Stel Finkelstein

Seventh International Workshop on High Performance Transaction Systems (HPTS)

Asilomar, California
14-17 September 1997

General Chair:	**David Vaskevitch**
Program Chair:	**Bruce Lindsay**
Program Committee:	
Sam DeFazio	**Jim Johnson**
Jeff Eppinger	**Gary Kelly**
Michael Franklin	**Ed Lassettre**
Dieter Gawlick	**Barbara Liskov**
Peter Gassner	**Susan Malaika**
Jim Gray	**Friedemann Schwenkreis**
Pat Helland	**Randy Smerik**
Joe Hellerstein	
Organization:	**Nancy Chapman**
	Diana Miller

Seventh International Workshop on High Performance Transaction Systems (HPTS)

Asilomar, California
14-17 September 1997

Contents

HPTS '97 Program vii

Seventh International HPTS Workshop Participants xi

Processes + Transactions = Distributed Applications
Gustavo Alonso, Institute of Information Systems,
Swiss Federal Institute of Technology 1

High Performance and Scalability Through Associative Client-Side Caching
Julie Basu, Stanford University; Meikel Pöss, Technical University of Berlin;
Arthur M. Keller, Stanford University 7

Design Transactions and Serializability
Philip A. Bernstein, Microsoft 13

Databases for Next Generation Telecommunications
Munir Cochinwala, K. C. Lee, Bell Communications Research 17

Issues to Argue About
George Copeland, IBM 21

The Coyote approach for Network Centric Service Applications: *Conversational Service Transactions, a Monitor and an Application Style*
Asit Dan, Francis Parr, IBM 23

Performance Archive Database (PAD) -- Data Mining Towards Self-Tuning Servers
Pranta Das, Girish Vaitheeswaran, T. K. Rengarajan, Sybase 49

Workload Management in Large Transaction Processing Systems
T. J. R. Dunn, IBM 65

Why PC Servers Won't Overtake Mainframe Servers Anytime Soon
Wayne Duquaine, Independent Consultant, Client/Server Interoperability 77

TIP: Gateway to Heterogenous Internet Transactions
Keith Evans, Johannes Klein, Tandem; Jim Lyon, Microsoft;
Francis Upton, Forte 81

Synergy Between Public Key Technology and TP Systems
Edward Felt, BEA Systems, Inc 87

Storage Metrics
Jim Gray, Goetz Graefe, Microsoft 91

Position
Don Haderle, IBM 93

Operational Data Stores Must Unite
James R. Hamilton, Microsoft; Patricia G. Selinger, IBM 95

Contents

CONTROL: Providing Impossibly Good Performance for Data-Intensive Applications
Joe Hellerstein, UC Berkeley 109

The Shock Of IRAM: Will Information Systems Be Ready for the Next Chip Technology?
Joe Hellerstein, UC Berkeley 113

Will non-determinism be a gating factor in Web transactions?
Tony Hey, University of Southampton; Charles Brett, C3B Consulting Ltd..... 115

TPMoniters - gone, but not forgotten?
Pete Homan, Mark Carges, BEA Systems, Inc..... 117

WorldFlow: A System for building Global Transactional Workflows
Mohan Kamath, Krithi Ramamritham, University of Massachusetts;
Narain Gehani, Daniel Liewen, Lucent Technologies/Bell Labs 119

Benchmarking Database Application Systems
Alfons Kemper, Donald Kossmann, Universität Passau 125

Caprera: An Activity Framework for Transaction Processing on Wide-Area Networks
Suresh Kumar, Eng-Kee Kwang, Tactica Corporation;
Divyakant Agrawal, UC Santa Barbara 131

Shirt Pocket Transactions
Tobin Lehman, IBM 141

Distributed Workload Balancing Deserves More Attention!
Benoit J Lhereux, NCR Corporation 147

Distributed Transactions in Java
M.C. Little, S. K. Shrivastava, University of Newcastle 151

Application Recovery: Closing in on an Elusive Goal
David Lomet, Microsoft 157

Embedded HTTP
Susan Malaika, IBM 161

Enterprise JavaBeans: Extending the JavaBeans component model to scalable three-tier applications
Vlada Matena, Mark Hapner, Rick Cattel, Shel Finkelstein,
Graham Hamilton, Sun Microsystems 167

Database Systems as a Reflection of the Changing Nature of High Performance Transaction Processing Systems
John McPherson, IBM 173

Data Management Issues in Biotechnology
Michael A. Olson, Molecular Applications Group 175

Contents

Isolation Level Testing
Pat O'Neil, Umass 179

Pipeline Server: A New Architecture for Server Performance on Modern Microprocessors
Michael Parkes, Rick Vicik, Charles Levine, Microsoft 183

Implementing Extended Transaction Models
Calton Pu, Roger Barga, Tong Zhou, Oregon Graduate Institute;
Shu-Wie Chen, Columbia University 185

Building the Global Payments System for the Next Millenium
Bill Reid, VISA International 191

Relational Data Access on the Web: The Argument for a Dataless Database
Eugene Shekita, Dan Ford, IBM 195

Self-Tuning Scalable Servers
Peter Spiro, David Campbell, Microsoft 199

Industry Standard Benchmarks, What's Real and What's Smoke and Mirrors
H. Reza Taheri, Hewlett Packard 203

A Framework for Configurable Distributed Transactions
S. M. Wheater, S. K. Shrivastava, University of Newcastle 205

High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++
Arthur Whitney, KX Systems; Dennis Shasha, Corant Institute, NYU;
Stevan Apter, Union Bank of Switzerland 211

Contents

Sunday: 1:00 - 5:00
Registration

Sunday: 6:00
Dinner

Sunday: 7:00 - 10:00
Reception

~~~~~~

Monday: 8:30
Kickoff / Keynotes *(Bruce Lindsay)*
David Vaskevitch - Microsoft
"Where You Want To Be"
Don Haderle - IBM
"Object Relational and Data Integration"

Monday: 10:00
Break

Monday: 10:30
Technology Futures *(Jim Gray)*
Dennis Roberson - NCR
"Smart Cards and how they will revolutionize transaction processing"
Ed Lassettre - IBM
"Great software needs greater hardware -- design for the future and plan to trade hardware for functionality and usability"
Justin Rattner - Intel
"Greater hardware -- the revolutions in communication, in computer architecture, and in computer economics"

Monday: 12:00
Lunch
~~~~~~

Monday: 13:30

The Impact of Business Issues on Application Programming Models (*Don Haderle*)

Pat Helland - Microsoft
Mark Carges - BEA
Ed Lassettre - IBM
Randy Smerik - Top End

Monday: 15:15

Break

Monday: 15:45

Network *(Randy Smerik)*

Vlada Matena - SUN
"Java Beans"
Santosh Shrivastava - Newcastle
"Java Transactions"
C. Mohan - IBM
"DB2/MQ integration"

Monday: 17:30

Break

Monday: 18:00

Dinner

Monday: 19:00

Evening I *(Susan Malaika)*

TPmonitor updates & Work-in-progress mini-talks

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Tuesday: 8:30

Applications *(Jim Johnson)*

James Chong - Charles Schwab
Munir Cochinwala - Bell Research
Pete Heisinger - VISA
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Tuesday: 10:15
Break

Tuesday: 10:45
Self Tuning *(Friedemann Schwenkreis)*
Pranta Das - Sybase
"Performance Archive Database: PAD"
Peter Spiro - Microsoft
"Self Tuning Scalable Servers"
Tim Dunn - IBM
"Workload Management in Large Transaction Processing Systems"

Tuesday: 12:00
Lunch

Tuesday: 13:30
Panel: Do Industry Standard Benchmarks have Value to the Buyer of Transaction Processing Systems? *(C.Brett)*
Reza Taheri - Hewlett Packard
Jim Johnson - Standish Group
Bruce Lindsay - IBM
Randy Smerik - Top End

Tuesday: 14:45
Break

Tuesday: 15:15
Is the Web Server a TPmonitor? *(Bruce Lindsay)*
Pat Helland - Microsoft Viper Xaction Server
Jeff Eppinger - TransArc Encina
Pete Homan - BEA Tuxedo
Susan Malaika - CICS
Randy Smerik - NCR - Top End

Tuesday: 17:00
Break

Tuesday: 18:00
Dinner

Tuesday: 19:00
Evening II
Consultants Couch *(Brett & Johnson & Boucher)*

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Wednesday: 8:30
Workload balancing and scaling *(Peter Gassner)*
Benoit Lheureux - NCR
"Distributed Workload Balancing Deserves More Attention:
Rick Vicik - Microsoft
"Pipeline Server: A New Architecture ..."
Eric Brewer - Inktome
"Scalable Web Crawler & Indexer"

Wednesday: 10:00
Break

Wednesday: 10:30
Selected Technologies and Expected Implications *(Sam deFazio)*
Joe Hellerstein - Berkeley
"The Shock of IRAM"
Wayne Duquaine - Consultant
"PC Servers Won't Overtake Mainframe Servers Anytime Soon"
Mike Caruso - Insyte
"Analytic Objects"
Toby Lehman - IBM
"Shirt Pocket Transactions"

Wednesday: 12:00
End - Lunch
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Processes + Transactions = Distributed Applications

Position Paper, HPTS'97

Gustavo Alonso
Institute of Information Systems, Swiss Federal Institute of Technology (ETH)
ETH Zentrum, Zürich CH-8092, Switzerland
E-mail: alonso@inf.ethz.ch

March 10, 1997

1 Introduction

In addition to the conventional operating system interpretation, the concept of *process* is starting to be used to refer to complex sequences of computer programs and data exchanges controlled by a meta-program. Thus, today one finds notions such as *process centered software engineering*, *business processes*, or *process based parallelism*. In fact, the idea seems to have widespread acceptance in many areas, in particular in those where computation is based on cluster of PC's and workstations or on environments involving heterogeneous platforms and applications. A careful analysis of areas such as business environments [Fry94], software engineering [BK94] or scientific data management [ILGP96, BSR96] reveals a surprising number of problems that are both pervasive and common to all of them. Such pervasiveness may explain the attention being devoted to *workflow* products, which are the current process support systems. It may also explain why some researchers consider workflow management to be just a reincarnation of job control languages.

Today's computing environments, however, have changed significantly since the conception of job control languages. In practice, the most widely available platforms for corporate computing are based on multiple stand alone computers linked by a network. Such clusters offer the possibility of implementing truly distributed systems, which can be done in two ways: *top-down*, the entire system is designed from the beginning as a distributed system, and *bottom-up*, where already existing applications are used as building blocks. The former approach offers many interesting research opportunities and considerable attention has been devoted to it. The latter approach is, above all, practical since in most cases both the hardware and software infrastructures are already in place and cannot be discarded. The notion of process described above, derived mainly from workflow management, targets precisely distributed systems built following a bottom-up strategy, which is likely to be the approach of choice for future distributed systems.

In this short paper, it is argued that the notion of process, augmented with transactional properties, can be a very powerful tool for designing and developing distributed applications over clusters of PCs and workstations. This is done by first defining the type of distributed applications being targeted (Section 2), showing how the notion of process supports such applications (Section 3), and finally pointing out the role transaction must play to complement the notion of process (Section 4). The paper concludes with an overview of how these systems may evolve (Section 5).

1

2 Distributed Applications

The distributed application addressed in this paper are those based on already existing, stand-alone tools located on clusters of PCs and workstations linked by a network. The idea is to provide a way to use these existing tools as building blocks of a higher level system in which the process acts as the blueprint for control and data flow. Typical examples are business processes in which several office tools such as spread-sheets, text editors, databases, and human decisions are combined into a higher level entity by encoding the business logic within the flow of control and data of the process. Among existing products, the ones most closely related to the notion of process used in this paper are workflow management systems. Existing workflow tools, however, target almost exclusively either business processes or imaging systems, suffer from severe limitations related to performance and functionality [AAEM97], and are not easy to apply in other areas [MVW96, BSR96]. Workflow management, however, is a first step towards supporting the development of complex applications over distributed systems using already existing tools. Its concepts can be generalized by extending the notion of process to any arbitrary sequence of tool invocations (a script or pipelined UNIX commands, for instance, are simple forms of processes). In this way, future workflow systems could act as a high level programming tools linking heterogeneous, stand-alone applications. Integrated with additional technology such as CORBA, queuing systems or TP-monitors, workflow management could also very well play the role of distributed operating systems in which to exploit the coarse parallelism and distributed characteristics of distributed processes.

3 Processes

A process can be seen as a description of an arbitrary sequence of application invocations along with the data flow between these applications. As such, the process acts as a the meta-program governing the interactions among existing applications. Each step within a process is an *activity*, which represents invocations of external applications. The flow of control within a process – what to execute next – is determined by control connectors labelled with *transition conditions*, usually boolean expressions based on data produced by the activities. Processes also include programming constructs that allow modular design and nesting, as well as the invocation of other processes.

During execution, the *process engine* navigates the description of the process determining which activities are to be invoked next. The procedure is very similar to that of executing any other program (more like an interpreted program than a compiled one). The description of the process is usually stored persistently in a database and the engine consults the database continuously to find out what is to be done next. Any changes to the process are also stored persistently (the status of executed activities, returned values, etc.), which opens up interesting possibilities in terms of recovery and overall reliability [KAGM96]. When an application is to be invoked, the workflow engine notifies an *application-agent* located at the same node where the program resides. The application agent then executes the program and returns the results of the program to the workflow engine.

These ideas hint at the possibility of considering the workflow engine as a distributed operating system executing programs that have been constructed using existing applications as programming

primitives. In practice, workflow engines do act as schedulers and resource allocators in distributed environments. But, unlike in the case of centralized operating systems, workflow engines have to contend with the network, which immediately brings up issues such as atomicity of invocations, interferences between concurrently executed processes, performance, and overall consistency. Here is where transactional notions should play an important part.

4 Transactions

Transactions are the conventional form of encapsulating database operations so as to provide *Atomicity*, *Consistency*, *Isolation*, and *Durability* [GR93]. They provide not only clean semantics to the interactions between concurrent executions but also a powerful abstraction in which to base optimizations to the architecture of a database system. This same ideas can be applied to processes so as to provide useful abstractions to reason about the correctness of the system, to express properties of the execution of processes, and to tune the overall architecture of process support systems. It is likely that transactional ideas applied to process support systems will be as successful as when applied to databases (which, to certain extent, has always been the goal of many advanced transaction models [Elm92]). However, in the case of processes, database-like transactions are too rigid. Some form of *light-weight* transactions should be used [BRS96]. Thus, within a process, instead of using a unique construct encompassing all transactional properties, e.g., *BOT/EOT*, several separate constructs should be used to group activities according to the desired semantics. Thus, one could consider *spheres of atomicity* (atomic units with the standard all or nothing semantics), *spheres of isolation* (isolation units, much like critical sections in traditional operating systems), and *spheres of persistence* (determining whether the activitys in the sphere are to be made persistent or not). Spheres can be naturally combined with the nested structures provided in most process specification languages. Such constructs could have the following semantics:

Spheres of Atomicity. Atomicity is the "most popular" property from transactions in that it addresses a common coordination problem. There have already been many suggestions regarding how to use atomicity in advanced transaction models [Elm92] and it has also been suggested as a basic mechanism for process navigation [Ley95]. In fact, process support systems may offer the only realistic scenario in which to use the many ideas related to compensation. For instance, activities could be declared to be *Basic* (non-atomic), *Semi-atomic*, *Atomic*, *Restartable*, or *Compensatable*. Basic activities are the default and correspond to non-atomic applications, i.e., those for which the system cannot guarantee atomicity. Semi-atomic activities are those providing enough information to implement a rollback method to be executed if the activity fails before completing its execution. Atomic activities are those that preserve atomicity by themselves, for instance, a transaction executed over an X/Open XA interface. Restartable activities [ELLR90] are those that can be invoked repeatedly until they eventually succeed. Compensatable activities are those that can be undone after they have finished using an user provided method attached to the activity interface[ELLR90]. Note that these categories can be applied at different levels, thereby allowing to group a set of activities into a semi-atomic block, for instance, or provide high level compensation for an entire sub-process by declaring it to be compensatable. The same navigation mechanisms used for processes could be used to drive operations related to atomicity properties.

Spheres of Isolation. The semantics behind the notion of spheres of isolation follow the ideas suggested in [AAE96], which point out that processes may require more a notion of synchronization in the traditional operating systems sense than the traditional database concept of serializability. For applications that demand a database like approach both spheres of isolation and spheres of atomicity could be used in a manner similar to that of nested or multilevel transactions [Wei91, Mos81], which provide a powerful mechanism to reason about recovery in applications with a complex structure [GR93].

Spheres of Persistence. Spheres of persistence could be used to avoid the overhead incurred by storing all process information in a database. Using databases for storage provides significant advantages in terms of reliability, monitoring and audit control [AAEM97, KAGM96], but introduces a considerable overhead. Within a process, a programmer could specify whether a set of activities is to be executed persistently or not. If an activity or a group of activities are embedded within a sphere of persistence, every step of the execution is recorded in the underlying database. This guarantees forward recoverability in the event of failures. The information related to activities not included in a sphere of persistence about the execution is maintained only in main memory. Upon completion or after pre-determined time intervals, this information could be checkpointed to the database in an off-line fashion, thereby avoiding the I/O overhead.

5 Processes + Transactions = Distributed Applications

In practice, the equation that serves as the title for this position paper should include *Standarization* on its left side. The amount of efforts being devoted to CORBA, OLE, SOM, DSOM, and ODBC, to mention a few, show the growing interest in being able to link together stand alone tools. In terms of products, the same goal can be ascribed to TP-monitors, persistent queuing systems, CORBA implementations, and workflow management systems. These products are slowly converging towards a common point, as proven by the many ongoing attempts at combining their functionality. For instance, TP-monitors implementing the execution guarantees in CORBA, CORBA extensions to TP-Monitors, efforts to combine workflow standards and CORBA under the notion of business objects, or queuing systems being added to workflow management systems. Once applications follow a given standard interface and these interfaces have been widely accepted, the task of linking together stand alone systems will be greatly simplified. But in addition to the standards there must a way to express interactions between stand alone applications. A very powerful paradigm for this purpose is processes, with the added advantage that the technology for process support is almost already in place. Certainly it will not be in the form of current workflow management systems, but it is likely that it will be in the form of a combination of mainly TP-monitors (for transactional guarantees), queuing systems (to allow asynchronous interactions), CORBA (and/or any other standard providing a common way to interact with applications), and workflow management (for process support). In a way, this is not very different from early transactional operating systems [GR93], but it is an idea that offers very interesting research and commercial opportunities and may give transactional concepts a significant relevance in future information systems.

References

[AAE96] G. Alonso, D. Agrawal, and A. El Abbadi. Process Synchronization in Workflow Management Systems. In *8th IEEE Symposium on Parallel and Distributed Processing (SPDS'97). New Orleans, Louisiana - October 23-26, 1996.*, October 1996.

[AAEM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and Limitations of Current Workflow Management Systems. *IEEE Expert (to appear)*, 1997.

[BK94] I.Z. Ben-Shaul and G.E. Kaiser. A paradigm for decentralized process modeling and its realization in the oz environment. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento. Italy, 1994.

[BRS96] Stephen Blott, Lukas Relly, and Hans-Jörg Schek. An open abstract-object storage system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[BSR96] A. Bonner, A. Shrufi, and S. Rozen. LabFlow-1: A Database Benchmark for High Throughput Workflow Management. In *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT96), Avignon, France*, March 1996.

[ELLR90] A.K. Elmagarmid, Y. Leu, W. Litwin, and M.E. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. In *Proc. of the 16th VLDB Conference*, August 1990.

[Elm92] A.K. Elmagarmid, editor. *Transaction Models for Advanced Database Applications*. Morgan-Kaufmann, 1992.

[Fry94] C. Frye. Move to Workflow Provokes Business Process Scrutiny. *Software Magazine*, pages 77–89, April 1994.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

[ILGP96] Y.E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A desktop Experiment Management Environment. In *Proceedings of the 22nd VLDB Conference, Mumbai (Bombay), India*, September 1996.

[KAGM96] M. Kamath, G. Alonso, R. Günthör, and C. Mohan. Providing High Availability in Very Large Workflow Management Systems. In *In Proceedings of the Fifth International Conference on Extending Database Technology (EDBT'96)* , Avignon, France, March 1996. Also available as IBM Research Report RJ9967, IBM Almaden Research Center, July 1995.

[Ley95] Frank Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *GI-Fachtagung Datenbanken in Büro Technik und Wissenschaft - BTW'95*, Dresden, Germany, March 1995. Springer Verlag.

[Mos81] J. Eliot B. Moss. Nested transactions: An approach to reliable computing. M.i.t. report mit-lcs-tr-260, M.I.T., Laboratory of Computer Science, April 1981.

[MVW96] J. Meidanis, G. Vossen, and M. Weske. Using Workflow Management in DNA Sequencing. In *Proceedings of the 1st International Conference on Cooperative Information Systems (CoopIS96), Brussles, Belgium*, June 1996.

[Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1), March 1991.

High Performance and Scalability Through Associative Client-Side Caching

Julie Basu
julie@cs.stanford.edu
Stanford University
Computer Science Department
and Oracle Corporation

Meikel Pöss
s_poess@ira.uka.de
Technical University of Berlin
Computer Science Department
D-10587 Berlin, Germany

Arthur M. Keller
ark@cs.stanford.edu
Stanford University
Computer Science Department
Stanford, CA 94305-9020, USA

March 24, 1997

Abstract

*A*Cache is an associative client-side caching scheme proposed in [6]. The cache at a client site dynamically loads query results and uses predicates based on the queries to describe its current contents. These predicate descriptions are used to determine query containment for local query evaluation at the client and also for cache maintenance. In this paper, we demonstrate via a simulation study that the A*Cache system provides high performance and scalability with respect to different workloads and large number of clients.*

1 Introduction

Client-server configuration is a popular architecture for modern databases. This setup involves one or more server processes that manage a repository of persistent data and handle requests for data retrieval and update from multiple client processes. Clients are autonomous entities that may be located on the same machine as the database server or on different machines. With the current prevalence of distributed computing, the latter scenario is most common, where client transactions are initiated from desktop workstations and communicate with the server through explicit messages across a local-area or a wide-area network.

The server response is a critical factor in the performance of a client-server database. Resources of the server are shared among all clients, and can become the bottlenecks in scaling the system to large database sizes and many clients. Optimizing the performance of the server has thus been a major focus of commercial systems. However, the revolution in computer hardware technology has made client resources relatively cheap and plentiful. Today's *smart* clients can perform intensive computations locally, using the database as a remote resource that is accessed only when necessary. Therefore, the system design must now take into consideration client memory and CPU for data caching and query evaluation purposes. Despite the pffotential cost of maintaining data cached at client sites, a number of recent studies for object-oriented databases [3, 4, 11] have demonstrated that increased client-side functionality generally improves the system performance.

A new client-side caching scheme for client-server databases was proposed in [6]. The client cache, henceforth called the *A*Cache*, dynamically loads query results during transaction execution,

and uses query predicates to formulate predicate descriptions of the cache contents. A*Cache supports *associative* data reuse across transactions — new queries are compared against the cache description usingff predicate-based reasoning [7, 8] to determine if the query can be evaluated locally. Predicate descriptions of client caches are also registered at the server, and are used to generate update notifications for cache maintenance. The clients the notifications to maintain the validity of their cache descriptions and data.

The A*Caffche scheme provides a number of benefits over ID-based caching for *navigational* purposes, as in [3]. A primary advanantage is that it supports the processing of associative queries locally at client sites, thus improving data reuse. Server indexes are not required for query execution at client sitesff — the clients may construct local indexes to faciliate query evaluation. The cache maintenance method is completely flexible, e.g., automatic refresh or invalidation upon update, and can vary by client and even by individual queries. Several new optimization and approximation techniques can be designed in this context; a detailed discussion of the design choices appears in [6].

Operating the A*Cache requires reasoning with predicates in a dynamic environment, naturally raising questions about its performance and scalability. Earlier papers have demonstrated the effectiveness of A*Cache in the presence of moderate-to-high update loads [1]. More results on the performance of the A*Cache scheme for different workload types will be presented in the full version of this paper. We present below a brief summary of our scalability results for the A*Cache scheme.

2 Overview of A*Cache Architecture

The persistent data store is resident at the server and transactionffs are initiated from client sites, with the server providing transactional facilities for shared data access and recovery. The configuration is non-shared memory, so that the address space of each client process is disjoint from that of the server and of other clients. Separate subsystems exist at each client site and at the central server for cache management. In this paper, we assume that a client runs a single transaction at any given time. Thus, local concurrency control and lock management issues are not considered at the client. Figure 1 shows an client-server A*Cache system with one client. Due to space constraints, we omit detailed description of the various components here; details of the system operation can be found in [1, 6].

3 Related Work

A couple of recent studies [2, 10] have examined associative access to a client cache. Both of these studies are related to the associative caching model presented in [6] ffbut are limited to read-only scenarios. The *semantic* caching study in [2] investigates cache replacement policies for no-update workloads. A cache manager called WATCHMAN for read-only caching of query results in data warehousing environments is presented in [10]. Neither of these two studies consider the important issue of cache maintenance when there are database updates, and the performance and scalability of the system under update loads.

4 Simulation Analysis

In this simulation study, we consider a single relation with 100,000 tuples of 200 bytes each in a Wisconsin-style scenario [5]. The relation has two indexed attributes *unique1* and *unique2* that are unclustered and clustered respectively. Each query is a linear range selection either on *unique1* or on *unique2*. The predicates in a cache description are linear intervals corresponding to query ranges, and they are scanned in sequence for checking query containment and for generating update notifications. Figure 2 shows our workload parameters for the simulation. The system paramaters such as CPU speeds of the server and client CPU speeds are omitted due to space constraints — they appear in [1].

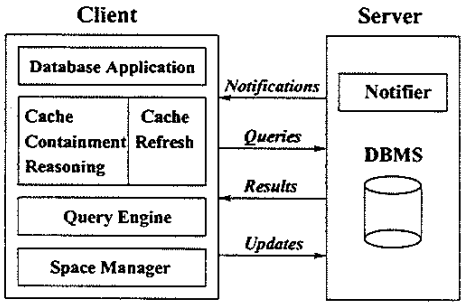


Figure 1: A*Cache System with 1 Client

DBsize	100,000	Database size in tuples
TupleSize	200	Tuple size in bytes
QueryLength	100	Query length in tuples
PrivateRatio	80%	Private region as a % of database size
PrivateAccessProb	50%	Access probability of private region
SharedAccessProb	20%	Access probability of shared part ($= 1 - PrivateAccessProb$)
PrivateWriteProb	20%	% of update transactions in private region
SharedWriteProb	20%	% of update transactions in shared region

Figure 2: Workload Parameters for Simulation

Our workloads model different degrees of shared data contention among clients. The database is logically divided into a shared and a non-shared (i.e., *private*) part. Each client can access data in the shared part and in its own private region. The private region for client i is defined by the linear interval : $[\frac{DBsize*PrivateRatio}{NumClients} * (i-1), \quad \frac{DBsize*PrivateRatio}{NumClients} * i)$. We define data access probabilities *PrivateAccessProb* and *SharedAccessProb*, and update probabilities *PrivateWriteProb* and *SharedWriteProb* for transactions in the private and shared regions of the database respectively. These quantities are the same for each client. Transactions consist of a single associative query or update, with an update transaction reading and writing all tuples that it accesses.

Figure 3 shows the A*Cache performance when the number of clients in the system is scaled up. The number of clients is varied from 20 to 70 while other parameters, such as the buffer size at the server, are held constant. Data access is by the clustered index *Unique2*. Half of the queries here are within the private region of each client and the *UpdateRatio* is 20%. Although A*Cache has a larger number of notifications with more clients, the query response time is better than the no-caching scheme. This result is due to utilization of the CPU power and memory of the additional clients by A*Cache. It shows that A*Cache is highly scalable with respect to the number of clients in the system.

In Figure 4, we varied the query length from 50 tuples (1KByte) to 300 tuples (6KBytes) for a database size of 10,000 tuples and 10 clients. Notifications increase with larger query lengths, but A*Cache performs much better than the no-caching case. The reason for the linear increase in the no-caching query ffresponse time is the almost linear increase in execution time at the server

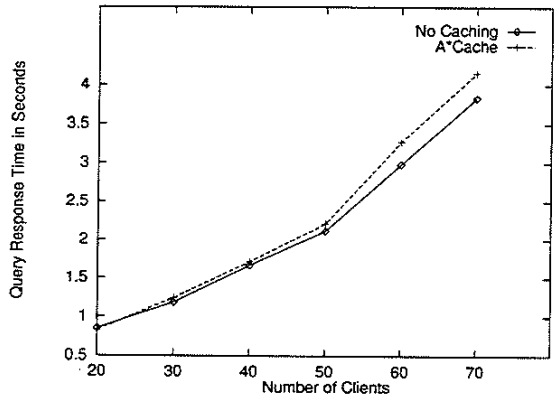


Figure 3: Varying Number of Clients

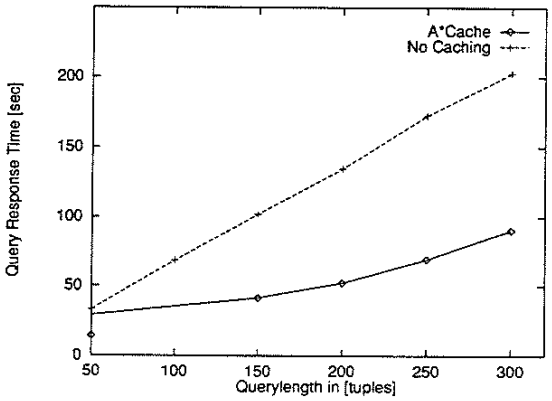


Figure 4: Varying Query Length

with increasing query length, causing the server CPU and buffer utilizations as well as the query response time to increase. In contrast, A*Cache re-uses local data effectively (*CacheHitRatio* = *63%*), so that majority of queries are executed at the client and not at the server. Therefore, the query response time for A*Cache increases less than linearly with the query length.

5 Implementation Approaches

To determine the feasibility of implementing the A*Cache scheme with current technology, we examined the issues in developing a prototype system with commercial database server, and a main-memory database (*SmallBase* [12]) serving as the client-side data store with query execution facilities. If the server is assumed to be a "blffack-box" opaque entity, the notifier must be an "add-on" external module. Although closer integration with the server would yield a more optimal implemenfftation, it is possible to use the data replication facilitites provided by the commercial systems to generate update notifications. For example, in Oracle, updates ffto a relation can be automatically inserted in a special table called a *snapshot log* [9], which can be subsequently examined by a separate notifier process. Likewise, client-side modules such as the cache containment reasoning system and the update handler can be layered on the basic query engine. In brief, extending a client-server system to support client-side A*Caches is very possible with today's technology. Our colleagues, Kurt Shoens and Marie-Anne Neimat, have been implementing sucffh a prototype system.

6 Conclusions

Associative caching is an important technique to improve the performance and scalability of a client-server database by better utilization of client CPU and memory. Our simulation experiments (only a few of which are reported in this abstract) demonstrate that A*Cache provides high performance even for moderately high update loads and under many different contention scenarios. The system is highly scalable with respect to the number of clients and query sizes, and performs consistently better than the no-caching scheme in most cases.

Future work includes the investigation of different cache maintenance policies, such as invalidation of infrequently-used query results and update of frequently-used ones. Using advanced techniques to determine query containment[7, 8] and to generate update notifications are other aspects of future work.

References

[1] J. Basu, M. Poess, and A. M. Keller, "Performance Analysis of Associative Caching in the Presence of Updates", Submitted for Publication, Feb. 1997. Available on the WEB at *http://www-db.stanford.edu/ basu/pub/acache_performance.ps*

[2] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, and M. Tan, "Semantic Data Caching and Replacement," *22nd Intl. Conference on Very Large Data Bases (VLDB 96)*, Bombay, India, September, 1996.

[3] M.J. Franklin, "Caching and Memory Management in Client-Server Database Systems," *PhD thesis*, University of Wisconsin-Madison, 1993.

[4] M.J. Franklin, B.T. Jonsson, and D. Kossmann, "Performance Tradeoffs for Client-Server Query Processing," *Proc. ACM SIGMOD Conf. on Management of Data*, 1996, Montreal, Canada, pp. 149-160.

[5] Jim Gray, "The Benchmark Handbook for Databases and Transaction Processing Systems,", Morgan-Kaufmann Publishers Inc., 1993.

[6] A.M. Keller and J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *The VLDB Journal*, Vol. 5, No. 1, Jan 1996, pp. 35-47.

[7] A.Y. Levy and Y. Sagiv, "Queries Independent of Updates," *19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993.

[8] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering Queries Using Views," *Proc. PODS Conf.*, 1995.

[9] Oracle Corporation, "Oracle7 Server Replication Manual," 1995.

[10] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," *Proc. VLDB Conf.*, Bombay, India, 1996.

[11] Y. Wang and L.A. Rowe, "Cache Consistency and Concurrency Control in a Client-Server DBMS Architecture," *Proc. ACM SIGMOD Conf. on Management of Data*, Denver, CO, May 1991, pp. 367-376.

[12] M. Heytens, S. Listgarten, M.-A. Neimat, and K. Wilkinson, "SmallBase: A Main-Memory DBMS For High-Performance Applications," Technical Report, Hewlett-Packard Laboratories, Dec 1994.

Design Transactions and Serializability[1]

Philip A. Bernstein[2]
Microsoft Corp.

1. Introduction

A common criticism of the classical ACID transaction model is that it's inappropriate for design applications. A popular database research topic in the mid-1980s was the development of models to control multi-user design activities, notably version and configuration management [1,3, 4], to fix the inadequacies of classical transaction concepts. Today, version and configuration management is well established as the standard solution for controlling multi-user design activities, though database research probably had little impact on this outcome.

While the mechanism of version and configuration management is quite different than classical ACID transactions, I believe they both can be explained using serializability theory, thereby providing a conceptual unification of these quite different mechanisms. I also believe that classical ACID transactions have a natural place in an environment where design transactions are managed using version and configuration management mechanisms. This position paper briefly sketches these ideas.

2. Explaining Version and Configuration Management Using Serializability

There are three main abstractions to consider in a version and configuration management model: versions, configurations, and workspaces. This section presents a somewhat oversimplified model with just enough detail to make the point about serializability. The basics will be familiar to anyone who has done software development using a configuration management system.

Consider a simple version model where each object can have many versions. Given a *database* D that supports versioning, an operation D->CreateVersionedObject returns the *root version* of a new versioned object. (The notation O->M means invoke method M on object O.)

Each version is in the *frozen* or *unfrozen* state. When it is created, it is unfrozen. The operation v->Freeze makes version v frozen, which means that its state can no longer be changed.

Versions of an object are related by a partial order. Given a frozen *version* v, v->CreateVersion returns a new version that's an immediate *successor* of v. A version can have many successors; every successor version beyond the first one is called a *branch*. The operation v->MergeVersion(u) merges the contents of a frozen version u into an unfrozen version v, thereby making v a successor of u. Thus, a version can have many predecessors.

A *configuration* is a versioned object that represents a set. In an object-oriented database system (OODB), or any DBMS that supports relationships as a primitive type, this can be represented by *containment relationships* from the configuration to its contained *items*. Thus, configuration isn't really a type, but rather is an arbitrary object that has outgoing containment relationships. Configurations can be nested. That is, a configuration can itself be an item in another configuration.

A *workspace* is a configuration that contains versions that have been checked out from a configuration C for a particular user. The checkout operation on a version v creates a new version of v and puts it into the workspace. To save space, I'll skip the details of checkout (see [3], for example). Suffice it to say that the ideal scenario is that the user: creates a new version C' of C; checks out versions of some objects from C' to the user's workspace; gets read-only (non-checked-out) versions of some other objects into the user's workspace; performs some design work on the checked out versions; at some later time checks in the

[1] Position paper for The 7th International Workshop on High Performance Transaction Systems, Sept. 1997.

[2] Author's address: Microsoft Corp., One Microsoft Way, Redmond, WA, 98052-6399.
Email: philbe@microsoft.com.

checked out items thereby replacing the previous versions of those items in configuration C', and then freezes C'. In this scenario, the configuration version C' represents a design transaction, which consists of the work performed to transform C into C'.

If the version graph for a configuration has no branches, then the design transactions represented by the configuration's frozen versions executed serially. Each design transaction finished before the next one started.

Things are more complicated if two designers concurrently change the same configuration. Suppose two users want to update objects in the same configuration C. Then they each must create a new version of C, say C' and C". User1 checks out versions from C' and user2 checks them out from C". Eventually user1 finishes his or her design work, checks in all the checked out versions, and freezes C'. So far, the versions of C represent a serial execution. The problem arises when user2 finishes, checks in everything, and freezes C". Now we have two sets of incompatible updates to C. It is *exactly* as if we had two replicas of C that were updated independently by two users, without cross-posting the updates to the other copy.

The solution to this problem is to merge C' and C". In large software development projects, this is done all the time. The goal is to have the final configuration include the logical design changes performed by each user. That is, the result should be semantically equivalent to a serial execution of the two design transactions. At the physical level, where we see conflicting detailed updates by the two users, we may not be able to understand the state of the merged configuration based on how the conflicts were resolved. However, at some more abstract level, in the sense of multi-level transactions [5], the result must be serializable. That is, serializability is as applicable to design transactions as it is to the world of short ACID transactions. The difference is in the mechanisms being used.

Looking at products, we find that many OODBs offer some version management capability. However, when a repository is implemented on top to support version and configuration management, the repository usually ignores the underlying versioning model. For example, IBM's Team Connection and Unisys' UREP are implemented on OODBs but implement their own version model. This is part of a broader theme where repositories re-implement many high-level functions of their underlying OODB platform, because the latter's functions are not quite right. See [2] for more examples.

An interesting sidelight is the role of savepoints in design transactions. This capability is obtained by allowing objects in a workspace to be versioned within the workspace. That is, after checking out an object, you can periodically freeze it and create a new version. Each of these versions is effectively a savepoint. There are several useful ways to manage savepoints using versioning. One is to create a new branch off of an earlier savepoint, thereby following a different line of development. Another is to freeze the whole workspace and create a new version of it, so you can restore the frozen version of the workspace (a savepoint) later on. A third is to tag each frozen version with a timestamp (user-intelligible date and time), so you can back up to all the versions that existed as of a particular time. This is particularly useful because designers associate states of their design with real time events. E.g., I want to return to the state of the workspace I had at lunchtime yesterday (the last time I had everything working correctly together). It's as if the designer froze the workspace configuration at that time and later restored that workspace version.

3. Where ACID Transactions Fit In

Within a workspace, a user operates on checked out versions of objects. In today's software development models, the workspace is usually a file directory and the versions are versions of files. However, as we move into the object-oriented world of component-based development, we will instead want the workspaces and versions to be stored in a database. This is already true today for some high-end CASE environments, and it will be increasingly true in the future.

When versions in a workspace are managed by a database system, updates must be protected by transactions. It's the best way we know of to keep the contents of the database consistent when concurrent updates are present. Although the workspace is typically single-user, a user often operates on the workspace using multiple tools. The tool developer needs to use transactions to ensure that the tool's updates to the workspace are consistent with respect to those of other tools. Classical short transactions are a good way to accomplish this goal.

Transactions are also useful to ensure the ACID properties of operations on the shared configuration, such as Freeze, CreateVersion, Checkout, and Checkin. In particular, checkout and checkin can be complex operations that must be atomic and isolated relative to other checkouts and checkins on the same objects.

Most transactions that are executed in this environment are on objects that are private to a workspace and to a single machine, namely, the workstation used by the designer who owns the workspace. In this environment, there is no need for data sharing across multiple workstations using a transactionally coherent client cache — a much researched and highly promoted feature of OODBs. While such a cache might be useful for the operations on the shared configuration, it is far from critical. Since operations on the shared configuration are infrequent, it is usually acceptable to perform those operations directly on the server. A degree-2-consistent cache of frequently accessed shared objects on the client seems sufficient. So far, in our repository product work at Microsoft, we have had no customer requests to support a transactionally coherent client cache.

4. Conclusions

This position paper is a snapshot of a work in progress. That work includes a general model of version and configuration management that is sufficiently general to explain the behavior of all software configuration management systems we know of. The serializability explanations presented here are less well-developed at this time, but are a subject on ongoing work. This work is being done in the context of the Microsoft Repository product, where we are working on adding version and configuration management capability to the first release of that product (shipped in February 1997 in Visual Basic 5.0).

Acknowledgments

The underlying model of version and configuration management used here was developed in collaboration with John Cheesman and Paul Sanders of Texas Instruments, Inc.

References

1. Bancilhon, Francois, Won Kim, Henry F. Korth, "A Model of CAD Transactions," *Proceedings of the 11th VLDB Conference*, Stockholm, 1985.

2. Bernstein, P.A., "Repositories and Object Oriented Databases," in Datenbankesysteme in Buro, Technik, und Wissenschaft (Proc. BTW '97 Conference), Springer, March 1997, p. 34-46.

3. Chou, Hong-Tai and Won Kim, "A Unifying Framework for Version Control in a CAD Environment," *Proceedings of the 12th VLDB Conference*, Kyoto, 1986.

4. Katz, R. H., "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, Dec. 1990.

5. Weikum, G. and H.-J. Schek, "Architectural Issues of Transaction Management in Multi-Layered Systems," *Proceedings of the 10th VLDB Conference*, Singpore, 1984.

Databases for Next Generation Telecommunications

Munir Cochinwala, K. C. Lee
Bell Communications Research
445 South St.
Morristown, N.J. 07960
munir@bellcore.com, klee@notes.cc.bellcore.com

March 31, 1997

Introduction

The Telecommunication industry is going through an economic and technical revolution. The Telecom Reform Act is taking the telecommunication industry from a monopoly to open competition. An important aspect of open competition is Number Portability, the ability to 'own' your telephone number irrespective of location. The FCC has decreed that 100 cities should have service provider number portability by 1998 [1, 3]. Service provider number portability means that a subscribers' phone number remains the same when they change service providers. Future versions of number portability will also allow for location and service portability.

Internet advances allow new and sophisticated services to be provided to users. The internet also allows the Regional Bell Companies to provide open access to their systems to conform with the Telecom Reform Act. Services such as incoming call screening and outgoing call screening can be easily administered by users through the use of scrollable lists. Currently, services are limited by the cumbersome telephone interface.

Bellcore is in the unique position of providing local access expertise to the long-distance carriers and long-distance expertise to local access providers such as the Regional Bells. We believe that these changes are providing interesting and large scale database problems that are of interest to the HPTS community. We are actively working on solving problems related to the advent of number portability and new services.

Call Processing

We now describe the network databases used to route a telephone call and the impact of number portability and other services on these databases.

A telephone number can be a physical or a logical number. A physical number is an actual terminating address to which a telephone call can be routed. An ordinary telephone number is an example of a physical number. A logical number requires a translation to a physical number. An example of a logical number is an 800 number.

A translation from a logical to physical number requires a query to a number translation database. The number translation database is called the SCP (Service Control Point) and is

considered part of the telecommunications network. The SCPs are network databases [2] that are replicated across the United States and Canada. The SCPs consist of the 800 number database (used for 800 number translation [4]) and the Line Information Database (used for calling card and third party call validation). Currently, there are 43 SCPs in Northern America.

Each of the SCP is completely replicated and has 75 million unique records. The total size of the data in each SCP is around 30 gigabytes. Each SCP is configured to handle a peak load of 450 reads a second and 40 updates a second. The SCPs get their updates for number translation and call validation from administrative systems, (SMS 800 for the 800 number database and DBAS for the LIDB databases). The SCPs also get updates to maintain billing information.

Impact of Changes

Introduction of number portability means that each number is a logical number and requires a translation. Thus, all telephone calls will require a query to the SCP and the database in the SCP will grow to include all the numbers (250 million). Now, the SCPs will have to handle 200 billion calls a day (number from 1993). A single SCP will have to handle 50 million calls a day which translates to approximately 550 calls a second throughout the day. The peak rate can be an order of magnitude greater. The number of updates will also increase significantly both in the administrative and the real-time billing updates. The size of the database will also increase by a factor of 4 to 300 gigabytes. Thus, a single SCP will have to handle a peak rate of 5000 queries and several hundred updates a second for a database size of 300 gigabytes. Such rates are difficult to achieve.

Adding new sophisticated services over the Internet further complicates the issue. Users can change services via the internet, increasing the frequency of updates to the SCPs. New services such as incoming call screening and outgoing call screening require lists of numbers to be maintained in the SCPs. These lists of numbers can grow arbitrarily large. Adding these services will increase the size of the SCP databases by 2-5K bytes per user. Now, the size of the SCP database grows to terabytes.

Each of the new service now requires complicated and multiple queries per telephone call. Imagine a user with outgoing call screening calling someone with incoming call screening. The initial dialed number has to be translated, screened via the outgoing call list and then screened via the incoming call screening list. Several queries have to be performed to complete a single call. This will further increase the load on the SCPs.

Conclusion

The problems of large, scale databases that can handle the workload generated by the changes due to the Telecom Reform Act are both interesting and challenging. The problems are further complicated by the reliability constraints on the telecommunication network. People expect reliable and robust communication from the telephone network. We are investigating partition and migration of data across SCPs to help with the solution.

References

[1] FCC 96-286 First Report and Order
CC Docket No. 95-116, July 2, 1996

[2] Amit Sheth Databases at Bellcore
Bellcore Technical Report, 1990.

[3] Report on Local Number Portability
Industry Numbering Committee (INC)

[4] A Recoverable Queue for Reliable Message Delivery Munir Cochinwala, Amin Abdul-Ghani, Kuo Lee, Chao-Chi Tong, Min Yu
Submitted to VLDB 97

Issues to Argue About:

George Copeland
IBM Austin
11400 Burnet Road
Austin, Texas 78758
copeland@austin.ibm.com
(512) 838-0267

A) Although objects were only mildly important a couple of years ago, now objects are THE way to do applications.

Reason:

- The web is the most significant thing to happen in the history of the computing industry.
- Java came with the web.
- Java is object-oriented.

B) Java (a package of interfaces - not the language) needs to be extended to support "managed objects":

- They have persistent identifiers that can be put in a URL or in the disk-version of an object that references the managed object.
- They have persistent state that is durable.
- They have access control on the object or object methods, not the data.
- They are concurrently accessible.
- They are recoverable via transactions.

The above properties are provided by an "instance manager" where managed objects live - like data lives in a database.

Reason:

- Objects model real-world entities.
- These real-world entities have the above properties.

C) HTTP and IIOP will replace proprietary protocols:

- HTTP for connectionless.
- IIOP for connectionful.

Reasons:

- All of the clients will support these - no install or download problems or overhead.
- Fewer protocols means fewer manageability problems.
- Directory and security can be done right.

The Coyote approach for Network Centric Service Applications:
Conversational Service Transactions, a Monitor and an Application Style

Asit Dan and Francis Parr
IBM T. J. Watson Research Center
Hawthorne, NY 10532

Abstract: The Internet stretches traditional strict transaction processing concepts in several directions. First, transactions spanning multiple independent organizations may need to address enforcement of pairwise legal agreements rather than *global* data consistency. Second, a new transaction processing paradigm is required that supports different views of *unit of business* for all participants, i.e., service providers as well as end consumers. There may be several related interactions between any two interacting parties dispersed in time creating a *long running conversation*. Hence, persistent records of business actions need to be kept. Additionally, some actions and groups of actions may be cancelable (however, this may not mean that all effects are undone, e.g., non refundable payments). Finally, the greater variability in response time for network computing creates a need for asynchronous and event driven processing, in which correct handling of reissued and cancelled requests is critical. This paper presents the COYOTE approach: an application development style and a monitor environment for supporting long running network applications and extended transaction models. We also provide a high level design for this monitor and describe briefly how network applications can be developed in this environment.

1 Introduction

1.1 Network Centric Computing is a New Context for Transaction Processing

The Internet and related technology trends (e.g., in end-user presentation interfaces and networking) have revolutionized the business transaction processing environment. *Business-to-end-user* and *Business-to-Business* interactions are both based on automated processing of asynchronous messages (i.e., service requests and responses). These interactions need to follow a long running conversation flow and need to have their effects made persistent. In such a *network-centric* business environment, independent business applications are created by autonomous organizations as *Conversational Service Transactions (CST)*, where the global or complete knowledge of interactions amongst all applications can not be known. Since transactions and transaction processing systems lie at the core of business computing, it is important to explore the extensions required to the classical transaction monitor [10, 5, 16] for supporting such new types of applications and how *network applications* can be developed in this new environment. (The additional requirements of such applications are summarized in Section 1.2).

In this paper, we advocate a service independence approach to application development and propose an execution environment for supporting long running conversational applications. We call the proposed approach COYOTE: for COver YOurself Transaction Environment, to emphasize that when several organizations participate in a business interaction across a public network, each organization can manage and be responsible for its own commitments and expectations but there is no party globally responsible for all activities contributing to the business interaction. In fact different participating organizations may have different views of where the boundaries of this particular

business interaction may lie. Localized execution of an interaction of a service application may well be a classical ACID transaction [16], or may even follow various extended transaction models already proposed in the literature [13, 15, 22, 25, 6, 9, 1].

1.2 Requirements on Infrastructure for Network Centric Service Applications

In addition to the traditional requirements for application development and runtime environment, service applications in a network centric environment need to have the following additional properties making them *Conversational Service Transactions (CST)*:

- *Service independence:* Each CST may be developed by an independent party where the complete interactions amongst all applications can not be known. Hence, the application development process is incremental.
- *Localized execution environment:* The processing of business logic is distributed and each component (i.e., CST) is run under a local application monitor. This also implies that application development and execution environments may be heterogeneous, and very little should be assumed of other execution environments.
- *Well known interfaces:* The interfaces of each CST as well as their operation semantics need to be known to other invoking applications. The interfaces are either known universally (e.g., EDI transactions) or are defined by the interacting parties via *service contracts* (see below).
- *Long running conversation:* Between any two parties, there may be many related interactions dispersed in time (e.g., delayed cancellation or modification of an earlier service request). Hence, the intermediate computation states of a conversation need to be durable.
- *Asynchronous invocations:* individual service action requests (making up the long-running conversation between client and server) are asynchronous, i.e., response to a request may not arrive immediately. There are many factors contributing to this requirement. First, since very little can be assumed of other CSTs and their execution environments, the response time of other CSTs may be unpredictable. Second, a CST itself may request services of other CSTs, adding to this unpredictability. Some CSTs may also require inputs from human operators. Finally, network response time and disconnected operations will also add to this unpredictability.
- *Compensation:* Since a CST may be long running, and traditional transactional execution of business logic across multiple organizations can not be assumed (see, further details in subsequent sections), *Compensation* of an earlier service request is a strong requirement. In real-world operations, this translates to refund of an earlier purchased item, cancellation of a reservation, exchanging items or changing the attributes of a previous request.
- *Coordinated invocations:* In many applications, it is natural to request a set of complementary services from independent applications in an *all-or-nothing* manner of execution. An example

of this requirement is coordinated purchases of items, such as, services of Hotel and Airline. The underlying runtime environment should ensure this all-or-nothing execution behavior.

- *Service contracts:* A service contract between two interacting parties (i.e., CSTs) documents their expected behaviors. It includes issues such as the interfaces that can be invoked, how long a conversation needs to be maintained and whether or not a service request can be compensated. The service contract needs to be enforced during execution time.
- *Query of interaction history:* A CST may need to query all its durable past interactions with other CSTs and the current states of interactions.
- *Site Autonomy, Privacy and Implementation Hiding:* A CST should be able to conceal from its requesters, whether parts of the service have been subcontracted out into the network; conversely the identity of the requester may sometimes need to be unavailable to any providers of subcontracted services.

1.3 Outline of the Paper

The remainder of the paper is organized as follows. In Section 2, we outline the Coyote approach: the conversational service transaction model, the Coyote monitor and the application development model. We further illustrate these concepts with an example. Subsequently, in Section 3, we describe in detail the essential concepts and building blocks, as well as itemize the functions provided by a COYOTE monitor. We provide further details of the compensation services supported by the COYOTE monitor, reliable execution of service requests, and service contract registration and enforcement in Sections 3.5, 3.6, and 3.7, respectively. In Section 4, we provide an overview of how the proposed COYOTE environment and services can be used for developing such service applications, and comments on the conveniences of this environment to the application developer. We then explore the relationships to earlier works: the ConTract model [25, 24], various Workflow/taskflow systems [1, 2, 12, 7, 20, 23] as well as component models [8, 21] for application development. Section 6 contains the summary and conclusions of the paper.

2 An overview of the COYOTE Approach

In this section we explain at a high level our proposed solution to meeting the needs of network centric service applications in terms of three components:

- Conversational Service Transaction model
- Coyote Monitor
- Coyote application development model.

We then illustrate the approach with an example.

2.1 Components of the Approach

2.1.1 Conversational Service Transaction Model

We treat requests for services as the fundamental building block for network centric service applications and define a simple integrity model for *conversational service transactions*. Each organization which wants to provide a network centric service application (i.e., CST) defines a set of well defined service access interfaces. These interfaces can be invoked by remote and/or other independent CSTs to access services provided by this one (see Figure 2(b)). We further advocate the use of an extended transaction model and a monitor supporting such pairwise interactions. The key concepts are:

1. Clients get service by having a *conversation* with the monitor at the server node,
2. The service requests which flow on these conversations are *encapsulated*, i.e., the client sees only a well defined service interface and does not know whether the implementation of the service involves sub-contracted services potentially at other nodes.
3. Valid and allowable service requests are defined in service contracts between interacting parties, and the service requests are monitored for enforcing this contract.
4. A client can request cancellation of a specific service request, a predefined grouping of requests or an entire conversation. However, there is no guarantee that the identified services will be completely undone; rather an application defined *compensation* action will be executed.
5. The monitor also provides runtime service interfaces to its local CSTs for coordinating their requests to other CSTs. A group of service requests to other CSTs are executed by the monitor in an *all-or-nothing* manner.
6. The monitor also provides to its local CSTs a persistent application level log and execution of service requests reliably in the face of reissued requests. This is required especially when response times (for the possibly distributed processing) do not meet the clients expectations.

2.1.2 Transaction Monitors and Specific Features of a Coyote Monitor

Coyote is a transaction monitor and hence, provides all the basic services expected of a transaction monitor. The basic functions of a standard transaction monitor are well known, e.g. CICS, Encina, Tuxedo [4, 16, 10]. We summarize them briefly here in order to highlight the additional properties of a Coyote monitor.

Transaction monitors are responsible for scheduling transaction invocations. Transaction programs or applications are registered to the transaction monitor in an administrative operation. Each registered application has a function name known to clients so that they can request it; associated with this function name is a registered entry point or program which is scheduled when a request for that function is received.

Incoming requests to the transaction monitor appear as messages typically from remote clients. An incoming request is logically queued by the transaction monitor until computing resource is available to execute it. This occurs when some other transaction completes and frees up a process or thread in a pool managed by the monitor. When a thread becomes available, the transaction monitor allocates it to the first queued incoming request, starts on the allocated thread the application program registered to the requested function, passing it any additional parameter data also in the request.

Having started the transactional application in response to an incoming message, the transaction monitor is responsible for supervising the execution of the transactional application. In particular it will intercept all outbound requests (to recoverable resource managers such as databases that may be local or remote) and to other remote transaction monitors if this transaction monitor can participate in managing a distributed transactional protocol.

Finally the transaction monitor is also responsible for the transactional behavior of the applications which it monitors. Definition of standard ACID transactional properties (i.e., atomicity, consistency, isolation and durability) can be found in [17, 16]. A primary function of standard transaction monitors is to provide a commit/abort protocol. This allows a transactional application to be ABORTED if either a serious error is detected by the monitor during processing of the transactional application or if the application explicitly requests it. The transaction monitor will then ensure that all effects of processing on behalf of this transaction instance are undone in the persistent resources of the participating resource managers (e.g., databases).

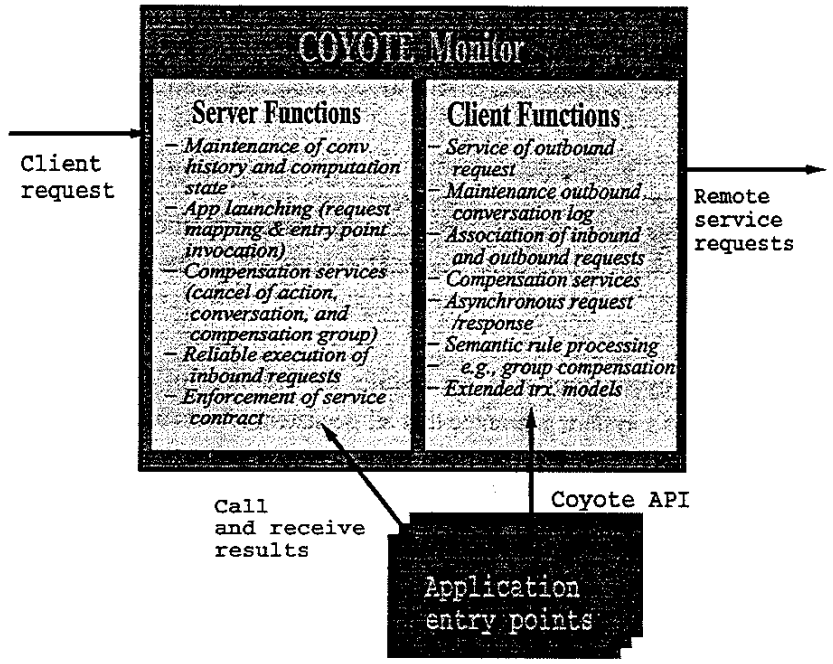


Figure 1: Coyote monitor

A schematic view of a Coyote monitor is shown in Figure 3. At this level the architectures of a classical ACID transaction monitor and a Coyote monitor are essentially the same. However,

a Coyote monitor differs from the above summary description of standard transaction monitor functions in several respects.

1. A server application started by a Coyote monitor is not by default an ACID transaction; rather it is a durable conversation for which there is no "system guaranteed" abort operation to undo all its effects.

2. An application level log of all interactions with the client and other remote resources and transaction managers is maintained by the Coyote monitor.

3. A service application registered with the Coyote monitor can have a group of programs or interfaces associated with it. The interfaces define the primary action, and its associated application defined compensation and modify actions.

4. Multiple related interactions that follow a logical conversation thread may be dispersed in time, and hence, the monitor maintains persistent conversation states and an interaction history.

5. The Coyote monitor has special support for enabling reliable execution of a service request on behalf of a particular conversation (with a particular user) even if action requests are reissued, and is reliably compensated (by the applicate defined compensation action) if the user so requests.

6. The Coyote monitor checks validity of invocation sequence as defined by the application during its registration.

7. The monitor also enforces service contract, i.e., checks to see if the user is allowed to invoke this method.

8. The Coyote monitor provides additional features to compensate an entire conversation or a defined group of service requests within the conversation, should compensation of the group be explicitly requested or automatically if some essential request within the group fails to execute successfully.

In short the novel features of a Coyote monitor is that it shifts the definition of a transactional application away from the concept of a system guarantee that all persistent effects are removed on failure, towards supporting the concept of persistent conversations in which it is easy to provide and manage application defined compensation of actions and provide the end-user with a reliable view of cancelling, reissuing and modifying particular service requests.

