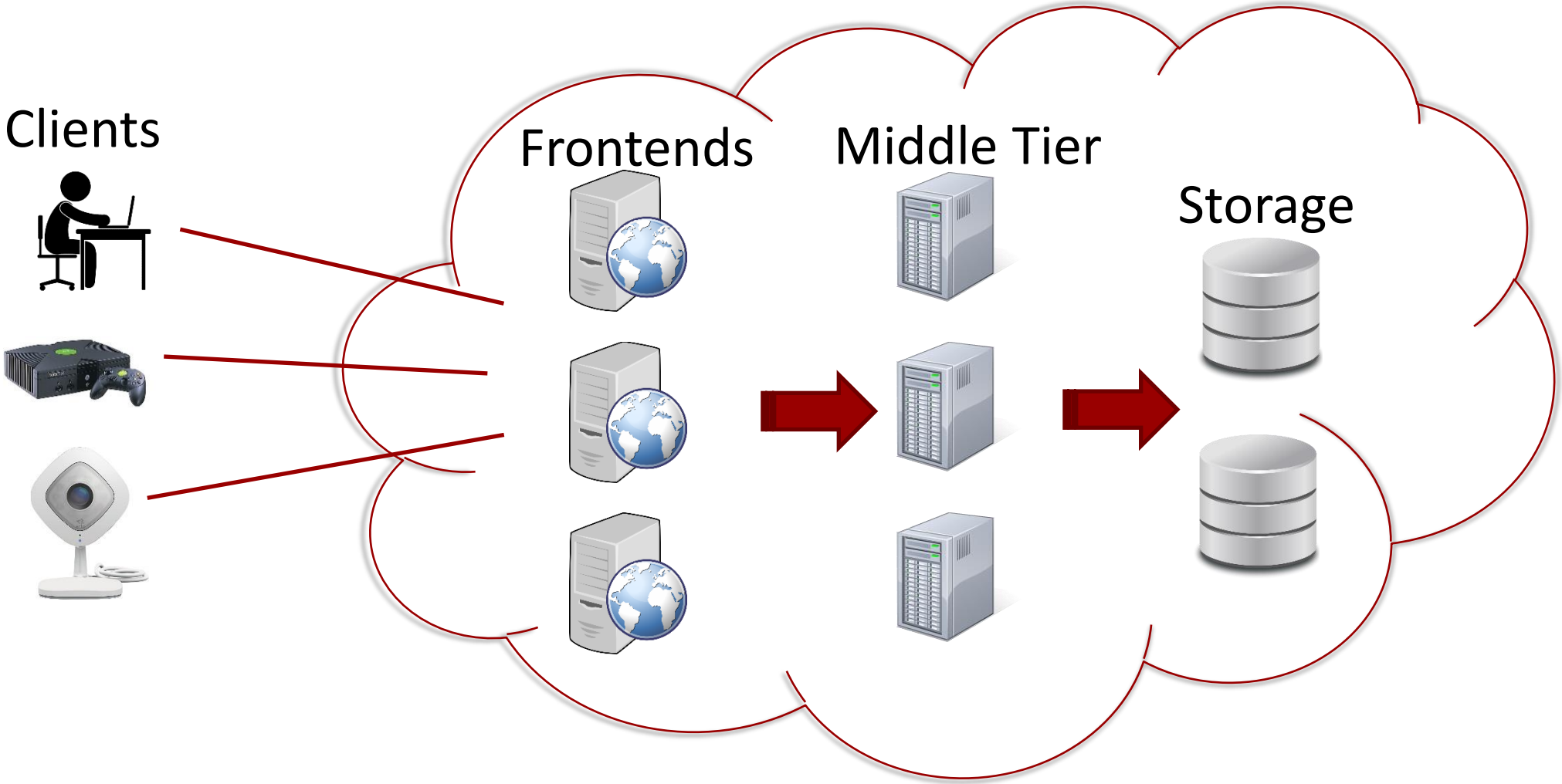
Orleans Transactions for Middle-Tier Stateful Applications

