# Geographically Distributed Transactional Applications

Mangesh Kasbekar

8th May, 2002

# Outline of the Talk

1. Introduction and Motivation
2. Problems with the Geographically Distributed Application model
3. Basics of transactions
4. Can rethinking application models be useful?
5. Conclusion and Q & A

# Outline of the Talk

1. **Introduction and Motivation**
2. Problems with the Geographically Distributed Application model
3. Basics of transactions
4. Can rethinking application models be useful?
5. Conclusion and Q & A

# Introduction and Motivation

Popular web-application platform:

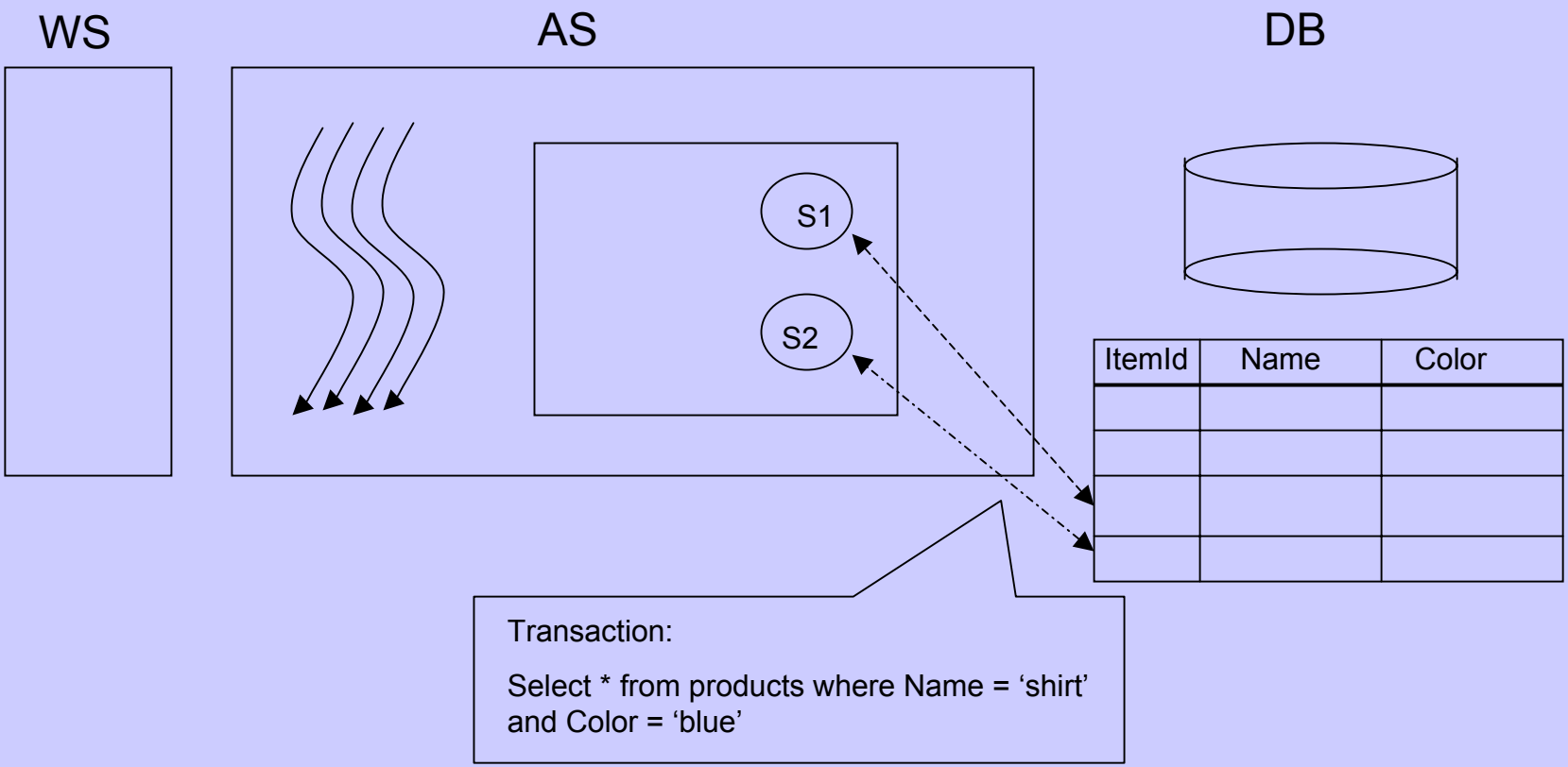
# Introduction and Motivation

The popular 3-tier web-application model:

- Web tier for presentation of the content
- Middle tier for request processing and business logic
- Database tier for persistent data and transactions

# Introduction and Motivation

A typical web app in e-commerce site

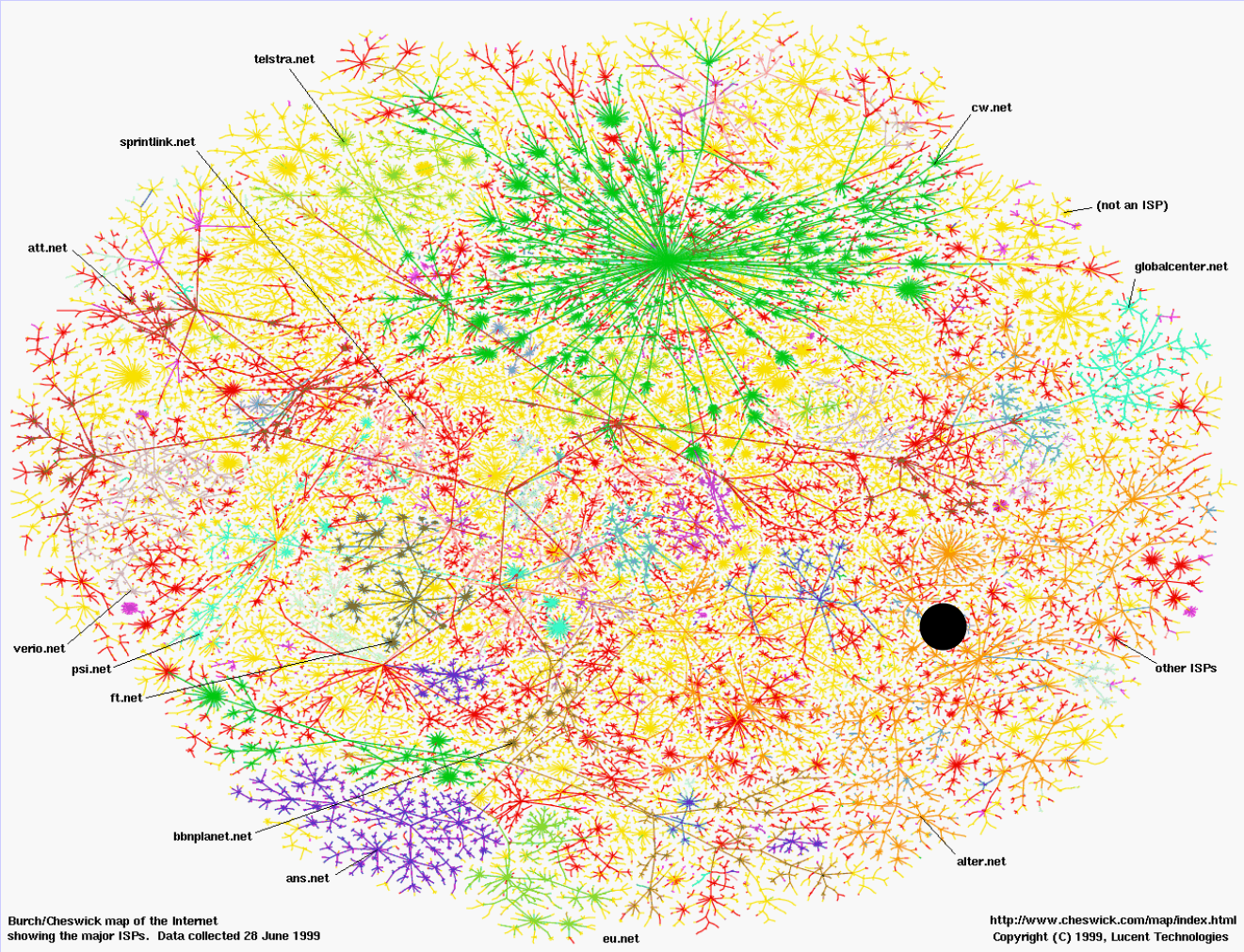


# Introduction and Motivation

## Problems with Centralized Applications

- Reliability issues:
  - Single point of failure
  - A single failure can potentially affect thousands of concurrent users
- Availability issues:
  - Internet partitions or connectivity problems can render the site unreachable from some part of the Internet
  - Unexpected high traffic volume can saturate network links
- Site performance as seen by end-users is unpredictable