2.1.3 Coyote Application Development model

A network centric service application has to guide a client which may be an end-user or possibly another unknown program through a unit of business. This is typically a long running interaction, in which there are short bursts of automated activity at the Coyote server, interspersed with periods of waiting for, subcontracted servers in the network to respond, the user to decide what to do next

or activities to happen in the real world as part of the service fulfillment, goods to be moved, money to transfer or time between a reservation and use to elapse.

Thus a Coyote application consists of:

- *Interfaces:* its service interfaces (specifically service contract) presented to clients; the application will also depend on the service interface definitions for any services which it uses from other applications in the network in a subcontracting mode.
- *Action methods:* the implementations of each short running action which can occur in providing the service; each action involves responding to a specific event, processing based on this event and the state of the service conversation, and sending out messages and action requests to the client and other servers.
- *Scheduling rules:* the definition of the application's trigger events, i.e., combinations of response messages, timeouts and user requests, which will trigger execution of an action by the application, and the rules to determine which action is to be scheduled when events occur.

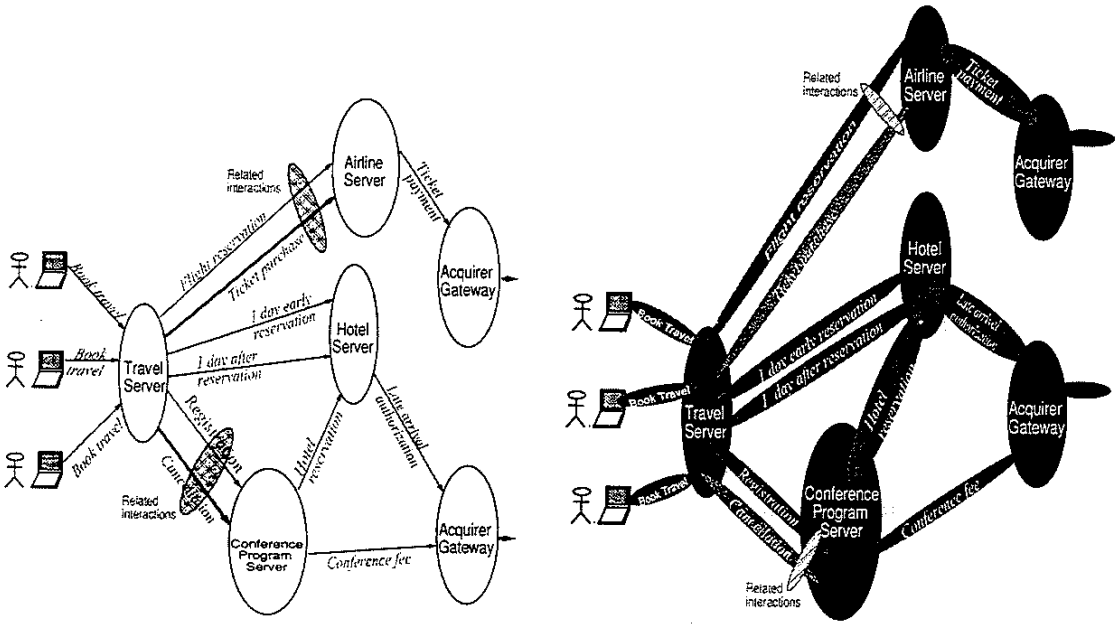
The developer of a network centric service application is required to provide all of the above information. The Coyote monitor plays the role of enforcing constraints defined in the service interface, gathering and saving the event data, interpreting the scheduling rules of the application, and then triggering the execution of action methods for the application following those rules.

We claim that the Coyote approach provides both a convenient and powerful structure for developing network centric service applications and a coherent model for characterizing their behavior.

2.2 An example: conference trip reservation

Consider a typical multi-organization long running business application: *Travel planning including conference registration.* [1] Figure 2(a) shows the participating organizations and their interactions. The *end-users* need to make complete travel plans associated with attending this conference. Each may run a local travel arrangement application on his/her desktop/laptop, but is more likely to contact an intermediary, *Travel Service* provider, that coordinates various end-user service requests, and maintains special business relationships with various business service providers (e.g., airlines, hotels). As part of the conference registration, the conference organizers provide (via the *Conference service application*) not only a seat at the conference and conference proceedings, but also arranges hotel accommodations for the entire duration of the conference. The conference organizers make special arrangements with the hotel (via the *Hotel Service application*) to provide this accommodation. They also collect appropriate fee (via the *Acquirer gateways*) through the credit service providers (e.g., Amex, Visa, MC). The end-user provides all the necessary information by processing an HTML form, and is typically unaware of all the business activities behind the scene.

[1] Note the multi-organizational, service independence and long running aspects of this example in contrast to the typical version of this example used in the earlier literature [25, 6, 13].



(a) Flow of service requests (b) COYOTE approach

Figure 2: Conference registration example

An end-user may also require a separate arrangement with the same or another hotel for an extended stay unrelated to the conference. For example, the end-user may decide to arrive early for some unrelated business activity (e.g., giving a talk at the local University) and stay a day longer beyond the conference for some sight-seeing. Additional activities may depend on various factors: availability of suitable flights, availability of hotel accommodation, and of course, no conflict in schedule. The conference provider is clearly not interested in providing services beyond what is absolutely required for conference attendance. The unpredictable schedule of a busy client may also require partial attendance or modification of the schedule on-the-fly.

The key requirements imposed by this overall application on the participating organizations and their underlying systems are as follows. Multiple organizations (e.g., Service providers for Travel, Conference, Hotel, Airline, etc) need to interact to provide all the required services to the end-user. All the services demanded by an end-user may not be known in advance, and may change during the course of the conference. For example, the requested and granted services may be cancelled by (or on-behalf of) the end-user at a later time. However, each organization is responsible for delivering its services. Each service provider develops its own application without the a-priori global knowledge of interactions across all organizations. Finally, the organizations do not need to be in constant communication during the course of this long running overall application.

Figure 2(b) illustrates the use of a COYOTE monitor for providing support for pairwise interactions. The lightly shaded areas (e.g., Registration, Cancellation) represent request interfaces, while

dark shaded areas (e.g., Hotel, Airline) indicate server nodes whose participation is typically not visible through the defined service request interfaces. Multiple interactions between server parties (e.g., Registration and Cancellation) may be related. Note that implementation of the "registration at the conference site" service, results in invocation of services that atomically perform reservation of a hotel bed and conference registration fee payment, as well as local book keeping such as updating the number of proceedings to be printed, etc.. Here, each transaction accesses/modifies its local database(s) in an atomic fashion, and hence, preserves desired integrity relationships within a site. Each organization is also responsible for managing its own interactions with other organizations and hence needs its own persistent application log of the interactions between them. We shall present this proposed extended transactional model (Coyote) by describing a monitor which supports it and provides useful services to server applications constructed with this approach.

In the above example, the travel server coordinates its interactions with conference, hotel and airline sites. The conference program server is responsible for ensuring all services that come with a conference registration, and coordinates its interactions with the hotel and the acquirer gateway servers. The end-user or the travel server never gets involved in the conference program server backend activities.

Each site also passes minimal (essential) information about its clients to other servers as well as hides its implementation of the ways it provides the services, particularly if it is dependent on the services of other sites. Without such information barriers, all participating sites may be aware of the relationship across all other (potentially hostile) sites. It may also jeopardize the businesses of intermediate sites since the end-client may directly contact final service providers. Additionally, the global state maintenance may impose unduly burden on each service provider.

3 Concepts and Services in the Coyote monitor

This section introduces the essential concepts of the Coyote model more formally to aid in the discussion. The central concept is that of a Coyote monitor. (In subsequent discussions, we will use the term Coyote to represent both the Coyote monitor and Coyote model depending on the context.) This is a system component located at a single node or processing engine. It receives messages, manages transactions and schedules server application processing.

3.1 Users, Conversations and Service Invocations, Action Requests

Users: The Coyote monitor understands the concept of a user on behalf of whom transactions can be executed, i.e., processing can be performed. The user is the individual asking for the transaction. Typically the user is located on some node different from the node where the the Coyote monitor resides and communicates by sending messages to this Coyote monitor. Local users could also exist at the Coyote monitor node but they would be treated by Coyote the same way as a remote user. The example expected to be typical is that of a person on a remote system running a Web browser and

sending HTTP requests to the Coyote monitor. A workstation based user executing an application at a remote site is the next leading example.

The Coyote monitor will keep a persistent record of the users which it knows and has executed work for as part of its directory and log. To identify a user it could eventually keep some Universal User Id generated by a certification agency but until standards for this become widely used it will probably keep a table of passwords for users established by this Coyote monitor.

The Coyote monitor will also keep a list of user ids which its local applications use to request services of other servers. These ids are defined as part of the application registration.

In a typical interaction, an end-user will browse through web content located at a particular Coyote monitor, looking for services of interest in an unstructured way. During this period, the user will typically not have identified himself by specifying a userid or password; Hence the interaction is anonymous at this point. When the user selects particular services and starts the process of requesting something, or attempts to discover what has happened to requests made by him/her at a previous time, identification via some sign-on process will be required.

Conversations: An identified user gets transactional services by starting a conversation with a Coyote monitor and flowing service requests on that conversation. The conversation is a grouping of service requests which is convenient and meaningful to the user. For example a user may choose to make all requests associated with a single travel trip a single conversation so that they are grouped together, can possibly be cancelled as a unit, and are separated from purchases of other items.

Each conversation has a single defined user who initiated it. Future work may address how these concepts can be extended to multi-party conversations.

Service invocations: Service invocations are the functional requests which flow on a conversation; examples would include making a specific hotel reservation or issuing tickets for a specified itinerary. To provide any useful business service usually requires more than one interaction with the end user, hence a service invocation is made up of multiple actions. An application available at a Coyote monitor, is defined as a named service with a collection of action methods.

A request identifies a service or transaction name. When the request arrives at the Coyote transaction manager, it causes processing of an instance of that named application to be scheduled. Various related actions (e.g., cancellation, modification) may be performed subsequently for an instance of a service request.

Action requests: Action requests are the atomic flows, represented by a single message from the user/client into the Coyote transaction monitor. These are specific requests to either start the service invocation, to cancel a previously started service invocation, to cancel or compensate for an active service invocation or a modify action to change what was previously done as part of that service. As an example consider the following sequence of action requests: (1) the initial action to initially make a hotel reservation (2) the modify action to confirm it with credit card number (3) the cancellation to clear it, as making up a service request associated with the application: *making a hotel reservation.*

3.2 Coyote Service Registration

The COYOTE monitor provides support for issuing, cancelling, reissuing and modifying service invocations, where cancellation implies taking some compensation action defined by the service. To provide this support, all services available at a Coyote server and their associated action methods must be **registered** with the monitor. Registration always involves defining:

- the name of the service ,
- each of the action functions: i.e. the primary issuing action for initially making the request, the compensate action (if it exists) and the modify action (if it exists),
- the signatures (prototypes) of each of the action functions, and
- the execution method for each of the action functions.

(We shall see in Section 3.7 some additional information also defined at registration.)

To explain what we mean by execution method we distinguish between registering **inbound** and **outbound** service requests to a Coyote monitor.

For every registered inbound service, the Coyote monitor is prepared to receive incoming action requests requesting that service. When this happens the monitor starts an application program (the indicated action) executing on a thread under its control to provide this part of the service. The execution method for an internal action is essentially the program entry point to be called on an appropriate Coyote managed thread when a request for that action is received.

The outbound services are the services which can be called by applications executing under the Coyote monitor to get at services provided by other service applications. These applications may be independent CSTs running at remote nodes or even in the local node under the control of a Coyote monitor. For actions which are part of outbound services, the registered execution method and the parameters of the IPC, RPC or HTTP requests are used to pass the request for action to the appropriate execution environment.

The function of the Coyote monitor is therefore to use its table of registered inbound services and their actions to schedule application processing when requests are received. It also intercepts all the outgoing action requests made by started applications and forward those requests, following information in its table of registered outbound services and actions.

The fact that a Coyote monitor treats inbound and outbound registrations separately providing different processing and services is illustrated diagrammatically in Figure 1.

3.3 Coyote Conversation Management and Request Processing

Conversation management: A user interaction with a Coyote monitor may optionally begin with an anonymous unstructured web browsing phase, but the proper conversation is established after the

user is reliably identified through some logon or password protocol. If the user is identified by some certified UUID in a public directory, then this can be used as identification, otherwise the Coyote monitor will use its own table of defined users and validate access by a password exchange. Once the user is identified and hence a record of previous persistent conversations of that user with this Coyote server is available, the user is given the choice of starting a new conversation or resuming a previously defined conversation. The Coyote monitor will in response either generate a new unique persistent conversation id or retrieve the identifier of the resumed conversation from its persistent log records.

The conversation id established in this way will then be passed in some form on all subsequent requests from this user. The monitor will maintain data structures so that conversation state and owning user identification can be efficiently recovered on receipt of this conversation id with each inbound request. Techniques for passing the conversation id on the request flow will vary depending on the implementation context for the Coyote monitor. If the requests into the monitor from clients are basic HTTP in a Web server environment, embedding hidden variables are one way of accomplishing this. If a Java shell applet is downloaded dynamically to client browsers starting a Coyote conversation, the conversation id may be passed as a formatted field with each request to the server monitor. For Object Oriented servers, the conversation itself could be an object and all subsequent service requests on this conversation rendered as method calls to this remote object using a protocol such as IIOP; in this case the conversation id is effectively being passed as the object reference on which methods are being called.

The conversation management interface to the monitor includes:

- start a new conversation
- close a specific conversation.

The close conversation command is issued when a conversation is completed and will not be used further for any action requests. Log records for the conversation will be kept for audit purposes but may be rolled out to some lower cost archival storage medium.

Processing inbound "new" action requests: Individual service invocations can now be made on the established conversation. A client will typically first issues an action for a "new" service invocation. These messages may not have a unique service invocation number established by the client; Hence when received, the server side Coyote monitor must do some processing to determine whether a received message is a valid new service invocation or whether it is a duplicate. This issue is taken up in subsection 3.6. The new action request starts the service invocation; multiple additional action requests can follow to modify or cancel it.

Each action request message includes fields for:

- the client supplied Service Invocation Number (SIN)
- the server supplied unique SIN

Client side unique numbering of requests is optional. For "new" action requests, server supplied SIN is always null on the incoming request (since it has not yet been allocated by the monitor). If the client does not uniquely number its service requests, (i.e., client SIN is null or omitted), the Coyote monitor will generate a unique SIN provided this does not appear to be a duplicate request. Specifically it will check that previous requests on this conversation for the same service function with the same request parameters for which no follow up modify or cancel has been received indicating that the client has been notified of the server SIN. All the information on requests needed to perform this checking is saved on the Coyote log as described in the following subsection. If the client SIN is supplied, this will be used to determine whether the "new" action is a duplicate. The processing of duplicate "new" actions is described with state diagrams in the section on Reliable Execution in Section 3.6. Having established that a valid (non-duplicated) new action request has been received, the Coyote monitor will generate a unique-within-conversation SIN for it and save it in the log. Parameters for the action request and if present the client's SIN will also be logged. The function identified in the request for action will then be looked up in the monitor's table of registered inbound service request functions. Assuming that a valid function has been requested, the monitor then uses the registered "new" program for this service request function, allocates an execution thread and starts the identified program on that thread. The server SIN, conversation id, user identification and action parameters are all made available to the launched application program instance.

The application program, launched to satisfy the incoming request, may generate during its execution multiple outbound action requests and associated outbound conversations to other services at this node or to other nodes. We will take up the monitor's processing of these outbound requests a little later in this subsection.

Eventually the launched service request program will complete. It then returns to the monitor with the output message to be returned to the requesting client. The monitor will free up the thread previously used by the application program, log any results, write the allocated server SIN into the reply message and send this message to the requesting client.

Processing inbound cancel and modify action requests: The processing of these requests differs from the processing of new action requests primarily in the method for associating a SIN with the received request. If client request numbering is used this association is used to determine the service request to associate with this action. If client request numbering is not used, then the server SIN must be supplied in cancel and modify actions and this will be used. The monitor then uses the appropriate "cancel" or "modify" program name in its table of registered inbound service request functions, and launches that application program on an allocated thread. In this case the application receives access not only to the user, conversation, SIN and action parameters, but also to the original parameters of the "new" service request which started this SIN. These parameters were logged by the monitor and are retrieved from the log and made available to the application to facilitate the modify or compensate processing.

Processing outbound action requests from a monitor started application: The Coyote monitor will intercept and handle outbound action requests from applications running under its control.

This may include requests to subsystems at this node or requests to other server nodes which may or may not be running under a Coyote monitor.

In order to request service from another server, the Coyote started application will establish an outbound conversation. This is similar to the inbound conversations used to pass service requests into Coyote. The application can start an outbound conversation using the Coyote interface OPEN_CONVERSATION(). The Coyote monitor starts a new logging thread as part of this conversation, and returns an unique *Coyote conversation number* to the application for later identification of this conversation.

The monitor will assign unique client SINs to all its outbound service request invocations and will log these SINs and the outbound action parameter lists in the Coyote log. It will also log reply parameters and any remote server SINs when it receives responses for these outbound requests. The monitor will associate these outbound requests with the conversation (on the input side) which launched the application making these requests. When the outbound requests are going to some resource manager or database not executing under a Coyote monitor (the most general and usual case unless Coyote monitors became widespread) there will be no notion of a Coyote conversation outbound, the servers handling these requests will have their own notion if any of conversation. Note that the outbound requests flowing from a CST to other CSTs will use the user id registered with the local CST. However, in some cases, the local CST may also use the user id associated with the inbound requests.

One thing which may also occur is that an application program executing under the monitor requests a service provided at the same monitor. This is processed as an outbound request from the invoking CST and inbound request to the CST to be invoked. The resulting processing is treated as part of the requesting conversation. A performance optimized monitor will typically execute a synchronous internal request on the thread of the caller.

3.4 The Coyote Log

As already alluded to in preceding subsections, the Coyote monitor will build and maintain a persistent log which records all incoming action requests, their associated client and server SINs and their input and reply parameter lists. For outbound service requests, the monitor stores similar information except that server SIN numbering cannot be relied upon in this case (just as client SINs were not required on the input side).

This information is maintained in a hierarchy:

```
user
    conversation
            inbound SIN + specific actions
                outbound actions.
```

Logging services are provided for the applications to retrieve this information, i.e., the log is presumed to flow over time into some relational database with indexes based on the hierarchy

above. However relevant action and parameter information is automatically made available to action programs when they are launched; The intent is that general searching is not part of normal "automated" request processing.

The same logging services used to build this system generated log are also available to Coyote application programs to build (as a separate log stream) their own application level audit log. Note that this is a very different intent (with rather different performance and structure requirements) from the recovery logs built by database and ACID transaction monitors to ensure transaction atomicity.

3.5 COYOTE Services for Compensation

In this section, we describe various COYOTE services used by an application for compensating a single or a group of actions.

We have commented above that ACID recoverable transactions are unlikely to be useful in recovery of a service invocation whose implementation is distributed across multiple organizations with no motivation to keep their database consistent. In this context cancellation or compensation is likely to be required (in many cases with multiple short conventional transactions embedded within the overall flow).

Coyote provides several levels of compensation support.

Compensation of action: The application may cancel an already granted service invocation, any time after it is started via an explicit request. The application passes the service invocation number to the Coyote monitor identifying this as a cancellation action. The Coyote retrieves all the parameters associated with the original request as well as the returned results from its log. The application may also be allowed to specify additional parameter values at cancellation time. If no such prior service invocation is found in the log or the service is already cancelled, or the cancellation request is too late, or an action request is still pending against that service invocation, an appropriate error code is returned to the application. In implementing the cancellation request, the Coyote monitor uses the fact that a specific *cancel* action was defined when the service interface was registered. If this were not the case, i.e. a service was registered with no defined cancel action, that would also be reported to the requesting application as an error

Modification of action: The application may also modify an active service invocation adding a new set of parameters. The Coyote monitor, as above, checks to see if the modify request should be honored, i.e., if it is allowed by the service interface. If so, the Coyote retrieves input parameters and returned results associated with the original service invocation. Any additional parameters passed by the client application on the modify action are also passed.

The processing of a modify is very similar to the processing of a cancel action . The differences are that (i) modify does not end the service invocation so that other actions in this service may follow (ii) modify lacks the implicit nesting semantics of cancel, i.e., unlike cancel, modify of a conversation is not defined, and hence, modify is not generated automatically as part of larger cancellation operations.

A typical example of a modify action in our example context, would be changing the parameters of a booking, either a date or an additional requirement such as a non-smoking room.

Compensation of conversation: As noted above service invocations are always part of a conversation. One conversation can include several service invocations. In particular in our illustrative example, the conference booking travel application would be implemented with a service request invocation to a hotel to get the room, and a separate service request invocation to the airlines to get the flight booking.

Either a client , or a Coyote application can request compensation of an entire conversation. This has simple nesting semantics. Compensation is called by the Coyote monitor for each of the service invocations in the conversation.

This corresponds to the user cancelling the entire trip. There may or may not be full refunds. Coyote will automatically call the appropriate compensation function in each of the logged service invocations for this conversation.

Compensation group: Sometimes it is desirable to group a set of service invocations, so that if any one fails the entire group is automatically compensated by the Coyote monitor. A Coyote application can define a compensation group using the `OpenCompGroup()` call to the Coyote monitor. The Coyote conversation number is passed as the input parameter, and the Coyote returns the *conversation group number*. Both the conversation number and the conversation group number are passed as parameters for service requests. For requests that are not part of a conversation group a null value is passed as conversation group number. The conversation group is aborted if any service request fails or if the application issues an explicit `CancelCompGroup()` call to the Coyote. A conversation group is closed by a `CloseCompGroup()` call.

Query of state: If less structured compensation and recovery logic is required in a particular application it can build explicit logic around the information available form the Coyote log. This information was described in Section 3.4.

3.6 COYOTE Services for Reliable Execution

An important characteristic of the network centric environment is that the network is inherently unreliable, and servers may be managed to widely varying responsiveness requirements. A request for remote service has an unpredictable response time, or may get no response at all. In order to be able to conduct useful business in this environment, the Coyote monitor provides services for reliable execution of remote requests. On the receiving side the Coyote monitor has to deal with a stream of incoming messages (service invocations and actions) which may have missing or duplicated members.

Human users of the internet have frequently to face the question: "Is there a remote server somewhere still working on my last request - or has it failed?". Application programs implementing network centric services will frequently be faced with this same choice. An important part of their transactional environment is having a consistent way to deal with it. The concepts of a conversational service transaction makes it possible to treat this problem because:

- a request for service is uniquely identified
- actions such as modifying or cancelling a service are architecturally defined.

A Coyote application calling out for services uniquely numbers service invocations. All action messages associated with that service invocation carry the service invocation number. Hence it is very easy for any remote server, Coyote or otherwise, to detect duplicate service invocations, and associate cancellations and modify actions with the proper service invocation. If there is no response to the request:

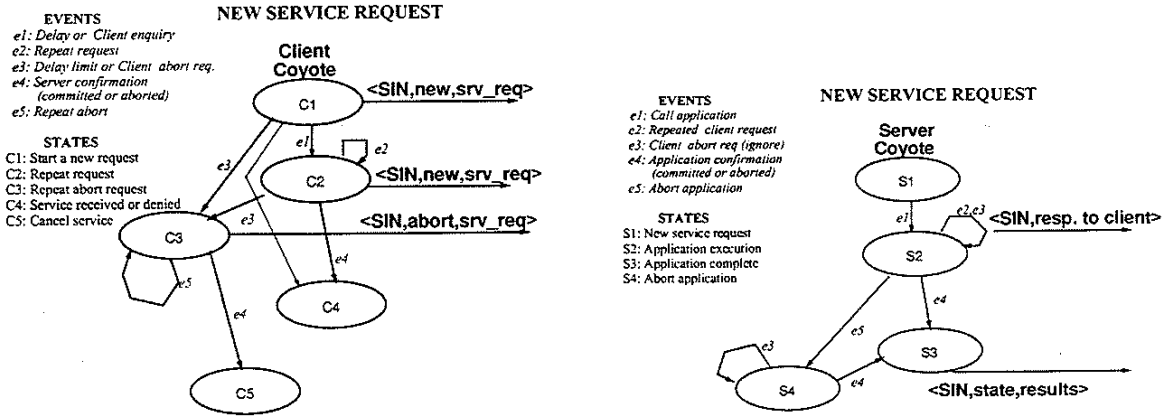
```
SIN# 1234 book hotel room   name: nnn
     hotel: hhh  dates: ddd     ...
```

within some expected timeout, then the resend action can still carry the same service invocation request number `SIN#1234` and hence there is no danger of ending up with two reservations.

When a Coyote monitor is communicating with a client - perhaps a simple browser - which does not uniquely number service invocations, then some heuristics are required to determine whether a request from the client is a "reissue" or whether it is a genuine independent service invocation. In the example context, if the server receives two reservations for Smith for the same night from the same client, should it assume that the previous reply message to the client got lost,(or the client got frustrated) or should it go ahead and make the second reservation. In our conversational service transaction model, whatever decision may be taken by the server, it will be recorded in the conversation log. This has the advantage that the client will be unambiguously informed using the conversation log whether or not the second message was treated as a separate service invocation (and given its own service invocation number). Furthermore this distinction is made in a generic way by Coyote, rather than as return information from the service application.

The preferred and more functional case is that the client does uniquely number its service invocations. Any client which is an application executing on another Coyote monitor will of course provide this. In this case, the Server Coyote will be able to do full and unambiguous checking for reliable execution. A simple example of the logical states in checking for repeat requests and cancellations between two Coyote monitors is illustrated with finite state machine diagrams in Figure 3.

This unique numbering of service invocations, and organization of the action methods within a service interface, also allows for sequencing rules on the actions within any service invocation. Specifically, as part of the service contract, the service will define the allowable sequences of action requests within any service invocation. For example, a service could allow at least one request for action `A`, followed by exactly one request for action `B`, followed by zero or more invocations of action `C` or `D`. In addition, the cancel can be issued at any point. These sequencing rules are formally defined as part of the service interface and checked on incoming action requests by the Coyote monitor. This is further explained in Section 3.7 where we discuss service contracts.



(a) Client Coyote states in reliable execution of a new request

(b) Server Coyote states in reliable execution of a new request

Figure 3: Execution states of a service request in clients and servers

3.7 Service Contract

When a service either inbound or outbound is registered to the Coyote monitor its semantics as seen by Coyote are registered in the form of a *service contract*. This encapsulates the following information:

- the name of the service
- the names and signatures of each of the action methods, and the cancel method
- the sequencing rules on the actions (see Section 3.6)
- authorizations on each action
- the responsiveness committed

This effectively defines a contract between requester and server. Note that different action sets within a service could be made available to different classes of user by using access control lists on each action method. The sequencing rules are used to manage possible loss or duplication of requests in flowing across the network. This sequencing would be checked and managed at the receiving Coyote server. Responsiveness is intended to give the client application some guidance as to how set useful timeouts. An average and a high percentile response time could be used to convey this.

4 COYOTE Application Development

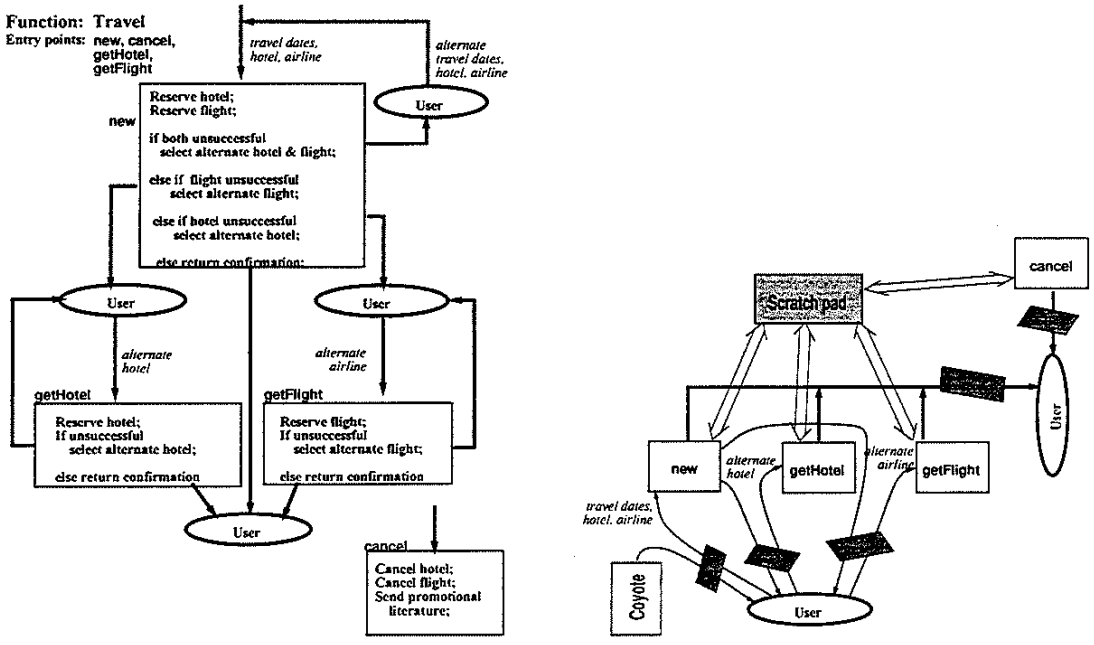
In Section 2.1.3 we gave an overview of the application development process indicating that the developer had to provide (1) service interfaces, (2) action definitions and (3) scheduling rules based on events.

The preceding section has described in more detail both the service contract defining the interface and the role of actions within a service invocation. In this section we will focus on the events and scheduling rules used to trigger valid sequences of actions to provide a complete service invocation

Events and scheduling rules play an important role in the design of network centric service applications, because the network environment drives the application designer to use parallel invocations of other network services and asynchronous messaging. Thus determining the "next step" in providing service may require determining which sets of responses, timeouts and additional requests from the client have been received. Trying to do this with conventional application processing logic is complex and potentially inefficient. A better solution is to allow the application to declaratively define events and triggering actions.

We use a travel reservation example to illustrate this point further.

4.1 Travel Booking



(a) Book travel pseudocode (b) Entry points in travel application

Figure 4: Application development

Figure 4(a) shows the logic of a travel booking application using sample pseudo-codes. The application logic is similar to the logic executed in the Travel Server of the conference registration example (see Figure 2). Upon receiving an user request along with the input parameters, the application first requests in parallel the Hotel and Airline (and also the Conference) servers for their services.[2] The customer later may cancel the BookTravel request (i.e., the entire set of travel arrangements on behalf of this customer), in response to which, the Coyote monitor in the Travel Server node finds all successful entries in its log and cancels corresponding requests. The log is maintained by the Coyote monitor and entries are created by intercepting all incoming and outgoing service requests. The Coyote monitor also facilitates reliable execution of all requests (see Section 3.6). The application code is executed upon receiving incoming service requests.

The above service requests in the BookTravel application can also be executed under various extended transaction semantics and compensation model. For example, the entire application may be viewed as an atomic operation. Under this semantics, if the execution of a service request to any of the other server fails, the application will be aborted and all the previous service requests under this application will be automatically compensated by the Coyote monitor. Alternatively, a subset of the requests (say, to the Hotel server and the corresponding car reservation request to the Rental server) may be combined as a compensation group. The Coyote monitor provides easy constructs for expressing this in the application, and leaves the details of its execution to the Coyote monitor.

An application developer first creates the application logic flow as shown in Figure 4(a). Each interaction with an user represents a break in the logic flow. This results in grouping of application steps into many small blocks, and a separate code segment can be developed for each such block (as supported by traditional TP monitors). The steps within a block can be expressed as transaction flow using the ConTract [25] model. However, the flow across blocks are guided by user interactions. Note however, the code segments for the blocks may share common variables that need to be persistent. The Coyote monitor stores the values of these variables in a conversational scratch pad which is hardened to a log upon exiting a block. These values are reinstantiated from the log and made available in the scratch pad when another related block for that application is invoked. Finally, the interaction with an user or a remote service node may require additional logic/processing. For example, an HTML form needs to be generated or an HTTP POST request processed for interactions with a human user. A different messaging format (e.g., EDI) may be used for interaction with an automated service node. The Coyote environment provides utilities and services for writing such applications. The complete application is defined to the Coyote environment by registering a set of ENTRY points and their associated application code segments and FORMs used by the application.

In this example an action triggering rule might be that:

```
after requests have been sent to airline and hotel services:

if both responses are received:
        => schedule action 1 (*advances to next step*)
```

[2]For ease of illustration, the request for conference service is omitted from the flow chart. Also in general such applications could be more complex, and may have many additional interactions with the user. For example, if the hotel is away from the conference location, the client may be asked if a car need to be rented from the Car Rental Server.

```
if one but not both responses received
and 90 secs have elapsed
        => schedule action 2 (*reports partial progress,
                              options for user*)
on late response (*user is told of partial progress*)
        => schedule action 3 (*saves response persistently
                              for next user interaction*)
```

Having these event and triggering rules of this sort stated explicitly simplifies the definition of the conversational service transaction and enables the monitor to schedule processing of it efficiently. The application may also use the rules based on a success or failures of a compensation group to trigger subsequent actions.

In summary, the Coyote monitor provides several important services to make writing business applications easier.

1. First, the Coyote monitor (as any transaction monitor) frees an application writer from the details of process management, application launching, state maintenance via log entry creation and management.
2. Second, it provides various extended transaction semantics and compensation constructs that are very useful in expressing application logic.
3. Third, it provides support for reliable execution of service requests that are useful in building distributed applications.
4. Next, Coyote approach provides an uniform model to write independent applications which are integrated via the monitor.
5. The Coyote provides services for form/message generation and processing for interactions with users or remote service nodes.
6. Finally, an important point to note here is that in the absence of Coyote services, individual applications need to duplicate these codes. Generally speaking, the code for these services is complex, but needs to be reliable. Hence, factoring out these code modules as Coyote services is important.

5 Comparison with Existing Work

5.1 Comparison with Traditional ACID Transaction Model

Many authors have identified shortcomings of the classical transaction model [13, 15, 22, 25, 9, 1]. These works address the issues of task dependency, semantics and correctness of execution. Here, we just recall the defining (ACID) properties of the classical transaction model [17, 16] to aid subsequent discussions. The key properties required of a transaction are: (a) *Atomicity* of the entire

set of operations performed by the transaction, (b) *Consistency* or data integrity across various data elements for preserving certain relationships (e.g., sum of checking and savings accounts of an user is constant after a transfer of money from one account to the other), (c) *Isolation* of operations and (d) *Durability* of operations. These properties made it feasible to maintain global integrity and consistency of data in a set of databases and helped in understanding the execution of transactions as a serializable stream of operations.

The above ACID properties have less relevance in the new Internet environment where end users are initiating potentially global business interactions, each one of which may span multiple independent enterprises. A participating organization cares much more about what it has committed to deliver and is legally obliged to do, rather than whether its database is consistent with the databases in partner organizations. For example, when a credit card company authorizes a charge amount against a particular card, it does not do so conditionally on whether some requesting transaction aborts; the authorization will just have some expiry date and will be valid if some confirmation request is received before then. Whereas the ACID transaction model emphasizes the ability of the transaction system and the resource managers to undo all effects on persistent data of a transaction which aborts, practical multi-party business interactions are more concerned with (1) have the business actions been durably recorded? (2) what application defined compensation actions are available if cancellation is desired? (3) what automatic expiry periods are required by the business and legal agreements between pairs of parties?

Note however, that the proposed conversational service transaction model for addressing the needs of the business service applications does not entirely replace the need for classical ACID transactions. It complements ACID and other advanced transaction models [13], and extends existing transaction processing environments for dealing with service independence and long running conversations. Within a single CST or conversational service transaction where data consistency issues are well understood and data integrity needs to be preserved (e.g., while executing individual service requests), ACID and other advanced transaction models will still be used. As has been suggested by previous works on extended transactions, there may be extended business interactions (i.e., non-ACID transactions) where the subportions may include ACID transactions. The earlier proposed various advanced transaction models seek flexibility in data dependency as well as improved concurrency [13, 9]. The Coyote monitor provides these services wherever needed. However, the Coyote monitor provides many additional services as detailed in earlier sections of this paper.

5.2 Conversational Service Transactions and Workflows

The advanced transaction [13, 9, 15, 22, 6] workflow [14, 20] and taskflow models [25, 24, 1, 2, 7, 12, 23] share many similarities in dealing with certain application abstractions, i.e., an application is composed of a sequence of (possibly) ACID steps. However, their design points are significantly different and their primary focus on application development issues and/or runtime environments are in general complementary. We first briefly summarize the design points for these systems and then contrast the Coyote monitor to these systems.

Objective: The design point for workflow systems emphasizes (1) easy modeling of business processes by composing pre-developed tasks, (2) role management (i.e., allocation of work to pools of human operators), and (3) maintenance of the workflow pattern in a general database structure. The implications are that the execution of these tasks can not monitored at a detailed level by the workflow monitor. In addition, the system is based on interpretation and the performance is traded for ease in expression. In contrast, the focus of advanced transaction models is on defining semantic language constructs and runtime services that improve fine-granularity concurrency and flexibility in data dependency expressions (e.g., nested transactions).

Recent works on transactional workflow systems [1, 2, 7, 12, 23] attempt to bridge this gap by creating semantic constructs at the script level for describing data or execution dependency. The monitor maintains execution state in terms of the tasks already executed and their results. Upon a system or application failure, the system can do both forward or backward recovery. Backward recovery is based on invoking a predefined set of compensating tasks.

Control flow: Typically, the execution flow of a workflow application is described in terms of the order in which the component tasks are executed. In contrast, general programming constructs capture more complex data and/or state dependency, particularly in the presence of asynchronous execution constructs (i.e., rules and events). Here, the number of execution states can be very large, and applications require appropriate interfaces for dynamic interactions with the monitor.

The steps or tasks can be related to application components in systems based on component architectures (e.g., COM, CORBA, JAVA Beans). However, the task invocation processes are different between these two classes of systems. In workflow systems, the steps or tasks are invoked based on predefined scripts, whereas the components are explicitly invoked by the embedded program logic in the later systems.

Step Granularity: The granularities of program steps (that are composed to create an application) in these systems may also differ significantly. In workflow or taskflow systems, the steps are of coarse granularities. Hence, the monitor (i.e., interpreter) does not coordinate the underlying resource management. In contrast, a TP monitor is part of the underlying runtime environment and coordinates resource accesses of an application via the X/Open protocol. In a taskflow or transactional workflow system, the

The design point of Coyote differs from that of workflow systems in that it emphasizes sub-second response time for each individual action requests. This is needed to be responsive to the end-user as well as to provide high throughput automated services. The workflow systems in contrast in general do not need this responsiveness. This is because they schedule more coarse-grained tasks either to human or to for automated processing. Additionally, in COYOTE we seek additional supports for dynamic non-predefined conversations between independent organizations. The Coyote also provides support for form (e.g., HTML) management and processing for ease in development of conversational applications.

5.3 Coyote, Existing TP Monitors and Component Architectures

Existing transaction systems Tuxedo [3], CICS [11], IMS [18] etc, have begun to offer internet gateways to them and support for network centric languages such as Java. Java itself is offering a transaction standard JTS which is actually a veneer on OMG's OTS transaction protocol. The proposed concept of network service transactions begins to provide a model and captures the essential semantics of service applications that these products expect to support. The Coyote monitor also provides some specific services to help in the mapping of input and output data to HTTP parameter lists and HTML pages.

The service transaction model and Coyote approach to application development also complements the application development based on various component architecture models, e.g., COM [8], CORBA [21] and JAVA Beans [19]. All of these architectures provide support and services for developing application components (e.g., Encapsulation, component registration and invocation, event services, transactional services, etc.). However, they all lack support for long running conversations (i.e., general support infrastructure, management of conversation instance, compensation of service request, group compensation, service contract enforcement, etc.). Hence, all of these platforms would benefit from the service transaction abstraction.

6 Summary and Conclusions

Service Transactions are an effective model to capture the semantics for adding service application in a network centric environment.

With the Coyote monitor we have defined the critical services which the transaction infrastructure need to provide to support network centric service applications. Key technical features provided by this monitor are:

- Transaction monitoring and Logging services
- Compensation model, Group compensation
- Automated invocation of applications and services
- Persistent queryable conversation state
- Reliable execution of service requests
- Services for generating Internet (HTML or Java) formatted output from the transactional environment.

The appropriate model for developing applications in this environment is based on:

- service actions which are themselves atomic
- long running persistent conversations built from them
- scheduling rules identifying the "next" action in response to messages or events

- use of the Coyote coordination and compensation services

This set of concepts, service transactions, monitor and application structure has considerable practical implications as to how one can most effectively construct *"middle tier"* servers in a network centric service environment.

Acknowledgments: The authors gratefully acknowledge the contributions of Ambuj Goyal and Tim Holloway who through many discussions helped in defining and refining some of these concepts. The discussion of application development builds on the concepts developed in IBM's MQSeries Three Tier product and discussions with Ian McCallion, Peter Lambros, Ian Robinson and Vernon Green. Dinkar Sitaram worked with us at an earlier stage, and contributed to an earlier prototype. He is also an author of a previous version of this paper.

References

[1] Alonso, G., D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, "Advanced Transaction Models in Workflow Contexts" In 12th *ICDE*, New Orleans, Lousiana, Feb. 1996.

[2] Attie, P., M. P. Singh, A. Sheth, M. Rusinkiewicz, "Specifying and Enforcing Intertask Dependencies", VLDB 1993, pp. 134-145.

[3] BEA Systems Homepage, *http://www.beasys.com*

[4] Bernstein, P. A., "Transaction Processing MONITORS", *CACM*, 33(11), Nov. 1990.

[5] Bernstein, P. A., M. Hsu, and B. Mann, "Implementing Recoverable Requests using Queues". In Proc. *ACM SIGMOD*, 1990.

[6] Biliris, A., S. Dar, N. Gehani, H. V. Jagadish, K. Ramamritham, "ASSET: A System for Supporting Extended Transactions", SIGMOD 1994, pp. 44-54.

[7] Breitbart, Y, A. Deacon, H. Schek, A. Sheth and G. Weikum, "Merging Application Centric and Data Centric Approaches to Support Transaction-Oriented Multi-System Workflows", *ACM SIGMOD Record*, 22(3), Sept. 1993.

[8] Chappell, D., "Understanding ActiveX and OLE: A guide for Developers and Managers", *Microsoft Press*, Redmond, Washington, 1996.

[9] Chrysanthis, P., and K. Ramamritham, "ACTA: The SAGA continues", pp. 349-397, in [2].

[10] *Customer Information Control System/ Enterprise Systems Architecture (CICS/ESA)*, IBM, 1991.

[11] IBM CICS Home Page, *http://www.hursley.ibm.com*

[12] Dayal, U., M. Hsu, and R. Ladin, "Organizing Long-Running Activities with Triggers and Transactions" *ACM SIGMOD Record*, pp. 204-210, 1990.

[13] "Database Transaction Models for Advanced Applications", Ed. A. K. Elmagarmid, Morgan-Kaufmann Publishers, 1992.

[14] *FlowMark*, SBOF-8427-00, IBM Corp., 1996.

[15] Garcia-Molina, H., and K. Salem, "SAGAS," *In Proc. of SIGMOD Conf.*, ACM, 1987, pp. 249–259.

[16] Gray, J., and A. Reuter "Transaction Processing: Concepts and Techniques," *Morgan Kaufmann Publishers*, 1993.

[17] Haerder, T., and A. Reuter "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, 1983, pp. 287–317.

[18] IBM IMS Homepage *http://www.software.ibm.com/data/ims*

[19] Java Beans Homepage, *http://splash.javasoft.com/beans*

[20] Kamath, M. and K. Ramamritham, " Modeling, Correctness and Systems Issues in Supporting Advanced Database Applications using Workflow Management Systems" Technical Report, TR 95-50, University of Massachusetts, Amherst, 1995.

[21] Object Management Group (OMG), *http://www.omg.org*

[22] Pu, C., G. Kaiser and N. Hutchinson, "Split Transactions for Open-Ended Activities" Proc. *VLDB*, 1988.

[23] Rusinkiewicz, M. and A. P. Sheth, "Specification and Execution of Transactional Workflows" *Modern Database Systems*, 1995, pp. 592-620.

[24] Schwenkreis, F., "APRICOTS - A Prototype Implementation of a ConTract System: Management of the Control Flow and the Communication System", Proc. of the 12th /em Symposium on Reliable Distributed Systems pp. 12-21, 1993.

[25] Waechter, H., and A. Reuter, "The ConTract Model", Chapter 7, pp. 219-263, in [13].

Performance Archive Database (PAD) – Data Mining Towards Self-Tuning Servers

Pranta Das *pranta@sybase.com*
Girish Vaitheeswaran *girish@sybase.com*
T.K. Rengarajan *ranga@sybase.com*

SPeeD (Server Performance Engineering & Development), Sybase, Inc.

Abstract

Today's transaction processing environments are very complex. As a result of this, performance tuning of database systems requires new levels of sophistication and expertise on the part of administrators as more and more database servers are being used for variegated applications. Specifically, the emergence of a new breed of Internet-based transaction processing applications have created more unknowns in the area of throughput and response time measurement. Also, the nature of traditional database application workloads typically range from high-throughput online transaction processes to ad-hoc DSS queries. The practice of running OLTP by day and DSS by night is a paradigm of the past. Attaining optimal performance has become a challenging task due to a significant increase in the number of configuration knobs being provided by Data Base Management Systems to an administrator. It is a non-trivial problem for a database administrator to gather the expertise in assessing the best options for these knobs that will result in optimal throughputs and response times among other things.

It is fairly tricky for a database engineer to accurately evaluate certain performance sensitive design decisions since most of these are subject to environmental and workload related considerations. Subsequently, the task of tuning for better performance is being delegated to the administrators and application developers. Ideally, every database engineer would like to design a self-tuning server. However, this requires a lot of background information.

The purpose of the Performance Archive Database (PAD) is an attempt to address this issue and thereby automate tuning as much as possible. PAD allows the DBMS to collect performance information from an active database server, archive the performance information in a database and finally either provide tuning recommendations to database administrators and application developers or perhaps feedback this information to the server so that it may tune itself. The basis for PAD is embedded in the form of monitor counters within the DBMS. These performance "probes" improve the overall observability of the DBMS, thereby enabling better tuning decisions.

PAD can be thought of as a training set in Performance Data Mining. The benefits of PAD are two-fold. On the one hand, it provides critical tuning information to administrators while on the other, it helps the DBMS engineer to gain a deeper understanding of customer application trends and work towards providing optimal out-of-the-box performance. This paper also provides some real-life examples where PAD proved to be useful.

Table of Contents

1	Introduction	3
	1.1 The Performance Tuning Problem	3
	1.2 Performance Data Mining	4
	1.3 Self-Tuning Servers	4
	1.4 Motivation behind PAD	4
2	Architecture of PAD	5
	2.1 ABCs of PAD	5
	2.2 The Collection Phase	6
	2.3 Contents of a PAD packet	6
	2.4 The Archival Phase	7
	2.5 PAD Schema	8
	2.6 Analysis and Feedback	9
3	Performance Tuning Heuristics	11
	3.1 Basis for Heuristics	11
	3.2 Deductive vs. Inductive Heuristics	11
4	Conclusion and Future Directions	13
	4.1 Learning from Packets	13
	4.2 Evolution towards a Self-Tuning Server	14
ACKNOWLEDGEMENTS		16
REFERENCES		16

Chapter 1
Introduction

1.1 The Performance Tuning Problem

Performance is the measure of efficiency of an application or multiple applications running in the same environment. It is usually measured in terms of throughput or response time. Tuning is an act of trying to optimize performance. The tuning layers in a typical database management system can be categorized as follows:

- *Application layer* : Involves good database design
- *Database layer*: Involves shared resources such as disk, transaction log, data cache etc.
- *Server layer*: Includes shared resources such as locks, CPUs, memory etc.
- *Network layer*: Includes the network or networks that connect the user to the engine.

In this paper, we will focus on the database layer, the server layer and the network layer only. We will assume good database design.

Performance tuning of database systems has become increasingly complicated as the nature and complexity of the applications that use databases grow. The contemporary transaction processing model is fairly intricate with database engines being used for servicing mixed workloads comprising OLTP, DSS and internet applications. Typical OLTP systems can run both interactive transactions, which are processed while the client waits for an answer, and batch transactions, which are submitted but may be processed later [1]. This diverse nature of workloads has intensified the need to have extremely well tuned database servers so that performance is never compromised.

Determining how to correctly tune the many configuration knobs provided by database management systems so as to attain optimal throughputs and response times for different workload situations is difficult. While one system may be very well tuned for OLTP environments, it may perform very poorly for DSS applications.

New data types (e.g. image, video, audio) and more complex query processing (rules, recursion, user-defined operations, etc.) will result in widely varying memory, processor and disk demands. The performance goals for workload class will vary widely as well and may not be related to their resource demands [2].

From the DBMS vendor's point of view, it is very difficult to predict every possible kind of target application or workload that the DBMS engine would be used for. Therefore, as a consolation, the DBMS engineer provides a number of configuration knobs that a DBA could use to tune his environment for optimal performance.

From the DBA's point of view, the tuning of low level performance knobs, that typically deal with the database, server and network layers and many of which are beyond the scope of his knowledge, is a complex task.

The advent of industry standard benchmark suites such as TPC-C and TPC-D, which provide a means of simulating real-life application workload scenarios for DBMS vendors, have enabled them to select the right defaults for certain hard-to-tune configuration knobs. However,

performance tuning still remains a manual, laborious and often complex chore for every database administrator.

1.2 Performance Data Mining

Just as regular data mining is used to search for patterns of behavior in a database[3], performance data mining can be defined to be the search for performance behavioral patterns in a database, which is populated by sampling monitor probes and configuration parameters from a DBMS during interesting workload runs. Based on the results of these searches, a DBMS can deduce certain facts and thereby recommend courses of action that are either:

- Manual i.e. an action that the DBA needs to perform, or
- Automatic i.e. an action the DBMS performs itself

In either case, the recommendation derived from the search, is a step towards tuning the DBMS better for optimal performance.

1.3 Self-Tuning Servers

An automatic tuning action performed by a DBMS, based on observations it made, is an act of self-tuning. A self-tuning database server automatically adapts to its environment, by monitoring and feeding back performance data at regular intervals while the database system is running and applies this data to adjust configuration parameters to optimize performance.

1.4 Motivation behind PAD

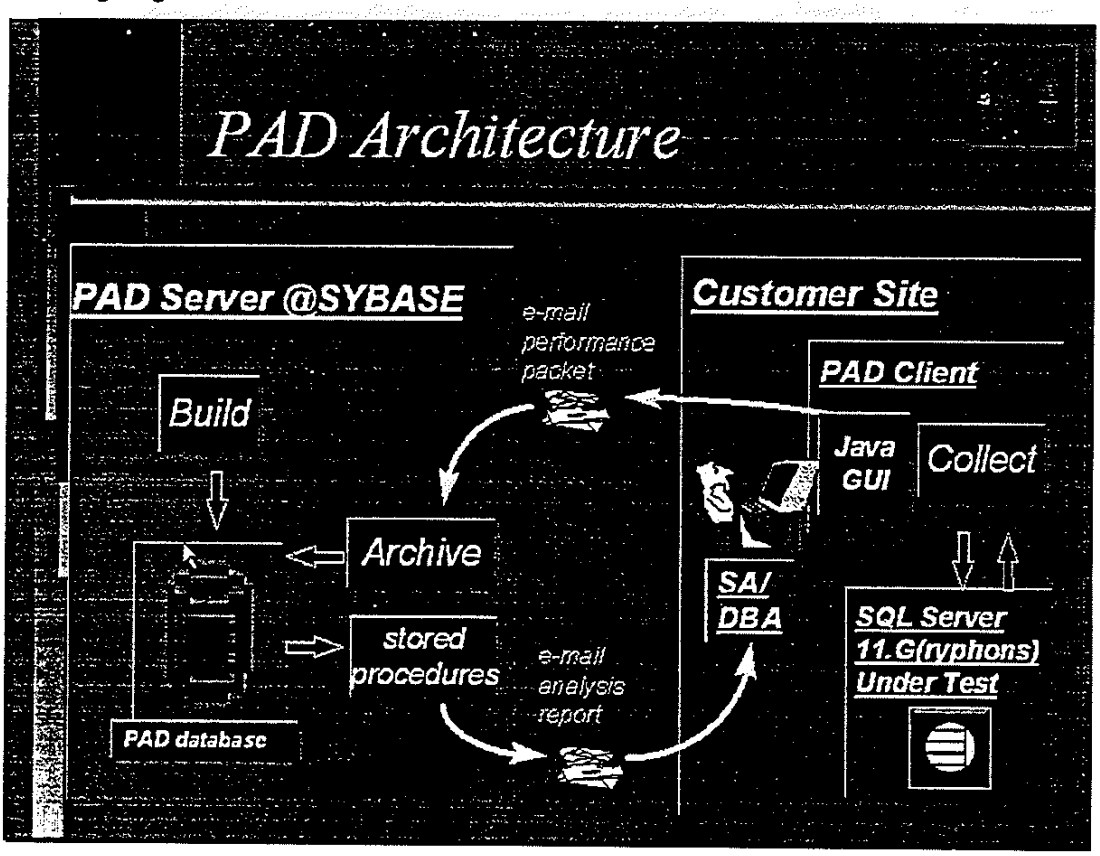
The Performance Archive Database (PAD) was developed as a tool that would address the complex problem of performance tuning. PAD was designed not only to provide database administrators a set of tuning recommendations that is specific to their environments, but also to provide valuable insight into customer applications and thereby enabling database system engineers to optimize out-of-the-box performance and perhaps someday be the basis for a self-tuning server.

Chapter 2
Architecture of PAD

PAD comprises of two primary components:

- The PAD server component which currently resides at Sybase, Inc.
- The PAD client component which is distributed together with the DBMS to all our customers

The architecture of PAD and the interaction between the client and the server is illustrated in the following diagram:



2.1 The ABCs of PAD

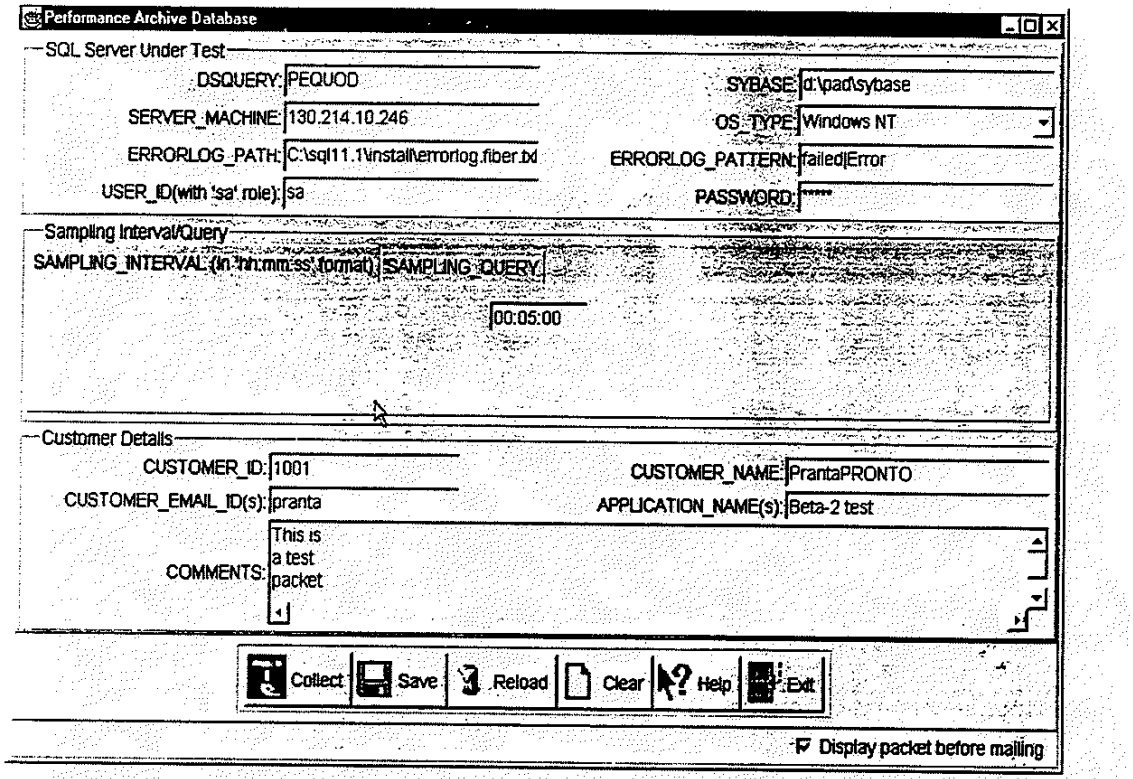
The PAD system consists of four modules:

- Archive: This is the module that intercepts an incoming performance packet from the customer and loads it into the database. It resides in the server component.
- Build: This is the module that creates the PAD database, defines the schema and the stored procedures. It also resides in the server component.
- Collect: This is the module that actually collects the monitor and configuration parameters from an operational SQL Server, formats them into a packet and sends the packet via e-mail to the PAD server at Sybase, Inc.
- stored procedures that analyze the performance packets after they are archived and send out an analysis report to the customer's system/database administrator who sent the packet.

2.2 The Collection Phase

The first step in using PAD is the collection step. Collection involves sampling of the system to be tuned for a specific sampling interval. The sampling interval should be chosen in such a way that only representative performance data is collected. Therefore, one of the main guidelines to be observed during the collect phase is to refrain from sampling when the system is idle since such a sample would not be instrumental in making intelligent inferences on performance.

The collection phase is facilitated using an easy-to-use Java GUI, a snapshot of which is shown below:



2.3 Contents of a PAD packet

During the collection step, monitor counter probes, instrumented within SQL Server code, get incremented in shared memory. These probes, which provide a microscopic view into system behavior, are read from shared memory and placed in a tokenized fashion into the performance packet. In addition, other information, which have a direct bearing on making performance tuning decisions, such as configuration parameters and device information, are also sampled and placed as unique tokens into the packet.

A PAD packet typically contains the following set of performance tokens:

- *Counter Tokens*: These include details such as number of CPU ticks, buffer cache hits, locks, etc.
- *Configuration Tokens*: These represent the SQL Server parameter settings existing in the environment that needs to be tuned.

- *Hardware Tokens*: This includes details pertaining to hardware configuration such as number of CPUs, system memory, CPU type, CPU speed etc.
- *Device Tokens*: This includes device-related information.
- *Customer Tokens*: These describe the details pertaining to the customer who collected the packet.

At the end of the collection step, the packet, which was formatted to contain all the relevant performance information, is mailed over the Internet to the padmaster@sybase.com.