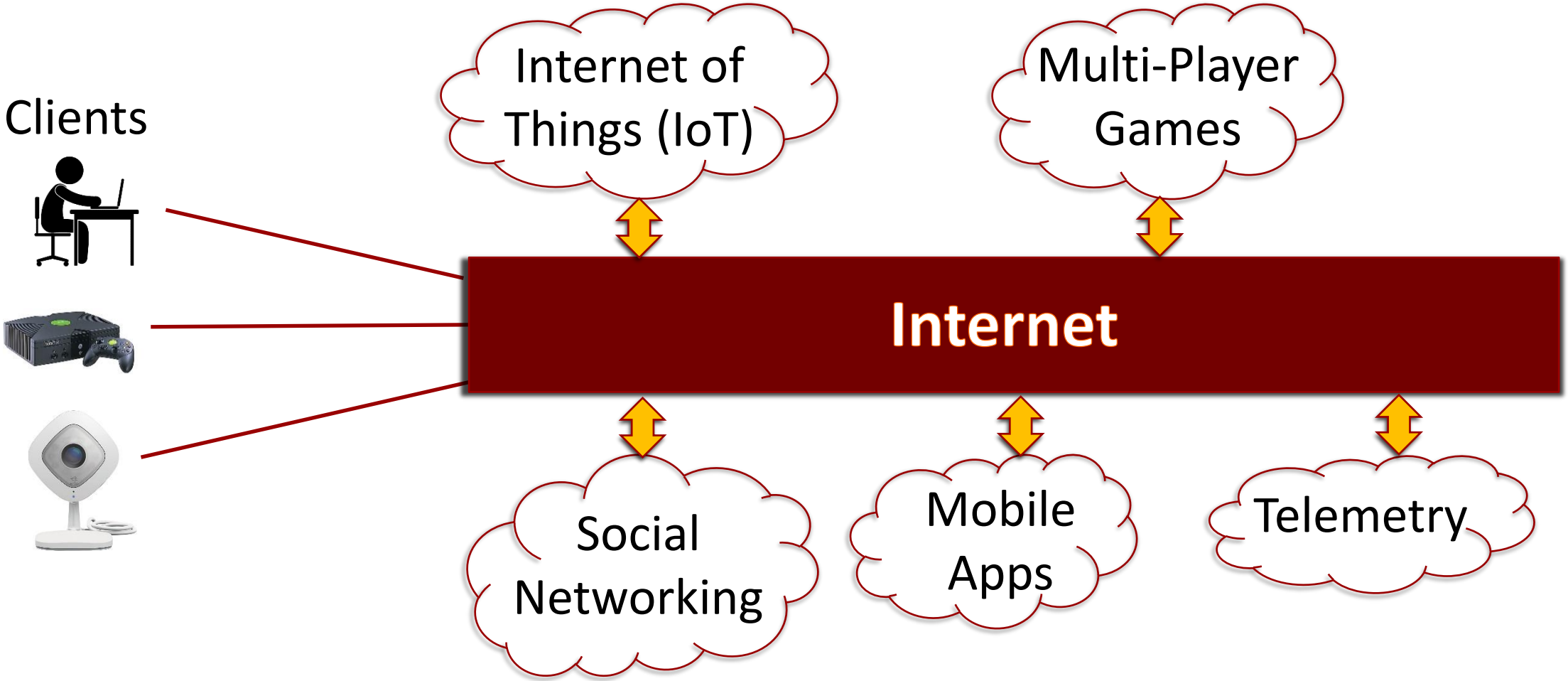
Philip Bernstein
Microsoft Research

18th HPTS
November 5, 2019

What's a Middle Tier?



Stateful Interactive Application Services



Stateful Object-Oriented Applications

- Application state is usually represented as objects
  - Naturally object-oriented, modeling real-world objects
- Examples of objects
  - Gaming: players, games, grid positions, lobbies, player profiles, leaderboards, in-game money, and weapon caches
  - Social: chat rooms, messages, photos, and news items
  - IoT: thermometers, motion detectors, cameras, GPS receivers, and virtual sensors built on top (room presence, traffic jams)

Orleans Programming Framework

- Orleans is an open-source cross-platform framework for building robust, scalable distributed applications on .NET
  - https://dotnet.github.io/orleans/
- Invented the Virtual Actor model
  - Objects are loaded and activated on demand
  - Deactivated after an idle period
- Supports scalability by load-balancing objects across servers

Fault Tolerance

- Object can save state at any time, e.g., to storage
- Virtual actor model automates fault-tolerance
- Orleans magic:
  A fault-tolerant DHT that maps object-ID to server-ID

```
public class Account
{
  int balance;

  Task Withdraw(int x);
    { if (balance >= x)
        { balance = balance - x;
          Save State;
          return (1);
      else return (0);
    }
}
```

Good news / Bad news

- Good news
  - Orleans automates scalability and fault tolerance
- Bad news
  - But not for actions across multiple objects
- Conclusion
  - Add transactions

Transaction Requirements

- All storage is in the cloud
  - Performs well despite cloud storage latency
  - No server-attached log
- Works with any cloud storage system
  - Files, BLOBs, key-value store, document (JSON) store, SQL DBMS,…
- Scales out elastically
  - Objects migrate between servers