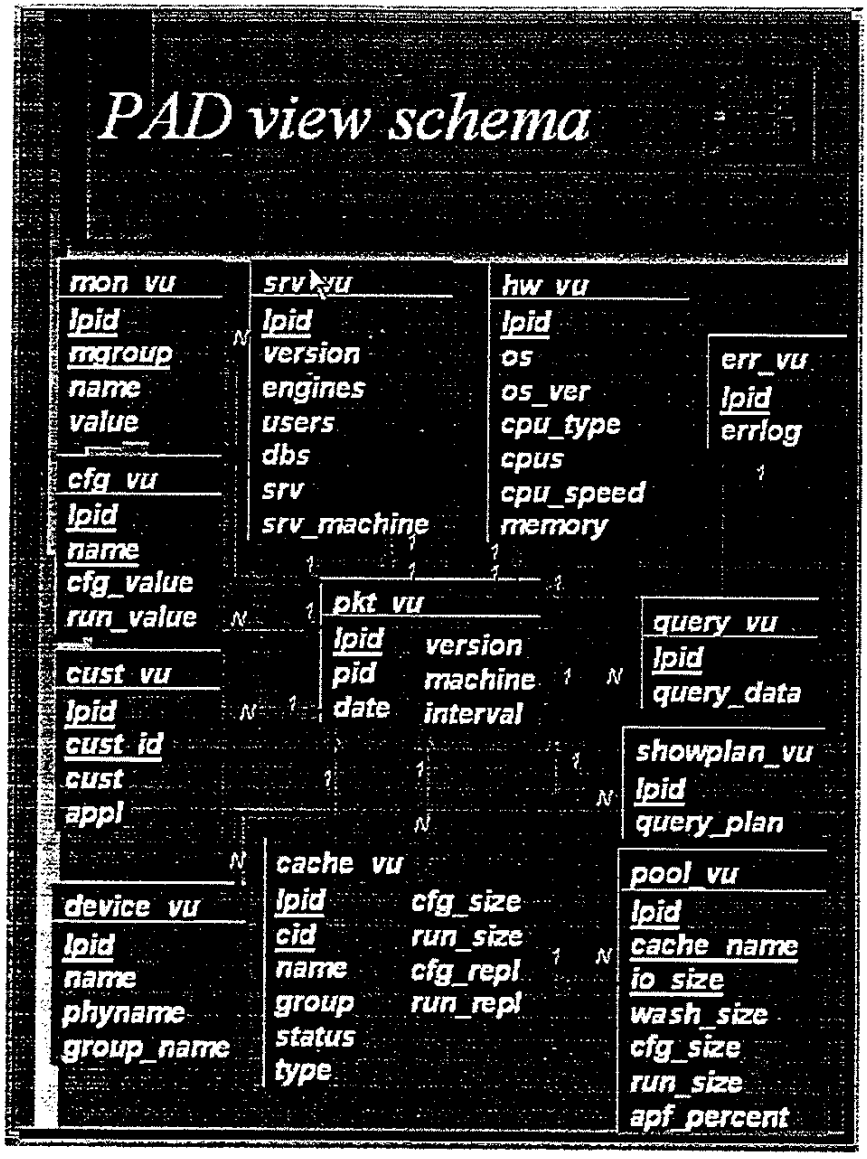
2.4 The Archival Phase

The second step involves intercepting the incoming collected packet and archiving the packet into the database. The tokenized packet is split up into various groups and each group is stored in a separate entity. The motivation behind archiving performance packets into a database has many advantages:

- Analysis of the packet can be done using ad-hoc SQL queries
- Performance packets from samples taken at different intervals could be compared and baselining can be done
- Specialized stored procedures could be run against a packet to identify possible performance bottlenecks
- The archived data can be used as a tool for learning from experience.

2.5 The PAD schema

The different views to the archived performance data in PAD can be easily understood from the view schema diagram below:



Each performance packet is governed by an entry in the packet view, where it is uniquely identifiable by a logical packet-id. All other views are directly or indirectly based upon the packet view with one-to-one or many-to-one relationships as shown above.

2.6 Analysis and Feedback

The third and final step involves analyzing the performance data contained in a packet and providing the feedback. This includes deducing performance recommendations for a specified packet. A detailed performance analysis of the system on different components of SQL Server such as memory management, kernel management, data cache management, procedure cache management, task management, transaction management and network management is provided and tuning recommendations for each of these components are made wherever applicable.

The database administrator can use the set of guidelines provided by this step and verify whether the performance has improved by collecting a subsequent packet. Such a comparison becomes possible very easily, since the first packet would automatically become a baseline, and every time a new packet with a higher throughput gets archived, it becomes the new baseline.

An example of an analysis feedback report sent by PAD is shown below:

Subject: PAD Analysis Report for packet-id 7
Date: Fri, 24 Jan 1997 12:06:53 -0800
From: padmaster@sybase.com
To: dba@ml.com

Sybase SQL Server System Performance Report

Server Name PDSQTS1
Version SQL Server/11.1/D/NT/OS 3.51/1/OPT/Wed Jan 22 02:51:50 PST
Packet Id 7
Run Date Fri Jan 24 11:31
Sample Interval 5 min.

Kernel Utilization

Engine Busy Utilization:

Engine 0	*100.0 %*
Engine 1	*100.0 %*
Engine 2	*99.9 %*
Engine 3	*100.0 %*

Summary: Total: 399.9 % Average: 100.0 %

CPU Yields by Engine	*per sec*	*per xact*	*count*	*% of total*
Engine 0	*0.0*	*0.0*	*8*	*15.4 %*
Engine 1	*0.1*	*0.0*	*15*	*28.8 %*
Engine 2	*0.1*	*0.0*	*21*	*40.4 %*
Engine 3	*0.0*	*0.0*	*8*	*15.4 %*
Total CPU Yields:	*0.2*	*0.0*	*52*	

Worker Process Management

	per sec	*per xact*	*count*	*% of total*
Worker Process Requests				
Requests Granted	*0.1*	*0.0*	*15*	*15.0 %*
Requests Denied	*0.3*	*0.0*	*85*	*85.0 %*
Total Requests	*0.3*	*0.0*	*100*	
Non Blocking Requests	*0.1*	*0.0*	*100*	*100.0 %*
Blocking Requests	*0.0*	*0.0*	*0*	*0.0 %*
Requests Retried	*0.2*	*0.0*	*45*	*45.0 %*
Requests Terminated	*0.0*	*0.0*	*5*	*5.0 %*
Due to Attentions	*0.0*	*0.0*	*3*	*3.0 %*
Worker Process Usage				
Total Used	*0.7*	*0.0*	*200*	*n/a*
Max. Ever Used at once	*0.3*	*0.0*	*80*	*n/a*
Memory Requests for Worker Processes				
Succeeded	*0.1*	*0.0*	*40*	*80.0 %*
Failed	*0.0*	*0.0*	*10*	*20.0 %*

Tuning Recommendations for Worker Process Management

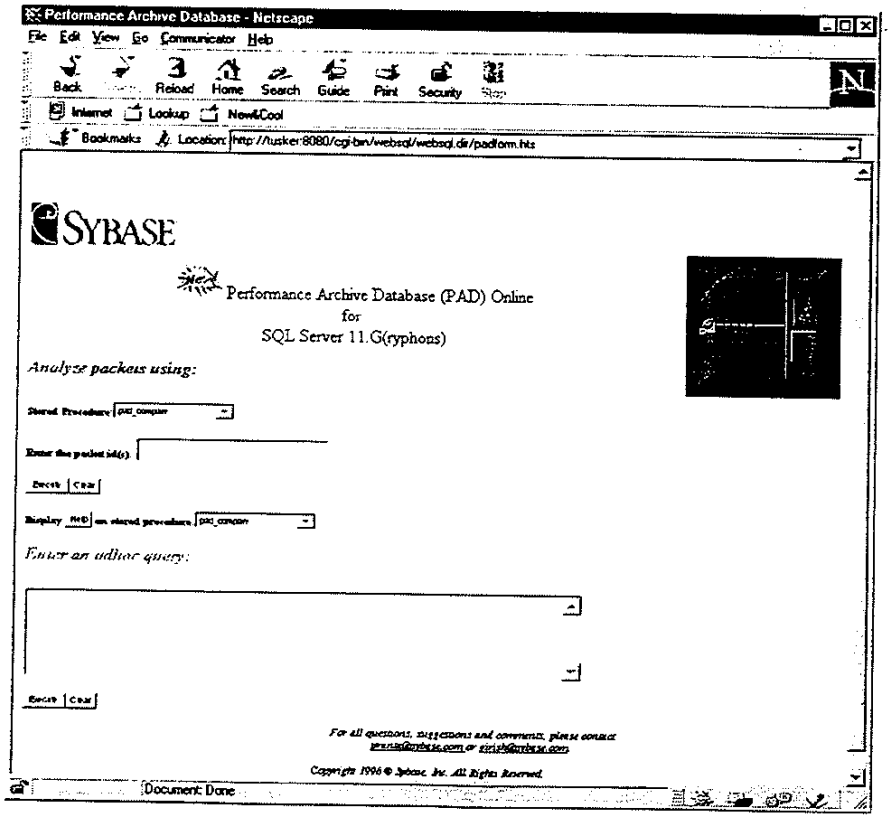
- Please consider increasing the 'number of worker processes' parameter by 25 and sample monitor counters again to verify.

As can be seen from the above report, PAD recognized the fact that there were 85% worker process requests denied and therefore it recommended to the DBA to increase the configured 'number of worker processes' by 25. In addition to recommendations such as this, PAD also has the capability to compare, based on some key metrics, the most recent packet with the archived baseline as shown in the next diagram:

Packet Comparison Report

Metric per second	Packet id 3	Packet id 7	% Difference
Transactions	122.3	111.2	-9
CPU Busy	40.00	39.31	-2
CPU Sleeps	0.00	0.00	0
CPU Idle	0.00	0.03	0
Context Switches	1164	1106	-5
Kernel Sleeps	1153	1096	-5
Buffer Searches	12909	11903	-8
Buffer Misses	840.5	776.4	-8
Buffer Washes	740.4	643.8	-13
Buffer Dirty	364.9	332.8	-9
Disk Polls	106.9	111.1	4
Empty Polls	0.39	8.28	2023
Completed Polls	1099	1163	6
Total I/O's	1099	1163	6
Total Reads	690.7	653.3	-5
Total Writes	408.0	509.3	25

Normally, the packet with the highest ever throughput automatically becomes the baseline. But customers may wish to compare a packet with a packet that is not necessarily the baseline. In such a case, a browser interface is provided using which, the customer can at any later point in time, perform further analysis, comparisons or even execute ad-hoc queries against the PAD repository. This web-based online interface is shown below:



Chapter 3
Performance Tuning Heuristics

3.1 Basis for Heuristics

A heuristic is an exploratory problem-solving procedure that utilizes self-educating techniques such as evaluation and feedback, to improve performance.

The basis and extent of the performance tuning heuristics implanted in PAD are directly proportional to the observability of the SQL Server itself. In order to make the SQL Server more observable, it has been instrumented with monitor counter probes throughout the code. In general, the Server Observability (σ) ratio can be defined as the number of probes (η) provided in a DBMS engine per KLOC (κ):

$$\sigma = \eta/\kappa$$

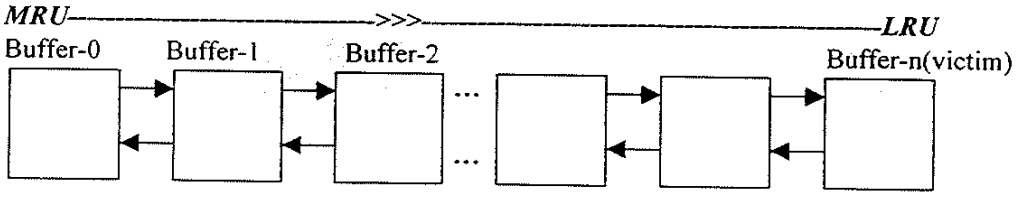
The value of σ is proportional to performance degradation that may result by turning on these monitor counters. Hence, there is a trade-off between observability and the amount of performance degradation, one is willing to sacrifice. As a compromise, only critical parts of the code are instrumented with monitor counter probes.

3.1 Deductive vs. Inductive Heuristics

Heuristics can be of two types:

- *Deductive heuristics*: These are heuristics that are deduced purely based on current observations.
- *Inductive heuristics*: These are heuristics that learn from prior experience and induce extrapolations.

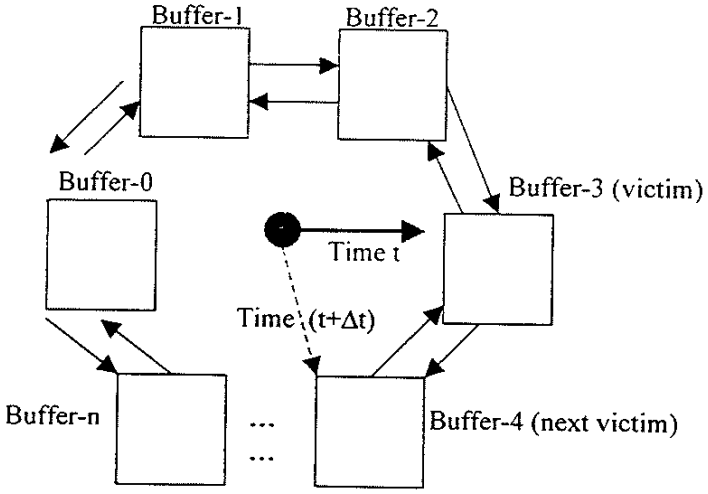
Currently most of the heuristics in PAD are deductive in nature. An example of a heuristic in PAD is the choice of a suitable cache-replacement policy for a given named-cache. There are two choices for cache replacement policies in SQL Server. The first is the strict LRU (Least Recently Used) policy, where each and every buffer in the named-cache is maintained in a doubly linked list with the most recently used buffers at the head of the chain and the least recently used ones at the tail. This is illustrated below:



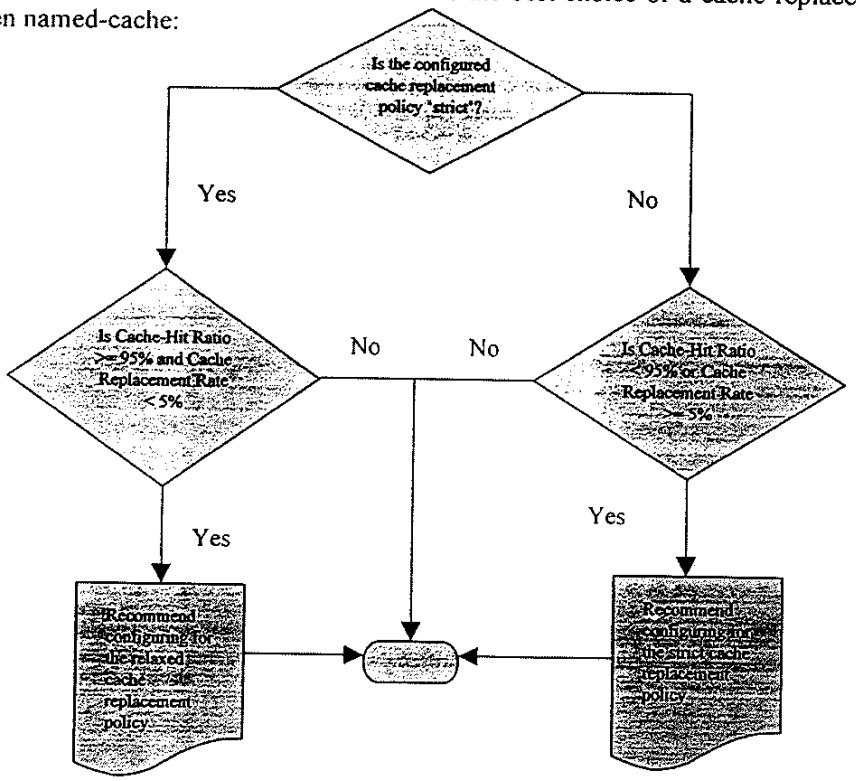
Under the 'strict replacement policy', whenever a buffer is reused, it is unlinked from it's original position in the chain and then re-linked at the MRU head of the chain. So, the buffer at the LRU end is the victim chosen when a replacement needs to take place. All these multiple link

operations in an SMP environment could be costly and may result in cache-to-cache transfers that will cause increased traffic over the bus introducing latency.

In a cache, where the cache-hit ratio is very high and replacement rate is considerably low, performing multiple link operations for maintaining the buffer chain in strict MRU-LRU order may not be so optimal. In such cases, the recommended policy for cache-replacement is the 'relaxed LRU replacement policy' (a.k.a 'clock replacement policy'). Here, the buffers are arranged in a circular linked list (like a clock) as shown below:



In the relaxed cache replacement policy, when a buffer is reused, it is not unlinked from it's original place in the circular chain. A pointer to the buffer, which will be the potential next victim, is maintained and this is updated to point to the next buffer every time a buffer is grabbed. This reduces the number of linking and unlinking operations that need to be done, and so it is ideal for caches with very little replacement. The following flow-chart illustrates one of the heuristics embedded in PAD that decides what is the best choice of a cache-replacement policy for a given named-cache:



Chapter 4
Conclusion and Future Directions

4.1 Learning from Packets

The performance data obtained from PAD packets from customers all over the world form a strong foundation for building a self-tuning server. In particular, this data helps us in:

- ***Providing better defaults for the configuration knobs:***
  For instance, the contents of one packet indicated that the customer had adequate number of disks configured so as to be able to issue a large number of asynchronous pre-fetch reads. However, this was hindered due to the lack of a sufficient number of disk I/O control block structures, which have a default value of 256. So, it may be a good idea to make the default number of structures a function of the number of devices configured.

  Analyzing PAD data is a first-step towards a future plan to reduce the number of configuration knobs provided to the database administrator thereby attaining out-of-the-box performance.

- ***Validating the accuracy of and improving the current algorithms in the SQL Server:***
  For example, one common observation, made from many packets, was that automatic checkpoint was being done way too often. The frequency of automatic checkpoints in the SQL Server is governed by the amount of time, a customer is willing to wait for recovery. This period of time is called 'recovery interval', and it's default is set to 5 minutes. It was earlier assumed that it takes approximately 10 milliseconds to recover 1 row. Therefore in 1 minute, it would be possible to recover 6000 rows. So, the frequency of automatic checkpoints for a 'recovery interval' of 5 minutes would be every 30,000 active log records i.e.:

  ```
  If (number of active log records >= 30,000)
  Then
        do automatic checkpoint
  End-if
  ```

  However, with the advent of faster recovery techniques in the server such as the use of asynchronous pre-fetch, it may take a shorter period of time to recover a single row. Hence, the frequency of automatic checkpoints may be reduced.

- ***Use the 80-20 rule for the most optimal out-of-the-box performance:***
  We have noticed that as a general thumb-rule, most production commercial database installations have ample idle periods. A system-initiated task, that operates during these idle cycles, is the SQL Server's housekeeper task. The housekeeper washes dirty buffers and launches the checkpoint process if necessary. These checkpoints are commonly referred to as 'free checkpoints'. Many packets have indicated the materialization of free checkpoints, thus reducing the burden of checkpointing during peak-loads and aiding faster recovery.

- *Provide tuning recommendations on certain configuration parameters and perhaps feedback these into the server:*
  Based on the monitor probe vis-a-vis configuration information, we have been able to suggest better tuning recommendations to the DBA.
  For example, we observed in some of the packets, that the server was configured with a very low value of 'deadlock checking period', which determines the frequency of deadlock checks the server makes, although the actual number of deadlocks was zero. A recommendation was automatically made by PAD to decrease the frequency of checking for deadlocks and let the engines perform more useful chores instead.

  It was noticed from certain packets that there was a high spinlock contention on some data caches. It was recommended by PAD that if multiple objects were bound to the same cache, they could be split up and the individual objects could be bound to separate caches, thus reducing the overall contention on the cache spinlock.

  Similarly, another packet revealed that there was a device semaphore contention and it could be recommended that the data be spread across multiple disk controllers to alleviate this.

  In another packet, it was observed that during query compilation the query optimizer felt that it could have incurred a lesser number of logical I/Os had there been a large buffer pool configured for a particular cache, to which an object was bound to. In such a case, PAD recommended that the user configure a large buffer pool for that particular cache.

  Today these recommendations are reported to the DBA automatically by PAD but the DBA has to manually implement the recommended configuration changes. In the future, we hope to automatically feedback these recommendations to the server, so that it may tune itself.

4.2 Evolution towards a Self-Tuning Server

The following are a set of general guidelines to evolve commercial database servers into self-tuning servers:

1. All configuration knobs must be dynamically tunable. Each knob must be observable. The need to tune a knob means there are advantages and disadvantages to changing the knob settings. Each knob must have one set of monitor counters that measure the advantages and another set of monitor counters that measure the disadvantages.

2. The monitor counters must be turned on always so self-tuning algorithms can depend on this feedback from the server. 64-bit integers must be used to keep track of the counters to avoid overflow. To make this scalable on SMP systems and to make this least intrusive, we need per-engine monitor counters. These counters must be monotonically increasing and never reset, except upon overflow.

3. Several recent monitor samples, made at frequent intervals, must be archived by the server. Summarized data from the recent samples must be presented as a "fake" system table. Each row of system table must have recent cumulative values, short-term and long-term averages. The computed values must be triggered by queries on this "fake table". A persistent version of this table must also be maintained to collect long-term performance patterns. At server

shutdown, data collected in the current invocation of the server must be added to the persistent table.

4. Self-tuning heuristics operate roughly the following way:
   - For each knob, the current configuration value, as well as the feedback from the server via monitor counters are known.
   - The heuristics evaluate whether it is advantageous to increase or decrease the configuration value. Based on these, the self-tuning component suggests a new value.

5. The self-tuning heuristics must be prototyped via semi-automatic means, like PAD at first. Once confidence grows in the self-tuning heuristics, these could be captured in a reusable common component. The Adaptive Component Architecture will allow us to execute this common component within the server to make it self-tuning. The self-tuning component can be invoked by the server when idle cycles are available via the housekeeper task. It can also be combined with a GUI and execute in browsers with manual overrides.

6. Self-tuning components for the server may then be developed by external vendors, experts and consultants. These components will vary in terms of quality of the software, quality of tuning, quickness of response to server behavior and pertinence to industry segments. DBAs will be able to choose self-tuning components that suit their needs.

Just like the strength of a building is only as good as that of it's foundation, the reliability and accuracy of self-tuning servers will be heavily dependent on the data that it monitors and gets fed back. We feel that PAD is a first step in the direction of creating self-tuning servers.

ACKNOWLEDGEMENTS:

We would like to thank Sundar Seshadri and Max Berenson for their initial work on PAD. We would also like to thank Ganesan Gopal and Sapan Panigrahi for their valuable comments and suggestions.

REFERENCES:

[1] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*

[2] Kurt P. Brown, Manish Mehta Michael J.Carey, Miron Livny, *Towards Automated Performance Tuning for Complex Workloads.* Technical Report #1247, University of Wisconsin, Madison.

[3] Marcel Holsheimer and Arno Siebes. *Data Mining: The Search for Knowledge in Databases.* Report CS-R9406, CWI, Amsterdam.

Workload Management in Large Transaction processing systems.

T.J.R.Dunn

IBM UK Laboratories
Hursley Park
Winchester
SO21 2JN
timd@vnet.ibm.com

1.0 Abstract

This paper will examine the need for workload management in a transaction processing environment in which there are multiple clients and servers. The paper will briefly look at those characteristics which are essential for a well performing workload manager. This is followed by an examination of the workload management component provided within CICS System Manager for AIX. A brief description is then provided of a configuration in which CICS System Manager for AIX played a key role. The configuration is believed to be the largest demonstration to date of transaction processing software in an open systems environment, consisting of 100 servers and 10,000 simulated users.

1.1 Introduction

In the past, the open systems environment has not been noted for the ability to build large, scalable, robust and manageable transaction processing systems within it. The processing capacity per machine has been low when compared with that of the mainframe. To achieve a similar total capacity would involve the linking of multiple machines. The linking of machines starts to create problems. There must be the ability to coordinate the multiple systems as a single entity. This means that there must be facilities to manage, individually or collectively, the systems and the transaction processing software on the systems. There must also be the ability to distribute the incoming work requests across the servers capable of processing that type of work. Items of work are requests to run transactions in a server. A transaction is a unit of work, a chunk of processing to be performed by a server on behalf of a client. There is the need to perform an N:N mapping between the clients and the servers. This mapping may well change as the performance or availability of servers changes.

The most simple form of the N:N mapping would be to statically allocate groups of users to specific servers. This is inflexible though and when a server becomes unavailable through failure or the need to service it the users assigned to that server find that they are unable to work. This is clearly not a desirable situation.

A more flexible and powerful answer is required. The management of the system and transaction processing software can be provided through systems management software and the distribution of work requests from clients can be provided by workload management software. Such software needs to be transparent to the users of the system. The focus in this paper is to look at workload management. To see what the requirements are for such software and then to look at one example, CICS Systems Manager for AIX (CICS SM) and to see how it was able to manage a workload across a large collection of servers.

1.2 Desirable Characteristics For Workload Management

Software which is to undertake the distribution of work requests from a collection of users to a pool of servers needs to have certain characteristics in order that it performs the job well and does not impede the processing of the work.

The following characteristics are seen as essential for workload management software:

1.2.1 Good Performance

Workload management software should inflict minimal overhead on a system. It is inevitable that there will be some overhead. The overhead which arises from workload management software can be split into a number of different types:

- The cost associated with making a routing decision. This is the process of determining exactly which server a piece of work should be routed to. There will be a number of factors to take into account when making a

decision, such as the processing load which the servers already have, the health of the servers, the cost of reaching the server - some servers maybe local others remote.

- The cost of collecting information so that a routing decision can be made. This is essentially monitoring of the servers. In addition to the overhead on the servers of actually collecting the information there will be a communication cost in passing the information around. The information needs to be made available to all points at which routing decisions are made. Making this information available on a regular basis to many routing points can represent a significant processing overhead unless carefully controlled.
- The cost of making bad decisions. If work is routed to the wrong server the user will observe poor response time, the server may well become overloaded and stressed. Severe overloading will cause work to be queued overloading other components of the system such as the network.

1.2.2 Reliable

Workload management software should be reliable. It should not fail. The code should be simple and robust. Users must not be forced to stop working because their work requests cannot be routed to a server for processing. If the workload management component is unable to make a routing decision, work should be able to continue. A default routing should come into action. The performance whilst this default routing is taking place is unlikely to be optimal since many items of work may also be defaulting to the same server, but at least work can continue. Consistency is also included under reliability. The routing decisions that are made should be deterministic.

1.2.3 Dynamic

Workload management should proceed unattended. It should be possible to cope with the loss or failure of resources and servers and make routing decisions around the failure. That is do not continue to route work to a server when it is no longer capable of processing work because it abended or was shut down for example.

1.2.4 Scalable

Workload management should be capable of running the smallest of configurations up to the largest without incurring an increasing overhead. Ideally there would be no additional cost as more servers or clients are added to the managed configuration. In practice a linearly increasing overhead is the aim.

1.2.5 Flexible

A design which is able to cope with multiple types of request is desirable. In a CICS environment for example, there are a number of types of work which can be routed, such as transactions, a dynamic program link (essentially a subroutine call) or access to resources held in a CICS server other than the current one. Other transaction processing systems have similar facilities. Workload management should be able to cope with as many of these facilities as possible and certainly the most common ones.

1.3 CICS SM Workload Management

CICS SM provides a workload management function. It manages the routing of work items within a CICS for AIX environment. It only runs in conjunction with CICS for AIX and does not support any other transaction manager. Figure 1 on page 3 shows the components of the workload management component of CICS SM.

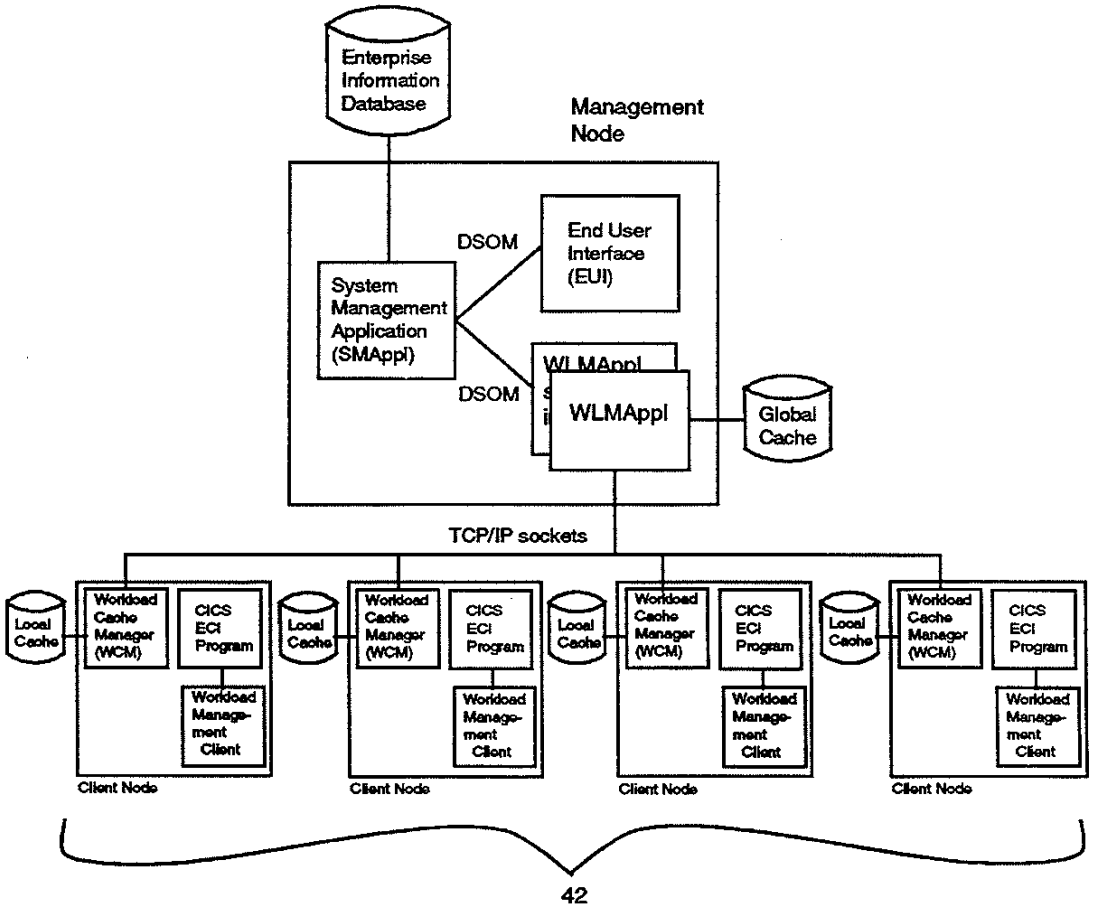


Figure 1. Workload Management Components and Interconnection

CICS SM Workload Management provides a means of dynamically distributing work over a pool of servers with the minimum of intervention. If a CICS for AIX region should fail, CICS SM Workload Management detects the failure. The work that was routed to the now failed region will be routed to the remaining regions capable of supporting that type of work. The re-balancing takes place without manual intervention, thus helping to provide significantly improved application availability. It is also possible to dynamically add new regions to a configuration and for work to be routed to all regions, including the addition.

Workload management can only be effective if there are multiple CICS for AIX regions which are capable of running a given application. Where an installation has multiple CICS for AIX regions each supporting a different application, it may make sense to restructure the CICS for AIX regions and making all applications available in all regions. By doing this, each application becomes available in multiple servers and there is then value in using workload management.

Routing decisions for transactions are made at the client. A client in this situation may be a CICS client, such as an ECI program, or a CICS for AIX region. In essence the sequence of events that takes place when a routing decision is required is as follows:

- The client generates a transaction request with a default server name that the transaction should be processed in.

  Note: If workload management is not active then this default server name will be used as the name of the server which is to receive the transaction request. Similarly if workload management is unable to make a routing decision then this default server name will also be used.

- CICS SM workload management makes a routing decision based on the data that is available to it at the time. The outcome of this decision is that the default server name that was supplied by the client is replaced with the name chosen by the workload management algorithm.
- The communication portion of the client determines the location of the server. If this is the first time that a particular server has been used then the location will be determined from cell directory services, otherwise the client will already have the location of the server.
- A call is made from the client to the server using the normal communication method.
- The transaction runs at the server.
- The response is received at the client. Workload management determines the response time for the transaction and records it in order to build up a picture of server performance.
- The client regains control.

The workload management component makes routing decisions on the basis of data that is provided to it about the CICS regions and the resources which are configured on those regions. This information is essential, since without it workload management does not know which resources are configured on which CICS regions. The data that is provided about the CICS regions and resources is obtained from the systems management(SMappl) component of CICS SM. The data is passed from the SMappl to the the workload management application (WLMAppl) when the CICS for AIX configuration is activated (the SMappl performs the acrtivation). The data on which CICS SM Workload Management makes its routing decisions is held in two types of cache held in memory, these are a global and a local cache. Both cache types are similar in nature. There is one global cache. There are many local caches, one on each machine on which routing decisions are required. The local caches operate independently of each other. Clients located on the same machine use the same local cache. Data is shared among clients using the same local cache.

In addition to the local cache on each node, there is also a workload cache manager(WCM) and a workload management client. The WCM is essentially responsible for managing the local cache. The workload management client is responsible for making the routing decisions. Each WCM establishes a communication link with the WLMAppl using TCP/IP sockets. This link is used to transfer data from the global cache to the local cache. Data transfer is always from the global cache downwards. If there is a change in the state or availability of a CICS for AIX region or resource the information will be passed to the WLMAppl and subsequently on to all the local caches which have down loaded information relating to the changed CICS for AIX region or resource. This can be done since the the WLMAppl keeps a record of which information resides in each of the local caches.

Initially both the global and local caches are empty. The global cache is populated when the SMappl passes configuration information to the WLMAppl. Population of the local caches occurs somewhat differently. When the first request to route a transaction occurs the workload management client tries to make a routing decision based on the information available in the local cache. As there is no entry in the local cache for that program, a request is made to the WCM to obtain that data. As a result the WCM issues a data request to the WLMAppl. The WLMAppl examines its global cache and passes the required data back to the WCM to store in the local cache. Through this use of local caches there is no cross system communication once the information has been passed down in order to make a routing decision. Each local cache holds a subset of the data that is held in the global cache. The information that is held in a particular local cache reflects the requests made by the CICS client on that node. In a typical environment, there would be many requested programs, and not all local caches would necessarily contain the same information because not all users would be running the same applications.

When a routing decision is being made, the workload management client looks to see which CICS for AIX regions are capable of running the requested application. It determines this from the definitions about the CICS for AIX regions and resources which are held in the workload manager local cache. Having identified the candidate regions, workload management calculates a score for each of the regions to which the piece of work could be routed to. The work is routed to the region with the lowest score. If several regions score equally, the algorithm selects one at random. This randomness ensures that work is spread evenly initially or after a quiet period. The total score of

a system is the sum of scores representing the load already on that system, its health, the quality of the connection to the selector and the extra load that deciding to route the current work item to this system would cause.

Once a routing decision has been made by the workload management client the information which is held in the local cache must be updated so that further routing decisions can take account of the changed position. This information reflects the volume of work which has been routed to each of the servers that this workload management client has selected so far as well as an indication of the performance of the servers. Each transaction which is capable of being routed has a load value associated with it. This load value is a measure of the weight of the processing that the transaction will impose on the server to which it is routed. No distinction is drawn between CPU and I/O processing. The load is actually set to the observed response time for the transaction. It can be difficult to set this load value in those situations where one transaction identifier is in fact used to invoke several different types of processing each of which will have a different response time.

Server performance is measured by comparing the observed response time for a routed transaction with the expected response time. If observed and expected are close the server is seen as performing well. It is important to set the expected response time for a transaction to a realistic value. Setting to a very large value for example would lead the workload management client to believe that work was being processed very quickly in a server and so it would route further work to that server. This would eventually lead to the the server becoming congested and poor performance being obtained.

The load and performance data in a local cache reflects the state of the servers as seen by the clients on that machine. This view of server load and performance could well vary from client machine to client machine. There is no consolidated view of server performance comprising of information from all the local caches.

1.4 Demonstrating CICS SM Workload Management

In order to illustrate the power and effectiveness of CICS SM Workload Management a large transaction processing system was built as a demonstration. The demonstration was built in July 1996 over 144 nodes of the RISC System/6000 SP Computer which is located at the Maui High Performance Computing Center in Maui. The demonstration consisted of 10,000 simulated users generating transactions which were then distributed across 100 servers using CICS SM Workload Management The primary purpose of the demonstration was to demonstrate workload management across a large number of systems. The focus was transaction processing. Because of this the decision was taken to simplify several other aspects of the configuration. Only a single application was used and there was no parallel database technology in use. Figure 2 on page 6 provides a summary of the overall configuration.

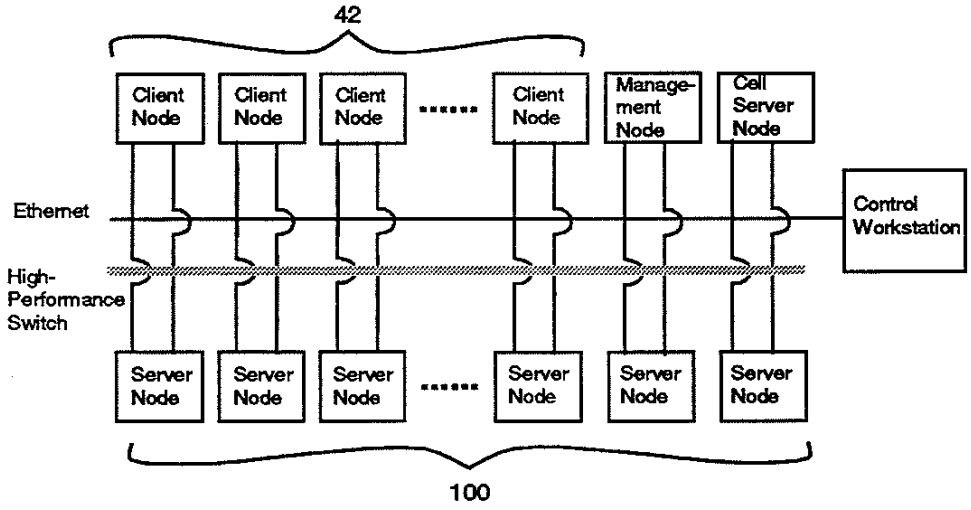


Figure 2. RISC System/6000 SP Node Configuration.

Looking at the configuration in a little more detail, we have:

1.4.1.1 Hardware

The RISC System/6000 SP computer on which the demonstration was built and run is a general-purpose, scalable, parallel system based on a distributed memory, message-passing architecture. The system consists of a number of processor nodes connected by means of a high-performance switch. The RISC System/6000 SP system provides good scalability, running anywhere from a single processor to a system containing 512 processor nodes. Each processor node has its own memory, disk, and peripherals such as network interfaces, tape drives, or external disks and executes its own copy of the AIX operating system.

Within a RISC System/6000 SP system, nodes are connected to a RISC System/6000 control workstation through an ethernet network. This network is generally referred to as the SP ethernet and is used for management purposes such as node installation.

In addition to the SP Ethernet, nodes in a RISC System/6000 SP are also connected through a low-latency, high-bandwidth connection between the nodes, know as the high-performance switch. The control workstation is not connected to the high-performance switch. Maximum use of the high-performance switch was made for inter-component communication, particularly the CICS client to CICS server communication.

1.4.1.2 Node allocation

Figure 2 shows 4 different types of node allocation. These are :

- Client
- Server
- Management
- Cell Server

1.4.1.2.1 Client: The client nodes were responsible for simulating user activity. This was achieved through the use of multiple instances of a multi-threaded program written in C which used the CICS External Call Interface (ECI) in order to communicate with the CICS servers. The CICS ECI provides a means for a non-CICS client application to call a CICS application, synchronously or asynchronously, as a subroutine. The client application communicates with the CICS server program using a scratchpad area. Typically this is populated with data, the subroutine call is made and the results are returned in the scratchpad area. A significant benefit of the CICS ECI interface is that it provides a means for any program on the workstation to access facilities available from a CICS

for AIX region. The client nodes also contained a CICS SM Workload Management local cache, a WCM and a workload management client. Figure 1 on page 3 shows the layout of the client nodes.

1.4.1.2.2 Server: The server nodes processed incoming requests to run workload transactions. The transaction environment was provided through the use of a CICS for AIX region and an Encina Structured File Server (SFS) (used to provide CICS system files). The application data was held within an instance of DB2 for AIX.

Each of the 100 server nodes was configured identically. All updates to application data were logged using CICS logging and DB2 for AIX logging facilities.

1.4.1.2.3 Management: The management node contained the CICS SM system management and central workload management components. There was only one such node.

1.4.1.2.4 Cell Server: The Cell Server provided the services of a DCE cell server. The DCE cell server provided an authentication service through the Security sever and a server location service through Cell Directory Services (CDS).

1.4.1.3 The application

In order to show work being managed across a system, an application was required. A version of the DebitCredit workload was used, this is an application that is not necessarily representative of a particular industry or type of application but is well understood.

DebitCredit represents a simple banking application in which customers visit branches of a bank and credit or debit sums of money to their account. Balances are maintained for the account holder, the teller and the branch. In addition an audit history is maintained of debit/credit history. There is a single transaction type in this application with all transactions performing the same operations, namely reading and updating the account, teller and branch tables and writing a record to the history table. The account number, teller number and branch number were generated at random by the client application.

1.5 Results

The transaction throughput of the whole configuration was measured using a collection of scripts and programs to perform the analysis. Measuring transaction throughput over a 1 hour period the configuration of 100 servers processed around 14 million transactions with a sustained 10 minute peak of 6289 transactions per second. All of these transactions were run under the control of CICS SM Workload Management

1.6 Discussion

There are a number of aspects of the configuration that are worthy of a brief discussion. These are issues that occurred during the development of the CICS SM Workload Management or during the building and running of the demonstration.

1.6.1 Investing time upfront

In building such a large configuration it was extremely important to have a very clear picture what was being built. A significant amount of time was invested at the start of the project to develop scripts which would allow each of the node types to be built and then removed. These scripts were developed and tested on two connected machines, not RISC System/6000 SP nodes. Once the environment had been built on these systems it was then torn down and rebuilt. At this point work on the RISC System/6000 SP commenced.

1.6.2 Ensuring stability

With a large population of users, 10000, and a large number of servers, 100, it is crucial to introduce some stability into the system. Where it is possible for work to be routed to any of the servers and there is little resistance to move then significant processing effort can be taken in constantly switching servers in order to find good response time. What happens is that initially a number of routing decisions are made to send work to a particular server. As more work is directed to this same server the response time for the routed work will degrade. With total freedom of movement, the workload management client will direct a clients next request to another region. In an environment with lots of short duration work it is likely that the same decision will be made for multiple pieces of work. The server will suddenly become empty, whereas previously it was heavily overworked. This swamping and desertion of servers will continue until some restraining influence is introduced. For this demonstration the 100 servers were divided into 13 groups. That is 12 groups of 8 regions and one group of 4 regions. The clients were then assigned to a group. The workload management client was then free to direct work to any one of the regions within the group to which the client was allocated. This meant that at most a client could only visit 8 regions. This is still sufficient flexibility to see a benefit from workload management, but it provides sufficient control that the system is stable.

1.6.3 Recognising the cost of moving

In making a routing decision in the workload management client it is important to recognise the cost, if any, of routing a piece of work to previously unvisited region. The implementation of the CICS ECI code means that there is such an overhead as the region has to perform initialisation work before it is able to process the work item that was routed to it. The workload management algorithm needs to recognise and accommodate this, otherwise the expected gain from routing work to a new region is unlikely to be realised.

1.6.4 Effectiveness of local caches

With multiple independent local caches there was concern that the information in the caches relating to the load and performance of the CICS for AIX regions would conflict and there would intense competition taking place between items of work sent from clients on different nodes to the same CICS for AIX regions. For example, two workload management clients on different systems could each decide that one particular CICS for AIX region was the best place to route work to and so they would each send large quantities of work. However the CICS SM Workload Management algorithm coped well with this.

1.6.5 Client based routing

Routing work from the client worked well. Performance evaluation measurements of the product showed that the throughput and response time obtained in a workload managed environment was identical to that obtained in a statically routed case. By routing from the client the overhead of the routing, effectively the cost of workload management, was borne on the client machine. Such machines tend to be less heavily utilised then the servers and so are more easily to bear the overhead.

1.6.6 Modelling vs monitoring

In the workload management algorithm the impact of the routed work on the servers was modelled rather than monitored. As work was routed to a region the load for that region was increased to represent the additional load that the new work item would place on the server. The alternative approach of monitoring the activity on the servers would have generated an overhead on the servers as well as generating a communications overhead and so reduced server capacity. The modelled approach worked well.

1.6.7 Systems administration

Systems administration with a high number of discrete systems can easily become a problem. The facilities of the RISC System/6000 SP helped significantly in this. From the control workstation it was possible to address all of the nodes, all the clients, all the servers or individual nodes as required.

1.7 Summary

The demonstration showed that it is feasible to manage work in a large configuration consisting of many client and servers. The building of such large configurations is not inherently a problem for most transaction managers. However building the configuration is not sufficient though. It is essential to be able to distribute incoming work across the servers in a flexible and realistic manner. The demonstration built across 144 nodes has shown that it is possible with CICS SM Workload Management.

CICS SM Workload Management scales extremely well. There is little additional overhead as further client nodes are added, since the routing decisions are always based on the information in the local cache. There is no need to communicate with other caches, where this to be necessary then the communications overhead would soon become prohibitive.

Because of its modelled approach to determining server load, as opposed to monitoring server activity CICS SM Workload Management, inflicts no measurable overhead on the server. This leaves the server free to process the work that it is supposed to, the applications.

Position Paper:

Why PC Servers Won't Overtake Mainframe Servers Anytime Soon

Wayne Duquaine
Grandview DB/DC Systems
10777 Cherry Ridge Road
Sebastopol, CA 95472 707-829-9633

Independent Consultant, Client/Server Interoperability

HPTS 1997

Abstract: My prior HPTS papers frequently sided with PCs becoming major Servers. The PC market now believes that it has reached Server status and will quickly become an "Enterprise Server" of choice. This paper discusses why current PC Server hardware and operating systems are still years away from kicking mainframes out of their F1000 Server role.

---

1. Introduction

Four years ago, conventional wisdom was that Unix was coming on strong, that Unix had legitimized itself as a Server, and large numbers of F1000 corporations were installing Unix Servers instead of upgrading their dinosaur-ish mainframes. It was clear that mainframes were doomed, and that the rapid advances in Unix-based hardware and operating systems, databases, and transaction processors would quickly overtake whatever niches the mainframe world still owned.

Today, conventional wisdom is that PCs using Windows NT Server are coming on strong, that PCs have legitimized themselves as Servers, and large numbers of F1000 corporations are installing NT Servers instead of upgrading their more expensive Unix systems. It is clear that Unix is doomed (it will get eaten by Windows NT Server), and that the rapid advances in PC-based hardware and operating systems, databases, and transaction processors will quickly overtake whatever niches the Unix world still owns. Those pesky mainframes will continue to hang around awhile, but ultimately they will be replaced by (Wolfpack) clustered PCs, who's superior ease of use and ease of scalability will win the war.

However, before the PC folks start crowing victory too soon, it would be instructive for them to analyze why Unix failed to dislodge the mainframe. The same Achilles' heels that nipped Unix's role in displacing mainframes are shared by PC Servers. These weaknesses can be summarized as follows:

- CPU-centric design.
- I/O Starvation under load.
- Poor Memory and I/O disaster recovery.
- Weak clustering (vs mainframes).
- Poor system tuning under load.
- Lack of sustainable 99.999 % uptime for an enterprise solution.

Each of these weaknesses will be discussed in the subsequent sections.

2. PC Servers: Kings of 1970s Mainframes

Today's PC Servers are by all metrics, equal to or superior to the mainframes of the 1970s. Unfortunately, it is not 1970 anymore. The essentially CPU-centric design of today's PC Servers reflects state of the art high-end mainframe design of the 1970s. This CPU-centricity leads to fundamental reliability issues, as well as throughput issues.

Since the early 1980s, mainframes have completely abandoned the CPU-centric view of the processor. All modern mainframe designs are memory-centric, coupled with redundant system buses. The CPUs (typically 4, 6, or 8 per system), are nothing more than fancy peripherals hanging off the buses. Additional engines, e.g. I/O Processors, Clustering Processors (aka Sysplex coupling facilities), and Service Processors are also hung off the system buses, replicated as needed based upon throughput and/or reliability needs.

Mainframe operating systems (e.g. MVS), exploit this view of the hardware. If a CPU crashes, MVS Alternate CPU Recovery (ACR) just shrugs it off. It can re-schedule the interrupted program in progress on a different CPU, back-step the PSW to just prior to the failing instruction, and re-start the failed program from where it left off. If the CPU fails while executing inside the bowels of the operating system, MVS Function Recovery Routines (FRRs) can clear up any CPU-specific locks, re-schedule the operating system service call on a different CPU, and re-execute the service. To date, none of the PC operating systems (including NT Server) provide such sophisticated levels of recovery. It is difficult, if not impossible, to achieve 99.999 % reliability and uptime in a mission-critical environment without such levels of recovery.

3. PC Servers: Masters of I/O Starvation

Part of "Amdahl's Law" states that in order to provide well-balanced system performance, there should be a minimum of 1 megabyte of sustained I/O throughput for each 1 MIP of processing power. By that metric, PC Servers fail miserably. Current PC Servers easily achieve 200+ MIPs performance, but have I/O architectures that have a hard time reaching even 40 megabytes per second sustained throughput (roughly equivalent to 1970 mainframe I/O performance). By contrast, a typical mainframe today can push sustained I/O throughput rates in excess of 500 megabytes per second.

Because of these system bottlenecks, I/O throughput becomes a limiting factor of how large a database, and how many users a PC Server (or clusters of PC Servers) can realistically scale up to. While it is certainly possible to put a terabyte of data on a PC Server, it is quite another to get high throughput against it in an update intensive and/or I/O intensive workload. Benchmarking such a PC workload against a mainframe that has more than 10 times the I/O throughput is a losing proposition. In addition, backing up, restoring, and logging also end up being hamstrung by the woefully inadequate I/O architecture that exists on PC Servers today.

While games can be played (cache large parts of the database in main memory, use battery backed up RAM, replicate parts to clustered servers, etc), a key fundamental limiting factor in PC scalability will continue to be its poor I/O architecture. Very large databases dictate large I/O throughput requirements, for updates, logging, etc. Building large databases and clustering on top of a PC I/O architecture that has feet of clay will not result in enhanced scalability.

4. Poor Memory and I/O Disaster Recovery

Current PC Servers have significant reliability problems when dealing with memory and I/O failures. Most current PC Servers rely upon Single-bit error detection and correction (SEC/SED) techniques for handling main memory errors. By contrast, mainframes utilize double-bit error correction and triple-bit error detection (DEC/TED) techniques. This is significant from a reliability (and hence uptime) perspective because of the dramatic reduction in size of RAM memory storage elements. 1970s style SEC/SED techniques are ok for memories of 64KB and 256KB RAMS, but become inadequate for large collections of 4M, 16MB, and higher RAM memories. Older memory technologies were only rarely afflicted by "background radiation" effects, such as alpha particles, because their RAM storage elements were fairly loosely packed and were as large or larger than a alpha particle. An "alpha hit" would at most knock out 1 memory cell. However, with the newer memory technologies, an alpha particle hit can typically take out 2 or more memory cells. On PCs, this results in a "crash and burn" situation, because the hardware cannot correct such errors. Since alpha particles are generated multiple times per day by everyday surroundings, (bricks, concrete, plaster, ...) this is a on-going reliability concern. Amdahl Corporation (purveyor of mainframes) learned this lesson the hard way in the early 1980s, when it had to undertake an expensive retrofit of the I/O processors on its 580 mainframes, because they were "crashing and burning" about once a month, due to alpha particle hits that were knocking out the I/O processor's 1 MB RAMs, which lacked the more powerful error-detection/correction techniques of the main memories. Hence, if alpha hits can cripple mainframes, they can also cripple PC Servers.

Mainframes also have very sophisticated memory recovery techniques, to keep the complex running, even when "uncorrectable" memory errors (aka large alpha hits) corrupt the memory pages that are running the operating system kernel. On a mainframe, if a memory error affects the OS kernel, the service processor reloads the page to a different part of main memory, changes the address registers to use the new page instead, and restarts the CPU processor that was affected by the error. By contrast, NT Server (and Unix) crash and burn in such situations.

I/O errors are another area where PC (and Unix) operating systems are significantly less reliable than mainframes. In a typical large server scenario (tera-byte size databases, etc), some form of serious I/O error (controller goes bonkers, disk drive goes bonkers, etc) occurs typically every other week. While PC Servers have RAID arrays to handle disk surface media errors, RAID is useless for preventing other errors such as Hot I/O or missing interrupts. Because of the large amount of DASD and the continual pounding, most large data centers see Hot I/O conditions at least once every two weeks. [A Hot I/O condition occurs when an I/O controller or its microcode gets confused, and keeps raining I/O interrupts on the main CPU. It does so at such a rapid rate that it can cause the main CPU to get swamped/overloaded with I/O interrupts, to the point that it cannot effectively get any other useful work done.] MVS has built in facilities to detect such conditions, and when they occur, it automatically tells the service processor to "fence off" (disable) the offending controller, by turning off its hardware interrupt gates. By contrast, PC and Unix systems end up crashing and burning in such situations. Worse yet, rebooting the system may not clear up the condition, so the PC Server just keeps being driven to its knees by an aberrant I/O controller. So much for enterprise level reliability.

The opposite of Hot I/O errors is the lack of getting a return acknowledgment from an I/O operation that was initiated (such as a disk read), because no reply was ever received back from the device or from its controller. These are referred to as "missing interrupts". MVS has facilitates to automatically retry when such errors occur, including using alternate I/O paths if necessary. Most PC and Unix operating systems by contrast, just hang when such conditions occur.

5. Weak Clustering

While we all know that "Wolfpack" and related PC clustering technologies are the greatest thing since sliced bread, it pales in comparison to the more sophisticated Sysplex clustering facilities that exist in the mainframe world. While the PC clustering techniques hope to scale up to support a dozen or so PC Servers, the mainframe today can cluster together over a hundred CPUs, providing aggregate support for over 800,000 users on one Sysplex complex. Suffice it to say, given the current design, it will be a while before Wolfpack and NT scale up to 800,000 users.

Mainframe software has been enhanced to use the Sysplex architecture for fully exploiting parallel database operations, flat file (VSAM) record sharing, and parallel logging across an entire Sysplex consisting of dozens of copies of the database server (DB2) and the transaction server (CICS). For example, in the new "Sysplex-enabled" version of CICS (5.0), on-the-fly logging to disk has been eliminated. The only time logs are now written to disk is at system (or CICS) shutdown. During normal operation, all logging is now directed to the Sysplex facility, which caches the logs in large, shared Sysplex memory. Since the Sysplex itself is serviced by a separate set of processors (part of the coupling facility), the main CPUs are no longer burdened by logging I/O overhead.

Lastly, new components in the mainframe operating system, called the "Workload Manager", are able to monitor utilization of the Sysplex, and in a rules driven fashion, can automatically start new (server) processes on any CPU in the complex as the workload on the system increases, and can automatically terminate server processes as the workload drops. Thus the system can dynamically increase or decrease its multi-programming level (MPL) on the fly. Such automatic system workload management is completely lacking for PC Server based operating systems.

6. Poor System Tuning Under Load

Mainframe MVS has some extremely sophisticated system tuning capabilities embedded within its System Resource Manager (SRM) component. It can tune the CPU utilization, memory usage, and I/O demand of database servers, transaction servers, and flat file programs, even down to individual users (e.g. favoring throughput or response time for one user or group of users for high priority operations). PC Server systems by contrast, are almost totally lacking in such tools. While NT nominally offers some tuning knobs, they are uneven in their impact, and difficult to predict. Under heavy loads, they are almost useless. Even moderate loads can cause unpredictable response times.

For example, start up SQL server, and crank up a suite of clients accessing it. At the same time start up a compile on the same system, and for good measure try running a few DOS .bat files at the same time. The system noticeably slows down (even screen re-painting suffers). Attempting to tune the system to favor the database server under load, usually results in sluggish

"foreground" (GUI user) response, even when the system is no longer under stress. This is because current NT tuning is fairly static, rather than being self-adaptive based on workload the way MVS's SRM is. As numerous enterprise studies have shown, erratic and unpredictable response times tend to drive end users batty.

7. No 99.999 % Uptime

The current PC RAS (reliability, serviceability, and reliability) philosophy is basically crash and re-boot quickly (e.g. "fail fast"). Both Unix and PC operating systems suffer from the "crash and burn" mentality for dealing with CPU, memory, and I/O failures. Because PC hardware is cheap, just replicate and cluster it. That will magically increase reliability.

This "cluster it and replicate" it approach using lower reliability PC Server operating systems has yet to be proven in a 24x7 365 days per year enterprise environment. And building a cluster and replicate architecture that is based upon a slow I/O architecture with poor recoverability means solving a number of non-trivial challenges.

What killed Unix as a central enterprise server is that it could not provide the same level of 99.999% uptime that MVS could. Typical Unix systems can run 1-2 weeks round-the-clock between crashes (about the same as NT Server). For F1000 corporations, this is entirely unacceptable. They are used to MVS levels of reliability, which can run for months on end without a crash. Because of this huge reliability mis-match, the great "downsizing trend" of the early 1990s has significantly slowed down, and in some corporations, reversed itself. Major banks and financial corporations (ala Wells Fargo, Schwab, etc) who 4 years ago were crowing that within 5 years all of their key mission critical stuff would be running on enterprise level Unix servers are now doing a 180 degree turn-around, because of the reliability issues. PC Servers offering the same levels of hardware and OS reliability as Unix are not going to fare any better at the enterprise level.

8. Bottom Line

The woefully inadequate I/O architecture of today's PC Servers will continue to act as a drag on how high PC Servers can scale, and how effective a MPL they can sustain. This will limit how much of an enterprise role they can play in supporting large numbers of users running I/O intensive workloads (such as database updates).

It will be significantly harder for PC Servers to replace mainframes at an enterprise level than most PC advocates want to 'fess up to. Fundamental hardware constraints, and recovery weaknesses in the OS indicate that it will be significantly longer in coming than most PC pundits want to admit. PC Servers suffer from many of the identical shortcomings that Unix Servers have. It is these shortcomings that have IS people backing away from using Unix servers as being central enterprise servers. They continue to be used as great departmental and branch office servers, but not as central enterprise servers. Until the analogous PC hardware and operating systems reliability and recoverability deficiencies are cured, PC Servers will encounter the same fate as their Unix brethren.

HPTS '97 Position Paper

Keith Evans
Johannes Klein
Jim Lyon
Francis Upton

TIP: Gateway to Heterogeneous Internet Transactions

There are de-facto (LU6.2 Synclevel 2) and de-jure (OSI TP) standards for heterogeneous transaction propagation and coordination. Both of these standards employ a tightly-coupled transaction protocol and application communications mechanism. i.e. they use a *one-pipe* model. The primary characteristic of this model is that the transaction protocol flows over the same conduit (pipe) as the application to application communications. The transaction tree is built implicitly as an application component communicates with another application component. This is achieved by the provision of a transaction protocol embedded within the communications services provided to the application (usually by a TP monitor).[1]

The use of a one-pipe model has the following consequences:

1. The application may use only the communications services supported with the transaction protocol (and no other), if it wishes transactions to be distributed. Work is required to invent a transaction protocol for each new communications protocol which comes along. As well as being wasteful (duplicative), this limits the possible communications mechanisms which may be used by an application.

2. To support the transaction protocol, a product must also implement the communications protocol. This significantly increases the complexity, which inhibits implementation. Hence, neither LU6.2 Synclevel 2 nor OSI TP are widely-implemented. Instead, numerous proprietary transaction protocols have been invented, which only work between themselves. This is wasteful and restricts interoperability.

The net of all this is that transactions may only be used where each component of a distributed application is running under the same TP monitor software, using only the communications services supported by that TP monitor, with a proprietary transaction protocol. This implies it must always be known in advance which application components are to be involved in a transaction, such that the user can arrange things to ensure all applications are written to use the

[1] In X/Open TP model terms, a Communications Resource Manager (CRM) cooperates with a local Transaction Manager to implicitly propagate transactions, the application uses the API of the CRM to communicate with other applications.

same TP monitor software. To limit the availability of the transaction paradigm in this way is undesirable since transactions greatly simplify distributed applications programming (the number of possible outcomes is reduced to two, and recovery in the event of failure is taken care of). Hence the development of new business applications is throttled by the restrictions imposed if transactions are to be used, and the complexity of writing distributed applications without them. This is especially true in the case of Internet transactions. There is no standard "transactional HTTP" protocol. And it is contradictory to the flexibility of the internet that it be known in advance which components will be involved in a distributed transaction, and hence to arrange it such that all components use the same proprietary transactional communications mechanism.

In summary, today's one-pipe standard transaction protocols have failed to become ubiquitous and consequently are of little practical use, and proprietary solutions are too restrictive to encourage use of the internet for distributed transaction processing. This limits exploitation of the inherent benefits of the internet for new business applications.

What is required is not a transactional version of HTTP, but a simple standard transaction propagation and coordination protocol which is *independent* of the application to application communications mechanism, and which may be employed with any such mechanism (present or future), including HTTP. This model of separation of transaction control and application communications is known as a *two-pipe* model. It is a testament to the simplicity and efficacy of this model that it is used by many existing proprietary transaction protocols. Leveraging this experience to specify a *standard* two-pipe transaction protocol is appropriate (and overdue).

Enter the Transaction Internet Protocol (TIP) [1]. TIP is a very simple two-phase commit protocol, which employs the two-pipe model. It achieves its simplicity by specifying only how different nodes agree on the outcome of a transaction; it requires that applications communicate via other protocols. It is intended that TIP be adopted by the IETF as an Internet Standard. While this protocol may not replace existing ones, it is expected to be relatively easy for many existing heterogeneous transaction managers to implement for communication with each other. It is assumed that TIP compliant transaction managers will offer a set of APIs to be used by local applications to import and export TIP transactions.

TIP offers two innovations compared to the current standard protocols which are particularly relevant to internet applications: 1) the *pull* method of transaction propagation, where a new participant requests an existing participant to add it to the transaction (the existing participant becomes the superior of the new participant). Among other things, this method allows a client to access an arbitrary set of servers and include them within the same transaction (this is a

typical web usage model); and 2) a method where transaction demarcation is controlled by the client application, but the client node does not participate in transaction commitment and recovery. This obviates the need for a full Transaction Manager implementation on the client system (i.e. with a log etc). This is most appropriate for internet applications where the client is typically running on a desktop machine using a web browser, with no local recoverable resources.

For example, consider a web-based electronic shopping basket application. The client (in the browser) requests books from the Moe's Books web server. The client begins a transaction with the Moe's server which starts the transaction locally and returns the transaction id to the client (via the TIP URL, see below). Moe's is currently the only recoverable participant in the transaction (the client is a volatile participant which may only request commitment or abort of the transaction, and be notified of the final outcome). A few good books are purchased from Moe's. The client then decides to buy some books from the Cody's Books server. When the client contacts Cody's, it passes the TIP URL (previously received from Moe's) and the Cody's server contacts the Moe's server (from the address information in the TIP URL) to *pull* the transaction from Moe's to Cody's, making the Moe's server the superior. The client is done and requests that the Moe's server commit the transaction (easy because the TIP URL contains the address information from the Moe's server). The Moe's server runs the 2PC protocol with the Cody's server and reports the outcome back to the client. During the 2PC protocol, each server handles the necessary coordination with local resources (typically via the XA interface). Should a failure occur, a recovery protocol is run only between the two servers, since the client is not a recoverable participant. Hence the distributed work is completed atomically. Note that communication between the TIP transaction managers takes place via means totally independent from communication between the application components (which in this example is vanilla HTTP between the web browser and server).

In order to facilitate the application use of TIP, a TIP URL is specified which defines the information necessary to propagate TIP transactions. This URL is passed between cooperating applications as the transaction context. The URL specifies the listening endpoint of the superior transaction manager, and a transaction identifier by which the superior knows the transaction.[2] This is all a subordinate application needs to supply in order for a local transaction manager to either pull a TIP transaction, or to put the application into the context of a previously *pushed* TIP transaction. Being architected, the TIP URL is also

[2] Whether a flat or hierarchical transaction tree is built depends upon which superior transaction manager listening end-point is passed via the TIP URL. i.e. it could be my own, or my superior's, end-point.

something an arbitrary application may look for in order to determine whether it should join a TIP transaction or not.[3]

The use of TIP confers the following benefits:

1. An application may choose whatever protocol mechanism is most appropriate to communicate with a partner.

2. There is no predetermination regarding which application components are to participate in a transaction. The only requirements are that, a) each node supports a TIP compliant transaction manager which manages the coordination of local resource managers with remote TIP transaction managers (the transaction managers on each node do not have to be the same), and b) cooperating applications agree about their communications data formats and protocol (part of which includes propagation of the TIP URL). TIP transaction trees can therefore be built up involving many arbitrary "TIP URL aware" applications, using many different communications mechanisms, and many different transaction managers.

3. As new communications protocols are developed, these can be used together with TIP to provide transactional versions of such protocols. There is no need for the development and implementation of new communications specific transaction coordination schemes. An application can avail itself of a new communications paradigm immediately.

4. Because of its simplicity and utility, it is expected that TIP will be widely implemented. TIP therefore offers a practical solution for heterogeneous transactional interoperability.

TIP is ideally suited to internet transactions where the actions of a client are entirely unpredictable regarding which components (e.g. web server CGI applications) will comprise a transaction (as shown in the "books" example above). TIP enables the development of web agency type applications, which act as brokers for the services of other providers. The agency application acts as the transaction coordinator. The agency provides screens to web browser clients, enabling them to open and close an electronic shopping basket, and in between to fill the basket with goodies selected from the various services offered. On close, the agency application commits the transaction (the participants of which were built up using the TIP push/pull mechanism, as the client filled their shopping basket). Providers to the agency simply need to write their service applications to recognise a TIP URL (present in an HTTP data-stream), and employ interfaces provided by a local TIP-compliant transaction manager to join

[3] See [1] for details of the TIP protocol and URL.

the transaction. The various distributed transaction managers then take care of transaction completion and recovery.

The TIP protocol itself is simple enough that implementation will not be onerous. Yet, it has characteristics that support high-performance. e.g. multiple TIP transactions may be multiplexed over a single TCP connection. Multiple TIP commands may also be pipelined together to reduce network latency and resource consumption. TIP is ready for prime-time. As HTTP evolves into an efficient server-to-server communications protocol, then the internet will be even better able to support enterprise transaction processing.

Summary

Today's standard transaction protocols employ a one-pipe model which renders them complex and hard to implement. Hence they are not widely deployed and are of little practical value. Use of proprietary transaction protocols inhibits interoperability and flexibility. These factors restrict the widespread use of the transaction programming paradigm, and limit development of new distributed business applications (especially on the internet).

TIP is a lightweight transaction protocol employing the two-pipe model. Being useful and simple to implement, TIP is expected to be widely deployed. It will therefore facilitate the widespread use of the transaction model, simplifying distributed applications programming. TIP frees the application from the limits imposed by TP monitor communications services, and enables the exploitation of new communications paradigms (while preserving transaction semantics). TIP includes methods which render it particularly applicable for use with the internet.

The net of this is to increase business application opportunities. Particularly, TIP will enable new internet applications which involve access to multiple web servers.

In the absence of any other practical widely-implemented standard transaction protocol, TIP represents a realistic opportunity for ubiquitous heterogeneous distributed transaction processing.

References

[1] Internet Draft, "Transaction Internet Protocol Version 2", J. Lyon, K. Evans, J. Klein. Available from:
ftp://ds.internic.net/internet-drafts/draft-lyon-itp-nodes-01.txt
http://www.tandem.com/INFOCTR/ARTICLES/TIPWEBAL/TIP.TXT

HPTS '97 Position Paper: Synergy Between Public Key Technology and TP Systems

Edward Felt
BEA Systems, Inc.
140 Allen Road
Liberty Corner, NJ 07938
ed.felt@beasys.com

Public key encryption and digital signature technology is especially well suited for use in transaction processing (TP) systems. This paper will provide an overview of what the technology has to offer, and how its benefits apply to TP.

Public key-based technology differs from traditional encryption algorithms because keys are *asymmetric:* there is one key for encryption known as a*public key,* and a second key known as the *private key* for decryption. The two keys are related by certain mathematical properties, yet one cannot determine a private key given its corresponding public key.

Because of this remarkable property, a user's public key may be disseminated widely and even published in various directories. The user's corresponding private key is kept secret. This allows anyone to encrypt data using the public key, but only the recipient, the holder of the private key, may decrypt it.

In actual practice implementing this type of encryption is more complex, and beyond the scope of this paper. Briefly, one needs a secure binding between a particular public key and an actual human being, known as a*digital certificate,* to prevent impersonation attacks. Also, a common technique known as a*digital envelope* enhances performance by reducing the amount of data processed by the public key algorithm.

Public key algorithms may also be run "backwards" to generate a*digital signature.* The purpose of a digital signature is not to hide data, but to prove the author's identity and the data's integrity. The author uses his/her private key to create a digital signature. Since only the author holds the private signature key, only the author may produce a legitimate signature. A signature is generated by computing a secure hash value from the data, and then "encrypting" this hash with the private key. Anyone wishing to verify a digital signature decrypts the hash value with the author's widely known public key, and then compares this value with an independently calculated hash.

With that brief introduction, let's move on to the importance of this technology within the context of a large distributed TP system.

The primary benefit offered by public key technology is increased security, essential to mission-critical bet-your-business TP systems:

- *Privacy:* preventing theft of data.
- *Integrity:* preventing undetected data tampering.
- *Authentication:* proving the originating user's or server's identity.
- *Non-repudiation:* preventing a legitimate user from denying previous actions.

These security needs are increasingly important as TP systems become involved with Internet commerce or distributed "intranet" applications running across a wide-area network.

To achieve high levels of performance and scalability, many TP systems limit the amount of session-specific state information stored centrally. Interactions with any particular user are typically brief and relatively infrequent, and systems often support thousands or tens of thousands of simultaneous users.

In this environment public key technology is very attractive for building a secure system which preserves TP's necessary performance and scalability characteristics. Information to ensure privacy, authenticity, and integrity may be associated with a client computer's requests: a digital envelope, a digital signature, or both. Roughly half of the computation effort is performed on a user's desktop machine, where CPU resources are essentially free. (Performing an encryption and digital signature on a small message takes under one Pentium CPU second.) Large multiprocessor server machines can linearly scale up to handle their share of the computations.

Unlike other security architectures, public key technology does not require communication with any third party in real time. This is an important performance consideration, especially given recent disproportional advances in computing power versus network latency and throughput. The self-contained nature of public key encryption/signature also reduces failure points.

Many TP systems support queued communication facilities. Because digital envelopes and signatures are associated with communication messages, they easily travel along with the data they protect, even when stored in a memory-based or disk-based queue. Communication models where there is little established relationship between the two parties, such as publish/subscribe or datagram messaging, also fit well with this technology.

In summary, public key technology offers the potential for excellent security, while meeting the performance, scalability, and reliability demands of large distributed TP systems. Challenges in this area involve management of digital certificates, establishment of a key

recovery/escrow infrastructure, and issues related to U.S. export policy.

References:

- RSA Laboratories' Answers to Frequently Asked Questions About Today's Cryptography: http://www.rsa.com/rsalabs/rsalabs .htm
- *Applied Cryptography*, Second Edition, By Bruce Schneier, John Wiley & Sons, 1996. http://www.counterpane.com/applie d.html

Author's Bio:

Ed Felt, a Staff Engineer with BEA Systems, has designed and written software forTUXEDO System releases 4, 5, and 6. He is co-inventor of the EventBroker publish/subscribe communication feature, and the System Event Monitor subsystem. Currently, Ed is responsible for enhancements to TUXEDO's security subsystem in release 6.3 and future releases.

Storage Metrics

Jim Gray, Goetz Graefe
Microsoft Research, 301 Howard St. #830, SF, CA 94105
{Gray, GoetzG}@Microsoft.com

Abstract:

Simple economic and performance arguments indicated appropriate lifetimes for main memory pages and sizes of main memory buffer pools. The fundamental tradeoffs are the prices and bandwidths of DRAMs and disks. They roughly indicate that current systems should have about 10 MB of DRAM per randomly accessed disk, and 100 MB of DRAM per sequentially accessed disk. They also indicate that 16 KB is a good size for index pages today. These rules-of-thumb change in predictable ways as technology ratios change.

The Five-Minute Rule Ten Years Later

Magnetic disk performance is improving quickly, but different aspects are improving at different rates. The charts in Figure 1 roughly characterize the performance improvement of disk systems over time. The caption describes each chart.

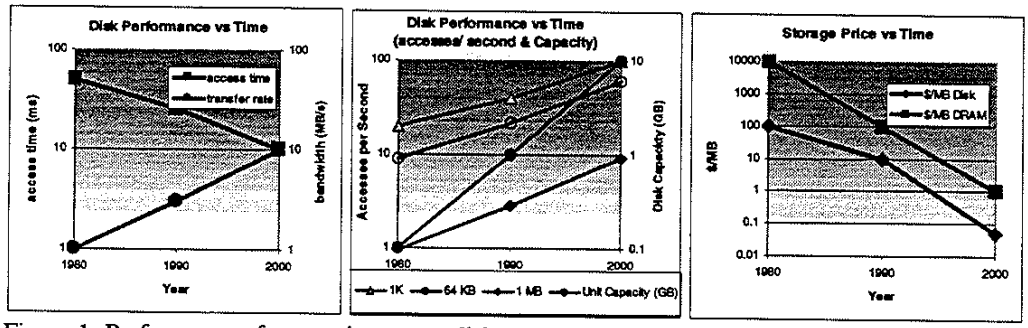


Figure 1: Performance of magnetic storage disks over time. The first two graphs show features that improved 10x or 100x while the third graph shows features that improved 10,000x in the same time. The graphs show that access times have improved relatively little, while prices have dropped by a factor of 10,000; unit capacity and accesses per second have grown by a factor of 100. Higher bandwidth and cheaper DRAM buffers have allowed larger pages. That is one theme of this paper.

In 1986, pages referenced every five minutes should have been kept in memory rather than reading them from disk each time [1]. Actually, the break-even point was 100 seconds but the rule anticipated that future technology ratios would move the break-even point to five minutes.

The five-minute rule is based on the tradeoff between the cost of DRAM and the cost of disk accesses. The tradeoff is that extra memory can save disk IOs. The beak-even point is met when the rent on the extra memory for cache ($/page/sec) exactly matches the savings in disk accesses per second ($/disk_access). The break even time is computed as

$$BreakEvenReferenceInterval = \frac{PagesPerMBofDRAM}{AccessPerSecondPerDisk} \times \frac{PricePerDiskDrive}{PricePerMBofDRAM} \text{seconds} \quad (1)$$

The disk price includes the cost of the cabinets and controllers (typically 10% extra.) The equations in [1] were more complex because they did not realize that you could factor out the depreciation period.

Figure 5 graphs the benefit/cost ratios for various entry sizes and page sizes for both current, and next-generation disks. The graphs indicate that, small pages have low benefit because they have low utility and high fixed disk read costs. Very large pages also have low benefit because utility grows only as the log of the page size, but transfer cost grows linearly with page size.

Table 4 and Figure 5 indicate that for current devices, index page sizes in the range of 8 KB to 32 KB are preferable to smaller and larger page sizes. By the year 2003, disks are predicted to have 40 MB/s transfer rates and so 8 KB pages will probably be too small.

Summary

The fact that disk access speeds have increased ten-fold in the last twenty years is impressive. But it pales when compared to the hundred-fold increase in disk unit capacity and the ten-thousand-fold decrease in storage costs (Figure 1). In part, growing page sizes sixteen-fold from 512 bytes to 8 KB has ameliorated these differential changes. This growth preserved the five-minute rule for randomly accessed pages. A hundred-second rule applies to sequentially accessed pages.

References

[1] J. Gray & G. F. Putzolu, "The Five-minute Rule for Trading Memory for Disc Accesses, and the 10 Byte Rule for Trading Memory for CPU Time," Proceedings of SIGMOD 87, June 1987, pp. 395-398.

[2] Dell-Microsoft TPC-C Executive summary: http://www.tpc.org/results/individual_results/Dell/dell.6100.es.pdf

[3] Sun-Oracle TPC-C Executive summary: http://www.tpc.org/results/individual_results/Sun/sun.ue6000.oracle.es.pdf

[4] Ordinal Corp. http://www.ordinal.com/

HPTS Position

Don Haderle
IBM
555 Bailey Ave.
San Jose, CA 95141
haderle@us.ibm.com

Databases will continue to address consumers needs for reliable data storage and high performance availability to it. Object relational databases are integrating alien data stores and computation (e.g., spatial stores). With respect to this view, they don't focus on storing the bits, but rather on integrating the capabilities of someone else's facility with the SQL language and, in some cases, with the operational management of the store so that there is consistent management for the administrators. In essence, they provide a framework for heterogeneous, federated database management.

This is a major theme for enterprises; viz., to aggregate multiple data sources and provide a bridge between existing operational systems and new operational systems which address deployment of new applications. This theme extends to cross enterprise into the domain of EDI and beyond. The quest is to add/modify applications and data items without disruption to existing application systems and/or databases; continuous reengineering.

This game has shifted to the middleware. There are several contenders for this position - object relational, corba, com, etc. The distinct advantage that an OR approach has over the others is the semantic integration of the facilities and the capabilities of the relational optimizers to formulate respectable performance in a complex topology. Corba offers object distribution, not object integration; that's left to the provider above Corba; similarly for OLE/COM. It seems to me that the eventual winner in the middleware will be Object Relational chosen for its semantic integration, with access deployed accross a small set of common pipes - viz., Corba, OLE.

This, in effect, becomes the next generation transactional database.

Operational Data Stores Must Unite

James R. Hamilton
Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399
JamesRH@microsoft.com

Pat Selinger
IBM Santa Teresa Lab
555 Bailey Ave.
San Jose, CA 95141
Pat@almaden.ibm.com

0. Introduction

We propose a unified operational data store called a Virtual Object File System (VOFS) as the next level of evolution toward integrated shared data that was begun with the invention of a shared database or file system. It is our belief that these concepts should be unified, just as the past decade had the theme of unifying object-oriented databases and relational databases. Such a unified data store would replace the private storage managers implemented today by individual subsystems such as databases, mail systems, workgroup shared file systems, queuing systems, purpose-built data management systems, and many others. We envision that an existing relational database management system would provide the infrastructure upon which VOFS is built. The RDB storage manager with some extensions will be the VOFS storage manager and the SQL Query Processor will process SQL queries against the underlying VOFS data store.

The proposal for VOFS combines two lines of research: 1) integrated storage, and 2) caching file systems supporting disconnected operation. In this paper, we present the background issues, propose the virtual object file system, describe how it solves current storage management problems, and discuss the advantages and disadvantages of such a system.

1. Background

Most persistent client applications and current server subsystems implement private storage managers uniquely tailored to the needs of that particular application or subsystem. Client applications typically implement persistence by storing into either individual file system files, or somewhat more complex storage systems built in the application upon the file system. Server side subsystems such as mail, queuing, net news (NNTP), vertical applications, HTTP, directory, groupware, workflow, and other collaborative frameworks generally implement a proprietary storage subsystem built upon the underlying file system. The advantage of each persistent subsystem implementing its own storage manager is that this storage manager is customized for the functional and performance needs of that dependent subsystem.

There are, however, significant disadvantages to the customized approach. The most obvious is that, while all storage managers share much in the way of common technology, each must implement and

maintain this technology. Although implementation and maintenance costs are increasing, they actually remain a small problem relative to the other implications of the independent storage manager strategy. The major cost of this strategy derives from the necessary co-existence of each disparate subsystem. Each subsystem has its own (different) mechanism to name persistent objects, navigate between persistent objects, apply transactions including persistent changes made by multiple subsystems, support a unified persistent object security model, selectively replicate the persistent store, support disconnected operation, backup and restore the persistent store, efficiently support shared objects, support hierarchical storage management of the persistent objects, recover from both system failure and user error, and support non-procedural query. These mechanisms are not unified across multiple subsystems. We also note that rapidly declining disk and memory costs, coupled with faster processors and nearly universal network connectivity are enabling dramatic increases in the amount of on-line data, yet the persistent stores of many subsystems are ill-equipped to scale to these data volumes. Many of these customized individual stores are facing the challenge of having to make substantial investments and reengineer in order to accommodate these data volumes, and in addition, scale for SMPs and hardware clusters working in parallel. The need for integration plus scalability is driving this virtual object file system proposal.

2. Our Proposal

We view this unification of database systems and file systems as a new step in the evolution of database management systems. In the past decade, object-oriented databases became unified with relational database systems, resulting in object-relational capabilities within extensible database systems. In a similar way, we envision bridging the separate technologies of file systems and object-relational databases, resulting in a virtual object file system supporting both capabilities in an integrated system.

The driving forces of this evolution are the rapidly increasing volumes of information online and available to each user, the cost of developing and maintaining independent data stores, and the cost to customers of administering and managing separate data stores. These factors are causing subsystems developers to review their storage management infrastructure investments. Increasingly, the conclusion is that storage management is not their core business and that development investment should be focused upon delivering product function while depending upon a vendor-supplied general purpose data management system.

In the next section, we will detail the driving forces behind this evolution, examining both client and server-based applications and their storage subsystem needs.

3. Current Subsystem Storage Architecture

3.1 Current Server-Side Application Subsystems:

Looking at some typical server applications, we see the entire spectrum, from applications already depending upon this approach to applications that are exclusively implemented upon proprietary data stores. For example, vertical applications such as Peoplesoft [24], Baan [1], SAP [26], and Oracle [22] are already dependent upon commercial database management systems, whereas applications such as queuing, mail, workflow, groupware and other collaborative servers are often still implemented using proprietary data stores. However, as such systems increasingly become "mission critical" and require high data volumes and large numbers of concurrent users, commercial data storage managers become reasonable solutions. For example, Jim Gray's 1995 HPTS paper, "Queues are Databases" [6] made exactly this argument for queuing systems.

Another class of server-side applications requires high speed, very short transactions with read-mostly data access. Web-based applications such as Lotus Domino [5] or Microsoft Internet Information Server [19] are prototypical examples. With these applications, there is a perception that performance matters more than robustness, so the argument to use a commercial data store is a more difficult one to make in these cases. However, we know that system resources are becoming less costly roughly in proportion to Moore's law [25], while administrative costs continue to grow at least as quickly as inflation and usually much faster, driven by increasing system complexity.

As we examine server-side systems, we see the following trends:

1) The evolution towards integrated data storage is already far advanced. Many server-based applications already use a

database management system for storing the data associated with their application. Nearly all of the remaining applications depend on the file system for centralized storage services.

2) Over time, in response to increasing demands for productivity and consistency, databases have enriched their functionality. Applications can share objects, rules, triggers, and relationships, while retaining all the benefits of database systems such as atomic actions, independence from the impact of concurrent users, durability, and integrity.
3) There has been a dramatic increase in the use of the same data by sequences of applications as well as by unrelated applications such as point of sale data being processed by data mining applications. Together with the growth of on-line data, the increased number of applications accessing the same data is driving significant advances in data storage scalability and heterogeneous data access.

To summarize, converging requirements are driving server-side systems to either share storage managers or, at minimum, implement independent storage managers of equivalent functionality. We believe that the major server-side storage managers have converging requirements and conclude that these requirements could be best served by implementing their function either in a single storage manager, or a small number of storage managers that are tightly integrated with underlying system services. Example shared services are, transaction management, security, naming, object browsing and navigation, replication, utilities, administration, management, and monitoring.

One of the most important effects of reducing the number of storage managers or building them upon a common set of infrastructure components is the reduction in administrative costs. It is well documented in both the trade and popular press that personal system administrative costs are out of control. Gartner Group estimates that 28% of the total cost of ownership of a desktop system is administrative and technical support [16]. A much greater concern highlighted in this same study is that 56% of the cost is in end user's wasted time due to system failures as well as "unproductive activities attributed to the extensibility of today's PC environment."

We propose a solution to both these problems through the Virtual Object File System. VOFS will be a single integrated data store that supports the advanced functions described earlier in the paper, yet still supports current or legacy interfaces and allows all existing applications to continue to run unchanged. The reduction of administration costs are perhaps less exciting than the new function made available but, in the end, solving this problem might be the most important contribution.

3.2 Current Client-side Application Systems

Next we look at client-side systems, a world much less evolved from a storage management perspective than current server systems. Today there are some fairly significant differences between the requirements of server-side storage managers and those of client systems. Client systems typically store personal data that is either produced by the user of the system or provided for the exclusive use of that user. Although current clients typically support far fewer users than server systems, somewhat less online storage, and a relatively larger number of programs with fairly simple storage requirements, we assert that client storage managers are beginning to face requirements very similar to those described in the previous paragraph on server storage management.

As we examine client application subsystems, we note the following trends:

1) There is an ever-larger storage management problem at the client, driven by rapidly increasing disk capacities and increasing amounts of information available to the client system as a result of the nearly universal connectivity provided by public and private networks. However, the weakness of current storage management solutions is the inability to use a single query interface for all local and internet-attached storage managers, along with escalating client administration costs.
2) Information integration is becoming increasingly important. Just as data warehouses are very valuable for operational data, we are discovering that the same is true of personal data such as mail, schedule, personal documents, presentations, etc. For example, "I recall recently seeing a reference to a meeting on the advertising budget but I don't recall if it was scheduled on my calendar, referenced in a mail message, or mentioned in a status report."

This question could be answered by querying all mail read, all documents received, and all meetings attended during this period. With current systems, many resource managers such as mail stores support query but only through a special purpose interface supported only by that one subsystem. Other resource managers such as calendars often support no query interface whatsoever. Integrating naming, navigation, query, transactions and security is clearly very useful.

3) The cost of administering individual stores per application grows with the proliferation of both applications and data. This drives customer organizations towards dependence upon a single resource manager for all client applications or to a dependence upon a small set of resource managers supporting centralized administration.
4) Client systems are no longer dependent strictly upon locally-stored data. They are now susceptible to the combined failure frequency of all networks and systems upon which they depend. Further, portable systems are frequently intentionally disconnected and used in transit, or in locations where connection is not possible, not affordable or inconvenient.

3.3. Implications of these Trends for Storage Subsystem Architecture

We see three possible solutions in response to these trends:

1) All storage managers support the same function and interface,
2) Multiple storage managers (a relational database and a file system) with integrated functional and administrative interfaces support all application subsystems, or
3) A single integrated storage manager that supports both relational access and file storage.

Briefly we will discuss each of these three options.

All storage managers support the same function and interface:

All of these applications could satisfy the functional requirements listed above by implementing a proprietary database-like layer. However, unless each application depends on the same uniform set of services, no progress will be made towards unification of naming, security, navigation, and non-procedural query. Further, if administration costs are to be reduced, then we will need to see administrative unification as well. And, if it were possible to implement functionally identical interfaces for dissimilar storage managers with support for common administration, would the resource footprint occupied by the multiple storage managers be affordable?

Another argument against this approach is that if application developers are freed from worrying about storage management complexities, then functionally richer yet less expensive applications will emerge. Further, some storage management problems are sufficiently complex that they likely can't be affordably solved in some applications. Support for disconnected operation is a prime example of one such function. In just the same way that the operating system has taken over many of the resource management functions that each developer originally needed to manage, we believe that considerable savings can be realized by providing a higher level of abstraction for persistent data access.

The advantages of making it easy to write better applications by providing better building blocks surround us. The WWW is an ideal example. Prior to the Tim Berners-Lee invention of HTML and the HTTP protocol [2], internet use was primarily restricted to those with good computer skills, most of whom were programmers, academics, or students. The availability of HTTP as a base infrastructure made much more information available to less computer-literate users, which fueled the development of ever-simpler browsers. These were so easy to use that many more users came online, which fueled massive corporate investments in putting information online, which again fed back into bringing more users online. This same principle applies equally to client side storage management. If it's easier to write information-intensive applications, then more are written which brings more users, which again fuels more development investment in improving these tools.

Most, if not all, agree that a high-level data access interface abstraction is required. Many have been proposed, with Microsoft OLE DB [17] being one of the more recent, for solving this problem. Such approaches make a significant contribution in providing uniform access to legacy data stores. However, we've argued throughout that considerable advantage can be realized if all the data is actually stored in a single integrated storage system. It's that next step that we are attempting to make here.

Multiple storage managers (a relational database and a file system) with integrated functional and administrative interfaces supporting all application subsystems:

One might argue that a complete integrated solution is overkill for many applications. Essentially, it is a compelling argument that storage-intensive applications can easily be implemented on a commercial database management system, but non-data intensive applications clearly don't need most of the function provided by the commercial data management system.

Therefore, the solution might be to depend upon several storage managers. We agree that many non-data intensive applications, especially those running on clients, can be hosted upon a file system satisfactorily; however, this is the wrong question. Rather than asking, "Is the file system sufficient?" we ask "Is there a way that these applications can be hosted upon a single integrated storage management system without consuming excessive resources?" As we observed earlier, there are considerable reasons to depend upon a single integrated storage manager supporting both relational access and file support at much reduced administrative costs.

A single integrated storage manager that supports both relational access and file storage:

This is perhaps the most radical and, if successful, the most elegant solution. However, it should be remembered, the goal is increased user function with reduced administrative costs through integrated storage. Integrated storage itself isn't the goal so, if we conclude that some aspect of the storage problem is not addressable in a single integrated data store with current technology, we should simply fall back to providing the second option, several storage managers with integrated functional and administrative interfaces.

This approach costs less than the other alternatives from a development perspective because, by delivering advanced storage management capability in a single common mechanism, the work is done only once. However, many excellent ideas have failed in the market place by forcing adopters to forget the past and discard their existing systems and investments. It is imperative that any integrated storage manager support significant legacy APIs.

Having discussed each of the three options, we argue in favor of the third, based on efficiency and engineering difficulty, as well as on the basis of the functional and administrative benefits of depending upon a single unified data store. For the remainder of this paper, we'll investigate the third alternative exclusively.

4. The Virtual Object File System

The challenge is to provide a single integrated data store that supports the advanced function described earlier in the paper, yet still supports legacy interfaces and allows all existing applications to continue to run unchanged. We propose the Virtual Object File System (VOFS) as a possible solution to this problem.

Although VOFS could be built from first principles, we believe it is easier to start with an existing database management system. From this system we'll be re-using the storage manager which, after some extensions are added, will form the basis of the VOFS storage manager. And, we will take the SQL Query Processor and run time system and use this as the basis for the VOFS SQL processor.

VOFS combines two lines of research: 1) integrated storage management, and 2) caching file systems supporting disconnected operation, exploiting the ideas proposed by Satyanarayanan et al on the Coda file system [27, 28, 29]. We extend the Coda work by combining it with integrated storage management, providing the advantages of Coda including support for disconnected operation to all storage dependent applications implemented on the VOFS storage sub-system.

Considerable research has been published on caching file systems supporting disconnected operation by the Coda team at Carnegie Mellon University [15], the Bayou Architecture team at Xerox PARC [4], and by P. Honeyman at the University of Michighan [9] as part of his cache consistency work. These research systems have progressed to the point where integration into commercial systems makes sense, however, the research to date has, for the most part, focused exclusively on file systems rather than storage managers in general. Client systems typically depend upon mail, calendar, web, net news (or other collaborative information resources), and often database access, in addition to file system access.

The VOFS client cache approach supports the following uses:

- Portable and disconnected operation
- Continued operation through network and server failures

- Improved program and data access performance through local caching
- Self installing and self personalizing subsystems
- Identical environment across many different systems (and supporting guest systems)
- All data ever experienced is always available

The following is an example of the use of such a VOFS in disconnected operation: the client system only has a cache (a copy) of the user's environment, system software, programs, and data. As a consequence, if the user uses a different system at home and at work, both systems can support identical environments. Further, users visiting a branch office in a different city can use any desktop in that office by simply connecting to whatever Internet Service Provider (ISP) is used in that area and authenticating themselves. If it's a different ISP from the one they normally use, an inter-ISP cache transfer will be done in much the same way that a bank card can withdraw cash from a different bank in a different city. After the inter-ISP transfer, if required, the user's environment trickles down to the local personal system cache.

We would implement the VOFS storage manager in much the same way that a typical file system is implemented, namely in the operating system kernel (protected) address space. The query processor and SQL runtime system is outside the kernel running in the same address space (process) as the calling program. We would implement the storage manager in the kernel address space so that it can be accessed from the file system interface via a mode switch to the protected kernel address space without requiring a context switch just as many file systems are implemented. The SQL Query Processor and Runtime system would be implemented outside the kernel to avoid dynamic and difficult to bound resource consumption problems. The memory resources consumed by the query processor are dynamic, related to the complexity of the query, and difficult to predict in advance, which make it difficult to safely host the query processor in the kernel address space. An alternative that may be feasible if implementing VOFS on a micro-kernel based operating system such as Next, Mach, Windows NT, and Apple's next major operating system shipment, Rhapsody, is to implement the query processor, run time system, and storage manager as a protected sub-system. Micro-kernel systems implement most operating system function in independent user space servers. Communications between application programs and called servers would be via highly-optimized Local Procedure Calls (LPC). On some research systems, the LPC is actually implemented without a context switch and is very nearly the same cost as the mode switch required to request a system service in a conventional operating system.

To solve the problem of making the file store available to the VOFS SQL query processor, it will be necessary to impose a relational schema on the file system. We need a representation for directories, a representation for files and some form of query scope support (the file, directory, or sub-tree upon which the query should operate). Technically the query scope support isn't required since some relational database management systems implement recursion, however, we feel that the direct reference is a more natural abstraction for file system users.

When designing the relational schema for the file system, we note that we will need to be able to store file and directory names and, for both, we need to store metadata such as creation time, last access time, last changed time, security access control list, etc. A directory entry contains zero or more files and directories so we'll need a representation of multiple entries for the directory. A file entry contains no sub-entries, only data. Both a file and directory will have names and metadata differing only in that directories may contain directories and files whereas files may only contain data. We will represent files and directories (and related metadata) in a single FileObjects table and we will represent the many to one relationship of directories to files and directories in a DirectoryEntries table.

The basic goal for each storage API implemented by VOFS will be to 1) faithfully implement the required API and efficiently match the semantics and performance expected of that storage API, 2) represent both the storage objects and the metadata describing those objects in a single format accessible by all APIs implemented by VOFS. This approach of sharing metadata will permit all the storage APIs to share the data store. Because the underlying metadata and storage format is the same, each API will be able to access objects stored by another API and, where meaningful, insert new or update existing objects.

An example of an SQL query against a file system is as follows: find all files in the directory \dir1\dir2 or any of its subdirectories whose name begins with HPTS:

Select name from vofs.FileOjbect
where SubDirectories('\dir1\dir2')
and type=file and name like 'HPTS%'

We will use the storage manager recovery log both to ensure that all file system operations are transactional and recoverable and to log all object references for later use by the cache manager. The cache manager is responsible for keeping recently accessed objects in the client cache and for evicting objects unlikely to be referenced in the near future (a modified least recently used algorithm will be used). We propose to use a conventional write-ahead logging (WAL) system. One potential concern with this approach is massive log growth caused by large changes to file data. For example, it's not uncommon for a desktop application to completely replace an entire file. There are many solutions to this problem including shadow paging and adding constraints to the buffer manager such as not over-writing a page until commit time (no undo record needed) or forcing new and changed pages to disk at transaction commit time (no redo record required unless media recovery is supported).

It's important to remember that, although the VOFS log manager will be writing more data to disk than would be required in a conventional file system, the write ahead log protocol allows us to avoid writing the changed data pages to disk at commit time and, therefore, only the log is written synchronously. Clearly the changed data page must eventually be written to disk, but for client systems with a VOFS running on a personal computer, there will be many periods of unused I/O bandwidth where this page can be asynchronously forced to disk.

If VOFS is running on a client system, the log can be transparently migrated up to the server either by trickling up to the server or by batch migration on re-connection. In the later case, it is possible to post-process the log during trickle migration to the server where undo records are discarded if the transaction has committed and redo records may be discarded as soon as the changed pages described by the log records are propagated to the server system. The log is saved indefinitely, as it represents the life history of the user who created it, and could be used for many purposes including cache management and to support other advanced access methods not described here.

5. Critique of VOFS

We assert that VOFS can help raise the storage management abstraction level available for programmers making them more productive, increase user productivity by offering functions previously unavailable, provide unique support for disconnected operation, and reduce client-side administration costs. However, there are a number of criticisms that can be levied against the general concept.

1. ***Difficulty of achieving a common unified subsystem:***
   Consider the mechanisms by which such a common unified VOFS system would be designed and built. There are several practical possibilities:

   A. De jure standard in which a set of potential providers joins together in a committee to identify and define a set of common interfaces that their planned products would meet. Examples of similar work are the ANSI X3H2 committee to define the SQL language standard. Such committees may work slowly and may be tempted to over-define in order to satisfy all members; rather than making the hard decision between two definitions aimed at solving the same problem, they may be inclined to adopt both mechanisms in order to satisfy all participants. Such a resulting system may have features and functions that are redundant, and therefore cost more to build. Such expense is justifiable only in proven marketplaces where the cost of not staying current with competition outweighs the cost of overbuilding.

   B. De facto standard in which a significant provider is able to define and build a set of common interfaces. This provider would either have to own the platform or have sufficient market share that the subsystem definitions it provides become, at least in that marketplace, a compelling definition for many providers to support. Examples of such systems in the operating system arena are MVS and Windows. There are many vendors competing for storage subsystem mindshare: Microsoft, Oracle, IBM, Novell, Informix, Sybase, all Unix operating system providers, and others. And these vendors are such strong competitors, that competitor X might find it difficult to adopt the interfaces promoted by competitor Y because it would provide Y with undue control in setting strategic directions and therefore Y would have an undue commercial advantage. De facto standards work best where there is only one strong dominant provider and other niche competitors adopt that direction to gain a foothold.

Alternatively, a strong provider could "do it all" and provide a unified platform on which they build their components and have enough marketplace presence that it doesn't matter if competitors might not play along. This is a scenario that is feasible and has been done before. When the interfaces and the concept has merit and provides significant real value, for example, Sun inventing and promoting Java, then this can be a very successful approach.

In summary, a common VOFS data store that requires the cooperation of natural competitors to define an acceptable set of primitives for that data store is quite difficult to achieve and would present a significant barrier to the adoption of the proposed VOFS. But the alternative of a significant vendor defining a set of interfaces could very possibly be successful if the idea has merit and significant value in the marketplace.

2. ***Reduced control and less ability to provide differentiation:***
Developers often prefer to implement their own solution rather than depend upon a previously-produced solution built by someone else. If an application vendor were to depend upon a VOFS, a general purpose storage manager, they would be giving up storage manager ownership and control and losing the ability to add unique optimizations aimed specifically at the needs of their application. And, since they would be sharing the same storage managers with competitors, they would be reducing the proportion of their system where they can innovate and differentiate themselves from their competitors.

A specific example illustrates this. One mechanism queuing subsystem software vendors use for competitive leverage is better performance, a significant proportion of which will come from specialized mechanisms for shorter pathlengths and fewer I/O's combined with special storage structures adapted for large volume of small-sized, short-lived data items. A general-purpose storage facility would not necessarily be tuned for this design point.

Another example is the fact that database systems use buffer managers to retain pages across multiple uses to reduce or avoid I/Os. Because the database engine or its administrators know the expected access patterns to disk data, they can assign buffers and their associated sizes and priorities to different data objects. For example, they can place indexes in a different buffer pool than data tables. In addition, because multi-page scans of data objects are frequently used, the database engine can, with its own proprietary storage mechanisms, provide techniques such as prefetching sequences of pages from a logical object, discarding pages from the buffer pool that are known to be read once (e.g. pages used for temporary storage during sorting), and similar mechanisms. Such mechanisms are known to provide significant performance leverage. Factors of 10 and 20 in performance compared to simple systems can be cited. While such mechanisms can be built into a common VOFS, there is a large and ever-increasing repertoire of them, and simply the volume of architecture work and implementation in an VOFS to accommodate even the significant techniques across all interested subsystems is daunting.

Consequently building their own proprietary mechanisms offers application developers the ability to:

A. use specialized data structures and innovative customized search techniques,

B. exploit platform features in ways that are uniquely tailored to the needs of that particular application or subsystem, and

C. provide enhanced customer value through providing the entire solution, not just a small percentage of it

The counter-arguments to these criticisms are based on pragmatic business sense and knowing the trends in computer evolution. In general, giving up control of data to another subsystem, possibly implemented by another vendor, is only done, from a developer's viewpoint when it is "worth it". Almost all application developers do this for file systems and operating systems, because the value gained from writing their own doesn't have sufficient benefit. We no longer write in assembler, but use high-level language compilers because of the saved implementation costs and the productivity gains. Just as programmers who use high level languages and file systems give up control in return for greater development speed, we believe that a Virtual Object File System could also be a similar good "buy". The software depended upon must have sufficient performance, function, and quality/reliability that a developer wants to use it

instead of writing his or her own. We believe that building on the file system or a higher level abstraction as an interface instead of bare-handing a proprietary storage manager is simpler and, unless the developer is very smart indeed, better performing and more reliable.

As for the application developer's concern of losing customer value, this is a practical business concern. If a single vendor supplies all parts of a solution needed by a customer, then that customers' business is more secure as a source of sustained revenue than a situation where the application developer provides only ten percent of the solution and the rest is commodity parts available to anyone. These are business model issues, and we point out that successful application developers make good judgements about tradeoffs, particularly where additional work would have the most leverage and revenue potential.

As a final counter-argument, with the foreseeable drops in hardware prices and the increasing costs of software development, maintenance, and administration, we believe that using more common software parts, including a virtual object file system if one is available will be compelling from a cost standpoint. To net it out, a vendor can afford to invest more in attractive, marketable function because they exploit common infrastructure and spend the savings on enhanced function above the infrastructure interface. History proves this out. Only a small number of very specialized application vendors build their own operating system or file system.

In summary, with common parts, vendors can exploit some of the savings generated to be able to add significant differentiating function to their applications. A vendor could deliver an integrated solution, especially if they offer more than one application, that saves development dollars and helps all those customers that buy several storage consuming applications from them. So a VOFS solution doesn't have to be cross-industry to have value; it can have value even for products offered by a single vendor.

*3. **Multi-platform issues and application builder motivation:***

A common VOFS across platforms (both operating systems and hardware) would motivate cross-platform application vendors to adopt the VOFS set of interfaces rather than continue to use their current proprietary ones. If a vendor is faced with the challenge of supporting, say, 10 operating system/hardware platforms, and the VOFS is available only on 5 of them, there is reduced motivation to adopt the higher level VOFS, because he has to continue to invest in the customized layers he has built on top of the file system for the other 5 platforms. Such a vendor would be motivated to move to a VOFS if it was available and supported by a reliable provider across all the platforms in which that application sells. Such a provider would further have to be committed to providing support for new operating system, device, and hardware features as rapidly as application vendors (especially a direct competitor who has not chosen to use the VOFS) could add them in their own customized layers. Providing a VOFS that is common across tens of platforms and operating systems will become either a responsibility of the platform provider, in which case commonality of function across platforms becomes a problem, or it is the responsibility of a single third party provider, in which case function is common, but platform exploitation may suffer. Or a set of third party providers could supply the VOFS functionality across different platforms; in this case, rapid platform exploitation still remains a problem, and in addition different VOFS providers will have different implementations and very likely different performance characteristics and different delivery schedules for new VOFS functions. Each of these factors reduces (but doesn't necessarily eliminate) the motivation for an application subsystem builder.

A pragmatic alternative would be a VOFS available on a single platform with sufficient revenue base to be financially attractive for application providers. One case where a high function storage manager is bundled in with the base operating system is the IBM AS/400 [10]. OS/400, the native operating system on the AS/400, includes a storage manager as part of the base operating system. The file system depends upon this storage manager as does a separately purchasable SQL Query Processor, mail system, spreadsheet, and many others. AS/400 applications are all built upon this single storage manager. This strongly supports the claim that were a storage manager included in the base operating system, developers are much more likely to depend upon it than write their own. It is also worth noting that the AS/400 is best known for ease-of-use and low administrative costs. Since all applications

running on an AS/400 share the same storage manager, they are able to achieve uniformity of naming, navigation, non-procedural object query, transaction, security, administration, management, monitoring, quotas, and limits. The Windows NT platform would also, if it had such a VOFS, be a candidate for a single-platform provider of a VOFS system.

In summary, lack of reliable, up-to-date, consistent function with consistent performance across platforms will be an inhibitor to the universal adoption of a VOFS approach. The counter-example is Sun's JAVA, which because it had compelling functionality, is rapidly being deployed on a variety of different platforms. We believe that VOFS could be successful either because it offers sufficiently compelling function that it will rapidly spread across platforms, or it could be embedded into the operating system of a popular platform and therefore application vendors for that platform could count on its presence.

4. ***Performance:***
This is the most commonly used reason for not basing an application on an existing or unified storage manager. Developers will agree that using an existing storage manager rather than writing a proprietary one is the right approach in general if it is a commodity part, has high quality, high reliability, and performs well. However, they are concerned, rightly so, about whether a general purpose storage manager with extra function not needed by their application is going to perform well. Issues cited for performance are specialized tuning, specialized data structures, need for lightweight functionality rather than full function, and lack of requirement for the full mechanisms of the underlying storage mechanism.

Mail systems, for example, often implement multiple mail objects within a single subsystem-provided storage entity (a file) and do the management of mail objects within the mail application. The simple mapping of each message to its own file would perform poorly, take too much disk space (cluster factors much larger than the average message size) and wouldn't scale well. Current mail applications have a design point that is different from either relational databases or file systems. The data is much smaller and is more volatile. Relational databases and file systems count on the fact that creating and deleting tables and files are infrequent and heavy-duty operations. The mail server application designer must, of necessity, build an optimized manager on top of such underlying systems, taking their (non-optimal) performance characteristics and design points into consideration. This argument is more applicable to the server systems whose performance and resource management is critical for the vendor's marketplace. Client systems typically have much more available resources, although they too have high performance requirements for some frequently used operations. The performance issue will certainly be a key issue for deciding whether a VOFS storage subsystem should be used for a given application. We know there are some server-side applications dependent upon real-time data access where general-purpose storage managers remain inadequate. One such application is high-end video and other continuous media severs. For a broad range of storage-based applications, however, it will be sufficient.

In general, performance of a general purpose VOFS storage manager will, at least on the server, be a barrier to adoption wherever the design points of an underlying subsystem are not a good match for the application subsystem. If there is a mismatch, early adopters of this VOFS in highly competitive markets may be less able to adapt to their environment and exploit the next new platform (operating system or hardware) feature compared to more intensive and proprietary investments by other competitors.

To satisfy such needs, the implementers of a VOFS system must do careful design and exploit underlying system features that can enhance performance such as the LPC cited above. Generally, frequently used functions will have much attention paid to performance tuning because it is so visible to all users. Further, as the cost of hardware drops, we have seen less motivation for developers to write in assembler, write their own file managers, and so on. Finally, as the availability and scalability needs of an application grow, the investment required to build such a scalable, highly available proprietary storage system becomes significant and possibly prohibitive as the hardware and operating system environment becomes richer (uniprocessors, SMP, clusters, shared nothing, shared disk, etc.)

5. ***Existence of other viable alternatives:***

A compelling case may be made for a VOFS if the marketplace of potential exploiters is large, due in part to the substantial engineering costs of building or maintaining other alternatives. However, certainly on servers, but to a much lesser extent on clients, there are several alternatives:

A. ***Extensible relational databases:*** Each merchant database vendor has built a data storage sub-system common across his or her supported delivery platforms so they would have little motivation to adopt a different level of platform interface. However, other application vendors who have not already built such a portability layer and who are not as motivated by performance or other competitive differentiation reasons to keep their customized data subsystems, are beginning to use SQL as a common interface. Example products include Information Advantage [13], which provides OLAP support built upon a relational data store. Another newly announced product following this trend is the DB2 OLAP Server [12] provides the Arbor ESSBASE API supported by the DB2 storage subsystem.

   For server-side systems, this is not precisely an argument against VOFS. In fact, VOFS is simply the next step in the evolution of extensible relational database systems to include explicit file system interface support.

B. ***File systems provided by the native platform:*** The argument for simply depending upon the native file system as an alternative to a VOFS is somewhat less compelling, but this solution may suffice for many applications. The concerns with file systems as an alternative to VOFS are the lack of atomicity, query capability, and the low-level of functionality offered. File systems are useful for application developers who do not want or need the full power of a relational database and for certain applications that do not obtain sufficient performance by mapping their techniques to SQL data types and the table/row paradigm.

C. ***Heterogeneous access systems with unified query and transaction management:*** These systems include Oracle Universal Server's DataCartridges [23] where cartridges are foreign data sources whose capabilities extend the application interface functionality to non-relational sources. Similarly IBM Research has the Garlic [7] research project which provides a unified extended SQL query capability across an extensible set of data stores, including relational databases, file systems, the internet, and special purpose repositories such as chemical or human genome databases. In this way, application developers can select to store their data in a relational system, or to keep it in file systems or special purpose databases while retaining the ability to query across the data as if it were in a single store.

Considerable research that has been done over the past 10 years in the area of extensible database management systems. At first glance it would appear that extensible database management systems and integrated storage management have little in common but, upon closer inspection, they are attempting to solve many of the same problems. This work on integrated storage attempts to show that most major server subsystems can be implemented against a single storage manager. Extensible database management system research attempts to show that a single database system can support non-traditional data types and applications such as CAD/CAM, office systems, statistical databases, VLSI design, expert systems, and text applications. In effect, integrated storage and extensible database management systems are really just two different approaches to solving much the same problem: that of storing all server data in a single store. Three fairly well known extensible database research efforts are Postgres [30], Starburst [8], and Exodus [3]. All have published limited research results showing competitive performance. Both Postgres and Starburst have evolved into the commercial products Informix [14] and DB2 [11] respectively, further strengthening the claim that they both work and have acceptable performance.

6. Conclusion

In the discussions above, the arguments against VOFS apply most strongly to the server-based application subsystems. These applications have

made substantial investments in proprietary stores, have the revenue base required to continue to maintain these stores, and the performance requirements to justify escalating investments for some time to come. However, many of these providers, such as Peoplesoft and SAP, have already chosen to depend upon extensible relational databases as their storage infrastructure and to instead focus their engineering investments in functional differentiation at the application level. We argue that the advantages of depending upon a general-purpose data store are here today, some significant applications have already made the switch, and we know of others that are considering such a move.

However, the argument on the client-side is much more obvious. On the client we currently have less advanced capabilities from a storage perspective and a quickly growing need for richer data management functionality such as transaction atomicity, support for disconnected operation, heterogeneous access to non-local data, and replication. For the most part, this investment has not yet occurred, as it already has for many server-side application subsystems. Client data storage requirements are rapidly increasing functionally, as are the requirements for more storage capacity, robustness, and performance. As a result client application subsystem providers will have the opportunity to make implementation choices that may include using a common storage subsystem such as VOFS. The concerns listed in Section 5 above still apply, however several of them are lessened for the following reasons:

1. Client systems typically have much more available resources, although they too have high performance requirements for certain frequently used operations.
2. There are relatively few different client platforms, so the issues of standardization or multi-platform support are not as significant a concern.
3. Control, tuning, and performance remain issues, but client application systems are rarely as driven by these concerns, and would be balanced by enhanced built-in functionality.
4. There are fewer alternatives to a VOFS subsystem on clients, only low-function personal databases or file systems.

Consequently the barriers for client-side application adoption of a unified data store are much less than those for servers, where the arguments in Section 5 are significant.

The argument that we have presented here is that application subsystems should be written against a single integrated storage manager, and that the rationale for not doing so is flawed in most cases. Through examples we have shown that an integrated storage management system supporting a wide range of applications could be built, and would have substantial advantages to both the application developer and, more importantly, the users of these systems. The AS/400, as an example, reinforces our argument that a single uniform storage manager can be built and, if bundled with the operating system, will be used by subsystem designers. Further, there is considerable evidence that the IBM AS/400 system is a much easier system to administer as a result of this decision to exploit a single storage manager.

Client-side applications will need to support disconnected operation, recoverable storage, the ability to change client systems without data loss or administrative work load, be recoverable, support versioning, support atomic update and transactional integrity, be scalable, and be extremely cheap to administer. Further there is a need for uniformity in client-side persistent object naming, security, navigation, and non-procedural query. In addition, client administration costs must be brought under control. So, while we face far more complicated storage management systems driven by these requirements, we also need to simplify client system administration. With this design, we support file system access, relational database access, full text searching and we can extend this store to support structured files and other higher function storage APIs typically built upon file systems such as OLE structured storage. And, in addition to the operational integration, we support integrated query across all data objects and the administration advantages of depending upon a single storage manager.

References:

[1] The Baan Company, http://www.baan.com

[2] Berners-Lee, T., "Information Management: A Proposal", 1989, http://www.w3.org/pub/WWW/History/1989/proposal.html

[3] Carey, M. J., D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vanderberg, "The EXODUS Extensible DBMS Project: An Overview", *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier Eds., Los Altos, CA: Morgan-Kauffman, 1989.

[4] Demers, A., K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch, "The Bayou Architecture: support for Data Sharing among Mobile Users", http://www.parc.xerox.com/csl/projects/bayou/pubs/ba-mcw-94/www/MobileWorkshop_1.html

[5] Lotus Domino: http://domino.lotus.com

[6] Gray, J. "Thesis: Queues are Databases", High Performance Transaction Processing (HPTS) Workshop Position Paper, 1995, http://www.research.microsoft.com/research/barc/gray/QueueIsDB.doc

[7] "The Garlic Project", http://www.almaden.ibm.com/cs/garlic/

[8] Hass, L. M., W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita, "Starburst Mid-Flight: As the Dust Clears", *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[9] Honeyman, P. and L. B. Huston, "Communications Consistency in Mobile File Systems", October 1995, http://www.citi.umich.edu/mobile.html

[10] *IBM Application System/400 Technology*, IBM publication: SA21-9540, 1988.

[11] IBM DB2 Family, http://www.software.ibm.com/data/db2

[12] IBM DB2 OLAP Server: "IBM Teams With Arbor Software to Enter Analytical Processing Software Arena", February 24, 1997. http://www.internet.ibm.com/news/2402.html

[13] Information Advantage Inc, http://www.infoadvan.com

[14] Informix Software, http://www.informix.com

[15] Kistler, J., and M. Satyanarayanan, "Disconnected Operation in the Coda File System", *ACM Transactions on Knowledge and Data Engineering*, March 1990.

[16] "Microsoft Strategy for Reducing cost of Owning PCs", Microsoft, 1996, http://www.microsoft.com/windows/zerowp.htm

[17] "White Paper: Universal Data Access – OLE DB" Microsoft Corp, 1996. http://www.microsoft.com/oledb/prodinfo/wpapers/wpapers.htm

[18] Microsoft Internet Explorer, http://www.microsoft.com/ie

[19] Microsoft Internet Information Server, http://www.microsoft.com/iis

[20] Netscape Communications Corporation, http://www.netscape.com

[21] ObjectStore: Object Design Inc, http://www.odi.com/

[22] Oracle Corp., Applications, http://www.oracle.com/products/applications

[23] "Oracle Developer Programme Network Computing Architecture Questions and Answers", http://www.oracle.com/nca/html/ncaqax.html

[24] PeopleSoft, Inc, http://www.peoplesoft.com

[25] Schaller, B., "The Origin, Nature, and Implications of "MOORE'S LAW"", http://www.research.microsoft.com/research/barc/gray/Moore_Law.html

[26] SAP AG, http://www.sap.com

[27] Satayanarayanan, M., "Fundamental Challenges in Mobile Computing", *Fifteenth ACM Symposium on Principles of Distributed Computing*, May 1996, Philadelphia, PA.

[28] Satayanarayanan, M., "Mobile Information Access", IEEE Personal Communications", Vol.3, No.1, February 1996.

[29] Satayanarayanan, M., "Coda: A Highly Available File System for a Distributed Workstation Environment", Proceedings of the Second IEEE Workshop on Workstation Operating Systems, Sep. 1989, Pacific Grove, CA.

[30] Stonebraker, M. and L.A. Rowe, "The Design of POSTGRES", Proceedings of ACM SIGMOD Conference, 1986.

CONTROL: Providing Impossibly Good Performance for Data-Intensive Applications

Joe Hellerstein
U.C. Berkeley
jmh@cs.berkeley.edu

August 8, 1997

1 Introduction

Two factors are converging to lead users to ask the impossible of data-intensive applications:

1. Dataset sizes are continually growing, and despite hardware advances data-intensive operations remain a leading source of wait time for users.
2. Advances in user interfaces are conspiring to make databases look slow:
   - PCs and the web have put computers directly into the hands of important but computer-naive users, who often want broad views of large datasets.
   - These demanding users expect "speed of thought" response time from mass data analysis tools, as they have gotten in the past from spreadsheets and similar programs.

There are two common techniques for handling "impossible" requests in today's data-intensive systems: precomputation and failure. If answers have been precomputed, they can be provided quickly. Since the answers to all queries cannot be precomputed, current systems either disallow some operations (MOLAP, spreadsheets and other GUIs) or have uselessly slow performance for them (ROLAP, SQL, etc.) These operations essentially require performance that is impossible for today's systems.

In the CONTROL project at Berkeley, we are developing techniques for *Continuous Output and Navigation Technology with Refinement On-Line*, to provide a viable third approach for handling data-intensive operations in an interactive fashion. We combine new approaches for data delivery with statistical estimation techniques, to provide progressively refining estimations for data-intensive processes. In addition to estimation, our techniques allow users to control processing online, at various levels of granularity. In essence, we are bringing user demands and computational possibilities into synch, by developing viable techniques in data processing and statistical estimation.