Transaction Programming Model

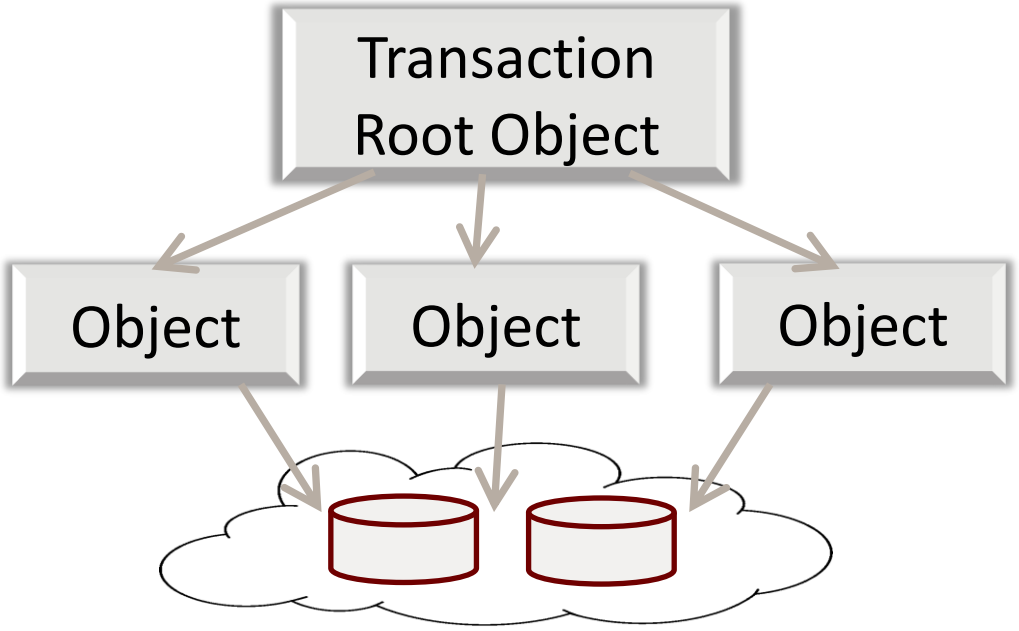
App server model is fine

```
public interface IAccountActor
{
    [TransactionOption.Required]
    Task Withdraw(uint amount);

    [TransactionOption.Required]
    Task Deposit(uint amount);

    [Transaction(TransactionOption.Required)]
    Task<uint> GetBalance();
}
```

Programming Model (cont'd)

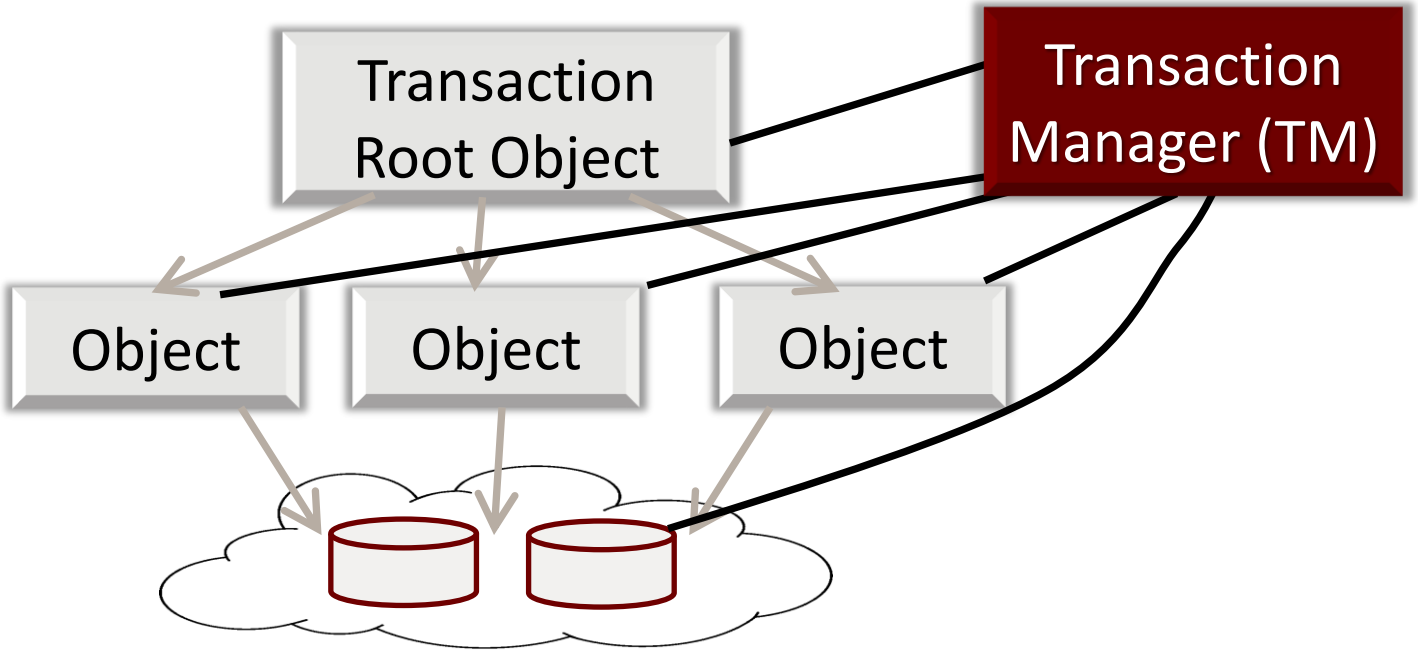
- Object's state is encapsulated in a class, e.g. MyStateClass
- Object class inherits from TransactionalObject<MyStateClass>
  - Implements generic transaction functionality
  - Start, Prepare, Commit, Abort, Read, and Write
  - Maintains logs of stable versions and active versions