2 Example Applications, Present and Future

2.1 Online Aggregation, OLAP and Data Mining

We have implemented a number of these ideas in the context of "Online Aggregation" queries in POSTGRES and Informix Universal Server. Online aggregation provides progressively refining estimations to SQL aggregates and GROUP BY queries. An example online aggregation interface appears on the left of Figure 1, for a query to find the average GPA per major. As a student in each major is fetched from the

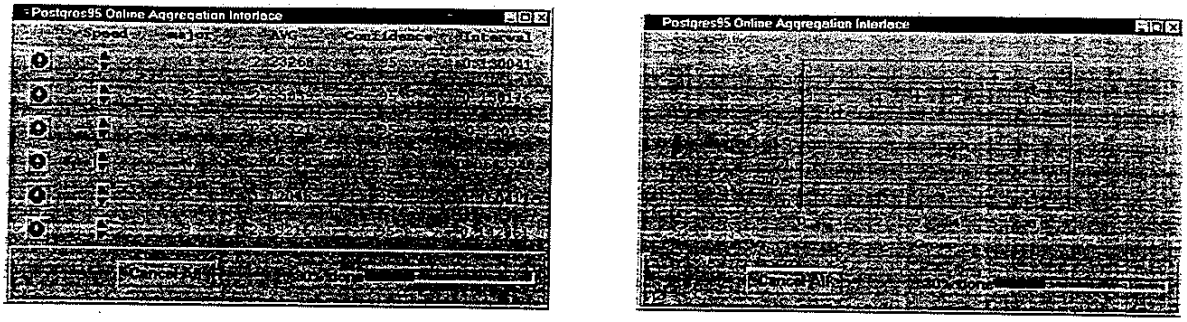


Figure 1: Two Alternate Multi-Group Online Aggregation Interfaces

database, a row is added to the UI for that major, containing the major name, the current estimate of the average GPA for the major, and a *confidence interval*. As additional students from that major are fetched, the estimate for the major refines and the confidence interval shrinks. Users can stop the processing on a row-by-row basis, and can control the relative speed at which the different groups' estimates are refined. An alternative interface appears on the right of Figure 1, in which the confidence intervals are presented as "error bars", which move and shrink as processing proceeds.

A natural extension of this work is for OLAP applications. MOLAP systems precompute all aggregates at all available levels of drill-down and roll-up. This amount of precomputation breaks down completely at 50Gb or so, and is problematic and expensive even for much smaller datasets. ROLAP systems scale better, but provide no guarantees on query performance, and typically don't have a truly interactive flavor. Integrating online aggregation into ROLAP solves the problem – users can slice and dice in a completely ad-hoc fashion (since answers need not be precomputed), and get immediate estimates to guide their navigation of the dataset. Moreover, the techniques used to support the speed buttons of Figure 1 can also allow the online ROLAP system to tune the running behavior to the users current focus: e.g. as a user drills down into a particular "sub-cube", the estimates for that sub-cube can be refined faster than those for other parts of the overall dataset.

Similarly, data mining applications are essentially long-running queries (or sequences of queries) that extract aggregated information from large datasets. They typically run for hours before producing any results. The techniques used in online aggregation are natural for these applications. We are currently developing an online association rules miner, which allows users to adjust threshholds on the fly, and focus effort onto particular rules of interest while the mining is under way.

2.2 PROponents: Progressively Refining Online GUI Components

Many supposedly interactive applications like spreadsheets become batch applications when faced with large datasets; the problem is exacerbated by our expectation of interactive behavior from GUI widgets. Programmers of GUI widgets are used to handling tiny datasets. As a result, they write the widgets to operate in batch mode, updating the screen only when they complete operation. This becomes extremely frustrating when the underlying operations involve large amounts of data.

Consider the common "list box" widget — it allows the user to bring up a list and scroll through it, or jump to particular points by typing prefixes of words in the list. Now imagine implementing a list box over gigabytes of unsorted, unindexed data resulting from an *ad hoc* query. This is likely to be rather unpleasant to use. Similarly unpleasant behavior arises when sorting, grouping, recalculating or cross-tabulating over large datasets — activites which are usually interactive in spreadsheets. Today's desktop applications carefully, almost imperceptibly discourage these behaviors over large unindexed datasets. There are many scenarios where such behavior is desirable, however; in such cases unwieldy database software (e.g., OLAP and Data Warehousing) is employed, requiring batch performance and lots of disk space to set up.

Many development environments and toolkits today include libraries of GUI components (widgets). We are working to develop progressively refining online components (PROponents) to let programmers easily develop GUIs for data-intensive applications. In some instances, online GUI components could present different interfaces than standard widgets, in order to represent the state of ongoing processing — a trivial example is the "heartbeat" given by many web browsers as they present text and images online. In other instances status information might be unnecessary. This nexus of user interfaces and data processing appears to be natural both to implement and to use.

2.3 Online Data Visualization

Data visualization is an increasingly important problem, as users desire interactive yet efficient modes for analyzing data. An inherent challenge in architecting a data visualization system is that it must present large volumes of information efficiently. This involves scanning, aggregating and rendering large datasets at point-and-click speeds. Typically these visualization systems do not draw a new screen until its image has been fully computed. Once again, this means batch-style performance for large datasets. This is particularly painful for visualization systems that are expressly intended to support browsing of large datasets.

A natural solution is to extend a visualization system to draw objects as soon as they are fetched from the database. For example, as soon as a tuple is fetched, a corresponding point can be plotted on a map or graph. An more complex alternative we are exploring is to combine this incremental, sampling-like access with network data encoding. We model the final state of the screen as a *single aggregate object* to be estimated. After scanning a number of tuples, we use them as a sample to estimate the first few coefficients of a wavelet encoding of the final image. As more tuples are scanned, this estimate is refined. The user sees the picture improve much the way that images become refined during network transmission. This may be particularly useful when a user pans or zooms on a canvas, when the accuracy of what is seen is not as important as the rough outlines of the moving picture.

3 The Economic Aspect

The transaction processing community retains a focus on large-scale, batch solutions to data processing. This is reflected in the TPC benchmarks, which reward the fastest system in a given price bracket. In Berkeley tradition, the CONTROL approach subverts this metric to bring data analysis to the masses.

Many TPC-D-style decision-support queries require only very rough estimates to satisfy users. A rule of thumb in the statistical community is that 20 observations (only 20 tuples!!) is all that is needed to begin providing a reasonable estimate. Queries that require only 20 tuples can be run on clients costing only dozens of dollars! However, because users sometimes do require complete answers (or think they do until they see their estimates converging), it probably makes sense to invest a bit more money in a client that can process a query to completion, albeit slowly. Caveats aside, the point remains: *the economic savings available via estimation is enormous, and is not reflected in benchmarking models like those of TPC-D.* In future, large-scale data analysis benchmarks must incorporate not only cost and time to completion, but also accuracy over time. If current vendors ignore this issue, the online estimators will soon catch them unawares.

4 Conclusion

CONTROL is demonstrating the viability of progressively refining online processing of large datasets. We are placing our basic techniques in the service of a number of applications, both on a large scale (online aggregation in Informix) and a small scale (PROponents). We maximize feedback, control and ease of use at the front end, using a healthy mix of new data processing and statistical techniques. We do not require any statistical sophistication from users. We believe that the CONTROL ideas will prove to be critical components of affordable, usable solutions for running "impossible" queries.

The Shock of IRAM: Will Information Systems Be Ready for the Next Chip Technology?

Joe Hellerstein
U.C. Berkeley
jmh@cs.berkeley.edu

Based on current technology trends, some computer architects are predicting a new generation of chips on the horizon: *Intelligent RAM* (IRAM) [1]. IRAM will combine RAM and CPU on a single chip. This could fundamentally shift the hardware bottlenecks we have carefully tuned for over the last 15 years. In particular, IRAM will provide two features of interest for data-intensive systems:

1. **High bandwidth I/O-to-RAM transfers:** I/O devices (disk, network) will be connected by individual serial lines, rather than by sharing an I/O bus. Patterson predicts that this will bring 10-40x improvements in I/O bandwidth [2]. Disk latency will stay about the same. A corrolary is that disk seek will become increasingly harmful.
2. **High bandwidth, low latency RAM-to-register transfers:** Essentially, 96 MB of RAM will behave like what we now think of as a Level-1 cache. Patterson predicts 100x improvement in memory bandwidth (over DRAM) and 10x reduction in memory latency [2]. A simple fallout is that cache-conscious algorithms as we think of them today may not be beneficial.

If IRAM is indeed the RISC of the '00s, then systems builders have some thinking to do. This suggests numerous modifications to today's systems, ranging from modest to radical. I include a sampling here.

- No-seek, elevator disk scheduling may make sense.
- Alternatively, giant block sizes may make sense.
- Reading an entire index into memory may be cheaper than traversing it.
- Nested loops join may be very fast.
- It will be cheap and increasingly beneficial to reorganize relations, possibly on the fly.
- Replacement selection may re-emerge as the sort of choice.
- NUMA to other processors' RAM will be far better than disk seeks.
- CPU-intensive processing (e.g. in ORDBMS) will be increasingly detrimental if it blocks the data pipeline.
- Lock granularity may be at odds with I/O transfer granularity.
- All bets are off for query optimization, resource allocation, and admission control.

A more fundamental question is lurking beneath the surface – regardless of exactly how the bottlenecks shift, should we expect to rearchitect our software systems over the next 10 years? Are we prepared to do so? We are currently in a phase of software systems entrenchment, with the computer architects trying to serve our needs. Is this likely to last, and is it wise?

References

[1] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent DRAM: IRAM. To appear, IEEE Micro, April 1997.

[2] David Patterson. Intelligent RAM (IRAM): Chips that remember and compute (Revised). Presented at the Industrial Liason Program (ILP) meeting, March 12, 1997 at U.C. Berkeley and at the DARPA Embbedded Systems Principle Investigator meeting on March 21, 1997 in Santa Fe, New Mexico. http://HTTP.CS.Berkeley.EDU/~pattrsn/talks/IRAM.DARPA.ps.

Will non-determinism be a gating factor in Web transactions?

Tony Hey
Professor, Department of Electronics & Computer Science, University of Southampton

Charles Brett
President, C3B Consulting Ltd.

Those wishing to conduct transactions over the Internet – specifically using the Web – tend to presume that the deterministic properties associated with conventional transaction processing are carried into the Internet. Yet the vagaries of network traffic through the Web produce a non-deterministic situation at the database. It may no longer be possible to guarantee that the same input will result in the same output. The issues are:

- will this limit the transactions that can be carried out?
- does the use of the Web as a front-end for transaction processing drive enterprises to new levels of risk exposure?

In conventional TP systems, great care is taken to ensure that changes to the underlying database obey the ACID principles, and in particular that they are repeatable - the same requested change will produce the same results each time. With multiple concurrent updates in progress, locking mechanisms are employed to ensure that different pieces of work do not interfere - a whole Unit of Work (UoW) either completes or is backed out. Access via dedicated terminal networks ensures that there is little delay between starting and finishing a piece of work. The first user to start a transaction against a database item will be the first to get the lock and complete his UoW. Locking assumes that things happen fast - the lock will not be held for long enough seriously to inconvenience others.

Access via the Web leads means unpredictable delays in network times, which may also be quite lengthy. The first person to enter a request to update data may or may not be the first to get the lock at the database. Another user accessing the data from a less congested part of the Web may well submit a later request which arrives first. However, on a different day, the order might be reversed.

We have, therefore, a non-deterministic situation where the outcome of a number of transactions entered into the database may vary in an unpredictable manner, due to the nature of the access network.

If the database is distributed, the problem is made worse. The synchronous approach - of using two-phase commit to ensure tables in several locations are updated more or less simultaneously - cannot be employed using the Web for the reasons given above; so replication will have to be used. However, the strategy of having a primary site, which is the only place for updating a given data item, and then propagating updates to other sites also has its problems. Access to the primary site is unpredictable (across the Web), and there may be several primary sites involved for the different data items involved in one UoW. Introducing long running transactions will also, likely, complicate matters further.

Given this, how is the global enterprise, which wishes customers to place orders via the Web, and which has several mirror or replicated sites around the world, to proceed? Is the only option to:

- allow updates to any local database, with reconciliation later
- accept a much larger degree of risk that some orders that have been committed to customers that cannot be satisfied?

We believe that the implicitly non-deterministic nature of these situations means that stochastic techniques may need to be employed to decide whether to fulfil an order or not. However, while this might be acceptable to a company taking orders for high-volume, low-value items, it is questionable whether financial securities could be traded on this basis. The use of the Web as a front-end may therefore be severely limited.

Asked in broader terms, the issue is whether the unpredictable scale of Internet delays may lead to non-deterministic results which will open the door to the Jeremiah's who want to question the ability of the Internet to support businesses and business transactions. Understanding whether non-determinism will (or will not) become a gating factor to the introduction and acceptance of Web transactions may yet become a significant contention point.

Antony Hey

Professor, Department of Electronics & Computer Science,
University of Southampton, Southampton, S017 1BJ, UK.
Tel: +44 1703 592749; Fax: +44 1703 592978
Email: ajgh@ecs.soton.ac.uk

Charles Brett

President, C3B Consulting Ltd.
St Swithun's Gate, Kingsgate Road, Winchester, SO2, UK
Tel: (+44) 1962 878333; Fax: (+44) 1962 878334
Email: spectrum@middlewarespectra.com; WWW: www. middlewarespectra.com

TPMonitors - gone, but not forgotten ?

Pete Homan
BEA Systems, Inc.
Sunnyvale, CA 94089
Tel(408)-743-4006
e-mail:pete.homan@beasys.com

Mark Carges
BEA Systems, Inc.
Sunnyvale, CA 94089
Tel(408)-542-4157
e-mail:mark.carges@beasys.com

The demise of TP monitors has long been predicted, including at previous HPTS meetings. There are people who claim they aren't needed - the 2tier client server vendors, who are now busily building TP monitor equivalents, having invented/discovered 3tier computing. Object systems, publish and subscribe systems, mom systems, and others all claim to be the enterprise middleware solution of the future.

How will we know which one is the winner, when TP monitors are gone, not just going?? The requirements of an enterprise production system that TP systems address haven't changed, new requirements have been added.

What's really going on is that TP monitor technology is being adapted and incorporated into lots of systems, from ORBs to Web Servers. A Gartner Group prediction, fast becoming the conventional wisdom, applies the label Object Transaction Manager (OTMs), to the systems that are the result of that evolution. What is an OTM?

As an enterprise middleware vendor and owner of TP monitor, MOM and distributed object technologies, BEA believes it can explain why it can be a successful provider of the next generation of enterprise middleware.

We would be interested to compare and contrast our view of the OTM future with other contestants in what clearly will be an interesting and important market.

We suggest this would make a suitable topic for a moderated panel discussion.

WorldFlow : A System For Building Global Transactional Workflows

(Extended Abstract)

Mohan Kamath & Krithi Ramamritham
Database Systems Lab
Department of Computer Science
University of Massachusetts
Amherst MA 01003

Narain Gehani & Daniel Lieuwen
Database Systems Research Department
Lucent Technologies / Bell Labs
700 Mountain Avenue
Murray Hill NJ 07974

Abstract

To meet the requirements of emerging electronic commerce applications there is a need to integrate aspects from Workflow Management, World Wide Web, Transaction Management, and Warehousing technology. We have designed and implemented WorldFlow for this purpose. It is a system for building transactional workflows that access Web based and non-Web based information sources. This paper introduces the WorldFlow system and highlights some of the issues that arise in implementing it. It also presents the architecture of WorldFlow and briefly discusses its implementation.

Contents:

1. Introduction
2. WorldFlow Overview
3. Problems in Implementing Workflows on the Web and Solutions
    1. Problems
        1. Network/Server Failures
        2. Efficiency
    2. Solutions
        1. Handling network/server failure
        2. Improving Efficiency
4. WorldFlow System Architecture & Implementation
5. Summary
6. References

(1) Introduction

The *World Wide Web* has been growing at a rapid rate to become a global information source. Many manufacturers, stores, banks, financial consultants, law firms and government offices publish information about their products and services - and even sell them - through the Web. Numerous studies indicate that the volume of electronic commerce on the Web will sharply increase. Thus, most information sources in a company will be Web accessible - internally for various groups within the company and externally for business partners and customers. To meet the requirements of emerging electronic commerce applications there is a need to integrate aspects from Workflow Management, World Wide Web, Transaction Management, and Warehousing technology. We will motivate this with some examples.

Consider the formation of a *virtual enterprise* which will make it possible to implement JIT (Just In Time) method of inventory control to save money on inventory, warehousing and handling costs. A manufacturing company can place orders with suppliers for the individual components needed to make the product immediately after it receives an order for the product. The suppliers use Web servers linked to their backend databases to process online orders. With such a system in place, the global internet can be effectively used to process orders round the clock and deliver the products in a timely manner while reducing the inventory cost. This example illustrates the benefits of automated business processes on the Web by allowing steps to automatically query/update information from

Web servers located both within and outside the organization. Such processes cannot be modeled using traditional transactions [4]. They require the use of extended transaction models or transactional workflows [8]. These in turn consists of a number of individual internet transactions.

Next let us consider the operation of an electronic brokerage service. Such services are essentially data warehouses that use both the ``push" and ``pull" methods for obtaining information. The stored information (for example catalogs) is obtained in advance (``push") while the rest (for example price and availability) is obtained dynamically (``pull") as the need arises. The brokerage services allow customers to browse through statically stored / dynamically fetched information and then issue transactions. To process customer transactions, the brokerage service might have to contact a bank to get payment and then place orders with the individual supplier(s) for the required items. Here again there is a need for transactional workflows. Transactional workflows can also be used by the brokerages (warehouses) for obtaining static/dynamic information. Examples of electronic brokerage services include shopping services on the Web and stock brokerages for trading stocks from around the globe.

The above two motivating examples illustrate the utility of a system that can be used to develop global transactional workflows. Hence we have developed and implemented the *WorldFlow* system. This paper discusses the key issue that arises in implementing WorldFlow, namely dealing with the unreliability of the Web in the form of server/network failures and providing efficiency. Then we present the architecture of *WorldFlow* and discuss its implementation. The rest of this paper is organized as follows. A brief introduction to *WorldFlow* is provided in Section 2. Section 3 discusses problems related to implementing workflows on the Web and discusses some specific solutions. Section 4 presents the architecture for *WorldFlow* and discusses its implementation. Section 5 concludes the paper with a summary.

---

(2) WorldFlow Overview

The basic principles underlying *WorldFlow* originate from workflow management. In this section we provide a brief overview of *WorldFlow*.

Conventional WFMSs consist mainly of a workflow engine, application agents and tools for modeling, administration and monitoring. The workflow engine and the tools communicate with a workflow database (WFDB) to store and update workflow relevant data. A workflow schema (workflow definition) is provided by a workflow designer with the aid of the modeling tool. The workflow schema can be used to instantiate several workflows. A workflow schema is essentially a directed graph with nodes representing the steps to be performed. The schema also specifies how data flows between the steps. Execution of a step usually corresponds to the invocation of a specified application program at a particular agent. At execution time, the agent is provided with the required inputs by the workflow engine and after the execution of the step, the output of the step is returned by the agent to the workflow engine. In case of failures, the WFDB facilitates forward recovery *i.e.*, each workflow is continued from the state it was in before the failure occurred rather than starting from the first step.

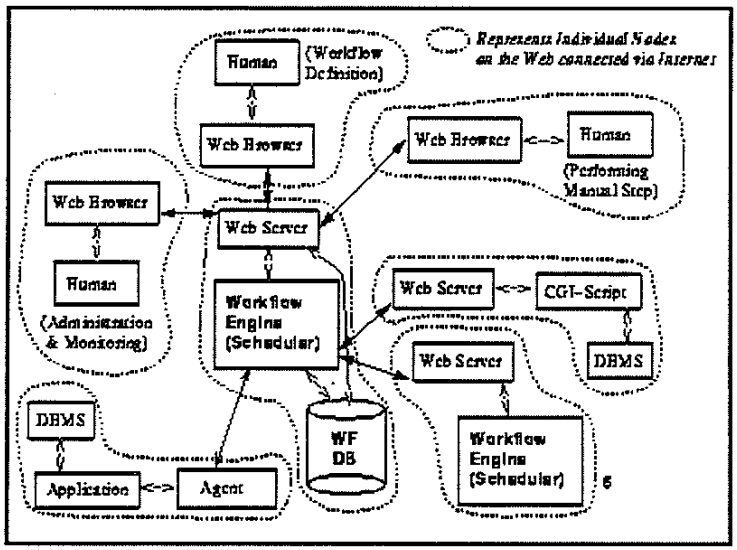


Figure 1: *Workflow Management incorporating the Web*

The Web is essentially a client-server based distributed system where browsers acting as clients communicate with Web servers using the HTTP protocol [11]. As shown in *Figure 1*, in *WorldFlow* Web servers can play the role of agents and the CGI-scripts (that often execute *transactions* against a backend database) can play the role of programs. Note that the *WorldFlow* engine is now a client of the Web servers. To facilitate inter-organizational workflows (where several steps are to be performed in a different organization rather than just accessing some information), a *WorldFlow* engine can transfer control of a workflow to another *WorldFlow* engine via a Web server. This paper focuses mainly on the interaction between the workflow engine and the Web servers, *i.e.*, business-to-business transactions.

Since customers and businesses want prompt service for their orders, it is important to ensure that workflows do not stall and make timely progress during their execution. It should be possible to track the status of a specific workflow that is in progress. Although our design incorporates a centralized scheduler (as it *simplifies tracking and ensuring progress* of workflows), multiple such schedulers can be used in parallel for load balancing and scalability within an organization. Transfer of control between workflow engines also ensures scalability of complicated inter-organizational workflows.

Our work assumes that a Web server assigned for each step will provide proper interfaces for performing the various functions (*e.g.*, placing a request, enquiring about the status of a request, canceling a request, modifying a request). The Web interfaces access data from backend DBMSs using a CGI script or some other access scheme (*e.g.*, the direct interface between HTML and SQL described in [7]). Each step in a workflow accesses data from at most one Web server.

---

(3) Problems in Implementing Workflows on the Web & Solutions

In this section, we briefly enumerate the problems that arise when we implement *WorldFlow* on the Web. We also provide outlines of specific solutions.

(3.1) Problems

(3.1.1) Network/Server Failures

- *Unreliability of the Web - Network Failure, Server Overloading:*
  The Web has been growing at an enormous rate and so has the traffic. Due to congestion, the *network can fail*, it may be impossible to establish a connection to a site. Even if a connection is established, the *delays in the network* sometimes cause timeout at the client. Web servers can also suffer from overloading resulting in the refusal of some connections. As shown in Figure 2, a client will not be able to determine whether the request (*i.e.*, the remote transaction) was processed or not. Such failures also pose problems while transferring workflow control over the Web from one workflow engine to another.

- *Web Server Failure:*
  If a Web server fails, some requests may never be completed and their responses never sent. Clients that sent those requests will timeout. A Web server processing updates can fail before a transaction is committed or after a transaction is committed but before the results are returned to the client. Even in this case, a client will not be able to determine whether the request was processed or not. A resubmission of the request can result in duplicate orders.

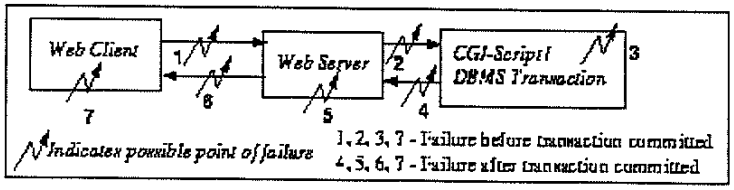


Figure 2: *Failure of Update request on the Web*

(3.1.2) Efficiency

To avoid losing potential business from customers, it is necessary for the system to be able to handle/manage all orders. Large organizations using workflows to provide and manage popular services will experience enormous loads - instantiating a few thousands

workflows per day would not be uncommon. The load on an organization's internet gateway will also be high and hence it is essential to improve efficiency by reducing the number of connections and the number of messages exchanged.

(3.2) Solutions

(3.2.1) Handling Network/Server failure

Since the *WorldFlow* engine is a client of the Web servers, the failure of the network, large delays, and Web server failure all mean the same since all problems cause connection timeouts. All that matters is whether the failure occurred before a request was processed or after a request was processed but before the response arrived. This is crucial for update steps (*e.g.*, steps ordering items from suppliers) of the workflow since each must be performed exactly once.

To ensure that each update operation is executed only once, a request has to be performed in an idempotent fashion. Web servers are stateless, and hence they cannot support idempotent operations, *i.e.*,they do not have the capability to discriminate between a new request and a duplicate request. Hence, it is necessary to have some way of testing (determining) whether the program was successfully executed or not at the remote server. If the state of the operation cannot be tested at the remote server, then it is not possible to achieve the ``exactly once" semantics. Only after determining where the request failed, can appropriate action be taken.

To handle requests idempotently on the Web, we have developed the EONS (Exactly ONce Semantics) protocol. By assigning a unique ID to each request (analogous to a purchase order number) before the request is made it can be ensured that each request is processed exactly once in spite of failures. The EONS protocol is quite similar to standard networking retransmission schemes like *Stop-and-Wait* and *Go-Back-n* [3] where the networking software uses counters to ensure that each packet is released to higher levels of software exactly once. The main difference is that the EONS protocol requires that this exactly-once guarantee hold even if the connection between the two parties is broken for an arbitrarily long period of time. Consequently, it requires persistent storage of request information (like the unique ID), while the networking protocols do not.

(3.2.2) Improving Efficiency

The solution we propose for improving efficiency makes use of dynamic batching and utilizes enhancements proposed to the HTTP protocol [11] that support persistent connections , *i.e.*, several URLs at a Web server can be accessed using a single connection. The workflow engine can batch steps from concurrent workflows that need to fetch or update data from the same Web server. Requests to Web servers are batched for a threshold batching period and are handled using a single persistent connection. Thus, significant savings can be achieved by avoiding several connection setup and tear down. An example is shown in Figure 3 where steps A, B, C and D are batched. The batching interval is a tradeoff and hence different types of batching schemes can be devised.

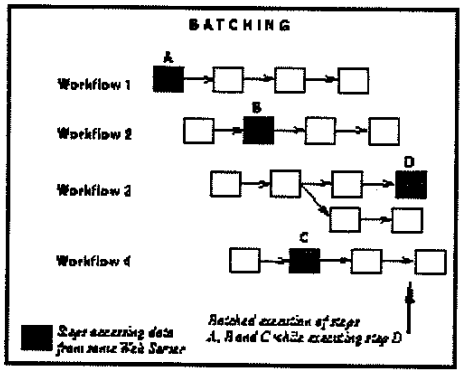


Figure 3: *Batching of Steps in Web Workflows*

The batching schemes will benefit a lot from emerging technologies that enable the implementation of *servlets* [1]. Servlets can be customized to process more complex and even multiple queries/updates in a single request. The batching scheme can then group multiple queries/updates and ship them in a single request to the servlet which in turn can process the request and send back a single reply containing the results of all the queries/updates.

(4) WorldFlow System Architecture & Implementation

The architecture we have developed for *WorldFlow* is shown in Figure 4. Other than the standard WFMS components, it also contains a Web module which is responsible for handling steps that need to access Web servers. It uses the information provided by the workflow schema for extracting the required output from the HTML file returned by the Web server and passes the results to the workflow engine. The Web module is responsible for enforcing the EONS protocol, and the workflow engine is responsible for batching. The Web module is also responsible for managing the persistent connections to remote Web servers while performing batching of requests.

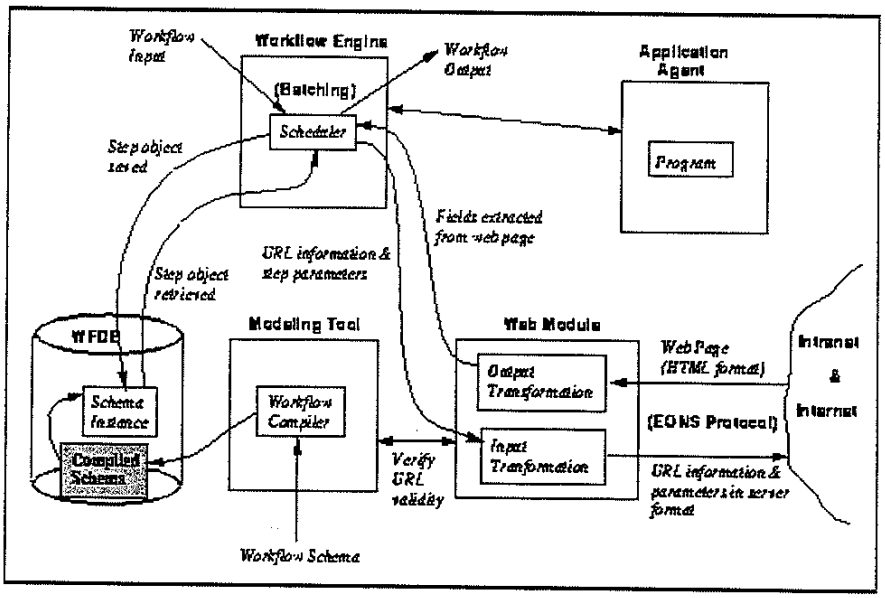


Figure 4: *WorldFlow* Architecture

We have implemeneted a prototype of *WorldFlow* by extending the prototype workflow management system we have developed to study failure handling schemes for workflows [6]. This prototype supports transactional workflows and has been completely implemented in Java and is portable across heterogeneous architectures. The workflow modeling language and the compiler were enhanced to handle URL specifications and the Web module was added to the run-time support. The EONS protocol is being implemented. Batching of requests will be implemented when the enabling technologies are available, *i.e.*, web servers using HTTP 1.1 and servlet development tools.

Using *WorldFlow* we have developed a sample brokerage service application called *StockAdvisor*. It is a tool for making smarter stock investment decisions. By selecting a company of interest, a user can get (i) the company's corporate profile information, (ii) stock performance of that company and other major players in the same sector and (iii) stock performance across all different sectors. This is achieved by accessing some local databases as well as other Web servers that provide corporate profile information and near real-time stock quotes. The EONS protocol will be used to support simulated trading.

(5) Summary

To meet the needs of emerging electronic commerce applications there is a need for systems that can support global transactional workflows. To address this need, we have developed and implemented *WorldFlow*, a workflow management system that can directly access local database and online information available on the Web. In this paper we addressed issues relevant to implementing transactional workflows on the Web. Specifically we discussed problems arising from network/server failures and also focused on efficiency.

Electronic brokerages where numerous value-added services are offered by a single agency with the ability to track and account for each order placed by a customer are becoming popular [5]. Much effort is being dedicated to the development of Electronic Data Interchange (EDI) standards [5], Business APIs (*e.g.*, SAP R/3 [2]) and Web servers to facilitate electronic commerce on the internet. These trends indicate the importance of our work in developing *WorldFlow* as a tool to build global cooperative information systems

for conducting business over the web. The Meteor2 project [10]has implemented workflows using the Web infrastructure, but they address a different set of issues. Their focus is more on human assisted business-to-business transactions as opposed to automated ones. While we provide a general infrastructure to develop workflows that access Web information sources, they have developed a very specific solution by embedding workflow related information in CGI scripts and executing them in a cascaded fashion. This can make tracking and failure handling difficult.

Worlkflows can be used as a tool in data warehousing environments for data collection from various sites [9]. Companies can have some of their data warehouse information accessible to business partners via web servers. Thus *WorldFlow* can be used as a tool for building global data warehouses by connecting the Web information sources in a desired manner.

References

1 Java servlet API, JavaSoft Document

2 *SAP R/3 System 3.1: The Foundations for Genuine Business on the Internet*, SAP Inc.

3 D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1992.

4 J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

5 R. Kalakota and A. B. Whinston. *Frontiers of Electronic Commerce*. Addison Wesley, 1996.

6 M. Kamath and K. Ramamritham. Failure Handling and Synchronization of Workflows in Parallel and Distributed Workflow Environments. Technical Report In preparation, University of Massachusetts, Computer Science Dept. (submitted for publication), 1997.

7 T. Nguyen and V. Srinivasan. Accessing Relational Databases from the World Wide Web. In *Proc. of ACM SIGMOD*, pages 529-540, Montreal, Canada, 1996.

8 M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond. W. Kim, Ed*, Addison Wesley, 1994.

9 S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, 26(1), March 1997.

10 A. Sheth et al. Supporting State-Wide immunization Tracking using Multi-Paradigm Workflow Technology. In *Proceedings of the 22nd VLDB Conference*, Bombay, India, 1996.

11 T. Berners-Lee and R. Fielding and H. Frystyk, editor. *Hypertext Transfer Protocol - HTTP/1.0*, Internet Engineering Task Force - HTTP Working Group document.

If you have any comments or questions please mail them to kamath@cs.umass.edu

Benchmarking Database Application Systems

Position Paper for HPTS 1997

Alfons Kemper Donald Kossmann

Universität Passau
Fakultät für Mathematik und Informatik
D-94030 Passau, Germany

⟨*last name*⟩@db.fmi.uni-passau.de
http://www.db.fmi.uni-passau.de/projects/SAP

Introduction

Today and in the past, the performance of *isolated* database management systems has been measured. To find out which is the fastest or "fastest per cost" database product, standard benchmarks such as TPC-C [Gra93] or TPC-D [TPC95] have been implemented and performed on alternative database systems. We believe that in future, it is going to become increasingly important to measure the performance of database systems together with popular database application systems. To find out which is the fastest or "fastest per cost" database product for a particular application system, a (standard) benchmark will be implemented using the application system and the benchmark will be performed using alternative database systems as the application system's backend.

Why is such an approach to benchmark the whole package consisting of a database and an application system important? It is obviously important for vendors of application systems and end users. Vendors of application systems, of course, need to know which database system works best for their system, and end users are only interested in the whole package so that "application-DBMS" benchmark results are more relevant to end users than the benchmark results of isolated database products published today. We think, however, that the vendors of database systems are also going to benefit significantly from benchmarking their products together with certain application systems: it is currently very difficult for database system vendors to market their products; if a database system vendor finds out that its product works particularly well with a certain application system, the database system vendor can specifically address the customers of that application system in order to sell its product.

Why is now the moment to start with such "application-DBMS" benchmarking? The answer to this question is that only two recent trends have made it possible and worthwhile to measure application systems and database systems together. First, some application systems used to have their own internal database management component. These systems have recently or are currently being reengineered so that they use a commercial (relational) database product as a backend for storage management. As a result, these

application systems can only now benefit from improvements to commercial database systems, and only now is it possible to compare different database products with such an application system in order to find out which one works best. Second, in many important areas, there are only few application systems that hold a significant market share; for example, SAP R/3 dominates the market for business administration systems. As a result, it is possible to carry out "application-DBMS" benchmarks that are relevant for a large number of end users with reasonable effort (i.e., considering only one or two application systems).

As an initial step towards benchmarking database systems in conjunction with application systems, we have carried out the TPC-D benchmark using SAP R/3 and a commercial relational database system. In the remainder of this position paper, we will briefly summarize the results of this study, and we will also discuss more explicitly why it is important and possible to measure database system performance in this way. A detailed description of the results of our performance experiments with SAP R/3 can be found in [DHKK97].

Benchmarking SAP R/3 and A Commercial RDBMS

As stated in the introduction, SAP is the market leader for business administration systems; it is, for example, used by several Fortune 500 companies and to an increasing extent also by smaller companies [AG]. SAP R/3 integrates all business processes of a company and provides modules for finance, material management, sales and distribution etc. SAP R/3 uses a conventional relational database system as backend, and it is possible to choose among several commercial systems when installing SAP R/3 (e.g., ADABAS, DB2, Informix, Oracle, SQL Server). The database system manages the SAP database which stores all business data (e.g., customer and supplier information, orders, ...) and all of SAP R/3-internal control data.

In the following, we will present the results of running the TPC-D power test [TPC95] on SAP R/3.[1] To run the TPC-D power test, we first loaded the records of the standard TPC-D benchmark into our SAP database; this was possible because SAP has pre-defined tables to store orders, customer data etc. and SAP R/3 offers a batch-input facility to load records from ASCII files. Then, we implemented the 17 queries and 2 update functions of the benchmark using SAP's query interfaces. For comparison, we also implemented the TPC-D benchmark directly on a commercial RDBMS, the same RDBMS we used as SAP R/3's backend in our installation.[2]

In all experiments reported here, we used the latest Release of SAP R/3 (Version 3.0E), and we carried out the experiments on a Sun SPARC station 20/612MP with two 60 MHz microprocessors, 256 MB main memory, and five 4 GB Seagate ST15230N disk drives. We used a TPC-D database with scaling factor SF=0.2 and tried to be as much as possible compliant with the standard specification of the benchmark. (We could not be fully TPC-D compliant because we were not able to load a database with a larger scaling factor, and SAP strongly partitions the TPC-D data, as we will see below.) Further experiments and more details on SAP R/3 and our implementation can be found in [DHKK97], and

[1] It would probably be also possible to run the TPC-C benchmark on SAP R/3. We have not tried this yet.

[2] Because of our license agreement, we are not allowed to reveal the name and details of the RDBMS.

the source code of all programs used in the experiments can be retrieved from our web pages [DHKK].

Schema, Database Size, and Loading Time

We will first compare the "original" TPC-D database, produced implementing the benchmark directly on an RDBMS, with the SAP database. The original TPC-D benchmark has eight tables (region, nation, supplier, part, partsupp, customer, order, and lineitem). The SAP database in our *vanilla* configuration, on the other hand, has more than 10,000 tables – 17 of which are actually used to store data used in the TPC-D benchmark. SAP has so many tables and strongly partitions the TPC-D data because it provides a great deal of functionality which is not tested in the TPC-D benchmark; for example, SAP R/3 can be used by multi-national enterprises which makes it necessary to provide special tables to store explanatory text in different languages at the same time.

The second observation we can make when comparing the original TPCD-DB and the SAP-DB is that the SAP-DB is about 10 times as large as the original TPCD-DB after loading all the records of the TPC-D benchmark with scaling factor 0.2; this is shown in Table 1. Again, this effect can be explained because SAP R/3's is a very powerful, general-purpose business administration system which requires having additional non-TPC-D fields in every table.[3] Another striking reason for this effect is that SAP R/3 uses 16-byte strings rather than 4-byte integers to represent key (and foreign key) attributes.

	Original TPCD-DB		SAP-DB	
	Data	Indexes	Data	Indexes
REGION	16	0	320	400
NATION	16	0	400	400
SUPPLIER	451	120	2.127	1.884
PART	6.144	1.792	79.485	83.525
PARTSUPP	32.310	5.275	102.045	44.455
CUSTOMER	7.929	1.463	37.805	26.355
ORDER	52.578	21.312	399.190	125.243
LINEITEM	171.704	72.860	2.191.844	558.746
Total	271.139	102.822	2.813.216	841.008

Table 1: DB Sizes in KB: Original TPCD-DB and SAP-DB; SF=0.2

Finally, it took us about a month to load the TPC-D data and generate the SAP-DB using SAP R/3's batch input facility. This extremely high loading time can be explained because SAP R/3 carries out extensive consistency checks for, say, every lineitem and order individually. For comparison, it took only a couple of hours to produce the original TPCD-DB directly using the RDBMS's bulkload facility and without carrying out any consistency checks.

TPC-D Power Test

Table 2 lists the results of performing the TPC-D power test. Again, we use the isolated RDBMS with the original TPCD-DB as a baseline for comparisons. On the SAP-DB,

[3]These fields were given default values in our experiments.

Queries and Update Fcts	RDBMS (TPCD-DB)	Native SQL (SAP-DB)	Open SQL (SAP-DB)
Q1	6m 09s	58m 59s	56m 18s
Q2	53s	3m 09s	34s
Q3	4m 03s	9m 02s	11m 51s
Q4	1m 45s	6m 18s	6m 38s
Q5	6m 39s	14m 42s	37m 27s
Q6	1m 20s	7m 28s	14m 06s
Q7	9m 03s	23m 05s	29m 24s
Q8	1m 54s	19m 04s	16m 37s
Q9	8m 42s	31m 33s	1h 7m 14s
Q10	5m 18s	33m 06s	57m 49s
Q11	5s	4m 37s	2m 23s
Q12	3m 15s	9m 48s	9m 36s
Q13	8s	19s	25s
Q14	6m 23s	10m 25s	21m 54s
Q15	3m 25s	13m 51s	28m 31s
Q16	13m 24s	3m 16s	3m 22s
Q17	11s	1m 50s	2m 13s
UF1	1m 40s	1h 46m 54s	1h 46m 54s
UF2	1m 48s	11m 35s	11m 35s
Total (Q1-Q17)	1h 12m 37s	4h 10m 32s	6h 06m 22s
Total (all)	1h 16m 05s	6h 09m 01s	8h 04m 51s

Table 2: TPC-D Power Test: Isolated RDBMS and SAP R/3

we implemented the TPC-D queries and update functions in two variants. The first variant exploits a special SAP query interface, called *Native SQL*. This interface allows to execute the benchmark queries directly with the RDBMS on the SAP-DB. In general, it is not recommendable to implement queries using Native SQL because the implementor of a Native SQL query might overlook intrinsic business process interpretations which are carried out implicitly by SAP R/3's standard query interface so that Native SQL queries are potentially error-prone; but, using Native SQL is nice for performance analyses because it isolates the effects caused by the differences between the original TPCD-DB and the SAP-DB. In Table 2, we can see that the total response time of the 17 queries is about 3.5 times as high in the large and partitioned SAP-DB as in the original TPCD-DB; for example, Query 1 is a simple table scan in the original TPCD-DB whereas it is a three-way join in the SAP-DB. The performance penalties of running the update functions on the SAP-DB is much higher in our implementation—here, we chose to use SAP R/3's standard batch input facility (in both SAP-DB variants) with all complex consistency checks turned on.[4]

The second variant we used to implement the TPC-D queries makes use of SAP R/3's standard query interfaces, called *Open SQL*. The Open SQL results shown in Table 2 are thus representative for the kind of performance a typical SAP R/3 user might observe for decision support queries. As shown in Table 2, the total running time of the 17 TPC-D queries is another 50% higher if Open SQL rather than Native SQL is used. This

[4]Even with these consistency checks turned on, it would be possible to achieve better performance for the update functions; in a system configuration that we used for a previous SAP R/3 release, for example, UF1 had a running time of 44 minutes.

additional performance degradation is mostly due to the following two reasons: (1) some queries (e.g., Q9 which involves an aggregate with an arithmetic expressions) are broken down into smaller subqueries which are executed by the RDBMS individually and whose results are combined (with additional overhead) by SAP R/3 in order to produce the overall query result. (2) For some queries, the optimizer of the RDBMS could not find the best query plan; this was partly the RDBMS's fault, and it was partly SAP R/3's fault because SAP R/3 translates Open SQL queries in such a way that in some situations the RDBMS cannot possibly find a good plan; for example, due to this translation, the RDBMS cannot well determine the selectivity of query predicates. The update functions showed in this variant the same performance as in the Native SQL variant because in our implementation of both variants the same complex consistency checks were performed by SAP R/3.

Conclusion

From the experiments and results shown in the previous section, we can see that due to peculiarities in the schema and query interfaces, it is virtually impossible to predict the performance of a database application system such as SAP R/3 even if comprehensive benchmark results for the underlying database management system exist. Therefore, it is crucial to benchmark an application system in order to find out which RDBMS to use and determine how powerful the hardware must be to meet the end user's requirements. Our study also shows that it is possible to do such "application-DBMS" benchmarks with reasonable effort. With the exception of getting started with SAP R/3 and loading the TPC-D benchmark database into SAP R/3, it is just as much effort to implement TPC-D on SAP R/3 as to implement TPC-D directly on an RDBMS. It is also possible to establish a standard TPC-D benchmark specification for SAP R/3; our Open SQL reports, for example, could be used in any SAP R/3 installation regardless of which RDBMS is used as a backend. So our message is:

Benchmark Database Application Systems—it's important and possible!

Of course, it is unclear whether it is possible to benchmark other database application systems (e.g., CAD systems) in less mature application areas. On the long run as the application systems, the corresponding database technology, and the market for the application systems matures, we believe, however, that "application-DBMS" benchmarks will become common practice in every application area.

References

[AG] SAP AG. R/3 system overview. `http://www.sap.com/r3/r3_over.htm`.

[DHKK] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance of SAP System R/3. `http://www.db.fmi.uni-passau.de/projects/SAP`.

[DHKK97] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance in the real world: TPC-D and SAP R/3. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Tucson, AZ, USA, May 1997.

[Gra93] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan-Kaufmann Publishers, San Mateo, CA, USA, 2. edition, 1993.

[TPC95] Transaction Processing Performance Council TPC. TPC benchmark D (decision support). Standard Specification 1.0, Transaction Processing Performance Council (TPC), May 1995.

Caprera™: An Activity Framework for Transaction Processing on Wide-Area Networks

Suresh Kumar
Vice President, Engineering
Tactica Corporation
suresh@tactica.com

Eng-Kee Kwang
Chief Technology Officer
Tactica Corporation
engkee@tactica.com

Divyakant Agrawal
Associate Professor
UC Santa Barbara
agrawal@cs.ucsb.edu

Abstract

Caprera software is an open framework for designing client/server applications that operate over a wide area network. The Caprera activity model that is used to extend transaction processing and transaction-oriented applications in an open environment including mobile and remote clients connected by wireless, phones, or Internet is described here. Since the Caprera framework enables off-line users on mobile platforms to interact with corporate transaction processing systems, Tactica™ Corporation markets its product as a programmable server software for off-line transaction processing (OFTP). This paper describes the design rationale and the OFTP architecture of Caprera software.

Introduction

As advances in computing and networking technologies march ahead, the ways in which information is accessed and managed grow at a faster pace. The explosion of information on the World Wide Web and the Internet and the race to exploit this information in a variety of ways are evidence of this. Wider availability and easier accessibility of technology, information, and tools have resulted in intense and increased competition among organizations in almost every conceivable market segment. The key ingredients for survivability and success of such organizations are to identify inefficiencies in their business processes and use advanced methodologies, frameworks and tools for increased automation to get a competitive edge. Business process re-engineering, workflow modeling, client/server computing, object-oriented methodologies, LAN/WAN technologies, and mobile/wireless networking are some of the factors bringing about radical changes in the information and data processing industry.

Proceedings of the Seventh High Performance Transaction Processing Systems (HPTS) Workshop 97. Asilomar/Pacific Grove, California, USA. September 1997.

Business and commercial enterprises' increased reliance on computer technology has been mainly due to the information and data processing needs of these organizations. Unlike scientific computing, in which computers have traditionally been used mainly for their computational capabilities, business enterprises have deployed computers primarily for data storage. The software component that oversees the information storage and processing of an organization is called a Database Management System (DBMS). A DBMS is a software layer that protects the critical organizational data and ensures that all user interactions maintain the integrity of the data. Furthermore, it provides efficient means to store and retrieve critical corporate information.

Under the existing model of data processing, referred to as *on-line transaction processing (OLTP)*, all user interactions, called *transactions*, are executed directly on the machine that stores the enterprise database. The OLTP model is extremely rigid in the ways in which it provides access to the information to its users. There are at least two important changes which require a complete overhaul of the traditional OLTP model. The first is the increasing reliance on re-engineered business processes and workflow automation for transaction processing. The second is extending the enterprise database connectivity to the employees in the field.

Large organizations are realizing that significant efficiencies will result by automating the execution of related activities needed to accomplish a business process. For example, a loan application from a customer to a lender institution involves a collection of related and dependent activities such as loan application, credit check, risk analysis, mailing acceptance/rejection letter etc. Current OLTP systems provide excellent support for executing a high volume of independent and short-duration transactions. Unfortunately, minimal support is available in most OLTP systems for controlling the execution of inter-dependent activities within a business process. The next generation of OLTP and DBMS systems that provide support for managing business processes are referred to as the workflow management systems (WFMS). Several products already exist that provide WFMS functionality. In addition, considerable research is directed to increase the expressibility, scalability, performance, and reliability of

WFMSs. In general, in most WFMSs, a business process is modeled as a directed graph called a workflow. The nodes in the graph represent the activities needed to complete the process and the arcs represent the control and data-flow among these activities. Most WFMSs provide a tool to model workflow and a run-time system to control the execution of activities of workflow instances. None of the WFMSs provide much support or tools to design the activities themselves.

Due to advances in the networking technologies, most employees of an enterprise have some level of connectivity to the enterprise-wide information system. Under the current OLTP paradigm, only those users who are directly connected to the enterprise network can access the enterprise-wide databases. This results in a layered information structure that duplicates work and cause inefficiencies. For example, an employee in the field may need an intermediary (e.g., an IS professional) to query and update the enterprise database from the field. A more appropriate model would be to download the information or data from the enterprise database directly on the remote user's machine. The remote user or the employee in the field can use this data to accomplish his/her tasks and upload the updates back to the enterprise database. This model of operation, which can be loosely termed as *off-line transaction processing (OFTP)*, will enable remote and mobile users to integrate with the enterprise-wide information infrastructure.

In this paper, we describe the design and implementation of Caprera software, a product primarily developed to extend database connectivity to mobile and remote users. In particular, this system provides a comprehensive framework for designing activities that are the building blocks of a business process. In addition, CapreraScript has built-in primitives for supporting workflow automation.

Caprera Activity Model

The Caprera system consists of an open, standards-based software framework that provides a complete set of software tools to efficiently build, deploy and manage transaction-oriented applications that operate over wide area networks such as phone lines, wireless networks, and the Internet. In this section, we describe the Caprera activity model as well as the rationale behind various design choices.

Design Rationale

A key goal of Caprera software is to provide transaction processing capability from mobile/remote platforms on host or corporate databases. Since there exist billions of dollars worth of investment in legacy applications on such databases, it is clear that Caprera architecture goals must be achieved without any modifications to either the host DBMS software or the host databases. Caprera software aims to provide database connectivity to its users over a WAN that includes mobile and remote users. Typically, the duration of interaction between Caprera software users and host databases is expected to be much larger than that between OLTP clients and host databases. The most widely used OLTP solution for ensuring data integrity is to lock data objects until transaction termination (often referred to as strict two-phase locking [5]). Although locks and two-phase locking are universally used in commercial DBMSs, it is commonly agreed that long duration locking may result in significant performance degradation. Since the Caprera software user interactions are expected to be long duration, due to slow network links as well as asynchrony due to mobility, the Caprera software cannot afford to use locking as the only means to solve the data integrity problem. A final requirement for Caprera framework design is to provide a uniform solution to fulfill the needs of all classes of Caprera software users such as mobile (intermittently connected) and remote (connected over slow links) clients.

Since user interactions in the Caprera architecture are long in duration, the term *activity* is used to distinguish Caprera software interactions from the standard transaction concept in databases. An activity enables Caprera software users to execute applications with transactional guarantees from mobile and remote client platforms. Traditionally, the transaction paradigm has been used to ensure atomicity of user interactions with databases. In order to simplify the application design, transactions are used to ensure that the execution of user transactions remains virtually "atomic" in spite of concurrent user interactions and system failures. These notions are referred to as ACID properties of transactions [7] or execution atomicity and failure recovery[2]. These notions in turn ensure the integrity of data in database management systems. Concurrency control protocols such as two-phase locking are used to ensure execution atomicity of transactions, and recovery mechanisms typically based on write-ahead logging [10] are used to support failure recovery in database systems.

Several proposals have been made to deal with long-lived transactions [6,12]. Although the proposed models can be adapted to handle activities that are both long-duration and transactional, none of the proposed long-lived models are ideally suited for all possible database workloads and environments. A general consensus among the database system researchers and practitioners is that ensuring atomicity of long-lived transaction at the system level may result in significant performance degradation (for example, due to locks being held for a very long time). Also, it is commonly agreed that the execution atomicity and failure recovery (to a lesser extent) for long-lived transactions can be handled best at the application level. Thus, instead of providing a

solution to enforce atomicity of long-lived transactions, a better approach is to provide a framework in which ensuring the atomicity at the application level becomes easier. Caprera software is designed with this rationale in mind. In particular, the system provides a framework that can be used to control the degree of atomicity at the application level.

Concurrency Control in Caprera Software

Figure 1 shows a three-tier architecture in which the Caprera system is deployed. As shown in this figure, the enterprise database is configured so that OLTP users can interact with the database via standard transactional interface using TP monitors and WAN users can interact with the host database via activities that execute in a distributed manner between servers and clients running Caprera software. From a correctness point of view, the Caprera framework must ensure execution atomicity not only of concurrent Caprera activities but also with respect to the transactions that are being executed at the host database. Caprera software uses two different approaches to control the concurrent execution of transactions and Caprera activities. These approaches can be classified as pessimistic and optimistic approaches to concurrency control [3,7].

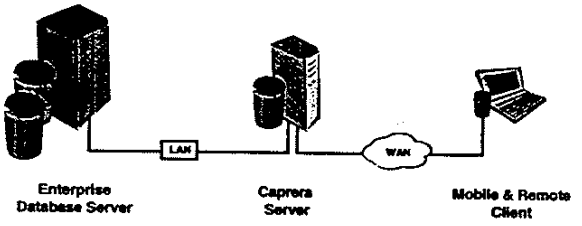


Figure 1: Three-tier Caprera System

In the pessimistic approach, Caprera activities are encapsulated in a transaction block. The semantics of a transaction block are such that interactions (i.e., read and write operations) from the activity to the host database are done by conforming to the host concurrency control mechanism. For example, if the host database uses two-phase locking, then all reads and writes from a Caprera activity result in setting locks on appropriate objects in the host database. This ensures that the Caprera activity is serializable with respect to host database transactions.

In the optimistic approach, Caprera activities are executed without performing any synchronization with respect to the concurrent host transactions. However, a certification check is performed before committing the updates of a Caprera activity. Unlike in optimistic concurrency control protocols [9,1], failed certification does not necessarily result in the abortion of Caprera activity. Instead, an exception is raised and it is up to the application designers to appropriately handle the exception at the application level. For example, the application may choose to ignore the exception, may take some action to resolve the conflicts[1], may redo the activity, or may abort the activity.

In order to accommodate both approaches, Caprera activities are designed so that updates to the host database are deferred until commit time. An advantage of this approach is that when an activity is executed as a transaction block, only read locks are held during the life-time of the activity. Write locks are needed for relatively short duration, in particular, only the termination phase of the activities are executed without synchronization (i.e., outside of a transaction block). Unguarded execution of Caprera activities will not result in host transactions viewing inconsistent data.

Updates from Caprera activities are incorporated in to the host database either when there were no conflicts (hence, the overall execution is serializable from the concurrency control point of view) or when the update conflicts have been resolved (indicating that there are no data inconsistencies at the application level even though the execution is not serializable). The certification check is the standard validation test from the optimistic concurrency control method [13,9]:

> *The read set of Caprera activities are validated against committed writes in the host database.*[2]

In order to avoid execution anomalies after the certification, an application designer must execute the certification and the commitment of a Caprera activity as a transaction block. In effect, this results in Caprera activity obtaining read and write locks during the termination phase which is significantly shorter than the entire life span of Caprera activities.

Note that the two approaches discussed above were primarily motivated to ensure correct execution of a Caprera activity in the presence of concurrent host transactions. The question then arises what about the execution of multiple Caprera activities? What if different activities employ different approaches, i.e., one activity is in a transaction block while another concurrent activity relies on update certification? It can be easily shown that if update certification is strictly enforced, i.e., activities that fail certification are aborted, and transaction block is used during certification, then the execution of concurrent host transactions and Caprera activities will be serializable and therefore correct [11,4].

Failure Recovery in Caprera Software

In contrast to the problem of concurrency control, where an unconstrained execution of a Caprera activity may potentially leave the host database in an inconsistent

[1] Caprera software provides a conflict resolution editor.
[2] The current implementation ensures that the data read from the host database is indeed the same at commit time.

state, the impact of failed Caprera activities on the host database is not critical. A recovery mechanism is used in database management systems to eliminate the effects of partially executed transactions that failed due to system crash or transaction aborts. In most commercial databases, in-place updates are used to execute write operations of transactions. Thus, if a failure occurs, the recovery component of a DBMS must undo the in-place updates of aborted transactions. Besides, in-place updates, DBMSs use write cache to reduce the I/O latency incurred while committing transactions. As a result of caching, it may be possible that committed transaction updates are not incorporated into the database prior to a failure. In this case, the recovery component must redo all such updates when the system recovers from failure. Write-ahead logging [10] is the most common technique to implement failure recovery in database management systems.

Caprera activities are structured so that all updates to the host database are deferred until termination[3]. Furthermore, a Caprera activity is either completely executed as a transaction block or the certification and update phase is executed as a transaction block. From the point of view of the host database, a Caprera activity is vulnerable to failure only after it performs its first update to the host database. Until then, failure of a Caprera activity has no effect on the host database. If transaction blocks are used to encapsulate the updates within an activity, then the updates are subjected to the concurrency control and failure recovery mechanisms of the host database. Hence, the recovery mechanism of the host database is sufficient for undoing the updates of aborted Caprera activities, as well as redoing the missing updates of committed Caprera activities during crash recovery.

Although the failure of a Caprera activity prior to the update phase does not impact the host database, it results in lost work for Caprera clients. In order to minimize communication between Caprera clients and servers, it is expected that Caprera activities will be designed to accomplish significant amount of work as compared to host transactions. Therefore, it is necessary to provide a mechanism in Caprera to facilitate *forward recovery*[4] of failed Caprera activities. Furthermore, since Caprera activities are long-lived, it may be useful to support partial backout or *backward recovery*[5] of Caprera activities.

Viewing a Caprera activity as a sequential program (for example, a shell script or a C++ program), the progress of an activity could be logged at every primitive statement level. Logging at such fine granularity will enable the Caprera users to minimize the amount of lost work as a result of failures or rollbacks of activities. Unfortunately, statement-level logging results in a significant overhead, since the program state must be saved on stable storage after the execution of every statement. Caprera strikes a balance in the tradeoff between logging overhead versus lost work due to failures and rollback by structuring Caprera activities as a sequence of coarse logical units.

A Caprera activity is a distributed object that embodies business rules and data. It is distributed because the execution of the Caprera activity involves multiple sites in the network. For example, a typical activity may be initiated at a Caprera server, followed by data collection/extraction from the host database. After the activity is ready, it may then migrate to an assigned Caprera client, which may be a mobile or remote platform, where the associated tasks are performed by the Caprera user. Once the tasks are completed by the user at a Caprera client site, the activity migrates back to the server for termination. The Caprera server terminates the activity after incorporating the updates into the host database. Figure 2 illustrates the life-cycle of a typical Caprera activity.

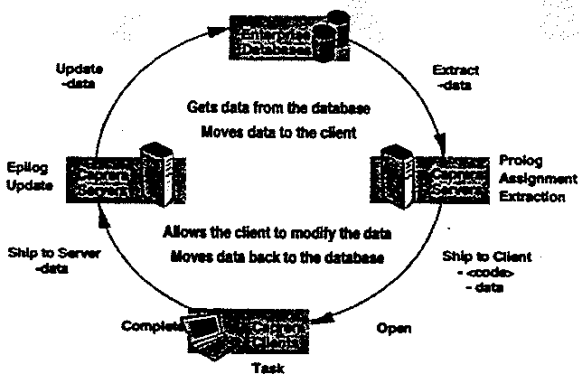


Figure 2: Life-cycle of a Caprera Activity

The various stages of an activity in the Caprera framework are structured as follows:

- Prolog: This stage of the activity involves initializing the state and the meta-information associated with the activity.
- Assignment: At this stage the input parameters, the business rules, and the initial state of the activity are used to assign the activity to an appropriate Caprera client. The assignment determines the client at

[3] Although it is expected that most Caprera activities will be structured in this manner, the current implementation is not constrained to adhere to this design philosophy. If an application chooses to update the host database in the middle of an activity it is permitted in the current Caprera framework implementation. This approach, however, weakens the failure recovery guarantees provided by Caprera software and must be strengthened at the application level.

[4] A resumption of a failed transaction from an intermediate point of execution is referred to as a forward recovery.

[5] When an execution of a transaction is rolled back to a prior control point - it is referred to a backward recovery. Most database management systems support a simple notion of backward recovery by backing out a transaction completely when it is aborted.

which the task associated with the activity will be performed.