Transaction Execution



- Root object initiates the transaction
- Transaction propagates on RPC's
- 2PL with object-granularity locking
  - Avoids deadlocks using Wait-Die
- Storage is oblivious to transactions

Transaction Manager



- One TM per cluster, which runs 2-phase commit (2PC)
- That's two round-trips to cloud storage.
- If 20ms per round-trip, object is locked for > 40ms
- ⇨ 25 TPS max on an object

Solution: Early Lock Release

- When object o receives T_1's Prepare, it releases T_1's lock
- If T_2 reads/writes o, it takes a "commit dependency" on T_1
  - ⇨ T_2 won't commit until after T_1 commits
- When T_2 terminates, it releases its locks
- Now T_3 can read/write o. And so on, until T_1 finishes Prepare
- After T_1 prepares, o writes prepare[T_2, T_3, …] in a batch (group-prepare)
- After T_1 commits, o writes commit[T_1] with next batch of prepared txns

Benefits: Early Lock Release

- Benefits
  - Conflicting transactions execute in parallel with 2PC
  - Enables group commit without a shared database log
  - Up to 20x throughput improvement with no extra latency
- Throughput of our centralized TM is 110K TPS

Customer feedback

- ☹ Single-object transaction must ask TM to validate its dependency
- ☹ Centralized TM is a single point of failure
  - ❖ It needs warm standby's
- ☹ Centralized TM is a potential bottleneck
- ☹ Centralized TM adds configuration complexity
  - ❖ Deploy a TM cluster alongside each Orleans cluster

Solution: One TM per Object

- Single-object transactions resolve dependencies locally
- No single point-of-failure
- Avoids configuration complexity
- Spreads TM load across objects
- Fringe benefit: TM's are naturally geo-distributed

Execution model

- If transaction root isn't persistent, it delegates TM role to a participant
- After root method terminates, its return is intercepted
  - It calls its local Transaction Agent to run 2PC
- Object storage must support ITransactionalStateStorage for its TM
  - Store → [list-of-states-to-prepare, commit-up-to, abort-after]
  - Load ←[committed-state, pending-states, txn-log, last-committed-timestamp]
- 2PC supports transfer of coordination to a one-phase participant

Status

- Released December 2018, v2.2.0.
- Used by Microsoft games
- We have a prototype that does operation logging
  - Objects manage state using *event sourcing*

Performance of Prototypes

- Max TM throughput is 110K TPS, but can be greatly improved
- Read-only txns have little effect on throughput and none on latency
- Write txn incurs 2 extra RPCs, 2 storage writes, and a deep copy
- Write-txn throughput on a single-object is 950 TPS
  - Vs. ~45 TPS w/o early lock release)
- TPC-C of prototype with operation logging, 5 nodes, 100 warehouses
  - 120,000 tpm - Orleans without transactions
  - 24,000 tpm – Orleans with transactions

Bibliography

- See https://research.microsoft.com/~philbe
- T. Eldeeb, P. Bernstein, “Transactions for Distributed Actors in the Cloud”, MSR-TR
- P.A. Bernstein, S Bykov, A. Geller, G. Kliot, J. Thelin: Orleans: Distributed Virtual Actors for Programmability and Scalability, MSR-TR
- P.A. Bernstein, M. Dashti, T. Kiefer, D. Maier: Indexing in an Actor-Oriented Database. CIDR 2017
- P.A. Bernstein, et al.: Geo-distribution of actor-based services. PACMPL 1 (OOPSLA 2017)

Acknowledgments

- Jason Bragg
- Tamer Eldeeb
- Sebastian Burckhardt
- Reuben Bond
- Sergey Bykov
- Christopher Meiklejohn
- Alejandro Tomsic

http://github.com/dotnet/orleans

http://dotnet.github.io/orleans