- Extraction: The input parameters, the assignment, and the business rules are used to extract the data that will be needed from the host database to execute the activity.
- Client Task(s): At this stage, the activity migrates to the client machine where the client tasks are performed.
- Update: The activity migrates back to Caprera server and business rules are used to update data in the host database using the data that was returned from the client machine.
- Epilog: The activity is completed after performing all the necessary clean up.

Figure 3 illustrates other types of activities that can also be supported by Caprera software. Each activity type is determined by the number of stages defined in the activity as shown.

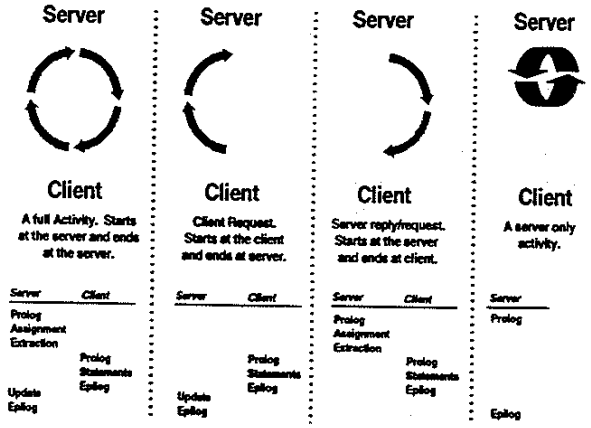


Figure 3: Variants of Caprera Activities

Failure recovery in Caprera software involves providing resiliency to both site and communication failures. In particular, a local persistent store (LPS) is used to log the run-time information associated with Caprera activities. Furthermore, transactional queuing mechanisms are used to deal with communication failures between Caprera clients and servers. In addition, control mechanisms are available for partial or complete rollback of an activity. A user-initiated abort or rollback request is admitted either to rollback an activity completely or to the beginning of any of the six stages in the activity.

Site and Communication Failures

Site failures in the Caprera architecture can be classified as host database failures, Caprera server failures, or Caprera client failures. A Caprera activity may be impacted by host database failures in two different contexts, either within a transaction block or without a transaction block. In the first case, Caprera relies on the host database recovery mechanism for safe recovery. Any partial host database updates, however, will be undone as part of the host crash recovery. In the latter case, i.e. non-transactional host database interactions from Caprera software, the Caprera server will time-out and retry the query.[6]

If local persistent store (LPS) is not available at a Caprera server, the only recourse to deal with server failures will be to abort all on-going activities. The only resiliency that can be provided in the absence of local persistent store is that activities that migrated to Caprera clients are not vulnerable to server failures. In order to achieve this resiliency, however, complete run-time and control information must remain with the activity at all times. On the other hand, availability of local persistent store eliminates the need for transmitting the entire run-time and control information back and forth between Caprera servers and clients. Local persistent store can be further exploited to increase the resiliency of Caprera activities. In particular, if a server failure occurs during the execution of an activity on the server (i.e., in the midst of the prolog, assignment, extraction, update or epilog), the server recovers the activity by re-executing the stage in which the activity failed.

Since a server failure may occur when an activity is interacting with the host database within a transaction block, the question arises what is the fate of that suspended transaction, which is now an orphan. Depending on the capability of the host database, either a passive or an active implementation strategy may be used to abort orphaned transaction blocks. If the server failure results in a lost connection indication to the host database, the latter will simply abort the transaction associated with such an activity and release all the resources (i.e., locks held). No recovery action is needed from the Caprera server side to clean up orphaned transactions. If disconnections are not handled by the host database, a pro-active approach is necessary to clean up orphaned transactions. Caprera software currently uses the passive approach and relies on the host database management system to abort orphaned transactions.

The failure recovery of a Caprera activity at the server relies on the local persistent store providing the necessary database functionality. In particular, the local persistent store at the server is a shared database in which the run-time and control information for Caprera activities is stored. Each stage of an activity is modeled as a transaction against the local persistent store. Server failures are thus easily handled by relying on the recovery component of the local persistent store. As a side-effect,

[6] The assumption being that application design is such that host database updates from Caprera software occur within a transaction block. If this is not the case then application-specific recovery mechanisms must be deployed to deal with non-transactional host database updates.

this approach also ensures that conflicting accesses to the shared run-time and control component of the local persistent store are synchronized. Although this may appear to be too restrictive, it must be noted that most of the run-time and control information is specific to activity instances. Therefore data conflicts at the local persistent store among concurrent activities are expected to be very rare.

Resiliency to client failures can be easily provided by exploiting the local persistent store at the Caprera server. That is, if a client failure occurs in the middle of a task execution, then the client can re-load the task portion of the failed activity on recovery. The assumption is that the server maintains all the information about an activity until the task is completed at the client and the completion is acknowledged to the server. This recovery model is appropriate for tasks that are automated at the clients. If the task execution at the client involves manual processing (e.g., data entry, sending electronic memos, etc.), then re-executing the task stage of an activity from the beginning may be wasteful. Furthermore, unlike other stages of an activity which are executed sequentially, the task execution is event driven. That is, the user is presented with a *to-do* list but the order in which the items on the list are completed is at the discretion of the user. User may reload the task from the server, or selectively undo/redo the items in the task that were either affected or were not done due to the failure.

As is apparent from Figure 1, communication failures may affect the connections between the Caprera server with the host database and with Caprera clients. Handling of communication failures to a certain extent is handled at the communication protocol level (e.g., TCP/IP deals with lost and out-of sequence messages). If the server interaction to the host database is transactional then Caprera software relies on the host database to deal with such failures (e.g., a lost TCP/IP connection will result in the ensuing transaction to be aborted). For non-transactional interactions, persistent queues (e.g., MQ Series from IBM) are used to deal with lost messages. Communication failures between Caprera server and clients are handled by relying on transactional queuing mechanisms.

Caprera Architecture

Caprera software is a client/server application-development environment for building transaction-oriented applications. An activity is the basic building block of a Caprera application. Applications are built as a collection of persistent objects in the Caprera system. These objects are activities, subset tables, views, rules and events. Subset tables define the data subsets that are selectively replicated to the clients using the activity framework. Views are the form definitions for presenting the subset data on the client. Rules and events are the triggers for initiating activities on the client or the server and can be used to design workflows in the application. Activities are transactional objects that encapsulate business rules using a Java-like scripting language called CapreraScript. Activity objects use subset tables, views, rules and event objects to represent a business process.

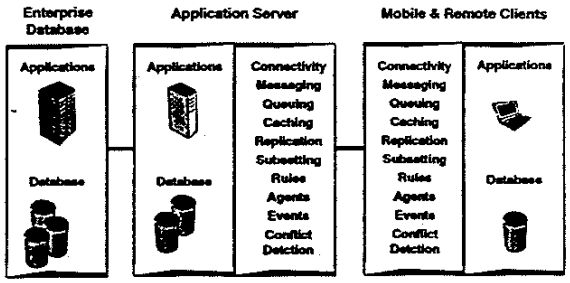


Figure 4: Underlying Technologies in Caprera Software

Caprera software provides the framework for OFTP by integrating a number of technologies as shown in Figure 4. Connectivity is just one of the components and Caprera software uses the rest of the technologies like messaging, queuing, caching, replication, subsetting, agents, rules, events, conflict detection and resolution framework for building OFTP applications. The following subsections describe in detail the internal structure of Caprera software.

Caprera Server

Figure 5 represents the layered architecture of a Caprera server. Each of the blocks in the diagram represents a major functional module such as Database and Transaction Managers, Communication Manager, Persistent Object Manager, OFTP workspace, Activity Manager, Rule and Event Managers, Security Manager and the CapreraServer interface. The persistent store for Caprera objects is a relational database called Local Persistent Store (LPS). LPS is also the keeper of the activity log on the server as well as on the client. All six stages of an activity are logged into the LPS, so that each stage can be recovered or rolled back in case of failures.

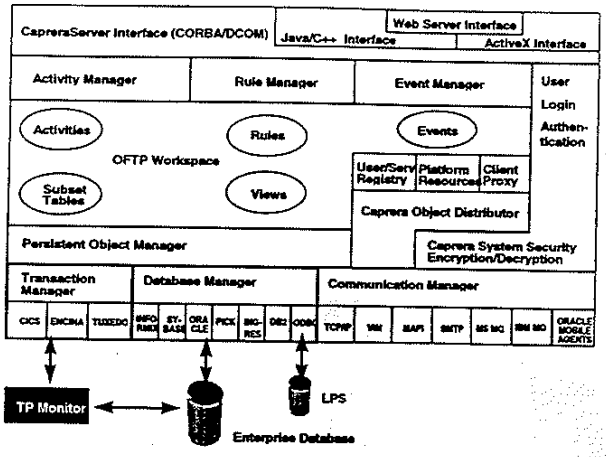


Figure 5: Caprera Server

All interactions with the LPS as well as the host databases are handled by Database Manager. Database Manager provides a number of native and ODBC adapters so that Caprera software can operate with a range of commercial database management systems. Database Manager makes use of the concurrency control and failure recovery mechanisms provided by the LPS as well as the host database. Transaction Manager complements the task of Database Manager by providing additional transactional support through integration with TP Monitors for host database connectivity.

Communication Manager provides connectivity between Caprera servers and Caprera clients. Communication between Caprera clients and servers is based on reliable message queuing to ensure guaranteed delivery of activities. Communication Manager supports a number of standard messaging protocols such as TCP/IP, VIM, MAPI, SMTP, MS MQ, IBM MQ and Oracle Mobile Agents.

Security Manager provides services based on RSA public/private key encryption algorithms for secure transmission of activities over public data networks.

Persistent Object Manager is responsible for storage and retrieval of Caprera system objects. Persistent Object Manager uses the LPS for storing Caprera objects and is responsible for the necessary format conversion from external (relational) representation of Caprera objects to their internal representation and vice-versa. When in memory, Caprera objects reside in the OFTP workspace and comprise of different object types such as *activities, rules, events, subset tables, views,* and other control objects. Note that the subset tables contain the data that has been extracted from the host databases. Views, on the other hand, provide the form definition (user interface) that is needed to display data in the subset tables to the users.

Caprera Object Distributor is responsible for distributing activities between the server and client. After the execution of Prolog, Assignment and Extraction stages, the activity is handed over to Caprera Object Distributor for distribution to the client. Caprera Object Distributor compresses the activity object and then uses the Security Manager to encrypt activity objects using RSA public/private encryption algorithms and then hands over the secure activity to the Communication Manager who is responsible for transporting the activity to the Caprera client.

Activity Manager is primarily responsible for scheduling the activities on the server. Each stage of an activity is executed in its own process space. The Activity Manager can perform load balancing by distributing activity execution to multiple servers on the network using a scheme based on the owner of the activity. Dynamic load balancing is achieved by distributing the activities to other servers based on the load of the currently executing activities. Activity Manager supports forward recovery by providing a framework for executing compensating[7] activities in case of data conflicts while checking data into host DBMS from the client.

Rule Manager is responsible for handling the rule objects as well as testing and firing of the rules. Similarly, Event Manager maintains the event objects and evaluates these objects against the event queue to ensure that appropriate actions are taken when a specified event occurs. Note that the activity, rule, and event objects are constructed using CapreraScript and Activity Manager, Rule and Event Managers employ the services of CapreraScript engine to interpret these objects.

Finally, the CapreraServer interface provides a means for interaction and manipulation of Caprera objects from other programs. Caprera software provides these interfaces through DCOM on Windows platforms as well as CORBA for all supported platforms. Caprera software provides a Web server interface so that the Caprera server can be accessed through a Web browser such as Netscape Navigator or Microsoft Internet Explorer.

Caprera Client

The overall structure of a Caprera client is illustrated in Figure 6. In general, most of the components in a Caprera client are light-weight versions of the same in the Caprera server. Also, some of the components are not present in a Caprera client. For example, Transaction Manager is not available in the client sub-system. The Database Manager at the client is primarily needed to manage the LPS at the client machine even though its functionality can be used to access any database on the client side.

[7] For backward recovery the compensation is used to undo the effects of the part of the activity that has already been executed. In contrast, for forward recovery the compensation is performed to the parts of the activities that have yet to execute.

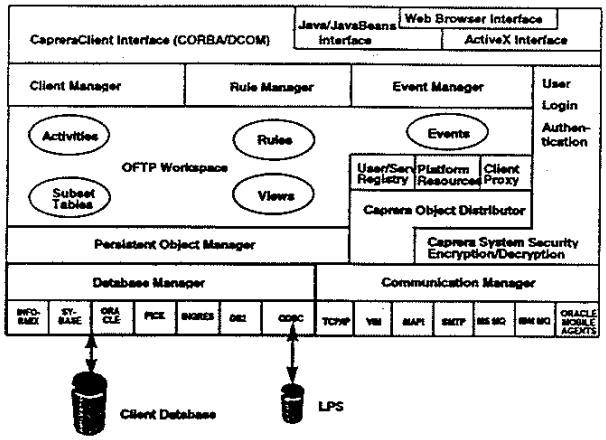


Figure 6: Caprera Client

The counterpart of Activity Manager on the client is the Client Manager. The Client Manager receives the activity object from the client Communication Manager, decrypts and uncompresses the activity object and brings the activity object back to life. Then the task stage of the activity is executed on the client. On completion of the client task, the activity is shipped back to the server securely. At the server, Update and Epilog stage of the activity is scheduled by the Activity Manager for execution.

Caprera software supports user login authentication while invoking the server or the client. It uses algorithms from the RSA suite to provide extensive security mechanisms for transporting activities over public data networks. On Windows platforms, inter-application communication is provided through ActiveX as well as Java Beans. This allows for tight integration of third party applications with Caprera client. Through these interfaces, Caprera client functionalities can be embedded in a Web browser.

OFTP Using Caprera Software

Caprera can be used in a variety of ways to accommodate the OFTP needs of large enterprises. In its simplest form, Caprera server can be used in a centralized manner as shown in Figure 1. This is referred to as a three-tiered OFTP architecture. Caprera server has LAN connectivity to enterprise database servers whereas Caprera clients may be connected over a WAN to Caprera server. The activity framework of Caprera is used to extract relevant data from the enterprise database and is transported to the client. The completed activity is sent back to the server from the client. The server incorporates the changes into the enterprise database.

In order to accommodate the needs of organizations with higher OFTP workloads, e.g., thousands of OFTP clients, Caprera software can be deployed in a distributed manner as shown in Figure 7. In a distributed OFTP architecture, multiple Caprera servers are used for sharing the load of activity execution. The activities can be partitioned statically for execution on specific servers, or distributed dynamically among all servers in order to obtain optimal load balancing.

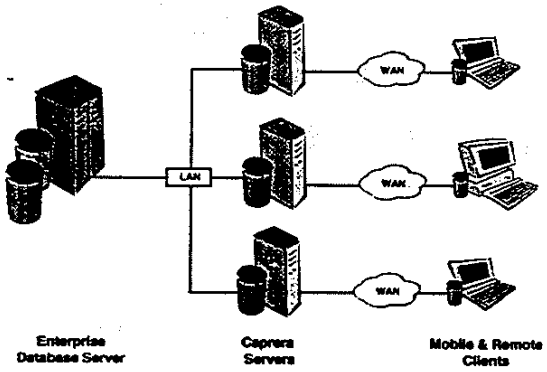


Figure 7: Distributed Caprera Architecture (Three-tier OFTP)

Figure 8 shows a hierarchical organization of Caprera servers to model enterprises that have a hierarchical structure. This is referred to as a four-tier OFTP architecture. At the top of the hierarchy is the headquarter (HQ) database system that provides OFTP capabilities via a Caprera server to its clients. A similar set-up is available to the OFTP clients for the regional offices and regional databases. However, in order to extend OFTP capabilities from HQ OFTP clients to regional databases as well as from regional OFTP clients to HQ database, the HQ Caprera server is connected to the regional Caprera server as shown in Figure 8. Note that due to the geographical distribution of HQ and regional offices, this connectivity is over a WAN.

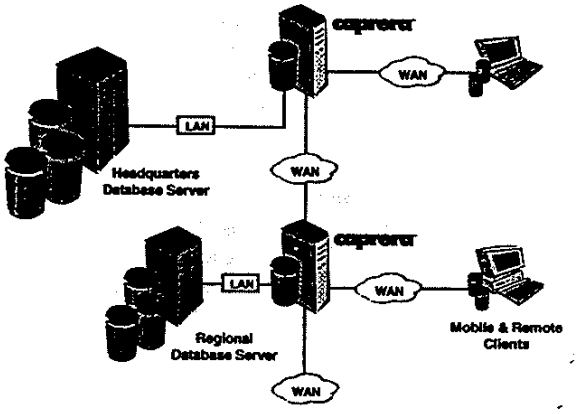


Figure 8: Hierarchical Caprera Architecture (Four-tier OFTP)

Finally, Caprera software's flexible architecture allows building information infrastructure that mirrors the real information model of large enterprises. In that sense, Caprera software can be used to support distributed multi-tier environment for OFTP, as shown in Figure 9.

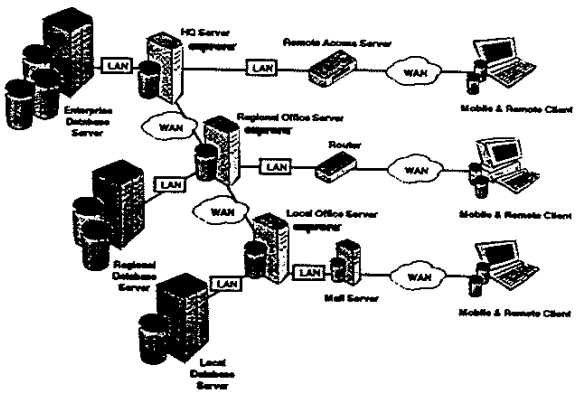


Figure 9: Generalized Caprera Architecture (Multi-tier OFTP)

Concluding Remarks

In summary, Caprera software is a commercial product that provides host database connectivity to mobile and remote users over a wide-area network. We refer to this as OFTP (*off-line transaction processing*) in contrast to the traditional OLTP. The foundation of the Caprera transaction and recovery architecture and the activity model is based on traditional database theory and concepts.

References

[1] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed Optimistic Concurrency Control with Reduced Rollback. *Distributed Computing, Springer-Verlag*, 2(1):45-59, January 1987.

[2] D. Agrawal and A. El Abbadi. Transaction Management in Database Systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 1-32, Morgan Kaufamann Publishers, 1992.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, Massachusetts, 1987.

[4] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5(5):203-216, May 1979.

[5] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624-633, November 1979.

[6] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249--259, May 1987.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[8] J. N. Gray. Notes on Database Systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393-481, 1978.

[9] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213-226, June 1981.

[10] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method supporting fine-granularity Locking and Partial Rollbacks using Write-ahead Logging. *ACM Transactions on Database Systems*, 17(1):94-162, March 1992.

[11] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631-653, October 1979.

[12] A. Reuter and H. Wachter. The ConTract Model. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219-263, Morgan Kaufamann Publishers, 1992.

[13] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transaction on Database Systems*, 4(2):180-209, June 1979.

[14] Caprera Technical Publications. Tactica Corporation.

Shirt Pocket Transactions

Tobin Lehman

IBM Almaden Research Center
650 Harry Road (K55/801), San Jose, CA-95120

- HPTS Position Paper -

1 Information Management in a Busy World

The world of business is growing more frantic and complex; decision time is shorter, the number of factors contributing to most business deals is larger, and the number of information sources is greater. Having the proper information at hand is the single-most important factor in any major business decision. Thus, the key for dealing with today's fast-paced and increasingly complex business world is building and maintaining an understanding of all parts of the business. However, this knowledge is not acquired easily; its acquisition is achieved by locating and managing the proper information. And, unfortunately, the trend for information management has gone from fully accessible centralized data to less accessible distributed data.

Information, the lifeblood of an enterprise, was once the sole responsibility of mainframes. Company data, in the form of rigid tabular, textual information, resided in a closely guarded centralized data center. Data processing applications could be directed to access business information residing in files and database systems in a central computing complex. In a sense, this was the golden age of data processing, as enterprise data would never again be so simple and so easy to manage – although this may not have been the view of a user, since the tools for accessing data were primitive and clumsy.

The arrival of the client/server model changed the organization of the data from one of a centralized system to one of a distributed system. With that came problems. Although some amount of company data became more accessible to the average worker – it was no longer guarded by the "glass house" guards – the total collection of company data became less accessible to any one person or company agency because there was less than full connectivity to all of the data. Much of the enterprise data was deposited onto either standalone PCs or PCs in disconnected networks. These were the dark times of information management.

In the last few years we have witnessed the new age of connectivity to information. Extra-nets have allowed individuals access to data both around their intranets and around the world. And, the semi connected world of client server is evolving into the fully connected world of the peer to peer model. Coincident with this connectivity is a new emergent technology, which brings an entirely new family of computing devices into the connected world. The old model of three tiers of computing (tier 3 is mainframes, tier 2 is servers, and tier 1 is desktop machines) is getting a new tier – tier 0. Tier 0 represents those devices that are more lighter/smaller/mobile than traditional desktop machines, such as PDAs and smart cell phones.

2 Smart Everything

We are on the brink of a new era in computing. The cost of an embedded computer (single chip computers comprising CPU, RAM, I/O ports and wireless communication) will soon be low enough ($10 range) to embed in most major (and some minor) home, automotive and business appliances. In addition, computing devices (PDAs) [USR Pilot, Apple Newton] and mobile devices (pagers, phones) [Nokia, Motorola] will merge, adopting features from each other to create a family of wireless computing products. A Nokia phone (the 9000 communicator) [Nokia 9000], for example, already supports most features of a PDA. Similarly, the popular USR Palm Pilot (PDA) [USR Pilot] has numerous wireless modem options already. One could expect that a future model of the Pilot will have pager and telephony capabilities.

3 New Age Transactions

At the first HPTS workshop in 1985, there were many discussions of transactions that involved buying an airline ticket or making a bank withdrawal. These classical transactions were served by very large centralized complexes running high performance transaction processing systems like IMS Fastpath (now IMS/ESA [IMS]), ACP (now TPF [TPF]) and SABRE [Sabre]. In the last 12 years, a significant change has taken place. Now, in this new information age, transactions are no longer debit/credit operations on a centralized store, they can be any durable exchange of information between two (or more) parties. And, rather than directly involving people, as was often previously the case, these newer transactions may well act completely autonomously. Acting on behalf of a user, but not necessarily under the user's direct control, a wireless USR Pilot (a person's mobile platform) may exchange data with home appliances, with the car, with other PDAs or with public institutions, such as stores, banks or agencies. Each of these interactions may involve a complete range of data types, from basic types like numbers, text, audio, and video to more complex types like applets and encapsulated types. In addition, the diversity of peers that our Palm Pilot talks to can also be large.

4 A Database System for Tier 0

What would we expect of the database system that ran on a tier 0 device? As an example, let's take our favorite mobile platform, the Palm Pilot. In a single day, our Pilot could be exchanging data (executing transactions) with several hundred different systems using a multitude of datatypes. The Pilot would be an information (push) client, our interface to the home/car/store/shopping mall/office, and the general database for all personal information. One of the common users for the Pilot would be to connect to devices (vending, pay phones, ATMs, etc) to deliver a key for access. And, although the Pilot might not retain large amounts of data (more than tens of megabytes), a large amount of data could conceivably pass through it. Today's Palm Pilot Professional comes with no disk memory (though 1 inch drives could be common in the future) and 1mb of RAM (upgradable to 2mb, or even possibly 4mb with after-market upgrades).

What database system will we run on our Pilot? Or, more appropriately, what are the requirements for a Tier Zero Database Management System (TZDB)?

Small footprint: The TZDB must be able to fit in a fraction of a megabyte of memory. Since the TZDB will no doubt grow over time to add functionality, any increase in tier 0 device memory will easily be absorbed by the DB and the increasing amounts of data.

Flexible data model: Given the diversity of data sources that the TZDB will interact with, the TZDB must able to store records with dynamically varying numbers and types of attributes. In other words, it should not have the rigid schema of a relational system.

Flexible type system: The TZDB must be able to load new types dynamically that can be compared and indexed (this is something often talked about by the Object-Relational systems, but rarely implemented).

Flexible management system: The TZDB must be able to load new operators dynamically that perform new data management functions.

Anonymous communication: The TZDB must allow clients to add and remove data from the information store without application-level knowledge of client and server machine addresses. When used in a producer/consumer mode, it's important that both producers and consumers can connect to the TZDB anonymously and asynchronously.

We believe that the standard Relational or Object-Relational Database systems: DB2, Oracle, Sybase, Informix and SQL Server, and the Object Database systems: ObjectStore, Versant and Objectivity do not (and will not) satisfy the above criteria. So, if not the name brand database systems, then what?

At the IBM Almaden Research Center, we have been working on Almaden TupleSpace as part of the Yoda project.[1] The Almaden TupleSpace is a lightweight, network oriented database system that is tuned to the needs of tier 0 devices. The spirit of the Almaden TupleSpace comes from the Tuplespaces used in the LINDA Project [Gelertner 85, Ahuja 86, Carriero 89]. In the mid-1980's, David Gelernter, a professor at Yale University, created the LINDA programming language, which was designed to address the communication problems of parallel programming. Part of this project was a concept known as Tuplespaces, which were designed to simplify data exchange between parts of a parallel program. TupleSpaces embodied three main principles:

- Anonymous communication
- Universal associative addressing
- Persistent data

Although the LINDA TupleSpace was originally intended as a global communication buffer for parallel programs (which are concerned mostly with synchronization and high performance), we found that it also works for distributed programs which often need a ubiquitous persistent communication buffer. The central concept of a LINDA Tuplespace was surprisingly simple – agents post unstructured tuples to a universally visible Tuplespace, consume tuples, and read tuples. LINDA was a bit hit in the parallel programming community [too many references to mention], but it has not enjoyed much visibility outside of that particular research area. More recently, SUN Microsystems has renewed interest in Tuplespaces, calling it Javaspaces [Javaspaces].

We use the Almaden TupleSpace (ATS) to simplify data exchange between peer to peer components in a generalized network, where the data itself is very dynamic. The ATS is able to handle dynamically changing data because it does not use (or require) a static data schema definition. And, because it is written in Java, the data transport between all platforms is standard.

[1] We felt that Yoda was a perfect icon for our project. Yoda is small, he is very powerful (much tougher than he looks) and he is one with the completely connected universe (the force flows through everything).

The ATS is a simple data manager that manages self-describing "tuples". A tuple is an ordered set of type/data pairs. Applications sharing information agree on the general structure of the data, though there is the ability for dynamic additions of new data and new types. Operations are somewhat simpler than a database system. Though the essence of the operations are the same: insert, delete, and query, the queries do not have the complexity of the SQL language. This is considered a plus for these applications that mostly move and store data, rather than execute elaborate queries over it.

The ATS model works well across a wide variety of applications. Consider a simple embedded environment in the home, where a number of computers embedded in appliances communicate via TupleSpace to exchange status information with each other, with the central home security system, with the homeowners Pilot, or with anyone who cares. Similarly, today's cars have an average of 30 on-board computers (the advanced cars have up to 60) – most of them not sharing data. With the automotive industry trend changing to put devices on a common data bus and powerline, these devices will be able to communicate with each other or with the central automotive security system (which then communicates to the driver's PDA). All information exchange can be done via a TupleSpace system.

Consider the Pilot that fits in your shirt pocket. You need the latest map information, corresponding to your current location, downloaded into it. Do you need to know the IP address or the URL of the map server? No. You just need to issue the request, "Map needed for the following coordinates" to the mobile TupleSpace server. The map server, listening for query requests, responds, putting the answer back into TupleSpace. The answer may be in the same format that it was in last week, or it may now contain new audio data that correspond to various interest points on the map. The TupleSpace is able to serve up map data to your Pilot, along with the new audio handling methods, if needed.

Our experience with ATS so far is that it is lightweight and flexible, yet it is powerful enough to serve as a real information store. It is a reasonable system to manage information in your home, your car, your PDA and your PC. It can easily interface with more sophisticated stores, such as a fully functional relational database system, to satisfy the information needs of any client. In fact, a TupleSpace works quite well as an interface to a database system, where the initial query and the answer set are both transmitted via tuples.

5 Conclusion

The small mobile or embedded computers (PDAs, sensors, pay phones, vending machines, home appliances, office equipment, etc), often referred to as tier 0 devices, represent a major change in how our computing infrastructure will function. Rather than use the desktop model of directed point to point communication over a LAN, tier 0 device communication will be autonomous and often anonymous. In addition, the data management needs of these new devices will not be heavily query oriented (no need for a heavy-weight query language like SQL), but will instead be more information transaction oriented. A new system, the Almaden TupleSpace, can connect the tier 0 devices, satisfy their data management needs, while keeping a connection to the tier 1, 2 and 3 information stores.

Many corporations will be incorporating tier 0 devices into their computing networks, as these devices are often the information generators upon which the company's business data is based. It is essential for the livelihood of the business that the data can flow from the tier 0 devices all the way back to the corporate data warehouse, where it can be analyzed with modern data mining techniques. The Almaden TupleSpace tier 0 database provides the data conduit to connect tier 0 devices to each other and to the main business computer networks.

6 References

[Ahuja 86] Ahuja, Carriero, and Gelernter, Linda and Friends, Computer, August 1986

[Apple Newton] http://www.apple.com

[Carriero 89] Carriero and Gelernter, Linda in Context, Comm of the ACM, Apr 1989, Vol32, No4

[Gelertner 85] D. Gelertner, Generative Communication in Linda, TOPLAS, Jan 1985, Vol7, No1

[IMS] http://www.software.ibm.com/data/ims/

[Javaspaces] http://chatsubo.javasoft.com/javaspaces/

[Nokia] http://www.nokia.com/comm9000.index.html

[Sabre] http://www.sabre.com, http://www.travel.sabre.com,

[TPF] http://www.s390.ibm.com/products/tpf/tpfitaa.html, http://www.blackbeard.com/tpf/book/tpf.history.html

[USR Pilot] http://www.usr.com/palm/500.html

HPTS '97 Position Paper: Distributed Workload Balancing Deserves More Attention!

Benoit J. Lheureux
TOP END Product Center, NCR Corporation
17095 Via del Campo
San Diego, CA 92127
benoit.lheureux@sandiegoca.ncr.com

The advent of widespread, large-scale message-based (passing or queueing) multi-tier, distributed computing with the associated abstraction of presentation, business and data access logic is fertile ground for deployment of effective workload balancing. (This is not a reference to workflow.) But despite lots of trade-rag and marketing collateral references to "workload balancing," there is little meaningful public discussion or analysis of the requirements—let alone application—of workload balancing in large-scale distributed systems. This brief paper disseminates some key observations about workload balancing that we have experienced with production users of TOP END—NCR's highly scalable, available transaction-oriented middleware—and is intended to be a catalyst for further discussion and analysis of a largely ignored attribute of distributed computing.

A key enabler for high-performance workload balancing is the inherent connectionless relationship between client and server application logic, inherent in distributed computing solutions based on message-oriented middleware. Although there is an opportunity for workload balancing choices to be made in connection-oriented distributed computing (at least at connection time), connectionless relationships give middleware a unique opportunity to decide, based on any number of factors, how each particular request should be dispositioned among multiple providers (processes) of the same service to most effectively utilize system resources. Regardless of whether or not a relationship between the client or server is connectionless, it is the notion that the middleware ensures location transparency—i.e. that requesters of services do not know where their service providers are located—that is important. Connection strings targeting a specific server or specific queue reduce the effectiveness of workload balancing by targeting all like requests to the same destination.

Although stateless services are more naturally suited for various workload balancing algorithms, it turns out that workload balancing algorithms also work for stateful services. In general, however, the resolution of control and the accuracy of workload balancing algorithms tends to decrease overall as the length of persistence of state information increases. Certainly, a key factor is whether or not a statefull server can support multiple states on behalf of multiple clients. In cases where a middleware forces a server to be exclusively allocated to one client this will significantly reduce a workload balancing algorithm's effectiveness since the number of candidates for any new request is significantly reduced.

Given very high throughput requirements, we have found that applications can effectively utilize workload balancing algorithms whose scope of influence in the distributed solution is hierarchical. That is, where workload balancing decisions for large numbers of requests are made at an intermediate node level across multiple servers (e.g. across one of several physical servers in a clustered database server configuration), then at a lower level across replicated "server objects" (i.e. where each server object is multiple processes that all read from a shared queue), then within a server object (such that the server object will increase or decrease its own number of processes entirely based on the ratio of requests on its shared queue to the number of processes currently running). The point is that there is value in having workload balancing decisions being made at

various stages in the delivery of an individual message, versus making one decision at the beginning of a message's delivery cycle that determines the final destination.

Having a unit of resolution for workload balancing control at the individual (or sometimes grouped) service resolution gives system designers fine granularity of control over how workload is dispositioned. This is in contrast to workload balancing of users across servers, regardless of the services utilized. Also, different workload balancing algorithms have value under different circumstances; for example, simple round-robin algorithms work well for large volumes of consistent duration, short-lived requests. Enhanced routing (which uses feedback about the workload at each server object that is periodically propagated to all its neighboring servers) is useful for transactions whose duration is less consistent.

Interestingly, the workload balancing features of middleware perfectly complement those algorithms used for large-scale Internet Web Server farms. The latter use algorithms that either manage the workload from large numbers of Internet browsers transparently at a hardware level (e.g. using CISCO routers) or at a logical level during the Domain Name Service resolution (e.g. smart URL to TCP address mapping). In both cases although the workload from large numbers of browsers is (ideally) evenly distributed across web servers there is typically no provision for dealing with workload balancing requirements on the *back* end of web servers, on the Intranet, where middleware has a dominant role.

Although it might be inferred from comments so far that workload balancing is primarily useful for performance (increasing overall throughput) reasons, we have successfully used workload balancing algorithms explicitly for high availability purposes. For example, a "local/remote ratio" workload balancing algorithm allows system designers to specify what proportion of similar requests are processed locally (on the same server) versus on a remote server. This has practical use in very large configurations, for example hub and spoke configurations (e.g. bank branch or retail store) where the local/remote ratio is set such that distributed server applications are run and utilized by default at each site but will automatically and immediately fail over to the data center's copy of the server application if the local site's server fails. This, for example, can virtually eliminate the need to have replicated (fail-over) servers at each site, since one or two servers at the data center can be shared by all remote locations for fail-over purposes.

Although we allow system designers to mix and match workload balancing algorithms, e.g. to use different workload balancing algorithms for the same service at different levels in the message delivery hierarchy, our experience (surprise!) is that system designers prefer to keep things simple. For example, we offer workload balancing algorithms for system designers to favor individual service request response time over total system throughput (or visa versa), but they typically use our workload balancing algorithms for straight-forward objectives like scalability (so it is easy to add servers to an existing configuration) or availability (for cost-effective fail-over configurations).

The observations and comments herein about workload balancing are intended to convey our belief, based on production experience, that workload balancing is a vital attribute of large-scale distributed computing. In addition to the obvious variations in actual workload balancing algorithms themselves, in terms of how message routing various decisions are made, there are many dimensions to the value that workload balancing offers to system designers. It would be good to hear more detail about other implementations of workload balancing in other middleware and transaction-oriented products, as well as to see more references to the value of workload balancing in the public IT domain.

A full-length paper describing TOP END's workload balancing features and how they are used by customers is available at the public TOP END web site at www.ncr.com/product/topend.

Distributed Transactions in Java

M.C. Little and S.K. Shrivastava
Department of Computing Science
University of Newcastle, Newcastle upon Tyne, NE1 7RU, England
email: m.c.little@ncl.ac.uk

1. Introduction

The Web frequently suffers from failures which can affect both the performance and consistency of applications running over it. For example, if a user purchases a cookie (a token) granting access to a newspaper site, it is important that the cookie is delivered and stored if the user's account is debited; a failure could prevent either from occurring, and leave the system in an indeterminate state. For resources such as documents, failures may simply be annoying to users; for commercial services, they can result in loss of revenue and credibility.

Atomic actions (atomic transactions) are a well-known technique for guaranteeing application consistency in the presence of failures. Web applications already exist which offer transactional guarantees to users. However, currently these guarantees only extend to resources used at Web servers, or between servers; browsers are not included, despite being a significant source of unreliability. Providing end-to-end transactional integrity between the browser and the application is important: in the previous example, the cookie *must* be delivered once the user's account has been debited. Cgi-scripts cannot provide this level of transactional integrity since replies sent *after* the transactions have completed may be lost, and replies sent *during* the transaction may need to be revoked if the transaction cannot complete [OSF96]. This is an inherent problem with the original "thin" client model of the Web, where browsers were functionally barren. With the advent of Java it is now possible to consider empowering browsers so that they can fully participate within transactional applications. However, to be widely applicable, we claim that any such transaction system must meet the following three requirements:

(i) it must support distributed, nested transactions;
(ii) it must not compromise the security policy imposed at the browser's site; and,
(iii) it must comply with appropriate standards.

We have designed and implemented the *JavaArjuna* system, a transaction toolkit that meets the above requirements. Our toolkit allows transactional applications to span Web browsers and servers. The toolkit supports application specific customisation - at build time as well as at run time - so that an application can be made transactional without compromising the security policies operational at browsers and servers. Finally, the toolkit complies with the OMG Object Transaction Service (OTS) and the Java Transaction Service (JTS) standards, enabling it to interoperate with other compliant objects and applications. JavaArjuna is unique in that at the time of writing (March 1997), we do not know of any other working system that simultaneously meets all of the above three requirements. A productised version of this system is expected to be commercially available towards the latter half of this year.

2. Transaction standards for distributed objects

The Object Management Group (OMG) has specified a Common Object Request Broker Architecture (CORBA) that at the basic level consists of the Object Request Broker (ORB) that enables distributed objects to interact with each other [OMG95]. At the next level a number of system level services have been specified. These services include persistence, concurrency control, events and transactions etc. We concentrate here on the Object Transaction Service (OTS) standard.

The OTS is a *protocol engine* intended to guarantee that transactional behaviour is obeyed, but it does not directly support all of the transaction properties. As such it depends on other system level services mentioned before for the required functionality. Although the OTS specification allows transactions to be nested, an implementation need not provide this functionality. The OTS provides interfaces that allow multiple distributed objects to cooperate in a transaction such that all objects commit or abort their changes together. However, the OTS does not require all objects to have transactional behaviour. Instead objects can choose not to support transactional operations at all, or to support it for some requests but not others.

The Transaction Service specification distinguishes between recoverable objects and transactional objects. Recoverable objects are those that contain the actual state that may be changed by a transaction and must therefore be informed when the transaction commits or aborts to ensure the consistency of the state changes. In contrast, a simple transactional object need not be a recoverable object if its state is actually implemented using other recoverable objects. A simple transactional object need not take part in the commit protocol used to determine the outcome of the transaction since it does not maintain any state itself, having delegated that responsibility to other recoverable objects which will take part in the commit process.

The Java Transaction Service (JTS) is a direct mapping of the OTS to the Java language using the standard Java IDL mapping [JTS96]. The JTS is *not* a new standard, but an OTS for Java. Therefore, it benefits from the interoperability with other OTS implementations. Applications and objects written using the JTS can make use of objects written in OTS, and vice versa. This presents advantages to application developers who may have legacy backoffice applications, and when building Web applications which encompass browsers and servers they may take advantage of more efficient compilation techniques at the servers.

For example, the figure below shows a transactional Java application executing at a web browser. The application encompasses JTS transactional objects residing within other browsers and OTS transactional objects residing within an ORB. The JTS objects are written in Java, whereas the OTS objects may be written in another language, e.g., C++ or Smalltalk.

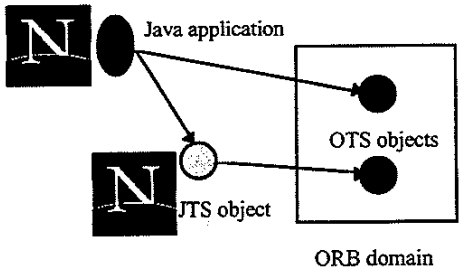


Figure 1: JTS and OTS interaction.

3. Impact of the Java security model

The use of Java to implement transactional applications raises some important security issues. Java security is imposed by a SecurityManager object, which defines what an applet can, and cannot do [ORA96]. Although eventually users should be able to provide their own SecurityManager, currently the Java interpreter provides the only implementation. Generally an applet cannot remotely communicate with a node other than the one from which it was loaded, neither can it write to the disk of the machine on which it is being run. If an applet is loaded directly from the user's disk then many of these restrictions are relaxed.

However, each implementer of a Java interpreter can provide a different security model and SecurityManager, which may impose different constraints. Therefore, an application written on one interpreter may not be able to execute on another. Moreover, the constraints imposed by a SecurityManager directly affect transactional objects which may require for example, to make state updates persistent by accessing the local disk. There are two obvious solutions to this problem:

- only allow local transactional objects to reside within an applet, with other objects and services (e.g., persistence) being accessed remotely.
- modify the Java language and the interpreter and provide a specific implementation of the SecurityManager which, for example, allows all objects to access the local disk [MA96].

Unfortunately, neither of these solutions is general enough. The first solution is unnecessarily restrictive in environments where SecurityManagers do allow applets to access local disks. The second solution lacks portability -the very reason for using Java- as it requires users to have access to specialised implementations.

Our approach, to be described in the next section, does not rely on modifying the language or the interpreter, yet it is flexible enough to enable an application to configure itself to make use of the resources a given SecurityManager permits.

3. JavaArjuna Implementation

3.1. Architecture

The approach we have taken to the problem of writing truly portable Java applications is to build an application support framework, *Gandiva*, which isolates applications and programmers from the differences between Java environments [SMW96]. The application can be dynamically reconfigured to take advantage of the environment in which it executes.

Software components are split into two separate entities: the *interface component* and the *implementation component*. The interactions between implementations can only occur through interfaces. A single interface can be used to access multiple implementations, and a single implementation can be accessed through multiple interfaces. The necessity of providing multiple interfaces to implementations has long been recognised. However, we take this further by allowing the bindings of interfaces to implementations, and the interfaces an implementation can be accessed through, to be dynamic and configurable. Applications are written only in terms of interfaces, and although an application can request a specific implementation, it occurs in a way that allows this request to be changed without modifying the application. Therefore, this allows the application to be adapted for each SecurityManager.

The framework has two components:

- *build-time*: programmers write applications without requiring knowledge about the environments on which they will execute. The build-time isolates the programmer and application from details such as how persistence and concurrency control are implemented.

- *run-time*: when an applet runs, it can be dynamically configured to adapt to the environment in which it executes, for example taking advantage of being able to write to a local disk.

The overall architecture of a JavaArjuna transaction system is shown below. The applet executes both on the Orb and Gandiva framework. The implementations within the applet may also involve the Orb or may bypass it, e.g., a particular persistence or concurrency control mechanism which does not involve remote communication.

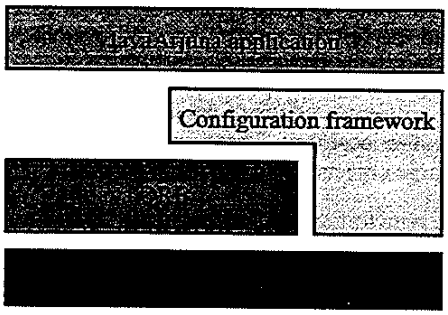


Figure 2: JavaArjuna architecture

3.2 Gandiva build-time support

The Gandiva build-time system offers support to programmers to construct applications from existing interfaces and to build new interfaces and implementations. Interfaces can be automatically generated from a high-level definition language, and contain the necessary code to interact with the run-time system to bind to an appropriate implementation (as described in the next section). To incorporate configurability into an application, the programmer creates a Configuration Management Object (CMO). The CMO contains *data* which specifies the

interface to implementation bindings for the application. The data may also specify alternate implementations, e.g., because of possible security restrictions. At bind time an interface interrogates the CMO to determine which implementation it requires, and then passes this information to the run-time system. Importantly for our purposes, the CMO data associated with an application can be specified at run time, therefore providing a way to configure the application for each user and environment.

3.3 Gandiva run-time support

The run-time framework consists primarily of an Inventory Object (IO), which keeps track of the types of implementations available. When an interface requires to be bound to an implementation, it interrogates the application CMO for the implementation type. It then requests an instance of this type from the IO. If the requested implementation does not exist, or cannot be used within the current environment, then the binding will fail. The interface can then attempt an alternate binding if one is specified by the CMO. Importantly, none of this is visible to the application, which simply attempts to create and use an object.

Figure 3 shows the resultant environment within which a Java application executes. Initially the application contains interfaces which are not bound to particular implementations. When an interface is first used, it consults the CMO residing within the application to determine which implementation class to request from the IO. If such an implementation exists the IO returns an instance of this class which the interface binds to. If the SecurityManager will not allow that implementation and an alternate is specified in the CMO, then one will be chosen until either an implementation if obtained, or no further alternates are available.

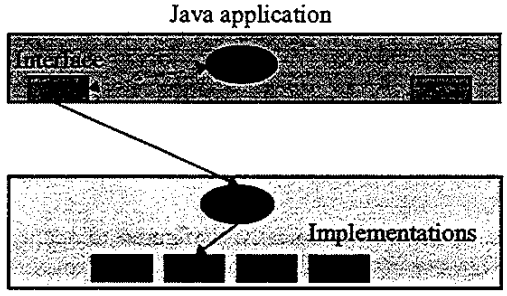


Figure 3: Application executing environment.

3.4 Implementations for transactional applications

For JavaArjuna, we have implemented the following interfaces and associated implementations, which enable a transactional application to execute in any Java-aware browser:

- persistence: in addition to remote implementations of persistence services, there are several implementations which require the ability to access the local disk. These implementations are tailored for specific object types and access patterns, e.g., storing large objects, or concurrently accessing related objects.
- concurrency: as with persistence, remote implementations exist. However, there are several implementations optimised for local use. For example, where sharing of objects between Java applications is not required then only concurrency between individual threads is supported.

3.5 The transactional toolkit

The JTS standard states that the classes defined within it are too low-level for most application programmers, requiring them to manage persistence, concurrency control etc. on behalf of every transactional object. Therefore, a higher-level API should be provided which attempts to hide much of these details from programmers. The API we have provided has been based on the experiences gained from extensive use of the Arjuna system [GDP95]. Distribution support is provided by the Orb on which the application resides. In JavaArjuna objects obtain desired properties through inheritance. The classes form a hierarchy, as depicted below:

```
StateManager                // Basic naming, persistence and recovery control
    LockManager             // Basic two-phase locking concurrency control
        User-Defined Classes
    Lock                    // Standard lock type for multiple readers/single writer
        User-Defined Lock Classes
    AtomicAction            // Implements atomic action control abstraction
    AbstractRecord          // Important utility class
        RecoveryRecord      // handles object recovery
        LockRecord          // handles object locking
        RecordList          // Intentions list
        other management record types
```

3.6 Building transactional applications

The API relieves programmers from having to explicitly register resources with a transaction. Neither do they have to manage persistence or concurrency control, which are managed on their behalf by the JavaArjuna classes `StateManager` and `LockManager`. To make use of atomic actions in an application, instances of the class `AtomicAction` must be declared by the programmer. The operations this class provides (`begin`, `abort`, `commit`) can then be used to start and manipulate atomic actions (including nested actions). The only objects controlled by the resulting atomic actions are those objects which are either instances of JavaArjuna classes or are user-defined classes derived from `LockManager` and hence are members of the hierarchy shown previously. Most JavaArjuna system classes are derived from the base class `StateManager`, which provides primitive facilities necessary for managing persistent and recoverable objects. These facilities include support for the activation and de-activation of objects, and state-based object recovery. Thus, instances of the class `StateManager` are the principal users of the object store service, which is provided through a suitable interface/implementation separation. The class `LockManager` uses the facilities of `StateManager` and provides the concurrency control (two-phase locking in the current implementation) required for implementing the serialisability property of atomic actions.

4 Concluding remarks

This paper has described the design and implementation of JavaArjuna, a standards compliant toolkit for the construction of fault-tolerant Web and Internet applications using atomic actions. The toolkit addresses the requirement for end-to-end transactional guarantees by allowing applications to be built which encompass Web browsers, rather than just Web servers. Transactional objects can reside within Web servers, and interact with objects and applications within other browsers or backoffice environments. As well as being standards compliant, the system does not compromise the security policy imposed at the browser's site. This means that applications can be built without requiring specific security policies, such as being able to write to the local disk. An application can be configured at build-time or run-time to adapt to the environment/user in which it runs, enabling the same application to execute anywhere.

All of the work presented here has been implemented, and simple applications have been constructed to test the system [MCL97]. In the full paper we shall describe the transactional toolkit and its configurable environment in more detail, and provide performance figures.

Selected References

[GDP95] "The Design and Implementation of Arjuna", G.D. Parrington et al, USENIX Computing Systems Journal, Vol. 8., No. 3, Summer 1995, pp. 253-306.

[JTS96] "JTS: A Java Transaction Service API", V. Matena and R. Cattell, Sun Microsystems, December 1996.

[SMW96] "The Design and Implementation of a Framework for Configurable Software", S. M. Wheater and M. C. Little, Proceedings of the 3rd International Workshop on Configurable Distributed Systems, May 1996, pp. 136-143.

[MCL97] M. C. Little, S. K. Shrivastava, S. J. Caughey, and D. B. Ingham, "Constructing Reliable Web Applications Using Atomic Actions", Proceedings of the 6th Web Conference, April 1997.

[ORA96] "Java in a Nutshell", O'Reilly and Associates, Inc., 1996.

[MA96] "Draft Pjava Design 1.2", M. Atkinson et al, Department of Computing Science, University of Glasgow, January 1996.

[OMG95] "CORBAservices: Common Object Services Specification", OMG Document Number 95-3-31, March 1995.

[OSF96] "Applications of the Secure Web Technology in Transaction Processing Systems", T. Sanfilippo and D. Weisman, The Open Group Research Institute, November 1996.

Application Recovery: Closing in on An Elusive Goal

David Lomet
Microsoft Research
Redmond, WA 98052
lomet@microsoft.com

Introduction

Persistent savepoints are the model that has usually dominated previous thinking on application recovery.

> "A *persistent savepoint* is a savepoint where the state of the transaction's resources is made stable ..., and enough control state is saved to stable storage so that on recovery the application can pick up from this point in its execution. ...
>
> If the system should fail and subsequently recover, it can restart any transaction at ... its last persistent savepoint operation. It doesn't have to run an exception handler because the transaction has its state and can simply pick up where it left off."{BeNe97}

The value of persistent savepoints is more than reducing lost work. Persistent savepoints simplify application programming compared to more explicit methods for coping with failures.

> "... the code is not only shorter than the [prior] solution... but simpler. ... everything related to the maintenance of persistent context is now taken care of by the Save_Work function, whereas the [prior] solution had to do the maintenance all by itself..." {GrRe93}

Traditionally, a savepoint has been viewed as the capture of an application's state in stable storage at the time a savepoint operation is executed. However, this is not necessary, any more than it is necessary for database recovery to materialize the final states of all pages of all committed transaction in stable storage. Instead, we recover via replay from the log.

> "Each resource manager participating in the transaction with the persistent savepoint is brought up in the state as of the most recent persistent savepoint. For that to work, the run-time system of the programming language has to be a resource manager, too; consequently, it also recovers its state to the most recent persistent savepoint. Its state includes the local variables, the call stack, and the program counter, the heap, the I/O streams, and so forth." {GrRe93}

Phoenix Goals

At Microsoft Research, we have started a project called ***Phoenix*** -

> "We suggest transactions using persistent savepoints be called *Phoenix transactions*, because they are reborn from the ashes."{GrRe93}

One technology we purse in ***Phoenix*** is application recovery from system failures. We call these *Phoenix* applications. Our intent is to exploit database redo recovery to reestablish application state, e.g. for simple applications like stored procedures. This has a number of benefits-

1. Application recovery increases *application availability*. Current recovery is limited to database state. Work can then resume on the database. But it can take much longer to clean up the "debris" of interrupted applications. Automating their recovery decreases the lengths of these outages.
2. Because we dispense with an explicit notion of a savepoint., the application programmer can forget about system failures as a source of application program error. The application can be written as a simple sequential program that is completely unaware of the crash-
   > "... unless it keeps [a] timer... and finds out that [execution] took surprisingly long to complete." {GrRe93}
3. A user of the application, end user or other software, may likewise be unaware that a crash has occurred, except for perhaps noticing a temporary increase in response time. An end user need not re-enter input data. Client software simply continues to execute.
4. Operations people who deal with and compensate for partially executed applications need only initiate recovery, a process that they are already familiar with for database recovery, and the recovered system state will include the recoverable application state.

The Technology Challenges

There has been prior research directed to achieving the ***Phoenix*** aims. A problem has been that the technology has been expensive in its impact on normal execution. We have addressed how to enable application recovery without serious impact on normal application performance? That is, we reduce the amount of data and control state that has

to be captured and written to stable storage, either to the "database" or to the log.

We treat application program state in the same manner as a database page. An application's state is one of the objects in the database cache, and whose flushing is controlled by the cache manager. As with database pages, we post to the log operations that change application state. Like any good redo scheme, we accumulate changes to an application's state in the cache, and only flush it to stable storage to control the length of the redo log and hence the cost of redo recovery. These flushes will be rare- and sometimes entirely unnecessary, as when the application terminates prior to the need to shorten the log. For example, we anticipate that most stored procedures can execute and complete within a single checkpoint interval- and hence their state need never be materialized on stable storage.

And, how exactly does one log operations for applications without their help? Application state can only be changed in two ways, via interactions with the world outside of itself, and by its own internal execution. We capture both as loggable operations. Since our applications are unaware of our efforts to make them recoverable, we seize upon the times when the application interacts with the rest of the recoverable resource manager (e.g. a database system) to do our logging.

There are a number of ways to log the effects of "normal" execution. For database pages, we can log entire page state or partial page state, or we can log what {GrRe93} calls "physiological" operations, as in ARIES {MHLPS92}. Given the size of log records, we point ourselves toward the later. But, in fact, in order to do really well in keeping log records small, we cannot restrict ourselves to operations that work purely on a single application's state.

Cache management is also impacted by the inclusion of applications as objects being managed. Application state is much larger than a page, and that by itself poses a problem because flushing needs to be atomic. In addition, application operations execute for unknown periods, making it difficult to seize a suitable operation consistent state to flush. Most dramatically, the very application operations that make logging efficient introduce flush order dependencies between objects in the cache.

Finally, application state is not a simple entity, but can be both scattered and complicated. The database system holds several parts of the state outside of the application *program state* (its code, local variables, registers, etc.). We need to capture and log changes for each of the several pieces involved. For example, temporary tables can be logged so as to make them stable. Other state is more intimately held in volatile database system data structures. These need to be captured with application program state and their updates logged. This includes database session state.

Phoenix Technology

This short overview cannot describe all the technology we have in mind. Indeed, **Phoenix** is a research project so some of the technology is in the process of being created. We sketch here our work that shows (i) how an application can be treated as a recoverable *database* object and how its interactions with the system and database objects can be logged; (ii) how this reduces the frequency of application flushing (checkpointing); and (iii) how logging costs are greatly reduced by logging only the identities of objects read and not their data values. This work is more completely described in {Lo97,Lo97a}.

There are two kinds of application operation, one for an application's execution between interactions, and the second for its interactions. Our intent is to replay the application from an earlier state to re-create the pre-crash state. Thus, we must:

- Re-execute an application between interactions. When starting at a state (e.g. at interaction i) encountered during normal execution, we produce the same state (as of interaction i+1) as produced during normal execution. This requires deterministic execution. All non-deterministic behavior is captured as interactions, and logged appropriately.
- Reproduce the effect of each interaction on the application state as of the interaction. We log whatever is necessary to do that. As a simple example, if the application read object X, we could include in the log record the value of X at the time it was originally read. During recovery, when the read is encountered, we use the value from the log record to update the application's input buffers.

We intercept *every* application interaction, as only interactions change its execution trajectory. Our resource manager wraps itself around the application, trapping every one of its external calls. At each call point, it logs the nature of the call and its effect on the application.

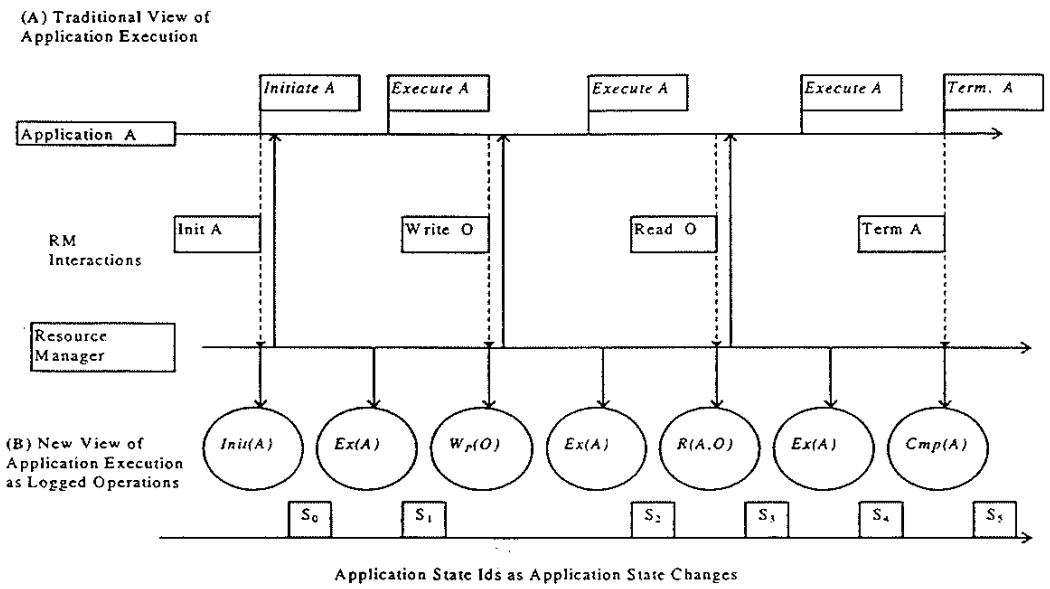


Figure 1

As well, the resource manager logs the application execution between these calls so that the application itself can be re-invoked to replay the transition to the next interaction. The resource manager invokes the operation by returning to the application after its system call, and the operation returns to the resource manager via its next system call. This stands on its head the execution call tree and permits the resource manager to orchestrate the replay of the application. This is illustrated in Figure 1.

Executions between interactions, i.e. Ex(A) operations, are thus logged as "physiological" operations. An Ex(A) reads application state A following interaction I, and transforms it by the execution of the application into the application state prior to interaction i+1, which is "written" by Ex(A). Ex(A) can be logged very compactly since all we need do is name the application that produces the change in state, and whatever parameters were returned to it at interaction i. That is, we replay the return of control to the application.

Logging interactions can be much more costly. The read of X above stored in its log record the entire value of X. If many pages of a multi-megabyte file are read, we have a problem. It is far better to *replay* the read operation during recovery, meaning we log the name of an object, e.g. X, instead of its value. That requires, when the read is replayed, that the object have the original value. The reduction in log record size is potentially enormous, so it is worth going to some effort to make this possible.

Here is the problem. After application A reads object X, X may be updated. Should A's read of X be replayed later, the updated value of X would be returned as the result. But recovery needs to re-create the value of X read by A so that it is available when A's read is replayed. This seeming contradiction can be overcome with sophisticated cache management.

The important observation is that a resource manager has two versions of actively updated objects, a cached volatile current version and a stable earlier version. The stable version enables us to re-create the version of X read by A. To guarantee this, our cache manager ensures that the updated X is not flushed until we no longer need to replay A's read of X. Replaying the read is no longer required when we have flushed a later state of A, or when A terminates. Recovery of A need continue only from the flushed, later state or need not be done (if A is terminated).

The cache manager enforces flush order dependencies on cached objects so that the more powerful operations (e.g. a "read" that reads X and application state A and writes A, i.e. A's input buffers) can be replayed. This requires keeping a flush dependency graph, called a **Write Graph** in {LoTu95}. The cache manager data structures

needed when reads are the only such "logical" operation are quite simple {Lo97}. When writes are treated as logical operations to avoid logging data values written, we must cope with circular dependencies.

Circular dependencies are not just a bookkeeping problem. Naively, all objects involved in a cycle must be flushed atomically together. To avoid this, as before, requires that the right versions are available when needed. By clever slight of hand, these versions can be on the log instead of re-created from earlier stable versions in the database {Lo97a}. We can then "unwind" each circular dependency and flush one object at a time. While one can use the log to effect atomic installation, what we describe in {Lo97a} is not atomic installation but rather permits much more flexible installation.

Discussion

We log application reads/writes of objects owned by the resource manager that manages (provides recovery for) the application as logical operations. This dramatically reduces logging cost for these operations. We needn't log the data values. Cache management is more complex but reduced logging cost outweighs this. This is how we "close in on an elusive goal", i.e. to recover applications with low normal execution cost.

There are some difficulties, however. For example, it can be very difficult to capture the perhaps diffuse state of the application, parts of which are traditionally held in resource manager volatile data structures. One needs to be very careful about the boundary for application state so that the log accurately captures all non-determinism. And one must know exactly what to include in installed application state (checkpoint).

Further, some applications are not easily "wrapped" by a resource manager. Client applications are at arms-length from server resource managers. Our read and write optimizations are not directly possible then. Nonetheless, the Phoenix techniques can contribute to efficient application recovery here as well {LoWe97}.

Not fully captured at the current level of abstraction is how to characterize *all* interactions in terms of reads and writes. Indeed, some interactions are inherently non-replayable even when local, e.g. reading the real-time clock. Is there a substitute for logging the data values that have been read or written?

Distributed applications deal with multiple resource managers. This means partial failures are possible, which are more difficult to handle than monolithic failures. {StYe85} describes application recovery in a distributed system. They do substantial logging and subtle log management involving logs at each site. It is highly desirable to make this activity cheaper and simpler.

Bibliography

{BeNe97} P. Bernstein and E. Newcomer, *Principles of Transaction Processing for the Systems Professional.* Morgan Kaufmann (1997) San Mateo, CA

{GrRe93} J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann (1993) San Mateo, CA

{Lo97} D Lomet, Persistent Applications Using Generalized Redo Recovery. Technical Report, June 1997.

{Lo97a} D. Lomet, Application Recovery with Logical Write Operations. Technical Report, June 1997.

{LoWe97} D. Lomet and G. Weikum, Efficient Transparent Application Recovery in Client-Server Systems. Technical Report 1997 (in preparation).

{LoTu95} D. Lomet and M. Tuttle, Redo recovery from system crashes. VLDB Conf. (Zurich, Switzerland) Sept. 1995

{LoTu97} D. Lomet and M. Tuttle, A Formal Treatment of Redo Recovery with Pragmatic Implications. DEC Cambridge Research Lab Technical Report (in prep)

{MHLPS92} C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. On Database Systems 17,1 (Mar. 1992) 94-162.

{StYe85} R. Strom, and S. Yemini, Optimistic Recovery in Distributed Systems. ACM Trans. On Computer Systems 3,3 (Aug. 1985) 204-226.

Embedded HTTP

http://www3.hursley.ibm.com/sm/embedded.html

Susan Malaika, IBM

Observations

In transaction processing, problems of scale often occur when many people try to access integrated shared resources concurrently. Bottlenecks include the processing power of the computer, operating system process management, and access to storage devices. Solutions include workload management across processors, lightweight threads, data buffering and replication.

On the Inter/Intra/Extranet, additional problems of scale arise when many people try to locate and access autonomous dispersed shared resources across networks. Additional problems and bottlenecks include locating relevant resources in large conglomerations of accessible systems, lack of integration, security, network bandwidth, and distances where the speed of light affects user response times. Solutions include crawlers, robots, indexers, searchers, integration through hyperlinks and middleware, encryption and third party authentication, caching and replication (on the client, near a group of clients, near a server or group of servers, at the server).

Various systems are embedding Internet protocols (such as HTTP), Java Virtual Machines, and ORBs for program to program communication. In a world populated by numerous network connected systems and devices, such as personal servers, real time sensors, coffee makers and cameras, that support Internet protocols, ***what are the additional problems of scale when many more systems and devices with embedded HTTP, ORBs and Java Virtual Machines, communicate with humans and with one another?***

The remainder of these notes outline current ways of accessing systems across the Internet, the use of embedded HTTP and its support in CICS. More information can be found at the Workshop on Embedded Web Technologies at the Sixth International Web Conference in Santa Clara in April 1997.

HTTP Overview

HTTP (HyperText Transfer Protocol) runs over TCP/IP and consists of requests from clients, often with human interfaces, to servers and their corresponding responses. HTTP was first developed in 1990 as part of the World Wide Web project at CERN in Switzerland to provide access from a variety of client systems to a variety of server applications and data. Currently HTTP 1.0/1.1 is used to communicate between Web browsers (and other clients such as PointCast), and various server systems and devices. HTTP includes provision for:

- Content and version negotiation
- GET and POST requests to send user input and to retrieve static or dynamic HTML pages from a server by naming a URL
- PUT requests to update resources on a server
- Request redirection

- Persistent connections with request pipelining
- Efficient cache control mechanisms

Proposals are underway to provide state management across requests within the HTTP protocol which is currently stateless.

Accessing systems across the Internet

Currently, most people who access existing systems across the Internet, do so in the following ways:

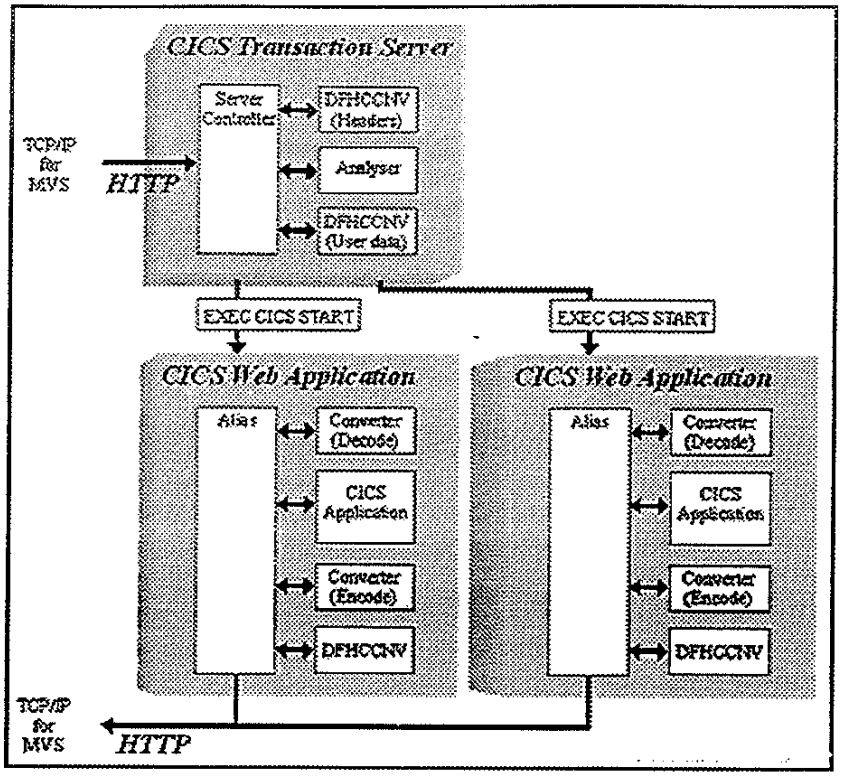
- **Using server side code:** where Web gateway code translates between the language of the Web server, such as CGI or NSAPI, and the language of the target system, such as SQL or IRC. Usually application specific software and/or configuration support is required in the gateway system. Examples of gateway software includeNet.Data for accessing databases, files and applications, MQSeries Internet Gateway, and the CICS Internet Gateway
- **Using client side code:** where a Java applet (client) is downloaded from a Web server through HTTP, which in turn communicates directly with the target system typically using the existing communication protocol of the target system such as the MQSeries Client for Java. Some Java clients such as the CICS Client for Java and the DE-Light Client for DCE Encina communicate with an intermediate gateway that resides between the Java client and the target system. Usually application specific code is required on the client which is downloaded as part of the applet.
- **Using embedded HTTP:** where the target system supports the HTTP protocol directly and maps the protocol to its own interfaces. Here are some examples of systems that support HTTP:
  1. Systems that use embeddable HTTP server software such as MicroServer from SpyGlass, ProWeb from 3Soft, and Jigsaw from W3C.
  2. Devices that incorporate embedded operating systems such as Embedded Internet for pSOSystem from Integrated Systems.
  3. Computer devices such as some Tektronix printers and Cisco Routers.
  4. Groupware such as Lotus Domino and broadcast systems such as PointCast.
  5. Transaction processing and database systems such as CICS.

CICS

CICS was first developed in the late 1960s. It runs on mainframes and other platforms such as Windows/NT and AIX. CICS's main function is to run applications efficiently on behalf of large numbers of concurrent users who generally perform repetitive short predictable activities such as withdrawing money from bank accounts or borrowing books from libraries. Often persistent resources such as files and databases are updated.

Native embedded HTTP support became available in mainframe CICS in 1996, as an alternative to We server gateways and Java based clients that already exist for CICS applications. Thus, it is possible to access CICS applications directly from a Web browser without an intermediate gateway, and without writing application specific code on a client or gateway.

The CICS Web Interface (Embedded HTTP in CICS)



The CICS Web interface provides access to CICS applications from Web browsers without requiring an intermediate server or gateway. Regular Web proxies, gateways, and firewalls can be placed between the browsers and the CICS systems. Here is a sample URL format that could be used to access a CICS application: **http://www.any.com/CICS/transid/programid?params** where

- **www.any.com** is the name of the mainframe system. Multiple HTTP servers on a single system can coexist using different ports.
- **CICS** is a convention to confirm that the request is intended for the CICS Web Interface.
- **transid** names the CICS transaction code to be used which governs security and performance etc.
- **programid** identifies the CICS application program.
- **params** usually include the input variables from the user which are then placed in the CICS application parameter area known as the COMMAREA.

Here are some of the components (also illustrated above) of the CICS Web Interface:

- **The analyzer** determines the format of the path. It receives the URL as input and selects the transid and userid for the CICS START (a way of initiating a CICS lightweight thread to run a transaction). There is one analyzer for the whole CICS system.
- **The decode routine** can be application specific. It receives the user input from the browser and creates the CICS COMMAREA for the application. It can also select the application program name.
- **The application** takes the output from the decode routine, which will very likely include the user input and creates the appropriate output HTML page.
- **The encode routine** takes the output from the application and adds the appropriate headers, e.g., HTTP.

- **HTML template routines** are supplied that can be used from the decode, application or encode routines, enabling the application to be unaware of the details of HTML.
- Routines are supplied to help with **user session management** across requests using CICS temporary storage or variables memory.

Links and References

- http://www-irl.iona.com/PR/:
  Object Requests Brokers from Iona
- http://www.sun.com/sparc/java/:
  Java Chips from Sun
- http://www.w3.org/pub/WWW/Protocols/Overview.html:
  W3C HTTP Overview
- http://www.ics.uci.edu/pub/ietf/http/:
  The IETF HTTP working group
- http://lcweb.loc.gov/global/internet/html.htm:
  HyperText Markup Language at the Library of Congress
- http://www.w3.org/pub/WWW/Addressing/URL/:
  URL (Universal Resource Locator)
- http://www.ibm.com/technology/books/webgate/:
  Web Gateway Tools, Wiley 1997
- http://hoohoo.ncsa.uiuc.edu/cgi/:
  CGI(Common Gateway Interface)
- http://home15.netscape.com/newsref/std/nsapi_vs_cgi.htm:
  NSAPI (Netscape Server Application Programming Interface) versus CGI
- http://www.jcc.com/sql_stnd.html:
  SQL Standards
- http://www.cs.hope.edu/~hahnfld/quikchat/:
  QuikChat - IRC (Internet Relay Chat) gateway
- http://www.software.ibm.com/data/net.data/:
  Net.Data
- http://www.hursley.ibm.com/mqseries/platforms:
  The MQSeries Internet Gateway and the MQSeries Client for Java
- http://www.hursley.ibm.com/cics/internet:
  CICS Internet Support
- http://www.hursley.ibm.com/cics/internet/cicsgw4j/index.htm:
  The CICS Gateway for Java
- http://www.transarc.com/afs/transarc.com/public/www/Public/ProdServ/Product/DELight:
  DE-Light Client for DCE Encina
- http://www.dreisoft.de/:
  ProWeb from 3Soft.
- http://www.spyglass.com/products/microserver/:
  MicroServer from SpyGlass
- http://www.w3.org/pub/WWW/Jigsaw/:
  Jigsaw from W3C
- http://www.isi.com/Products/pSOS/embed.html:
  Embedded Internet for pSOSystem from Integrated Systems
- http://www.tek.com/Color_Printers/products/phaserlink.html:

PhaserLink software for Tektronix printers

- http://www.cisco.com/warp/public/732/clickstart/click_pa.htn: ClickStart for Cisco Routers
- http://pioneer.pointcast.com/products/iserver/techp.html: PointCast Technical Papers
- http://diagnostics.stanford.edu/www6/EWTPositionPaper.html: Workshop on Embedded Web Technologies at the Sixth International Web Conference

---

Susan Malaika, malaika@vnet.ibm.com
phone: **+1 (408) 463 2522**, fax: **+1 (408) 463 3834**
IBM Santa Teresa (DRYA, C376), 555 Bailey Avenue, San Jose, CA 96141, US
Date Updated: *17 April 1997*

Enterprise JavaBeans: Extending the JavaBeans component model to scalable three-tier applications

Vlada Matena, Mark Hapner, Rick Cattell, Shel Finkelstein,
and Graham Hamilton, Sun Microsystems Inc.

Introduction

This extended abstract provides a high-level overview of the Enterprise JavaBeans architecture. At the workshop, we will present a more detailed description of the architecture, with the focus on the transaction and state management. We will also discuss several key design decisions.

Overview of JavaBeans

JavaBeans [http://splash.javasoft.com/beans] is the component model for applications written in Java. Components are called *beans*. JavaBeans defines the standard that allows a bean written by one developer to be easily reused and customized by another developer without the latter developer having access to the bean's source.

JavaBeans has become the de facto standard for building applications in Java. A number of application builder tools provide support for JavaBeans. Applications built using JavaBeans are portable across many platforms that support Java.

JavaBeans does not define a standard for distributed applications. While it is possible to use JavaBeans to build two-tier enterprise applications, the architecture does not provide direct support for building three-tier applications. Although it is possible to develop three-tier applications using JavaBeans, the application programmer has to understand system-level concepts, such as state management and security, in order to develop a three-tier application.

Enterprise JavaBeans

The goal of Enterprise JavaBeans is to extend JavaBeans to support the development and deployment of three-tier (or multi-tier) applications.

While preserving the key advantages of JavaBeans (simple component model, customization, application portability, tools support), Enterprise JavaBeans adds support for the features required for the development of three-tier enterprise-level applications. Enterprise

JavaBeans defines support for the following:

- Client/server distribution
- Transactions
- Scalable state management
- Security
- Support for deployment.

The Enterprise JavaBeans APIs can be divided into two parts. One part defines the APIs used at deployment time to introspect an application and discover the various meta-data such as security attributes. The other part defines the interface that bind the application code with the underlying platform at runtime.

JavaBeans and Enterprise JavaBeans together provide a full set of capabilities for development of three-tier enterprise applications in Java.

Server bean

A server bean is one type of enterprise bean. A server bean implements business logic that executes synchronously with a client program. The client invokes a server bean using Java method invocation. If a client executes in a transaction scope, a server bean can execute in the same or new transaction scope.

Server bean is currently the only type of enterprise bean supported. Future versions of Enterprise JavaBeans may define other types of enterprise bean. For example, asynchronous messaging can be implemented using enterprise beans that are activated when a message arrives on a queue.

Beans transaction server

A beans transaction server (transaction server for short) is any container that is capable of hosting enterprise beans at runtime. While a transaction server can be built from scratch in Java, it is possible to provide a transaction server as an extension of an existing TP monitor, database management system, or any other server platform.

We have worked with the leading vendors of transaction processing platforms to ensure that the protocols for transaction and state management in Enterprise JavaBeans are compatible with those used in their existing platforms. Therefore, we believe that it is practical for a vendor to provide a transaction server as a thin Java layer on top of an existing platform.

Support for deployment

Enterprise JavaBeans defines the *enterprise beans package* as a standard way for packaging enterprise beans for deployment. Since an enterprise beans package will be understood by all compliant transaction servers, a packaged application can be deployed on any compliant platform.

An enterprise beans package augments the standard Java Archive (JAR) file format with a standard for passing deployment time information. The deployment time information is formatted as beans (i.e. serialized Java objects) to take advantage of existing tools.

The deployment time information includes attributes that control transaction scopes, transaction timeouts, resource quotas, security attributes, etc.

Distributed programming

Enterprise JavaBeans provides flexible support for client/server distribution. Enterprise JavaBeans does not depend on or mandate using any particular communication protocol.

An application programmer writes local Java code both for the client and the server side of an application. The communications stubs that bind the application with the underlying communication infrastructure are generated automatically at deployment time. The application programmer does not have to use an Interface Definition Language (IDL) to allow generation of communication stubs. Rather, the communication stubs are generated using Java language introspection of the application class files.

The programmer must follow certain restrictions on the types of method arguments and results. The restrictions on method parameter types were chosen to allow straightforward mapping of the Java method signatures to the commonly used communication protocols (IIOP, DCOM).

Transaction and state management

Easy-to-use support for transactions and scalable state management is a key goal of Enterprise JavaBeans.

A server bean developer can choose the state management style that best fits the application. The supported state management styles are:

- Method-duration
- Transaction-duration
- Object-duration.

The selection of the style is done through a declarative property of the server bean. The programmer does not use any API calls to manage the state of a component. The underlying transaction server automatically activates and deactivates an instance of a server bean based on the server bean's declarative property.

Server beans that have non-trivial set-up overhead may be developed as *serially reusable*. A transaction server is allowed (but not required) to pool instances of a serially reusable component.

Instances are not allowed to share data directly. Enterprise JavaBeans defines a *shared properties* mechanism to allow for managed sharing of data among instances of the same server bean. Shared properties have transactional isolation semantics. Transactions on shared properties are bracketed by the duration of a method invocation on the component.

Transactions

The transaction and state management protocol was designed to allow application developers to easily develop a server bean such that it can be used both as a top-level component (i.e. the component initiates a transaction) and as a nested component.

There are two API calls through which a component controls transaction management:

- commitWork
- rollbackWork

CommitWork tells a transaction server that the instance is in a consistent state. A transaction server attempts to commit the transaction if the instance is the transaction initiator.

RollbackWork marks the transaction as "rollback-only". A rollback-only transaction can never commit.

More sophisticated server beans can register a *Synchronization* callback interface. If an instance registers a Synchronization interface, a transaction server calls the instance before and after the completion of the transaction. Synchronization allows a component to vote on the outcome of a transaction, write any cached data to a resource manager, and to be notified of the outcome of a transaction.

Security

We expect that the majority of server beans will be developed as *security-unaware*. Security checking for security-unaware applications is performed outside of the

application. The deployment information included as part of an enterprise beans package may include an access control list and an effective runtime identity for each server bean. Access may be controlled at the level of individual methods.

Enterprise JavaBeans provides an API that allows *security-aware* server beans to access security information such as the identity of the client, or to change the identity associated with the runtime execution of a server bean instance.

Customizable components

A unique feature of Enterprise JavaBeans is the support for component customization. It is possible to develop a generic business component that can be customized at later stages of the development/deployment workflow without having access to the source code of the component.

A server bean is a serialized Java object that can be instantiated and manipulated at design time. Using the JavaBeans BeanInfo mechanism, a server bean developer can specify *design time properties*. Design time properties can be modified using visual tools at later stages of the workflow.

A developer may provide a visual bean editor with a more complex server bean. A bean editor controls how a server bean is manipulated at the various stages of the development workflow.

The JavaBeans architecture makes it possible for builder tools from multiple vendors to be used at different stages of the development workflow.

Acknowledgments

Enterprise JavaBeans is a broad effort that includes contribution from numerous groups at Sun and at partner companies. The ongoing review process of the specification has been extremely valuable and the many comments that we have received helped to shape the specification.

Database Systems as a Reflection of the Changing Nature of
High Performance Transaction Processing Systems

John McPherson

IBM Toronto Laboratory
North York, Ontario
johnmc@torolab.vnet.ibm.com

Introduction

The nature of high performance transaction processing systems (HPTS's) has evolved significantly. These systems have changed from very specialized, centralized systems implemented as specialized code into distributed systems built with very high function building blocks. Central to these systems is the need for concurrent processing of many transactions acting on data. Data is at the heart of these systems and as such, a database system is usually at the heart of these systems.

The nature of the underlying processor architectures, network infrastructure, applications, and supporting products has changed significantly over time. The success of database systems as demonstrated by the fast and steady growth of database management system revenue has been possible because database products have evolved to exploit the changes in hardware and software infrastructure, and to incorporate, over time, many of the functions that originate as complementary products or applications. This position paper will briefly review some of the trends we've seen in database systems and some of the challenges database systems face.

Trends and Challenges

Because data is fundamental to high performance transaction systems, the management of data is fundamental to these system. Supporting the ACID properties (Atomicity, Consistency, Isolation, and Durability) is a hallmark of these systems and the need to support these properties really has not changed. What has evolved is the way in which this is done. Concurrency control and recovery protocols have evolved, and the utilities for such activities as backup, restore, reorganization, load, unload, replication and so on have slowly evolved from serial, batch utilities to utilities that exploit parallel architectures, are incremental, and are done concurrently with normal processing. There is great variance in commercial database management systems in their support for true 7 x 24 operation, and continued focus is needed. The real challenge, however, is to provide self-manageability of all aspects of the system.

Clearly high performance is a requirement for HPTS's - it's in the name! As the nature of the applications changes, the nature of the performance enhancements has and will change. New indexing structures such as bit mapped indexes are just one form of enhancement we are seeing in general purpose database management systems as they expand to support new kinds of systems. Exploitation of parallel hardware continues to be a focus, with SMP's with larger number of CPU's and support for various flavors of NUMA architectures. Support for clusters of SMP's is also important, with clusters of PC's on the horizon as a viable HPTS platform. Parallel query processing techniques, manageability of these complex systems, and exploitation of the inherent redundancy of clusters of SMP's for availability are where the current challenges are.

Today's HPTS's are generally very complex distributed systems, with multi-tiered architectures for not only the operational data, but also for the analyis and data warehousing functions. Application logic is spread across all tiers including being pushed into the database itself in the form of triggers, user defined functions, and constraints. Support for application logic in the database has several advantages including providing centralized control of business rules and performance. Performance advantages can be realized because the application logic is applied closer to the data, and interestingly, in parallel systems because it enables parallel execution of application logic. An example is the execution of triggers or constaints in parallel in the database. These are not simple systems to write application logic for, however. A challenge is to develop application development environments that permit easy partitioning of application logic and debugging support for applica-

tions whose logic is spread across multiple systems. One should point out that support for stored procedures and user defined functions written in common application development languages such as C, Java, or Basic, instead of proprietary vendor languages help make development of such applications easier.

A related force is the need to integrate data from multiple data sources. This requires support for heterogenous data access, distributed transaction management, security services, directory services and other distributed system services. Many vendors are in the process of providing architectures and products supporting componentized environments. The challenge for database vendors will be to ensure they are not relegated to just a role as one of the data resource managers, but that they evolve to subsume much of the higher level middleware function.

Commercial database management systems are evolving to support new kinds of applications with the support for greater diversity of data types and functions. The ability to easily model and support new applications is the commercial driving force behind object relational systems. Not only is there opportunity for application developers to add support for their own peculiar data types and function, but there is an opportunity for database vendors and third-party software developers to develop plug-in functionality that can be sold or given away to enhance the database functionality. There are several challenges including continued support for well-integrated object relational features which support increased modeling capability. Support for underlying performance enhancements such as specialized indexing structures for new kinds of data, query rewrite technology, and query optimization technology to maximize processing efficiencies of queries supporting this new modeling capability is also necessary.

Database management systems in HPTS's need to do more than just store data and support high numbers of updates and reads against the database. Database management systems must also provide the ability to analyze and get business information out of the data they hold. OLAP support is being integrated into most major commercial database management systems in various forms. Support ranges from support for CUBE and ROLLUP operations, to support for specialized join processing and indexing structures. OLAP is just one form of processing that is important, however. Other forms of analysis and data mining are being supported by some database vendors, and this is adding a very important new dimension to the power of database management systems. The ability to mine the data managed by these systems is a very powerful feature. Data analysis capabilities will continue to evolve and will be embedded in database management systems over time.

The incredible growth of the internet and related technologies has driven many changes in database management systems. Not only does the internet present a new infrastructure for client-server applications, it has also driven new language (JAVA), and a whole set of new distributed application interfaces and services including JDBC and Java Transaction Services (JTS). Database management system vendors have responded with many products supporting web-based application development and web-related technologies and this will continue. This is really just supporting the infrastructure provided by the internet technology explosion. One of the biggest challenges is not related so much to the internet infrastructure and technologies as it is with the incredibly complex data the internet provides access to. It is not simply an incredibly complex web of data interconnected by URL's. It is an incredibly complex interconnection of "databases" with very diverse interfaces and content, all interconnected with URL's. There is no universal data model used and one of the big challenges is to make the data accessible on the internet easily exploitable by database applications.

Conclusions

There are of course many other trends and challenges. Workflow, queuing, data warehousing issues such as data cleansing are just a few that have not been discussed here. If commercial database management systems are going to continue to increase in demand and increase their market value, they are going to have to respond to these trends and challenges. If they do not, other products will. There is tremendous opportunity in this rapidly changing field, both for increasing market share and for decreasng market share. The winners are those that respond the fastest to these evolving trends and market needs.

Data Management Issues in Biotechnology

Michael A. Olson
Molecular Applications Group
mao@mag.com

1. Introduction

Over the past decade, a new area of scientific specialization has arisen. This new area is often referred to as "biotechnology" or "bioinformatics." In either case, the distinguishing characteristics of the new area include its focus on DNA and protein sequences in understanding diseases, and its marriage of computational techniques with conventional laboratory work to deduce facts about the sequences and, ultimately, how they work.

Biologists working in this field are involved in very important, and very exciting, research. Most current methods of treating disease deal with symptoms, rather than with the disease itself. For example, cancer treatments based on radiation and chemotherapy are aimed at destroying tumors after they have formed. They do not address the causes of cancer, and do not eliminate the possibility of its recurrence.

Genetic researchers are searching for the link between the DNA sequence that makes up the human genome and diseases like cancer. Once this link is established, investigators can explore the process that results in the outbreak of cancer in healthy individuals, and can look for ways to interfere with that process. This much less invasive procedure can genuinely prevent the outbreak of disease, rather than merely treating its symptoms.

Genetic research requires the storage and manipulation of complex information on computer systems. DNA is a double helix of complementary bases chemically bound into very long chains. These chains are represented on computer systems as long strings composed of four letters, representing the four bases from which sequences are formed. These strings have rich internal structure, as well as physical structure in three dimensions. Both varieties of structure must be manipulated algorithmically.

In addition to its complexity, data used by biotechnologists is large. DNA and protein sequences are very long. Single strands millions of bases long must be manipulated as single values. There are many such sequences to store, and the number is increasing rapidly. A single laboratory can produce upwards of 40,000 new sequences a week, and must use computer systems to sift through them for interesting research candidates.

Researchers working in biotechnology have adopted a variety of data management techniques to solve problems over the years. No standards have ever been enforced, and systems to store and deliver biological data have often been designed by working biologists, rather than by data management experts.

Predictably, these systems were effective at solving the short-term problems confronting their inventors, but have not aged gracefully. Common problems for existing systems include size constraints, poor support for queries, an inability to support multiple concurrent users, and a propensity to lose data to system crashes or file corruption.

Despite pronounced interest, no single technology has emerged for storing biological data or for delivering it to applications. An enormous market awaits the first successful developer of a robust, standard system. However, there are serious technical issues that must be solved.

This paper focuses on some of those issues.

2. Data Management Issues in Biotechnology

Broadly speaking, the issues confronting the biotechnology industry are similar to those already solved by the database community for others. Federated databases, content queries, scalability, and concurrency have all been implemented and are in production in other markets. In the details, however, the biotechnologists have a different problem from that confronted by other database users.

The core difference is due to the nature of scientific inquiry itself. Biotechnology is, fundamentally, the storage and use of information critical to the scientific process of disease characterization and drug discovery.

That science changes rapidly. Today's best practices are improved upon, and occasionally repealed, by tomorrow's discoveries. As a result, pieces of knowledge stored in the database become more or less certain over time. Most extant database applications treat the database as a repository of facts. Database applications extract these facts, consider them, and act on them. In general, facts never become falsehoods. A sale is always a sale; a circuit diagram remains a circuit diagram; a video stays a video. In contrast, a gene may not actually be a gene.

Any data management system for biotechnology must be robust to imprecision. More importantly, the data management system must allow users to add new scientific techniques to their arsenal as those techniques are discovered. The new techniques must coexist with older ones, even if some of them are mutually contradictory. Scientists are used to believing several things simultaneously, and they expect the same of their computers.

2.1 The Status Quo

Existing biotechnology databases are broken into two major groups, with a small number of outliers. By far the largest volume of biological data is stored in flat files, in formats defined by individuals in most cases and consortia in a few. These flat files usually store sequences in a format optimized for search and retrieval, but other kinds of information are also stored in files. Flat files typically do not support incremental updates well, and in general do not provide concurrent access to readers and writers.

The second most significant repository for biological data is relational databases. Sybase and Oracle are (in order) the market leaders, with a small number of DB2 and Informix databases in production use. Most of the large pharmaceutical companies have installed relational databases, because they have the budget to administer them.

A very small number of databases in production use are object databases. Some of the software vendors in the space have based new products on object databases. These products have so far not seen broad market acceptance, based partly on customer reluctance to bring another database system in house, and partly on their novelty.

The databases that do exist tend to specialize in providing a particular search service. For example, several of the flat file databases let users search for sequences that match a target more or less exactly, using well-known matching and scoring algorithms. Others deliver sequences and everything known about them, but only if the user specifies a name for the sequence.

Some of the databases are publicly administered, accept submissions from the community at large, and are available to anyone for searches. Others are proprietary, and are developed by a particular research group or company for its own use. Public databases typically contain some redundant and poor-quality data. So do many proprietary databases, though the private repositories are usually of higher quality than the public ones.

The challenge confronting the biotechnology industry is finding important information in this collection of databases. At present, users generally search for matching sequences when they get a new sequence, and look up each of the resulting sequences by name in all the public and proprietary databases available to see if anything can be deduced about the new sequence. Integrating the information, running the query, and resolving referential integrity problems and contradictions in the results are major problems.

2.2 *Federated Databases*

The problem of integrating all the existing databases into a single logical store for biological information is really just the federated database problem, solved with more or less success by database vendors for other customers over the years.

There are about fifty widely available databases of information in use by the biotechnology community, with new ones being added every few months. Of these, perhaps ten are in wide use. The databases typically assign different semantics to similarly-named attributes, sometimes include references among themselves, and overlap imperfectly in the information that they capture.

Commercially-available federated database systems could probably go a fair distance toward integrating these databases. However, there are some important core problems that aren't addressed simply by federating the existing databases into a single large schema.

First, the existing databases themselves are not very good. Much of the data that they contain is of dubious value, and the scientific community would really like to have the databases cleaned up before much more research is based on their contents.

Second, the existing databases are approaching the edge in their ability to store and find information. They were designed to store a few thousand sequences; at present, that many new sequences is generated every day by laboratories specializing in sequence production. To some extent, this problem is due to a triumph in the sequencing of the human genome. The early predictions of how fast the community would be able to produce sequences were off by more than a factor of 100, due to very rapid innovation in sequencing technology that produced at least one Nobel prize and several successful companies.

Finally, federation must allow the schema to evolve. New information is being discovered about sequences all the time, and scientists want to store it with all the information they have already captured. Extending a federated schema in a way consistent with the semantics and assumptions that underlie all its constituent databases is an open research problem.

2.3 *Support for Interesting Queries*

Most existing databases in use by biotechnologists store very complicated data in a very simple data model. DNA and protein sequences are rich in internal and physical structure. Almost all databases that store sequences store them as opaque, very long, text fields, totally ignoring structure. Generally, some textual fields are attached to the sequences, and users are able to search for sequences based on names and keywords stored in the text fields.

Some exceptions to the rule exist. For example, several databases provide a service called *homology search*. Homology is the term used by biologists to describe meaningful similarity. Computer scientists generally think of it as a form of string matching, but it is actually more subtle. For example, the match may be on three-dimensional structure, which is not represented directly in the string of bases or proteins, and collections of bases or proteins may substitute for others under certain conditions, with a penalty in the precision of the match.

The databases that provide homology searches provide no other mechanism for looking up sequences. The only way to do a query is to start with a sequence. The only result is a list of homologous sequences, with a score attached to each that captures its similarity to the query sequence.

Scientists have discovered that they need to search all their databases based on content, and that they need to use *ad hoc* query techniques that allow them to combine predicates in new ways after the database is created.

Again, *ad hoc* queries are generally well-supported in most commercial data management systems. The problem in applying these to biotechnology is that they do not capture the data model in use by the scientists. For example, homology searching is critical in sequence databases. Any new database system must permit the definition of new data types and operations to support new searching techniques.

Worse, the existing databases, because they were designed when the number of sequences was small, do linear searches to find homologous sequences during queries. With the number of sequences exploding, better techniques, including indexed access methods, must be invented. This is an open research problem, and one that is critical to solve.

2.4 Genomic Data Mining

The goal of most biotechnologists is simple: Using computers and the laboratory, search an enormous number of sequences for links to disease, and understand how the link works.

The goal is much easier to state than to achieve. In its pursuit, biologists want to apply techniques analogous to data mining on conventional databases. The analytical techniques that support the mining techniques are different from those used in retail or telecommunications, but their purpose is the same. They assist intelligent users in reducing their search space, and provide hypotheses for testing outside the computer system.

The main problem with data mining, as applied directly to biotechnology, is that so little biological data is numeric. Most data mining tools work well on numeric or character data, but no so well on more complicated data types like DNA sequences or three-dimensional structures. Histograms, analysis of variance, clustering, and other techniques may well be interesting in biotechnology, but it is not clear how to apply them to biological data types.

3. Summary

Biotechnologists have discovered that they need to use database systems to further research. To do so, they must solve a number of difficult problems.

There is a rich set of existing tools which are useful to the scientists and so will never go away. These existing tools – databases and analytical engines – comprise a legacy with which new data management tools must interoperate.

The data volumes, need for concurrent access, and rapidly changing data model tax many existing data management systems.

The scientists operate on enormously complex data values, and want to issue *ad hoc* queries against their databases with simple graphical tools.

The final goal of biotechnologists is to understand disease formation, and need new ideas and algorithms to do that.

Isolation Level Testing

Position Paper for HPTS-97
September 7-10, 1997
Pat O'Neil
UMass/Boston
poneil@cs.umb.edu

There has been a good deal of activity in the area of Isolation Levels in the past few years, and I feel there is a potential for making new advances in this area, with a commercial impact.

In 1995, a group of us (Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Betty O'Neil, and Pat O'Neil) published a paper we called "ANSI ISOLATION" in SIGMOD, [BBGMOO95], that demonstrated numerous flaws in the "Phenomena" method of Isolation Level definition. Specifically, we demonstrated that the ANSI Isolation definition was wrong in a number of ways. This paper also introduced a new concurrency method we called "Snapshot Isolation", that provided a method for reads to never block, reading snapshot data from the Before Image log. What was new about this is the EVEN READS PERFORMED BY UPDATE TRANSACTIONS would read snapshot data. Here is how this works.

A transaction executing under *Snapshot Isolation* always reads data from a *snapshot* of the (committed) data at the time the transaction started, called its *Start-Timestamp.* A transaction running in Snapshot Isolation is never blocked attempting a read as long as the snapshot data from its Start-Timestamp can be maintained. The transaction's writes (updates, inserts, and deletes) will also be reflected in this snapshot, to be read again if the transaction accesses the data a second time. Updates by other transactions active after the transaction Start-Timestamp are invisible to the transaction.

When the transaction T1 is ready to commit, it gets a *Commit-Timestamp,* which is larger than any existing Start-Timestamp or Commit-Timestamp. The transaction successfully commits only if no other transaction T2, with a Commit-Timestamp in T1's interval [*Start-Timestamp, Commit-Timestamp*], wrote data that T1 also wrote. Otherwise, T1 will abort. This feature, called the *First-committer-wins* policy, prevents lost updates caused by two different transactions making different updates of a piece of data based on the same original version.

Of course Snapshot Isolation is not perfectly serializable in the classical sense, since transactions can read data at one instant and write data at a later instant. There must be ways in which non-commutative interactions between transactions can occur. In fact, there are two that we were able to find.

Example 1. Write Skew. Consider a transaction that is meant to maintain a constraint, that two bank balances x and y (for husband and wife) should sum to non-negative number (i.e., $x + y \geq 0$). But consider the single-value history:

H1: r1[x=50] r1[y=50] r2[x=50] r2[y=50] w1[y=-40] w2[x=-40] c1 c2

H1 has the same inter-transactional dataflows as could occur under Snapshot Isolation (there is no choice of versions read by the transactions). While T1 and T2 both act properly in isolation to ensure that $x + y \geq 0$, the constraint fails to hold in H2. There is a problem simply because the two transactions are updating different things (thus First-committer-wins doesn't stop the updates) and each depending on the other thing updated remaining the same.

Example 2. Predicate Write Skew. A second example is basically the same idea, except with a predicate constraint. Say we have a table of employee_task_assignments, with columns including empid, task_day, and hours_load, and there is a restriction that the sum of task hours

for an employee in a given day cannot exceed 8 hours. But two transactions can each read the sum of tasks for the same employee on the same day as a sum:

```
select sum(hours_load) from employee_task_assignments
   where empid = :eno and task_day = :date;
```

. . . determine there are only 6 hours of load on this day, both of them add a 2 hour task, and thus cause non-serializable behavior. Note that Snapshot Isolation has a First-Committer-Wins policy even for index data, but here there are new index entries being created so there is no conflict on the same data.

Problems such as Write Skew and Predicate Write Skew seem to be uncommon in practice. Snapshot Isolation was implemented by Oracle Corporation and used in the TPC-C benchmark [TPCC, TPCC1]. Francois Raab and Tom Sawyer were both called on to validate that there was no concurrency problem in the TPC-C benchmark application that would arise under Snapshot Isolation (this is the only criterion that needs to be obeyed). When you think about it, the fact that Snapshot Isolation executes the TPC-C application in a serializable fashion is an important illustration of how useful it is as an isolation level. Snapshot Isolation is currently offered by ORACLE as an isolation level, called SERIALIZABLE, in Release 7.3 [OR7.3, OR95].

As we were working on the ANSI ISOLATION paper, it became clear that there were a number of areas of rigorous treatment that were inadequate in investigating isolation levels involving multi-version concurrency. There were even a number of questions about more traditional isolation levels that we found difficult to resolve, such as this:

- What is the definition of an isolation level? When are two implemented isolation levels equivalent? (E.g., is page-level locking equivalent to row-level locking?)
- What does it mean for one isolation level to be "weaker" than another?
- How can we determine if one isolation level is more useful than another? (I.e., better performing, or likely to cause fewer concurrency errors in practice.)
- How can we determine when a method of implementing an isolation level is error free?

But I want to concentrate in the remainder of this position paper on a particular area of research in this field, which I call *Isolation Level Testing.* Isolation Level Testing is supposed to give us a utility to answer the following question:

- How can we tell if an application program will run error-free under a given isolation level?

The motivation for this problem comes from the question that was asked of Francois Raab and Tom Sawyer: Given the TPC-C application, will it run error-free (without isolation errors) under Snapshot Isolation? The only way Francois and Tom knew to answer this question was to think about it very hard for many hours. THERE WAS NO KNOWN THEORETICAL RESULT THEY COULD APPLY TO ANSWER THE QUESTION.

Now if Francois and Tom have this problem, imagine the problems faced by a DBA with a large application, trying to decide if any serious errors will crop up as a result of running under Cursor Stability instead of Repeatable Read! The IBM manuals are no help. They essentially give the advice that Cursor Stability offers better concurrency, but BE CAREFUL!

I believe I have the beginnings of an approach to create a utility to list the isolation errors that can arise for a given database application under various isolation levels. Given such a list, there are workarounds in most applications to avoid these isolation errors. For example, in Example 1, we can make a rule that any constraint on a number of balances such as this has a materialized sum of balances which must be updated initially in making a change to any balance. Such an approach doesn't work where there are ad-hoc updates that can interact, but many problems of this kind can be fixed if we can list them for the applications writers.

I am writing this in March, the middle of my school term, and will not be able to give too much time to this research until late May. Hopefully, by the time HPTS occurs in September, I will have more I can say.

References.

[BBGMOO95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil, "A Critique of ANSI SQL Isolation Levels", Proc. ACM SIGMOD 1995, pp. 1-10.

[OR7.3] Oracle 7 Server Application Developer's Guide, Release 7.3, Part No. A32536-1, Chapter 3, Section on READ COMMITTED and SERIALIZABLE Isolation.

[OR95] Ken Jacobs, Roger Bamford, Greg Doherty, Karl Haas, Merrill Holt, Franco Putzolu, Brian Quigley, "Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7," Oracle White Paper, Part No. A33745.

[TPCC1] TPC-C Individual Results, http://www.tpc.org/execsum_TPCC.html.

[TPCC] TPC-C Benchmark Delivers 11,456 tpmC, Oracle Press Release December, 1996, http://www.oracle.com/corporate/press/html/tpc2.html, Executive Summary available at http://www.tpc.org/execsum_TPCC.html.

Pipeline Server: A New Architecture for Server Performance on Modern Microprocessors

Michael Parkes, Rick Vicik, Charles Levine
mparkes@microsoft.com, rickv@microsoft.com, clevine@microsoft.com
Microsoft SQL Server
One Microsoft Way
Redmond, WA 98052

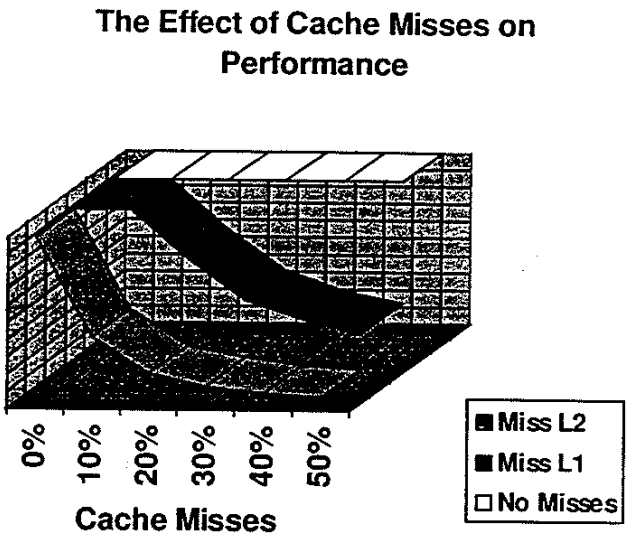
Introduction

Modern microprocessors such as the Intel PentiumPro and the Digital Alpha are increasingly limited by the speed of the off-chip memory systems. In server applications such as databases, current microprocessors spend the majority of their cycles waiting on memory stalls. As clock speeds continue to increase, the magnitude of this effect is only going to get worse. If current trends continue the scalability of databases and other servers will start to slide towards two CPUs per node.

In this position paper, we examine the effects of the evolution of computer hardware and suggest a new implementation approach, which we call Pipeline Server, that can dramatically increase server performance.

The Effect of Cache Misses on Performance

Almost all current hardware uses caches to mask the difference in performance between CPUs and main memory. However, the steady increase in the CPU performance relative to the memory sub-system is dramatically limiting CPUs' potential performance. The base reason for this is the limited spatial locality of memory references in a large number of applications. Caches do not work well if memory references do not have a high degree of locality. Databases typically exhibit poor locality properties. This problem is not intractable; it merely requires that applications be designed using a new approach. This issue cannot be ignored as the graph indicates what happens to the performance of a typical CPU (e.g., 200Mhz Pentium Pro) as the percentage of L1 and L2 cache misses increases.



A Better Way

The results of our research at Microsoft indicate that there is a better way to design and implement high performance servers (such as a database) which significantly improves cache locality characteristics. We believe that this methodology not only leads to worthwhile improvements in scalability but also opens the way to improved

methods of exposing and managing parallelism. As might be expected it is hard to detail all the facets of such a methodology in a short paper. However, the following section gives a brief insight of the techniques employed.

A typical SMP database such as Microsoft SQL Server uses threads to allow multiple queries to execute in parallel. However, in many cases these threads have very poor cache locality. Thus, work has been carried out to investigate alternative organizations. One promising technique is Pipeline Server, which restructures work within a server to optimize for cache locality and parallelism. Our experience with a prototype has provided proof of concept that practical systems can be built using this framework.

An application designed around Pipeline Server decomposes the normal, linear thread of execution into discrete service centers. Each service center processes packets of work. Packets describe units of work corresponding to some portion of the larger problem. As each service center processes a packet, it generates one or more new packets which are queued onto the appropriate service centers. In designing the service centers, any operation which might stall on an external event (e.g., disk or network IO) should be separated into different steps. This batching of operations yields significant improvements in spatial and hence cache locality. Pipeline Server improves performance by dramatically increasing overall cache locality. As the level of concurrency increases, more work is be processed at each service center per pass.

Pipeline Server also facilitates higher degrees of parallelism than traditional linear programming techniques. Consider the following example.

UPDATE T1 SET C1 = C1 + 10 WHERE C2 LIKE 'SMITH%'

On table T1 having a nonclustered index on C2.

Using traditional linear programming, the predicate would be evaluated to find each record in the nonclustered index starting with SMITH. As each record is found, the corresponding row from the base table would be retrieved and updated. The sequence would loop until there were no more SMITHs. In contrast, Pipeline Server evaluates all the SMITHs in one or more blocks of the index and generates packets of work for each record. The next service center retrieves base table records into the cache. Rather than have a single IO outstanding for the current record, a large number of simultaneous IOs can be in flight, limited by the degree of parallelism the IO system can sustain. Unlike more conventional approaches to parallelizing server execution, parallel execution is a natural consequence of the model rather than a handcrafted extension. In this example, parallel execution occurs in the buffer manager, the lock manager, the logger, the index page scanner, the data page scanner, the updater, and the execution control manager.

Pipeline server increases performance through two key techniques: (1) improved cache behavior, and (2) increased parallelism. Cache behavior improves by using out-of-order execution to boxcar work at discrete service centers within the server. Increased parallelism results from a finer level of granularity in the units of work combined with a breadth first execution model. A subtle side effect of the finer granularity is that near optimal load balancing across CPUs can be achieved automatically even in the presence of radically changing workloads.

Conclusion

CPU performance is increasing rapidly and memory speed is not keeping pace. This relative inbalance between CPU and memory speed is dramatically shifting the bottleneck from CPU cycles to memory accesses. Architectures which optimize memory access can afford an increase in CPU instructions and still outperform traditional linear programming techniques. Pipeline Server is one example of such an architecture.

Implementing Extended Transaction Models*

Calton Pu, Roger Barga, Tong Zhou
Dept. of Computer Science and Engineering
Oregon Graduate Institute
P.O. BOX 91000
Portland, OR 97291-1000
email: calton@cse.ogi.edu

Shu-Wie Chen
Dept. of Computer Science
Columbia University
New York, NY 10027

1 Introduction

After some initial resistance, atomic transactions have been accepted as the standard building block for data-centric applications that involve updates. The initial resistance was due to the complaints by the application designers that ACID properties [10], in particular serializability, are unnecessarily strong (or sometimes too weak) for their specific application. Transactions prevailed primarily because of the high degree of data integrity and software maturity offered by OLTP (online transaction processing) systems and RDBMSs (relational database management systems). Gradually, application designers accepted the OLTP/RDBMS packages and tailored their applications around them.

Partially in response to the need for customization and extension of TP systems for new complex applications, many extended transaction models (ETMs) have been proposed, e.g., split/join of transactions [19] and Epsilon Serializability [21, 22]. These ETMs typically address a subset of problems raised by application designers by extending or relaxing the ACID properties in specific ways. Both split/join transactions and Epsilon Serializability are examples of "pure extensions" that augment the ACID properties and remain compatible with ACID. For example, applications that run under serializability may run unchanged under Epsilon Serializability. The main advantage of ETMs is their ability to provide predictable correctness properties during the execution of extended transactions.

Despite the large supply of theoretical ETMs, the question on the usefulness of ETMs remains. In practice, it is quite difficult for application designers to think seriously of ETMs without the availability of sufficiently mature software. From the users' side, application designers usually observe that ETMs fit their application semantics better than ACID, but the issue of software maturity remains a roadblock to the use of ETMs in real world applications. From the vendors' side, it is difficult to justify the investment to implement ETMs for production use without serious applications and clearly identified market pressure.

To resolve this tradeoff, of ACID properties with mature software on the one hand, and flexible ETMs without mature software on the other hand, we embarked on a long term program to develop and implement ETMs on top of production TP monitors. The program started with our experimentation [20] using the Camelot software [8] and continued with the Transarc Encina TP monitor when it became available. In this position paper, we summarize the results of our first 7 years in this program and outline current research activities.

2 Design for Implementability

Our approach, called *design for implementability*, was inspired by the observation of so many ETMs proposed, only a few prototyped, and none commercialized (to our knowledge). In retrospect, this observation is not surprising since even object-oriented database management systems, which preceded ETMs

*This research has been partially supported by funding from NSF (grant IRI-9510112), DARPA, New York State CAT-CIS, Intel, Oki Electric, Texas Instruments, AT&T Foundation, plus equipment from Digital Equipment Corp. and software from Transarc Corp.

by several years, struggled with serious software maturity issues throughout their history. Custom code implementing ETMs is difficult to maintain, port and update, not because of any inherent difficulties with ETMs, but because of the difficulty of maintaining a new set of significant software modules using current software engineering technology and tools. Therefore, cost/benefit analysis of an ETM implementation usually would indicate a high cost (and low benefit) for any particular ETM.

In the Design for Implementability approach, we create concepts, design the algorithms and build the systems so they can be implemented incrementally on top of software currently in production use. The main advantages are: fast and direct technology transfer, incremental development and deployment, and software reuse. This approach is analogous to:

1. Product design for easy and reliable manufacturing, where concept designers and manufacturing engineers work together.

2. Design for testability of electronic components, where chips are designed to include easy and/or automated testing.

3. Stepwise refinement of software in-the-large, where modules and interfaces are refined to augment the entire system.

Applying design for implementability to ETMs means extra work, in addition to the research on ETMs *per se*: a "good" ETM should pass a new implementability test, i.e., whether it can be implemented incrementally on a production TP monitor. To alleviate the dependency of the implementability test on specific TP monitor products, we adopted the commonly accepted TP monitor architecture [4, 9] as a product-independent yardstick for measuring the implementability of ETMs over the range of TP products that conform to the architecture in some way.

In our efforts to demonstrate the implementability of ETMs, we have adopted several techniques previously used in other areas of database and software research. First, from the programming language and software engineering communities we have found the concepts of reflection, meta interfaces, and open implementation [12] very useful in the definition of new interfaces for ETMs. For example, meta interfaces are particularly useful in the composition of the existing interface for the classic atomic transaction model, ETMs, and the user control of all models. Second, from the operating systems community we have adopted the concepts of microprotocols [17] and specialization [7, 18] for the restructuring of TP monitor components.

In the following section, we outline three basic components of an extended TP system that supports ETMs. The first component extends primarily concurrency control, using the Reflective Transaction Framework [2, 3], to implement *adaptors* to add functionality to a TP system such as Transarc Encina. The other two components extend distributed coordination [27] and crash recovery [6], which use microprotocols and specialization, to restructure the two-phase commit protocol and TP recovery system, respectively. Together, the three components form the basic foundation of ETM support on production software. Of course this does not mean shrink-wrapped ETM support tomorrow. Instead, we consider these results to be the beginning of ETM support for real world applications.

3 Summary of Results

3.1 RTF

Barga and Pu [2] proposed the Reflective Transaction Framework (RTF) to implement a number of ETMs and semantic-based concurrency control, including the split and join of transactions, cooperative groups, and Epsilon Serializability. The details of RTF and how it helps bridge the gap between ETMs and current OLTP monitors are described in a companion submission [1] to HPTS'97. We summarize the main ideas here to make this paper self-contained.

The Reflective Transaction Framework is a software framework that systematically extends both functionality and interface of a conventional transaction processing system. The design of the framework is based on a synthesis of techniques: *computational reflection* to effectively expose internal state and functions of a conventional TP monitor, *meta object protocols* to provide explicit descriptions of extended transaction behaviors, and good software

practice for abstraction and modularity. The result is a framework that a programmer can use to inspect, extend, and modify the services of a conventional TP monitor to implement new extended transaction functionality.

The implementation of the Reflective Transaction Framework introduces *transaction adapters*, reflective software modules designed to be implemented as a thin software layer on top of TP monitor software. Transaction adapters leverage, to the extent possible, the existing functionality of the underlying TP monitor, eliminating unnecessary code development. Transaction adapters provide a collection of extended transaction functions, such as *delegation*, *dependency management*, *transaction cooperation* and *group structuring*, and provide flexible substrate so that programmers can adjust selected aspects of transaction execution to implement extended transaction behaviors, and define new application interfaces to access these new extended transaction behaviors.

A prototype of the Reflective Transaction Framework has been implemented on production transaction processing software, namely the Encina Toolkit. The Encina Toolkit provides core transaction processing services to several modern TP monitors, including Transarc's Encina, IBM's CICS/6000, and DEC's ACMS/xp TP monitor. As such, the resulting Reflective Transaction Framework implementation can be used with any of these TP monitors. Our initial evaluations of the framework have shown that it offers a general method to define and efficiently implement a wide range of advanced transaction models and semantics-based concurrency control protocols on an industrial-grade TP monitor in a flexible, yet principled, manner. What is more, the framework preserves the original TP monitor functionality and application interfaces, so extended transaction functionality can be introduced without requiring changes to existing applications.

3.2 Open Commit Protocol

Zhou and Pu [27] introduced the Open Commit Protocol (OCP) to support *distributed* extended transactions. The details of OCP are described in a companion submission [26] to HPTS'97. We summarize the main ideas here to make this paper self-contained.

Open Coordination Protocol (OCP) is a flexible coordination facility for systematically building optimized coordination protocols for distributed extended transactions and transactional workflows [28]. OCP's "openness" stands for its general functionalities: (1) it can be used to build coordination protocols for different [extended] transaction or workflow management primitives, (2) it can be used to ensure different global correctness criteria, and (3) it allows different optimization combinations.

The main idea behind OCP is to decompose existing coordination protocols (e.g., two-phase commit protocol and its variants) into fine-grain *microprotocols*, which are then composed and specialized with respect to particular situations for flexibility, reliability, and performance.

By applying OCP, both existing coordination protocols and new protocols could be developed. For example, presumed-abort (PA) variant of the two-phase commit protocol [16, 15], open commit protocol [23], optimistic commit protocol [13], unilateral commit [11], etc. And, existing optimizations or new optimizations could be incorporated into these protocols as well. For instance, read-only, last-agent, voting reliable, etc. [24, 25]. We have developed new coordination protocols for a variety of distributed extended transaction management primitives, like `Delegate`, `Split_tran`, `Join_tran_group`, etc.

When applying OCP to build a specific distributed coordination protocol, four steps are followed: instantiation, adaptation, specialization preparation, and specialization. During *instantiation*, we select a set of microprotocols in OCP for the target protocol and supply parameters to each of them via its functional interface. During *adaptation*, we customize some microprotocols via their meta interface. During *specialization preparation*, we first choose a set of quasi-invariants to reflect the nature of some desired protocol optimizations (e.g., read-only); then we customize the validation function associated with each of these quasi-invariants. During *specialization*, we first compose those microprotocols and quasi-invariants in previous steps, and then apply specialization (e.g., invoke a *specializer*) to produce optimized protocol instances.

3.3 MARS

Chen [6] introduced the Modular Architecture for Recovery Systems (MARS) to build flexible and efficient recovery systems to support extended transactions. The details of MARS are described in a companion submission [5] to HPTS'97. We summarize the main ideas of MARS here to make this paper self-contained.

The MARS architecture is based on the observation that any recovery algorithm that implements transaction-oriented recovery must perform three tasks: identify the transactions to be aborted and committed, identify the operations associated with each transaction, and recover individual transactions by removing the effects of aborted transactions and inserting the effects of committed transactions. These tasks correspond to the three MARS recovery modules: transaction states analysis, transaction operations analysis, and transaction recovery. In keeping with traditional recovery systems, we organize these recovery modules so that the two analysis modules generate a recovery plan which is then executed by the recovery module. In this manner, we maintain backward-compatibility with existing recovery systems.

For each MARS recovery module, we have developed efficient, crash-aware algorithms which have been further decomposed into a set of recovery microprotocols that can be combined in various ways to implement different recovery functionality. We have also studied the ACTA framework to determine how it can be used to identify the recovery properties of transaction models. In particular, we have considered the effect of transaction dependencies on transaction states analysis and of operation delegation on transaction operations analysis.

Combined with the MARS architecture, these results allow us to the design and construct flexible, efficient extended recovery systems using the following four-step process: (1) provide the ACTA definition of the transaction model; (2) analyze the ACTA definition to determine the recovery properties of the model; (3) determine the required recovery microprotocols needed to support the recovery properties; (4) compose the required recovery microprotocols following the MARS architecture to produce the desired recovery system.

4 Discussion

With the near term availability of ETMs implemented as prototype adaptors on top of Encina, we have started the discussions with users about the experimental use of these ETMs in real world applications. Three example applications are logistics, health care, and electronic commerce. All of them use transaction restructuring (split an join) to manage relatively long transactions. The logistics application is in the context of the Advanced Logistics Program (DARPA), from which we receive funding. The health care application is in collaboration with Tactica's application development group, which works with Visiting Nurses Health Services. The electronic commerce application is in collaboration with ADP, an industrial member of Data-Intensive Systems Center (an alliance of faculty members from Oregon Graduate Institute and Portland State University). ADP provides data processing services to a large number of used car dealerships nation wide.

One of the lessons learned in the interaction with users is the need for integrated workflow facilities when applying ETMs. In contrast to atomic transactions, many ETMs are designed to support complex applications with long activities with sophisticated internal structure. Perhaps not surprisingly, these applications benefit from both the management of workflow and customizable correctness properties provided by ETMs. Liu and Pu [14] have recently combined split and join transactions with an activity model to form the Transactional Activity composition Model (TAM). TAM supports the restructuring, e.g., split and join, of activities (which include workflow as a subcase) with customizable correctness properties. One of the major results obtained in the TAM research is the preservation of these correctness properties throughout activity restructuring.

In addition to using Transarc Encina as the OLTP monitor on which we are building ETMs and TAM, we are also investigating the use of Tactica's Caprera middleware as another foundational component for the implementation of TAM. Tactica is a startup company located in Beaverton, Oregon, and Caprera is their main product line in the Offline Transaction Processing (OFTP) area. Caprera provides the middleware necessary for mobile and disconnected "transaction processing", including workflow and business rule support.

At first glance, our Design for Implementability approach (Section 2) may appear superficially similar to Microsoft's mantra to *embrace and extend*. For example, OLE and ActiveX can be seen as extensions of the object-orientation thrust started by Borland's successful C++ products about 5 years ago. Similarly, Microsoft's entry into the Internet software area in 1995 can be seen as extensions of Netscape's software architecture. While we also advocate the extension of existing software and interfaces, there is an important difference between Design for Implementability on the one hand and Embrace and Extend on the other hand. Microsoft develops code entirely owned and controlled by Microsoft. In contrast, we develop open solutions, both in terms of interfaces and code, that can be adapted and adopted by the research community and the technology transferred to industry at large, including Microsoft.

5 Conclusion

We believe that extended transaction models (ETMs) will be useful in new, complex applications the same way atomic transactions have been useful in banking, airlines, and other mission critical applications. The "theoretical" reason for their usefulness is the same: the preservation of well defined correctness properties. Similarly, the practical achievement of predictable and correct application execution requires software maturity, e.g., TP monitors for atomic transactions. We consider the software maturity issue from the beginning in the development and evaluation of an ETM, not as a follow-on stage.

During the last seven years, we have been designing and implementing ETMs through an approach we call Design for Implementability (Section 2), that takes into account the software maturity issue from the start. When applied to ETMs, the implementability requirement led us to a number of techniques from programming language, software engineering, and operating system areas, including reflection, open implementation, microprotocols, specialization (discussed in Section 3). Using these techniques, we have applied this approach to the main components of an ETM "extended monitor": extended concurrency control, extended crash recovery, and extended distributed coordination. In the implementation of ETMs, we have been using the TP monitor architecture as an abstract foundation, and the Transarc Encina as a concrete experimental vehicle.

Even though we made promising progress both in the kind of ETMs implemented and in the breadth of TP monitor components extended, we recognize an inherent limitation of ETMs. In our interactions with users, we have learned that by themselves ETMs are usually insufficient for advanced applications, which typically also need workflow and activity management. The ongoing work on the transaction activity composition model (TAM) is building on the results summarized in this position paper (Section 3). Leveraging on our experience, we are using Design for Implementability to implement ETMs and activities on top of production software such as Transarc Encina TP monitor and Tactica's Caprera middleware.

References

[1] R. Barga and C. Pu. Evolving applications and extended transaction processing: How can conventional transaction processing systems catch up? Department of Computer Science and Engineering, Oregon Graduate Institute; March 1997.

[2] R. Barga and C. Pu. A practical and modular implementation technique of extended transaction models. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[3] R. Barga and C. Pu. Reflection on a legacy transaction processing monitor. In *Proceedings of the ACM Reflection'96 Conference*, San Francisco, April 1996.

[4] P.A. Bernstein. Transaction processing monitors. *Communications of ACM*, 33(11):75–86, November 1990.

[5] S-W. Chen and C. Pu. Recovery for extended transaction models. Department of Computer Science and Engineering, Oregon Graduate Institute; March 1997.

[6] Shu-Wie Chen. *Recovery for Extended Transaction Models.* PhD thesis, Department of Computer Science, Columbia University, February 1997.

[7] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, June 1993.

[8] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1993.

[10] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[11] Meichun Hsu and Abraham Silberschatz. Unilateral commit: A new paradigm for reliable distributed transaction processing. In *Proceedings of the 1991 IEEE Conference on Data Engineering*, February 1991.

[12] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. See `http://www.xerox.com/PARC/spl/eca/oi.html` for updates.

[13] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of 1991 ACM SIGMOD*, pages 88–97, Denver, Colorado, May 1991.

[14] L. Liu and C. Pu. A reflective framework for organizing and restructuring complex open-ended activities. Department of Computing Sciences, University of Alberta; February 1997.

[15] C. Mohan, B. Lindsay, and R. Obermark. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.

[16] C. Mohan and Bruce Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of 2nd ACM SIGACT/SIGOPS Symposium on PODC*, Montreal, Canada, August 1983.

[17] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[18] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Colorado, December 1995.

[19] C. Pu, G.E. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 27–36, Los Angeles, August 1988.

[20] C. Pu, F. Korz, and R. Lehman. An experiment on measuring application performance over the Internet. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, San Diego, May 1991. ACM/SIGMETRICS.

[21] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, May 1991.

[22] K. Ramamrithan and C. Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6), December 1995.

[23] Kurt Rothermel and Stefan Pappe. Open commit protocols for the tree of processes model. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 236–244, 1990.

[24] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the 1993 IEEE Conference on Data Engineering*, Vienna, Austria, February 1993.

[25] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases*, 3(4):325–360, 1995.

[26] T. Zhou and C. Pu. Ocp – a coordination facility for distributed extended transactions and transactional workflows. Department of Computer Science and Engineering, Oregon Graduate Institute; March 1997.

[27] T. Zhou, C. Pu, and L. Liu. Open commit protocol. In *Proceedings of the 1996 Conference on Parallel and Distributed Information Systems*, Miami, December 1996.

[28] Tong Zhou, Calton Pu, and Ling Liu. Adaptable, Efficient, and Modular Coordination of Distributed Extended Transactions. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

Building the Global Payments System for the Next Millenium

A Challenge for High Performance Transaction Processing
Mainframe versus UNIX/NT Server Solutions

Bill Reid
Visa International

HPTS 97
September, 1997
Pebble Beach, CA

Background

Visa is a worldwide association of financial institutions with one of the most recognizable and powerful product brand names and logos. It is the preferred payment brand and the largest consumer payment system in the world. Visa plays a pivotal role in payments processing as well as advancing new payment products and technologies to benefit its 20,700 Member financial institutions around the globe, their cardholders, and the global economy. Visa's 561 million cards are accepted at more than 13 million locations worldwide, including more than 323,000 ATMs in 109 countries in the Visa Global ATM Network. Visa currently processes in excess of 20 billion transactions per year with a value of over $1 trillion US dollars.

For more than two decades, Visa has been the leader in electronic payments. New technological advances, many pioneered by Visa, and the global acceptance of the Visa brand have provided added value that has benefited participating financial institutions and all parties to the payments process.

Consumers and merchants associate this familiar brand with global acceptance, consistent reliability, accessibility, and quality service. Recent global research has shown that the Visa brand is positioned strongly for future use in the new online electronic payments environment.

While its reach, brand name, and strategies are global, Visa acts locally, adapting to local market needs and preferences. Visa is owned not by stockholders in a conventional sense, but by an association of members—financial institutions around the world. Recognizing that each of the markets

they serve approaches the business of consumer payments in different ways, Visa members customize and market Visa products and services to appeal to their local markets. The success of this approach is evidenced by market research that shows that while Visa provides global coverage, acceptance and service, it is deemed by consumers to also have a close local tie to them.

Visa divides its membership into six regions: Asia-Pacific; Canada; Central Europe, Middle East Africa (CEMEA); European Union (EU); Latin America and Caribbean; and the United States. Each of the six regions operates with the same rights and obligations, and each is responsible for adapting programs and activities to its local markets. Visa operates like a unified holding company. Each region has its own board of directors responsible for regional and local operations. Select members of these regional boards serve on the Visa International Board of Directors, thereby ensuring that Visa "thinks globally, and acts locally."

While the six regions operate independently, they share a common mission and a common vision. Visa International headquarters, located in the San Francisco Bay Area, provides the cohesion that guarantees consistency across all regions. The role of Visa International is two-fold, as an "enabler" and "influencer," fully supporting Visa member profitability and competitiveness in the financial services industry. The Visa International staff is responsible for long-range planning and interregional activities, including the establishment of international policy and operating guidelines for the use of the Visa brand. This staff is also responsible for product development and enhancement, global marketing, advanced payment systems strategies, and the operation and upgrading of VisaNet. This overarching framework enables Visa members to reap benefits from the value of the Visa brand, core products and processing capabilities and provides them with a structure that will have a major influence in implementing new technologies for future electronic payments.

The Current Processing Environment

Over the past two decades, Visa has developed and evolved a number of proprietary automated systems which support global financial payments. Collectively, these systems comprise VisaNet. Since the beginning, Visa has effectively built upon its initial credit card processing model and has added processing in support of a wide variety of ubiquitous card products and services among which are deposit access cards (such as ATM, Check Card ,Delta(UK)), commercial cards, electronic bill paying, and stored value cards (VisaCash). Collectively, the automated support systems have become known as VisaNet.

Core processing within VisaNet consists of payment authorization, clearing and settlement. Significant other processing activities occur in concurrent and post transactional support of all Visa card products and services. While a number of new Visa systems have been developed in recent years in a client/server UNIX platform model, the key workhorse system applications (33M LOC in Assembler/Cobol) continue to run on mainframe legacy, high performance entities.

Authorizations are processed at four (4) operating centers worldwide running IBM's TPF 4.1. Each center is backed-up by one of the other centers with complete real-time synchronization of data amongst all centers. The authorization 24x7 operating model mandates true 100% availability and reliability worldwide for our Members and their cardholders. Visa has realized a remarkable, near perfect record with less than 20 minutes of total primetime outages in the past five years and actually ran for 4+ years without losing one second of global connectivity.. The maximum peak volume at any one center during the 1997 holiday period is projected to be at a sustained rate of about 2800 transaction messages per second. While the Visa TPF platforms will continue to operate in a tightly-coupled configuration for the remained of this year, all platforms and applications have been upgraded to support a highly parallel, loosely-coupled (HPLC) configuration as of January, 1998.

In June of this year, the tightly-coupled configuration running on an IBM 6-way mainframe processor broke all TPF processing records by reaching a sustained 3500+ transaction messages per second with momentary spikes hitting well over 4000 transaction messages per second.

Clearing and Settlement is performed on an MVS platform. The primary functions of this platform are to collect completed transactions worldwide throughout the day, perform stringent financial edits and rules compliance validations, compute individual transaction value, assess appropriate fees and charges, compute net financial positions for each Visa Member, generate reports, deliver transaction data and reports to Members, and initiate funds movement via the settlement bank. The majority of settlement processing must occur within a four hour timeframe each evening and there is zero tolerance for pushing processing out. Processing volume for a peak night during the 1997 holiday period will exceed 90 million transactions with a value of approximately $3 billion USD.

The Challenge

Despite the volumes and stringent processing time constraints, the VisaNet systems are currently meeting the processing challenge. However, the current legacy architecture does not lend itself well for a rapidly evolving and expanding electronic payments marketplace worldwide. VisaNet is based upon mature technologies and architecture, the question is, are they too mature to meet the processing challenges in the next millenium ?

Visa processing volumes continue to grow at a worldwide rate in excess of 20% per year. Based upon marketplace trends, this growth will continue well into the next millenium. This will result in VisaNet processing volumes of 40 billion transactions per annum by the year 2000. Visa will realize peak day volume processing volumes for single center Authorization of 5250 transaction messages per second and 190 million transactions for Clearing & Settlement. Activating HPLC for TPF and moving to parallel sysplex for MVS should provide the required processing power to meet the volume demands, but what about Visa's ability to be responsive to the marketplace , an accelerating rate of daily changes and individual Member and country customization requirements ? The current systems are very complex, difficult to maintain, yet powerful and stable. They are not flexible nor nimble enough to respond to rapid changes for new or modified applications. Data/information extensibility continues to play an increasingly critical role. The flow of this data/information amongst disparate operating platforms is complex and does not always meet real-time access demands.

Visa has committed to a series of multi-year programs for the total revamping of the current VisaNet systems including the applications, computer systems, networks and processes. It will be based upon a component and bus architecture and object design. After nearly a two year evaluation of current and future technology trends , analysis of the current processing component flows and capabilities, and the future processing requirements in support of Visa into the next millenium, it has been decided to start the march away from reliance on the mainframe and the current Visa processing architecture. A presentation will be given at HPTS which will outline the rationale for starting the move towards the UNIX/NT arena, the performance expectations, and the overall timelines for achieving our goals.

Relational Data Access on the Web: The Argument for a Dataless Database
(position paper)

Eugene Shekita, Dan Ford
IBM Almaden Research Center
650 Harry Road
San Jose, CA, 95120
{shekita,daford}@almaden.ibm.com

1 Introduction

The web is a fantastic medium for browsing unstructured data. On the other hand, when it comes to searching structured data (i.e., relational data), the web can be an exercise in frustration. Most relational data on the web can only be accessed indirectly, via an HTML view of the data, and often the view provided isn't adequate. Underneath the covers a web site may have a wonderful database with a powerful query language. But the view of the data provided by the web site may make it impossible to get at what's needed or force users to sift through pages of uninteresting data.

Another problem with relational data access on the web is the inability to express queries that combine, aggregate, and compare data from multiple web sites. For example, suppose that 100 different on-line suppliers sell a widget that some company needs. The only way a purchasing manager at that company can find the supplier with the lowest price is to go to all 100 web sites and manually jot down the one with the lowest price.

2 One Solution: The Dataless Database

One way to solve these problems is to build a virtual database that does nothing but integrate other databases. Such a database would have no data per se, and hence the name *dataless database* (DDB).[1] The way a DDB might work is shown in Figure 1. It would provide a full SQL API, and connect to remote SQL databases over the web via the Java Database Connectivity API (JDBC) [2, 1] or some other standard. The DDB would also connect to a local web server via some mechanism like Microsoft's Active Server Page, which provides a stored-procedure-like capability for web servers. Alternatively, something like IBM's Net.Data or Informix's Web Blade could provide the connection. Browsers or some other client tool would be used to view the DDB's data. Finally, the DDB itself could export a JDBC interface to other DDBs.

[1]The term *dataless database* was coined by Bruce Lindsay during an impassioned lunch talk on the future of database research.

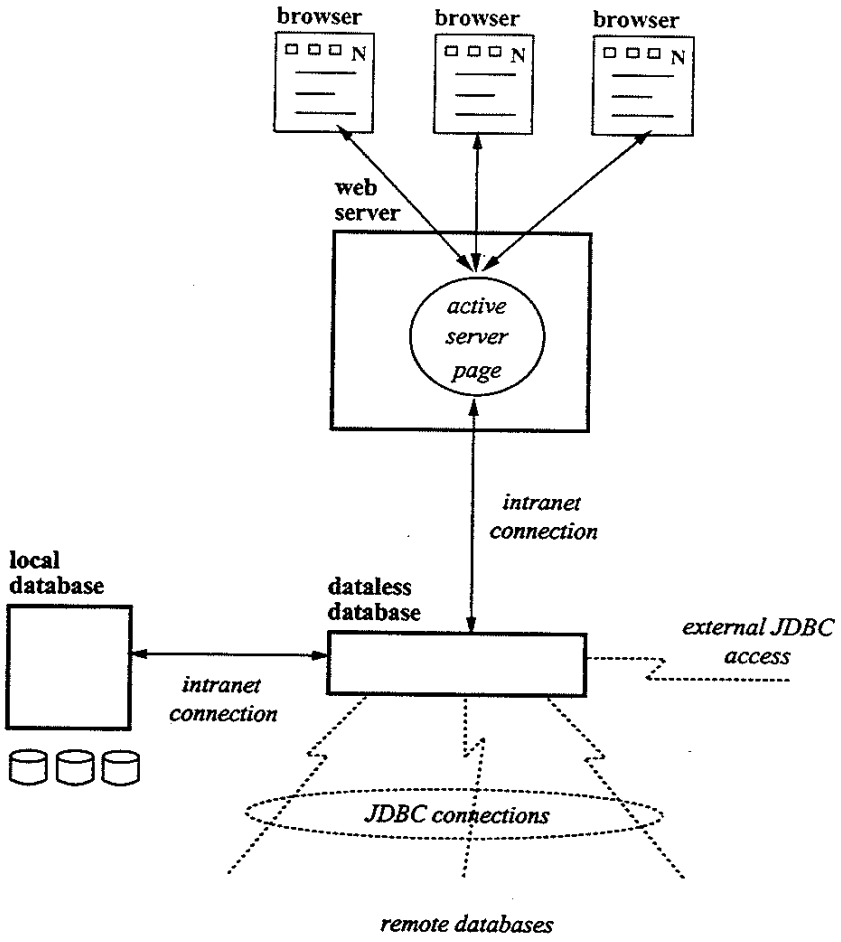


Figure 1: Dataless Database

2.1 Writing DDB Applications

The advantage of supporting a SQL API in the DDB is that today's RDBMS client tools could be used without modification to build applications. The DDB would take care of exposing remote tables as if they were stored locally. Most users are unlikely to undertake the job of writing applications, let alone maintain a DDB, however. This role could be filled by on-line "middle-men", who would provide canned web applications for accessing specific types of data. For example, a web-based CD Buyer's Guide might maintain a DDB with connections to on-line CD sellers. The CD Buyer's Guide would provide a canned Java application allowing users to search for the least costly seller of a particular CD. People would be willing to pay for these kind of services if it provided real value to them. Some sort of cybercash payment scheme is important here, since individual transactions would tend to be small.

3 Access Controls

One of the most important issues with a DDB is controlling access to data. Obviously, most corporate data is too valuable to be made freely accessible to any DDB on the web. Therefore, tight access controls would have to be put on most databases.

Security issues will certainly limit how much data becomes accessible over the web. If there was some value in it for them, however, companies would be likely to provide database access in controlled situations. For example, CD sellers are likely to provide database access to our hypothetical CD Buyer's Guide if it translates into more sales for them. Another example might be a large multi-national construction company with thousands of suppliers. To cut costs on collecting bids from suppliers, the construction company could demand that all suppliers provide DDB access to their databases. The construction company could go so far as dictating the views that each supplier must provide for the DDB.

4 Research Issues

One research issue is how to build a DDB. The naive approach is to take an existing RDBMS, rip out its storage management, and hook in JDBC access via SQL table functions. This is not likely to perform well, however. Deeper integration is needed for JDBC access. In particular, the query optimizer needs to get involved. Among other things, the optimizer should determine what predicates to execute locally and when to push-down predicates into JDBC access. It should also determine when it is cost-effective to build temporary indexes to help speed up a query.

There are also research issues about what data to cache in a DDB. Some data is likely to be static and a good candidate for caching, while other data is not. How is this communicated to the DDB? Perhaps some form of replication can be used instead of caching. Can an extension to JDBC be used as a channel for replication? Obviously, standards are required here. Of course, replicating a table to thousands of DDBs would not be feasible — the load on the database server would be too big. Perhaps multicast protocols can be used for replication [3].

Another research issue has to do with schema evolution. In controlled situations like the construction company example, schema changes would be communicated so that DDB applications can be updated to reflect the changes. But what about less controlled situations? How can schema changes at a remote site be conveyed to a DDB application? Can some sort of meta-schema standard be developed to communicate changes?

5 Conclusions

The web is all about being connected to information. Unfortunately, some of the most important business information, namely structured relational data, is largely disconnected from the web. The *dataless database* (DDB) was proposed as way to bridge this gap. Using open standards like SQL and JDBC, a DDB would be able to combine, aggregate, and compare relational data from multiple web sites. By enabling companies to connect their databases, DDBs have the potential to dramatically cut costs and create exciting new business opportunities.

References

[1] Hamilton, Catell, and Fisher. In *JDBC Database Access with Java: A Tutorial and Annotated Reference.* Addison Wesley, 1996.

[2] JavaSoft Inc. The JDBC Database Access API (http://splash.javasoft.com/jdbc).

[3] Tibco Inc. (http://www.tibco.com).

Self-Tuning Scalable Servers

Peter Spiro – Microsoft Corporation
David Campbell – Microsoft Corporation

1. Introduction

As database developers we've created a significant paradox with our products.

On some apps we've managed orders of magnitude performance improvements (tpc-h). We have all sorts of new functionality: replication, hashing, bitmap indexes, parallelism, more 'joins' than you can shake a stick at, etc. We recover most of the time after system crashes. We supposedly even have a next great wave: object relational technology. Furthermore we've managed to drive prices down in order to get good tpc results (even though it's not clear customers will actually see those same prices). So all this results in lots of revenue and for this we should pat ourselves on the back.

But our major failure:

We still have not yet produced the industrial strength product that mere mortals can deploy to solve real customer problems.

An analogy: we've managed to create a car that gets 100s/mpg, it's crash-proof, it can't get lost, it has every feature imaginable – **BUT** no one can drive it! You need a company-certified driver simply to get from point A to B, which of course we'll rent to you at an exorbitant price.

If anything this dysfunction is getting worse not better.

2. Complexity and Scalability

One of the main contributors to increased complexity is the concept of scalablility. If we're building only a high-end product then it's somewhat acceptable to have a very complex product, with hundreds of knobs, that requires consultants/gurus to optimally tune the database for the target application. Conversely if we're building a low-end product, we can have very few knobs, customers can live with fixed settings. But if we're aiming for a product that actually scales from the low-end to the high-end, then the spectrum is much greater. It's this broad spectrum that exposes the limitations of current products.

The low-end might not even want logging/recovery, whereas the high-end wants some sort of continuous spooling capability and integration with HSM systems. The low-end is concerned with memory requirements between the different apps that may be started on the client machine, whereas a server often has exclusive use of a machine.

Another contributor to the current problem is that in the mid-range to high-end, advances in new functionality (app, dbms, hardware) are allowing systems to be used in such a fashion that it's too complex for even the expert consultants to tune correctly.

For example, layered products such as PeopleSoft or SAP creating unbelievably complex environments, queries, or mixed workloads on top of the database system. No individual application programmer would ever develop systems like that. WEB enabled database access will also contribute towards unknown loads and access patterns.

Within the dbms software, the big push to 'parallelism' has created this tremendously complex execution model. What is seemingly a simple query ends up with an incredible flow pattern of sorts, joins, merges and other types of data flow, all needing scratch space, and all interacting with each other. The memory and

disk requirements for large DSS types of queries need to be carefully coordinated with other current activity on the system. Clusters compound the problem.

The object-relational capabilities create new access patterns, they have different locking semantics, and there are often calls out of the database system to external functions, which don't coordinate resource requirements/usage with the dbms software. Some of these systems might even manage their own buffer pool of objects thereby competing with the dbms buffer pool.

Another 'advance' causing complexity is hardware. Because of the additional hardware/os capabilities customers are able to add storage to systems online, add machines to clusters, even reconfigure data between disks. As a result the physical configuration of the platform now presents a moving target for the dbms software. A database, which was carefully tuned at some prior time for optimal memory and disk usage, may be out of date. Certain devices may be overloaded, and other unused.

And finally, because we've done a pretty good job of allowing all these types of capabilities and changes, customers are buying it. They're responding to market pressures and they're aggressively using these new features. And they're subsequently evolving their systems on the fly, which in turn has created a jump in complexity.

So to recap:

1. DBMS vendors succeed at creating functionally rich, fairly online, fast, complex software.
2. Solution providers create monster apps/queries layered on the database software.
3. Market pressures force customers to use all the stuff.
4. Fame and riches for dbms developers.
5. Result: chaos in the mgmt and tuning space.

3. The Problem Space

So it's clear the database software needs to solve this problem. This section will describe the path towards a solution.

It's useful to partition the problem space.

1. Data placement. Physical layout of data is a super-important aspect of most real systems. Most systems can deterministically partition data across spindles. Some systems allow data from different tables to be located on the same database page. If tables can be laid out in some sort of physical order scans will proceed much faster. Some systems will fragment record or leave forwarding ptrs. Virtually all aspects of storage placement will have an impact on performance of a running system, and hence this space is a prime candidate for self-tuning. For example, it'd be nice to defrag records, or partition hot database pages across multiple spindles, or relocate records onto the same database page that are accessed together.

2. Appropriate algorithms. When a system is designed to scale over a broad range, it becomes clear that there are often different algorithms that are better suited for different usage patterns. For example, a low-end system might be very concerned about disk footprint. In this case, when a database is created it's beneficial to have the system tables occupy as little space as possible. If a dbms allocated storage in extents of eight 32K pages, we'd end up with minimum database sizes in the megabyte range. Unacceptable. These systems want small databases to be under 100K. They want to allocate single 1K pages for tables. But large systems would like to allocate large extents because they could then perform efficient I/O for scans.

   These kinds of tradeoffs occur at a surprisingly large number of places within a database product. It's quite simple, different algorithms are more appropriate for different usage patterns. For example, low-end systems would like a small amount of space allocated to lock resources. In fact single user-systems would want no lock manager activity or resource allocations. The low-end

doesn't wanted backup/restore or logging. They'd prefer file copy as a backup mechanism. They want to be able to mail databases.

On the other hand a high-end system needs the commit stall finely tuned, the lock manager and buffer manager need large hash tables so as to avoid collisions, it wants to do 64K I/Os, etc, etc.

3. Shared resources. An important aspect of a self-tuning system is that the subsystems within a database product need to be correctly integrated. This sounds simple but it's not always the case, in fact I'm not aware of any product that has solved this problem adequately. In other words, each subsystem needs to be making 'decisions' based on the product as a whole, not just their individual behavioral patterns.

   The most pressing problem in this space is competing memory requirements, most often between query execution (sorts, joins, etc) and the buffer pool. Another problem is multiple queries competing with each other for memory. Other subsystems such as logging/recovery, backup, and stored procedure caches also dynamically compete for memory and are very often not coordinated.

   For example, say an index is being built, this involves sorting (the more memory the better), if there's very little buffer pool activity, should the index build get 75% of the available system memory? What if other queries subsequently start during the index build? Can the index build switch to using less memory in the midst of sorting index terms? What if OLTP types of queries start pulling records into the buffer pool, do they get priority for system memory? Then imagine a replication transaction starts which subsequently scans the log, how much memory does it get? And then because this is a scalable product, imagine it's running on a low-end system, and some other app like Word or Excel startup, thereby creating more competition for system memory. Chaos.

4. Really clever stuff. There's also a class of capabilities that would allow systems to run better: adding access paths based on query usage, pre-fetching data into the buffer pool based on observed data access patterns, etc.

4. Towards self-tuning systems

Data placement and appropriate algorithms.

First we need data. We can't make any intelligent decisions without data. Thus the first step is a uniform mechanism within the product that captures data for much of the activity within the server. This is actually fairly simple to build: a global section of memory with appropriate entries for various timings and events. Within the product, when event X occurs update the info in the stats collection. For example, trips into the buffer, index splits, duration of I/O stalls, size of group commit, lock queue lengths, average log I/O size, the number of merge passes for sorts, etc, etc. So it's easy to get the data, the tricky part is capturing the type of data with which the software can use to make intelligent decisions.

A second step is ensuring that the system will be able to use the data.

The first and most straightforward, which was mentioned above, is that you need to be able to modify some internal behavior based on the stats. Assume the logging sequence used a 16K memory buffer, and then the stats indicated that during formatting of log records there were on average 20K worth of data waiting to be logged. Thus it would be worthwhile to allocate more memory to the log buffers to simplify the logging sequence.

Or if the stats indicated there were too many forwarded records, an internal defrag utility should start moving records and reclaiming space. Another good use would be to partition hot data across mutiple spindles.

The commit sequence, do we simply flush dirty pages to disk or write to a log? Again, this could be a low-end high-end choice.

For both 'data placement' and 'appropriate algorithms' there's nothing magical about allowing self-tuning systems. Yes, data collection and having the software move records or go thru different algorithms will be fairly widespread throughout the server. It's a good bit of work, but with the proper focus and goals these can be accomplished. Simple matter of programming.

Shared resources.

The shared resource problem is actually quite tricky. First we need an appropriate mechanism to allow resource sharing. That is, every use of memory in the product should come from one common pool. And then we need a way to easily expand or contract the memory usage of a subsystem. For example, if the lock manager runs out of lock blocks, it needs to get more memory from the memory allocator. We could use the buffer pool as the collector of all memory and then dole it out to other subsystems. This creates complexity both in the buffer manager and in the fact that the acquired memory might not be virtually contiguous. But this is a solvable problem.

The other interesting aspect to resource sharing is the policy: who gets what amount of memory when? There's not been nearly enough research/effort in this space. Queries will compete with queries, and they'll also compete with the buffer pool and other subsystems such as procedure caches, recovery, backup/restore. A simple policy model would be based on some fixed amount of memory a query could use when it started and then it would be kept constant for the life of the query execution. A more ambitious policy would be to dynamically adjust memory usage while queries are running.

These are very real problems that need to be solved before systems perform well and become self-managing.

Clever stuff.

Finally there are the AI types of capabilities that can potentially move systems towards being easier to use and configure. These systems might also make use of the data capture mechanisms described above.

A good example might be a buffer pool that learns about reference patterns. For example, in a master detail kind of relationship the buffer pool could track the fetch patterns, and subsequently prefetch the detail records when the master is fetched. Or the system could automatically cluster the different records together on the same database page. Better yet, it should vertically partition the record so that only the hot fields are clustered on the same page.

Creating additional indexes is another prime candidate for useful progress in this space. As everyone knows, virtually no one in the world understands query optimizers. Getting the optimizer to use particular access paths is tricky and dangerous (the system should be able to do a better job). Hence it's reasonable to think that the system could record which queries resulted in a table scan because there wasn't a suitable index and capture enough data to know that indexes on certain key fields would produce faster results.

5. So who's going to do it?

Through hardware and software improvements we now have dbms products that generally solve customer problems. High-performance, no problem. Storage mechanisms, all generally there. Parallelism, not quite completed but a solved problem for the most part. VLDB, done. Simple and correct query optimizers, that'd be nice.

What to do next? Forget that object stuff, we really need to focus on the basics: we need to make the systems easier to use. That's the next great opportunity.

Industry standard benchmarks, what's real and what's smoke and mirrors

H. Reza Taheri
Hewlett Packard

Industry-standard benchmarks have become a mainstay of the transaction processing world since the formation of the TPC and the introduction of the TPC-A benchmark. This became even more evident when the TPC-C benchmark replaced TPC-A: if you don't have TPC-C results for your product, you don't have a product. Almost all end-users demand TPC-C results to put a product on their short list to make a purchasing decision. Despite all the complaints about the benchmark — it is synthetic and unrealistic; it is too easy to cheat with TPC-C and the safeguards against cheating are easy to defeat; it bears little resemblance to real customer workloads — it has become ubiquitous.

The first TPC-C disclosure boasted of a performance of 54 tpmC and a price/performance of $3,483 per tpmC. A representative recent result comes in at 23,143 tpmC with a $118 per tpmC price/performance. Have transaction processing systems grown 400+ fold in performance in 5 short years? Of course not.

But to dismiss the benchmark as a marketing ploy is plain oversimplification. Good industry standard benchmarks, such as TPC-C, are invaluable to both the vendor and the user despite their shortcomings. We will discuss how the 2-3 orders of magnitude performance increase has come about and how much of it is benchmark showmanship and how much of it is real. In particular, we will cover the early days of TPC-C and contrast it with how the benchmark is run these days. A lot has gone into producing these 4-digit tpmC numbers: advances in CPU technology, multiprocessing, storage and connectivity, operating system performance, and finally DBMS optimizations. Does the end-user benefit from any of these? (Yes!)

Finally, we will look into our crystal ball to see what's coming down the TP benchmarking road in the next year or so.

A Framework for Configurable Distributed Transactions

S.M. Wheater and S.K. Shrivastava
Department of Computing Science
University of Newcastle, Newcastle upon Tyne, NE1 7RU, England
email: stuart.wheater@ncl.ac.uk

1. Introduction

We consider a computation model in which application programs manipulate persistent (long-lived) objects under the control of atomic actions (atomic transactions). The Common Object Request Broker Architecture (CORBA) together with its services is a well known example of a distributed object model that will support the above model. At the basic level CORBA consists of the Object Request Broker (ORB) that enables distributed objects to interact with each other. At the next level a number of system level services have been specified. These services include persistence, concurrency control and Object Transaction Service (OTS). The OTS is a *protocol engine* intended to guarantee that transactional behaviour is obeyed, relying on other system level services to support the ACID properties [1]. Such a structure has the advantage that an application can be bound to any given set of compliant system services for obtaining transactional behaviour. There is therefore considerable scope for customising a transactional application (say for performance improvements) by choosing the most appropriate implementations of individual services. In this paper we explore this idea in detail and present the design of a transaction framework (a distributed object transaction support system) that permits a transactional application to be customised (configured) to a degree that has not been possible before. A very modular approach is presented that enables individual objects to be made atomic, with each object bound to concurrency control, persistence and recovery services in an application specific manner. Furthermore, these bindings can be changed at run time. So for example, it is possible for an object to switch from pessimistic to optimistic concurrency control at run time.

The framework to be presented here is based on our experience of designing and implementing a number of transaction services for distributed objects. These include the Arjuna system [2], the OTS version of the Arjuna system that we have recently completed and a Java transaction system [3]. This experience has enabled us to design a set of 'implementation neutral' service interfaces that permit considerable scope in the provision of a wide variety of implementations, all conforming to the same interface specification.

2. Diversity of requirements

The distributed object transaction support system is required to be configurable for several reasons; we enumerate some of them here:

(i) An important requirement which needs to be addressed by an implementation of the support system is object server management policy, indicating the relationship between passive persistent object states (on stable storage) and active objects (objects loaded in servers which are capable of having operations performed on them). Most applications could do with some control over object to server binding, including enabling and disabling of caching of objects at clients. It may be desirable to permit caching of read mostly objects and prevent caching of write mostly objects. Whats more, it may be desirable to dynamically switch on/off these as access patterns on an object vary.

(ii) Objects have different concurrency control requirements, so the system should be able to support these, e.g., pessimistic and optimistic.

(iii) The atomic action structure may need to be extended to provide more flexible structures, such as split transactions, glued and coloured actions.
(iv) The support for storage and retrieval of the persistence state of the transactional object can take many forms and will depend on the non-functional requirements of the objects such as performance and availability. To allow high availability it may be required to have the persistence support to perform replication. On the other hand it may be desirable to use some high performance log-based object store, or special server on a dedicated machine, may be required.

3. Transactional object support system

3.1. Overall structure

To implement ACID properties, the transaction framework (distributed object transaction support system) must monitor the operations performed on transactional objects and the beginning and ending of transactions. If an operation on an object will compromise one of the ACID properties, the system either prevents the operation from being performed or any effects of the operation from becoming visible.

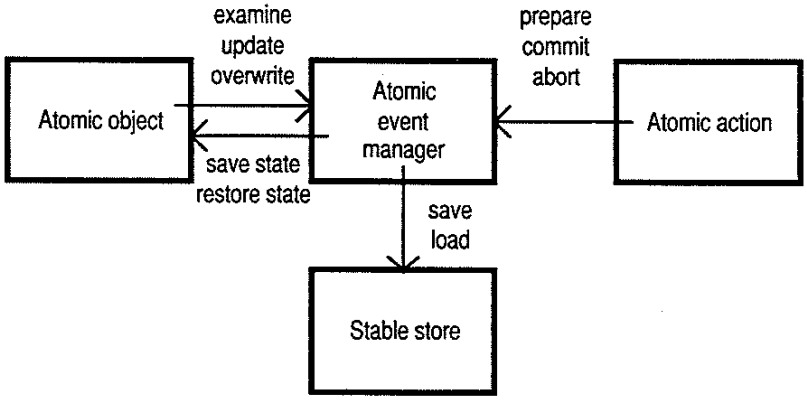


Figure 1: Transactional object support system events.

Fig. 1, illustrates the main structure of the transactional object support system and the events that can occur within it. An atomic object is responsible for generating events `examine`, `update` and `overwrite` to indicate that the object is about to be examined, updated or overwritten, respectively. An atomic action generates `prepare`, `commit` and `abort` events. The *event manager* acts on these events by directing these events to its concurrency control, recovery and persistence services to which the atomic object and the atomic action object are bound. Reconfiguration along the lines hinted in the previous section is made possible essentially by changing these bindings to different implementations of these services. The event manager is also responsible for generating events corresponding to saving and restoring the states of objects for recovery purposes, and saving and loading the states of objects to and from stable store to ensure state changes are durable.

3.2. Design of the distributed object transaction support system

As can be appreciated from the previous sub-section, the idea behind the design of the object transaction support system is to provide interfaces that isolate applications from the actual services responsible for implementing the ACID properties. This allows us to choose specific implementations on a per object basis, without functionally affecting applications. The distributed object transaction support system has been implemented using the Gandiva application building framework [4]. Standard RPC technology hides the differences between local and remote

invocations. The framework also supports a variety of object to server mapping policies. Software components are split into two separate entities: the *interface component* and the *implementation component*. The interactions between implementations can only occur through interfaces. A single interface can be used to access multiple implementations, and a single implementation can be accessed through multiple interfaces. Gandiva permits the bindings of interfaces to implementations to be dynamic and configurable. Applications are written only in terms of interfaces, and although an application can request a specific implementation, it occurs in a way that allows this request to be changed without modifying the application.

The event manager shown in fig. 1 has been split into four constituent classes, concurrency control (CC), persistence (P), recovery (R) and a coordinating class. Each instance of CC, P and R has two interfaces: atomic object manager interface (operations `examine`, `update` and `overwrite`) and atomic action manager interface (operations `prepare`, `commit` and `abort`); the coordinating class has only the atomic object manager interface. The resulting structure is illustrated in fig. 2.

The atomic action manager interface is used to isolate the atomic action from the details of transaction processing. An atomic action will maintain a list of atomic action manager objects, which are processed when the transaction ends. During the execution of an atomic action, instances of atomic action manager may be added to the list in response to specific events. The processing that is performed on the list when the action ends differs depending on whether the action commits or aborts. If it commits, the processing of this list takes the form of a two-phase commit protocol, using the `prepare()` and `commit()` operations of the atomic action manager objects. If it aborts the `abort()` operation is called. To make the preparing phase more flexible the prepare operation can be performed in multiple rounds. The `prepare()` operation takes an integer to indicate which round of prepare it is for. The return status of the `prepare()` operation will indicate if the transaction manager is fully prepared or requires to be involved in another round of preparation.

The atomic object manager interface is used to isolate atomic objects from the details of their support. The interface provides `examine()`, `update()` and `overwrite()` operations that are called to indicate to the atomic event manager that the object is about to be examined, updated or overwritten, respectively.

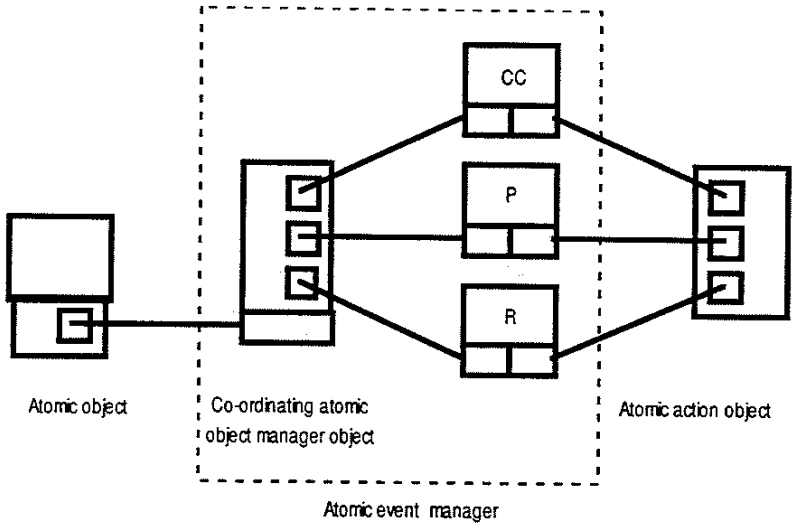


Figure 2: Object structure of the transactional object support system.

4. Examples

4.1. Mixing optimistic and pessimistic concurrency control

This simple example illustrates how two types of concurrency control can simultaneously be used within the same application. The application consists of a single atomic action *t* which performs operations on two objects *a* and *b*. Object *a* is using pessimistic concurrency control whereas object b is using optimistic concurrency control. The operations oper1 and oper2 performed by *t* are assumed to be update operations. The bodies of these operations therefore contain calls on the `update` operations of atomic object manager interface.

```
Action t;

t.begin();

a.oper1();
b.oper2();

if (. . .)
    t.commit();
else
    t.abort();
```

Within `oper1` the `update` operation will be invoked on *a*'s atomic object manager, which in turn will invoke `update` operations on the concurrency control, persistence and recovery (in that order); only if all these objects return true will the operation be allowed. The concurrency control object will perform conflict detection with the locks held on *a*, if none of the locks conflict the concurrency control will return true. The persistence support object will ensure that the object contains the current state of the object. The recovery support object will ensure that appropriate recover information is recorded. The sequence of events for operation `oper2` is the same. However, the optimistic concurrency controller for b will always return true.

If *t* commits, the objects which support *a* and *b* will each will be asked to prepare; for object *a* the concurrency control and recovery support objects will simply reply PREPARE_OK, whereas *a*'s persistence support object will first save a copy of *a*'s state and then return PREPARE_OK. Process is simpler for object *b*, except that *b*'s persistence support object will verify that the state within the object store is that which `oper2` was performed upon.

4.2. Mixed-mode persistence implementation

Our framework can support multiple implementations of persistence. A simple implementation will consist of each object forcing its state on stable store during `prepare`. We describe here how an optimised log-based persistence can be supported (without affecting application objects of course). We assume a two stage log management policy: in the first stage, a log of various object updates is constructed in volatile store and in the second stage this log is forced on to the stable store. This way, multiple force operations on the stable store from individual objects is avoided. A mix mode of operation is possible: some objects are using log-based persistence, whereas others are using the simple strategy. This is made possible because the framework supports multiple rounds of `prepare` during commit processing. Objects requiring multiple rounds return 'prepare continue' response, eventually returning 'prepare ok'.

4.3. Object server management policies

The need for exercising control over object server management policy, indicating the relationship between passive persistent object states (on stable storage) and active objects (objects loaded in servers which are capable of having operations performed on them) was briefly mentioned in section two. For the sake of illustration, assume that the following two policies:

- For each persistent object state there exists at most a single active object: this means that no co-ordination is required to maintain the properties of serialisability, failure atomicity and permanence of effect. This model can provide high performance, but the service will become unavailable if the server managing the object fails.
- For each persistent object state there can exist many active objects, co-located on the same node: the co-ordination required can be performed via fast single node inter-process communication mechanisms such as shared memory. This model can tolerate the failure of a server containing an active object, but not the failure of the entire node.

Our framework will permit an object to be managed by any such server. We have implemented and tested several object management policies.

5. Concluding Remarks

The framework presented here permits an application designer to choose the atomic object support system implementations that suits the requirements of the application. Considerable degree of freedom exists in selecting from a variety of concurrency control, persistence and server management implementations to control non-functional behaviour of application, while leaving functional characteristics unchanged. An application can be bound to the desired implementations at build time and the bindings can be changed at run time. The framework and a set of implementations, some of which have been discussed above, have been designed and implemented.

Selected References

[1] CORBA services: Common Object Services Specification, OMG Document No. 95-3-31, March 1995.

[2] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M. Little, "The design and implementation of Arjuna", USENIX Computing Systems Journal, vol. 8 (3), pp. 255-308, Summer 1995.

[3] M.C. Little and S.K. Shrivastava, "Distributed transactions in Java", submitted to this HPTS workshop.

[4] S. M. Wheater and M.C. Little, "The design and implementation of a framework for configurable software", Third IEEE Intl. Conf. on Configurable Distributed Systems, CDS96, May 1996, Annapolis, Maryland, pp. 136-146.

High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++ *

Arthur Whitney
KX Systems
1105 Harker Avenue
Palo Alto, CA 94031

Dennis Shasha
Courant Institute, NYU
251 Mercer Street
New York, NY 10012

Stevan Apter
Union Bank of Switzerland
299 Park Avenue
New York, NY 10171

Abstract

Imagine an application environment in which subsecond response to thousands of events gives the user a distinct competitive advantage, yet transactional guarantees are important. Imagine also that the data fits comfortably into a few gigabytes of Random Access Memory. These attributes characterize many financial trading applications.

Which engine should one use in such a case? IBM FastPath, Sybase, Oracle, or Object Store? We argue that an unconventional approach is called for: we use a list-based language called K having optimized support for bulk array operators, and that integrates networking, and a graphical user interface. Locking is unnecessary when single-threading such applications because the data fits into memory, obviating the need to go to disk except for logging purposes. Multithreading can be handled for OLTP applications by analyzing the arguments to transactions.

The result is a (private, size-reduced) TPC/B benchmark that achieves 25,000 transactions per second with full recoverability and TCP/IP overhead on an (167 Megahertz) UltraSparc I.

Further, hot disaster recovery can be done with far less overhead than required by two phase commit by using a sequential state machine approach. We show how to exploit multiple processors without complicating the language or our basic framework.

1 A Brief Description to the K Language

K is list-based language integrating bulk operators, interprocess communication, and graphical user interface facilities, designed and implemented by Arthur Whitney. It is extremely concise. For example, a simple spreadsheet has been implemented in about 100 characters, including the graphical user interface.

*Partly supported by NSF grants #IRI-9224601 and #IRI-9531554

Authors' e-mail addresses : nnyapt@ny.ubs.com (Steve Apter), shasha@cs.nyu.edu (Dennis Shasha), asap@netcom.com (Arthur Whitney)

Unlike most other list-based languages, K is extremely fast. For example, sorting three million records in memory takes two seconds on an IBM 990. K is also small. The entire runtime system, written in C, fits in 300,000 bytes.

The basic data structures are arbitrary length multi-lists and dictionaries (associative arrays). Binary operations on equi-length multi-lists include element-by-element addition, multiplication, subtraction, and division. max, and min. Boolean comparison ($<, >, =$) operators are also element-by-element returning a list of 1s and 0s. These operators are overloaded with scalar variants as in APL.

Rearrangement operators include sort, duplicate removal, and partition. Partition on a list L returns a list of sublists of positions where a given sublist of positions consists of all those positions in L having equal values. For example applying partition to the list 105 20 30 20 30 30 105 returns a list consisting of sublists 0 6 (positions having value 105), 1 3 (positions having value 20), and 2 4 5 (positions having value 30).

Location functions include "member" (i.e. is a value in a list), "find" (viz., find first place in a list having a value), and "where" (find all places in a list having a value). Find can be optimized using hashing.

Two-dimensional tables are built from multiple lists (a fully vertically partitioned representation in which each attribute is associated with a list). Each value in an attribute can itself be a list. Vertical partitioning works well for decision support, because only the attributes necessary for a query are brought in. (Also, the scanning rates in K are very fast. For example, 8 million records in RAM can be scanned in 1.9 seconds on an UltraSparc I to check that a field value is greater than 5 using the "where" feature mentioned above.)

The operators above allow K to implement relational algebra and aggregates including group by and having clauses. A library for all of relational algebra fits comfortably on two pages. Since K treats names as regular data, one can perform second-order queries. Suppose, for example, that you want to union certain fields of a group of tables, where the group is presented as a list of names. SQL doesn't allow this, but K's second order facility makes this a short one liner.

K has unified file input/output with TCP/IP support, and has a graphical user interface built on top of X-Windows in which the values of variables can determine their color. This is useful on Wall Street where painting losses red is a useful visual cue.

Note: The Sigmod reviewers found it annoying that we had no references for K. The language is in fact proprietary, even its description. We therefore depend on the reader's cultural knowledge of APL as well as the descriptions above to get a sense of what the language is.

2 Transaction Processing Strategy

Typical Wall Street programs enter a trade and update the trader's position as well as many subsidiary risk tables and back office (bill-clearing) databases. A message may consist of many transaction instances (hereafter just transactions). Our basic (single-threaded) strategy is to log all incoming messages onto disk and then to perform the transactions in those messages on the in-memory version of the database. Recovery consists of replaying the logged transactions on disk after reproducing the state from an appropriate dump. (In our applications so far, this is the dump of the state as of last night.) Message logging is sufficient because the database is updated sequentially.

The basic strategy is slow, however, if every transaction is logged separately. So, we write many transactions (each transaction is represented by its transaction type and all its parameters) at a time to disk. Once they are written to disk, they are secure against main memory failure. Recovery is done by replaying the log.

Warm start recovery might be slow even with group logging, because one might have to recover from a transaction log that is very long. In the Wall Street setting, we are worried not only about warm start recovery (processor failure) and media failure (local disk) but site failure as well (remember the World Trade Center bombing?). So, we are interested in remote backups. Further, we want to dump the database state periodically for fast recovery in case of a massive power failure.

For these reasons, the backup sites behave differently from the primary. On the primary, a single process logs a batch of transactions and applies them to the in-memory database. In every backup, a **logger process** will receive transactions and log them both on disk and in memory, but is not responsible for applying the transaction to the in-memory database. Once a transaction t is logged on backup b, b will report that t *is stable on* b. A commit message for transaction t will be sent back to the client after it is stable at the primary and a sufficient number of backups (how many is a policy issue) and the primary completes the transaction in its main memory database.

After logging and replying to the client, the logger process sends large groups of transactions asynchronously to the **worker process** which performs the transaction against the main memory database. The logger also periodically instructs the worker process to dump its (the worker's) state to disk. While the worker dumps its state, new transactions will accumulate in its buffer.

One might wonder how the worker will ever catch up to the primary. Note that the worker will have a huge number of transactions in the buffer and won't have to log to disk. In our experiments, the in-memory performance of K on TPC/B yields transaction rates of slightly over 50,000 transactions per second on a (167 MHz) UltraSparc I in the absence of TCP/IP and logging vs. 25,000 with the overhead of TCP/IP and logging. So the worker processes can catch up to the primary at the 50,000 transactions per second rate when it is finished dumping.

Like two phase commit, this strategy ensures that a sufficient number of backups can be brought up-to-date, but there is negligible overhead per transaction. The net effect is that we can continue processing seamlessly if the primary fails since at least one of the backups will be up-to-date. In the unlikely event, that there is a memory failure of all sites, we can recover by rolling forward from the most recent dump.

Ordering the Transaction Arrivals

We now must solve a distributed coordination problem: transactions must be applied in the same order to all data servers. A simple method is to have a single **time server** processor that assigns a sequence number to each transaction and then funnels them to all data servers. Failure of this processor however could cause the entire system to be unavailable.

Handling time server failures requires coordination that is as hard as the consensus problem[8] and is therefore prone to blocking, but can be made safe using ISIS process groups[2] and live with high probability. Both schemes entail little overhead during failure-free execution (two messages per server per collection of transactions).

3 Multithreading without Concurrency Control

How can one exploit multiple processors? A good first impulse should be to partition the database and assign one process to each partition. That will make better use of the disks on a single server. If one can partition the database into several servers, then the entire problem goes away.

If partitioning is impossible or inconvenient, we can achieve parallelism across processors or to parallel disk arms by using multiple K processes while ensuring that transactions APPEAR to execute in the order in which they arrive. Provided all sites do this, we are assured of consistency across the sites. Note that we are asking for a tighter constraint than serializability. We are asking for a serializable execution that is equivalent to a particular order of transactions (the arrival order of transactions).

The algorithm OBEYORDER makes use of a programmer-provided predicate called CONFLICT that will determine whether two transactions conflict in the normal serializability sense [1]. For example, for TPC/B, CONFLICT(t1, t2) will hold if and only if t1 and t2 access the same account, branch or teller (and then only if we are worried about negative balances); all other updates are commutative. OBEYORDER works as follows:

1. Log the transactions in batches on disk as they arrive.

2. Construct a graph whose nodes are the transactions in a batch. Form directed edges from the undirected CONFLICT relationship as follows. If t conflicts with t' and t has an earlier sequence number than t', then form a directed edge (t, t'). This produces a graph G = (T, E) where T is the set of all the transactions in the batch (or batches) and E is the set of directed conflict edges.

3. Execute T in parallel, respecting the order implied by E: That is, execute the roots of G in parallel. Then remove those nodes from G, forming a new graph G', and execute the roots of G' in parallel. Continue until there are no more nodes left.

Theorem 3.1 *Algorithm OBEYORDER yields an execution that is equivalent to a serial execution of transactions in ascending order of sequence number.*

4 Related Work

There has been excellent research in main-memory databases. Telecommunications databases are often main-memory[6, 5]. Toby Lehman and Mike Carey have looked at many main memory issues having to do with query processing[15], indexes (small fanout trees known as T-trees work well)[16], concurrency control[18], and recovery[17]. Some of these ideas found their way into the Starburst main-memory storage manager[20] where the authors find that the overhead of locking can dominate the cost of database accesses.

The Smallbase project led by Marie-Anne Neimat also uses T-trees as well as hash indexes[12]. The project supports a subset of SQL. As in our approach, Smallbase serializes transactions. Smallbase plans to implement undo/redo recovery through value logging. On their size-reduced TPC/B benchmarks, they achieve 20,000 transactions per second through their storage manager

interface without recoverability, serializability, or TCP/IP. In a nice use of the P2 configurable database manager, Thomas and Batory achieved 100,000 transactions per second on an HP 755 (99 MHz)[22] assuming collision-free hashing and without recoverability.

Li and Naughton[19] proposed a multiprocessor main memory databases. One thread groups input transactions into an input queue, though they don't use that queue for recovery. They write modified records to their log at commit. Their checkpoint thread sniffs the log and applies the changes to a shadow database. Periodically, it writes the shadow database to disk. Unlike them, our input queue is also the log since we use operation-based recovery. Further, our checkpoint process is completely asynchronous to the execution of the primary. They have one important safety feature that we lack: they allow rollbacks due to transaction logic or to exceptional conditions. The reason is that they keep the before-images of all data items.

In the Dali project, Jagadish, Dan Lieuwen, Phil Bohannon, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan have implemented multi-level recovery algorithms (including fuzzy checkpointing) and recoverable T-tree concurrency control algorithms[13, 4, 3]. Margaret Eich and her students have also worked on concurrency control and recovery[7]. Hector Garcia-Molina and Ken Salem have been among the first to observe that concurrency control might be unnecessary in a main memory environment[9].

ISIS[10] takes a complementary approach that ends almost at the same point. The ISIS mechanism delivers messages in order to the primary and backup site(s). Each site is a standard concurrent database management system, however, so some form of coordination is needed to ensure that conflicting transactions commit in the same order at all sites. The claim is that the overhead for this coordination is lower than that of two phase commit.

5 Conclusions

Relational database systems grew up in a time of RAM scarcity. In such an environment, concurrency control/two phase commit/replication server style solutions made sense. Today, many applications fit comfortably in RAM. In such an environment, different rules apply (as predicted by Garcia-Molina, Eich, and Lehman), perhaps enabling us to do without concurrency control for many applications. This would avoid a host of problems having to do with blocking, deadlock, two phase commit, and implementation complexity.

A language having well-implemented list-based bulk operations yields a system that is simple and powerful. APL, not relational algebra, may be a better starting point for such a language.

The performance gains from this approach follow from

1. The possibility (realized in K) of a very efficient implementation. (In the full paper, we illustrate comparisons on the same machine with a commercial database system and see speedups of 4 to 10 times.)

2. No overhead to enter and exit a database management system, because there is no separate database management system.

3. No overhead for concurrency control.

4. Simpler logging and recovery code.

Further, applications can be made more reliable in K than using relational systems because

1. There is no deadlock recovery code (because there are no deadlocks).

2. One can realize true hot backups through replicated state machines with much less overhead than required by commit protocols, because operations are processed (or appear to be processed) sequentially.

6 Acknowledgments

We would like to thank Tom Brown, Marc Donner, Toby Lehman, Mike Rosenberg, and John Turek for their comments on this paper. Wayne Miraglia and Don Orth helped greatly with their ideas and implementations.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman *Concurrency Control and Recovery in Database Systems* Addison-Wesley, 1987.

[2] Ken Birman "The Process Group Approach to Reliable Distributed Computing" Communications of the ACM 1993, pp. 37-48

[3] Philip Bohannon, Dan Lieuwen, Rajeev Rastogi, Avi Silberschatz, S. Sudarshan The Architecture of the Dali Main-memory Storage Manager personal communication, 1996

[4] Philip Bohannon, Rajeev Rastogi, Avi Silberschatz, S. Sudarshan Multi-level Recovery in the Dali Main-memory Storage Manager personal communication, 1996

[5] S. K. Cha et al. "Object-Oriented Design of a Main-Memory DBMS for Real-Time Applications" Proc of the Int Workshop on Real-Time Computing Systems and App (RTCSA 1995)

[6] M. Driouche, Y. Gicquel, Brigitte Kerherv, G. Le Gac, Yann Lepetit, G. Nicaud "Sabrina-RT, A Distributed DBMS for Telecommunications." EDBT 1988: 594-599

[7] Margaret H. Eich Main Memory Database Research Directions. IWDM 1989: 251-268

[8] M. J. Fischer, N. A. Lynch, and M. S. Patterson "Impossibility of Distributed Consensus with One Faulty Process" JACM April 1985

[9] Hector Garcia-Molina, Kenneth Salem Main Memory Database Systems: An Overview. TKDE 4(6): 509-516 (1992)

[10] Richard Gostanian, rg@isis.com personal communication

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques* San Mateo, Calif.: Morgan-Kaufmann, 1992.

[12] Michael Heytens, Sheralyn Listgarten, Marie-Anne Neimat, Kevin Wilkinson Smallbase: A Main-Memory DBMS for High-Performance Applications (release 4.2) Personal communication, September 7, 1995

[13] H. V. Jagadish, Daniel F. Lieuwen, Rajeev Rastogi, Abraham Silberschatz, S. Sudarshan Dali: A High Performance Main Memory Storage Manager. VLDB 1994: 48-59

[14] Ralph Kimball "Data Warehouse Architect" DBMS, August 1996, vol. 9, no. 9, pp. 14-16

[15] Tobin J. Lehman, Michael J. Carey Query Processing in Main Memory Database Management Systems. SIGMOD Conference 1986: 239-250

[16] Tobin J. Lehman, Michael J. Carey: A Study of Index Structures for Main Memory Database Management Systems. VLDB 1986: 294-303

[17] Tobin J. Lehman, Michael J. Carey: A Recovery Algorithm for A High-Performance Memory-Resident Database System. SIGMOD Conference 1987: 104-117

[18] Tobin J. Lehman, Michael J. Carey: A Concurrency Control Algorithm for Memory-Resident Database Systems. FODO 1989: 490-504

[19] Kai Li, Jeffrey F. Naughton Multiprocessor Main Memory Transaction Processing. DPDS 1988: 177-187

[20] Tobin J. Lehman, Eugene J. Shekita, Luis-Felipe Cabrera An Evaluation of Starburst's Memory Resident Storage Component. TKDE 4(6): 555-566 (1992)

[21] Joel Richardson "Supporting Lists in a Data Model (A Timely Approach)" VLDB-92, pp. 127-138

[22] Jeff Thomas and Don Batory "TPC-B on Smallbase and P2" manuscript obtainable from batory@cs.utexas.edu, August 1995