# Introduction and Motivation

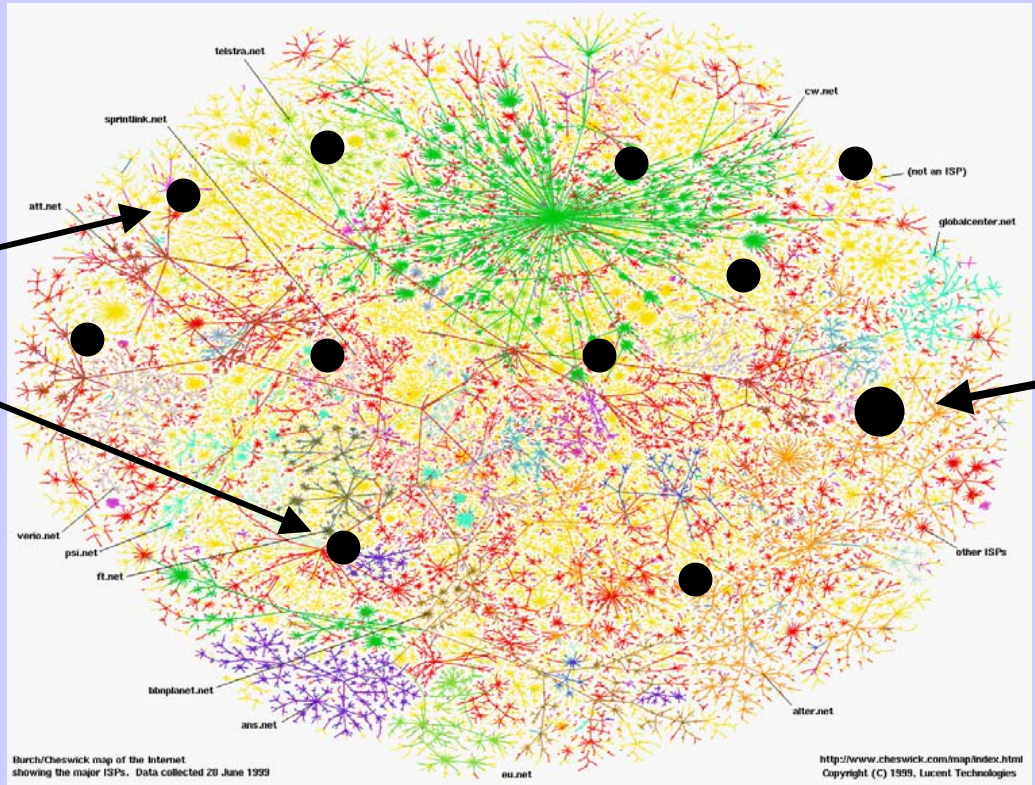


# Introduction and Motivation

## A Solution:

Geographic distribution and replication of the site

Edge servers




Origin server

# Introduction and Motivation

- Edge servers run the web-tier
- Edge servers run the web-tier and the middle tier
- Full replication of the site: All edge servers contain all the three tiers of the application, with replicated databases.
  - Edge servers run the web tier, the middle tier and have a db-cache, not a full database

# Introduction and Motivation

Other examples of geo-distributed applications

- Using web services available somewhere on the internet
- Grid computing, using idle cycles of machines all around the world for computation

# Introduction and Motivation

Basic requirements from the distributed applications

- correctness
- at least the same level of performance
- at least the same level of reliability
- better availability

Other requirements

- security
- monitoring
- manageability

# Outline of the Talk

1. Introduction and Motivation
2. Problems with the Geographically Distributed Application model
3. Basics of transactions
4. Can rethinking application models be useful?
5. Conclusion and Q & A

# Problems with the model

Problems:

- A very very unreliable platform
- Transactions and the (un)scalability of Database Replication schemes
- Consistency of the application replicas
- Finally, the CAP theorem

# Problems with the model

Platform:

- Very high latencies, losses and congestion:
  - 20mS to 1000mS roundtrip latencies
  - 2-20% packet loss
- Bad routing / badly configured routers
  - Asian countries connected via Los Angeles
  - Nepal to Bombay around the globe
- Frequent Network partitions

Compare it with a LAN / Server farm environment

# Problems with the model

Replicated databases:

- Eager scheme: each update under a transaction is propagated to all replicas before it can commit
  - Strict consistency among the replicas
  - Does not scale, too much communication
  - Blocks during network partitions
  - Deadlock hazard is high

- Lazy scheme: updating transactions commit without propagating updates, and updates are propagated to the replicas lazily
  - Weak consistency among the replicas
  - Transaction collisions are high, compensation frequently needed

# Problems with the model

Problems with wide-area transactions:

- Longer transactions: poor resource utilization
- No autonomy: if coordinator is unavailable, no transaction can be committed
- The 2PC protocol detects all problems by using timeouts:
  - Higher chances of failures due to partitions
  - Longer timeouts
  - Longer blocking for participants
  - Higher chances of heuristic commitment decisions

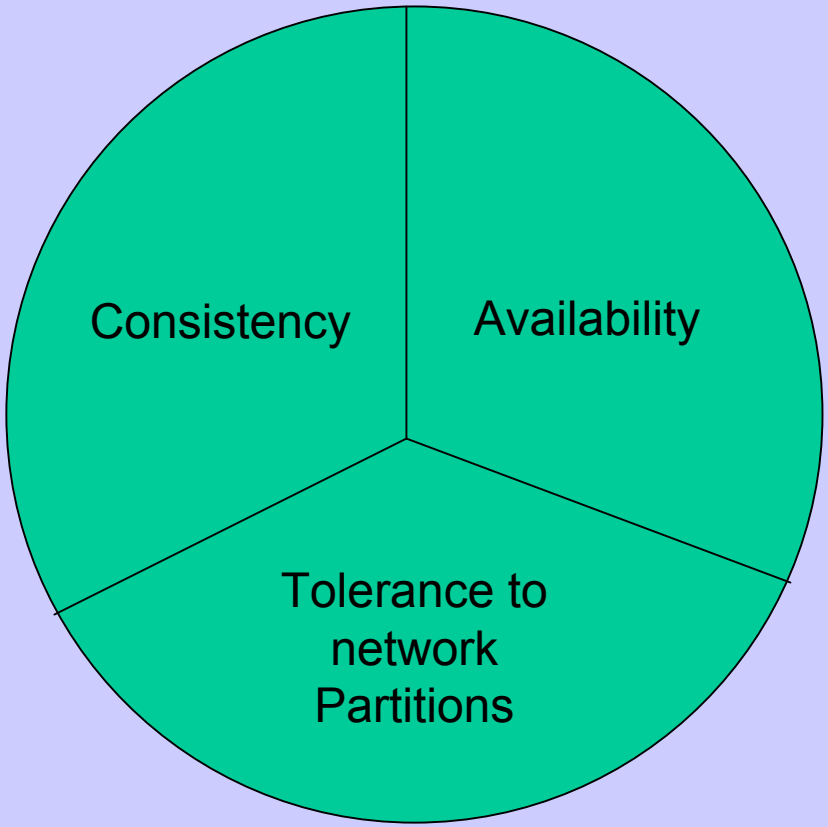
# Problems with the model

Consistency of the application replicas

- Applications are highly stateful
- Need to replicate the state to guard against single-site failures
- Need to keep the replicas consistent, so that if a single user is served from two different machines in a single session, no inconsistencies result.
  - Eager updates
  - Lazy updates

# Problems with the model

The so-called CAP theorem:




Theorem:

You can have at most two of the three

E.g.

- Cluster databases choose A and C
- DNS chooses A and P

# Problems with the model

Finally, some good news

For web applications,

- A high percentage of the databases accesses are read-only transactions (no update propagation across replicas is needed)
- A large volume of data is slow changing, database caching on the edge can help
- A good percentage of writes do not conflict (Single-writer)

# Outline of the Talk

1. Introduction and Motivation
2. Problems with the Geographically Distributed Application model
3. Basics of transactions
4. Can rethinking application models be useful?
5. Conclusion and Q & A

# Basics of Transaction Processing

Properties of Transactions

- ACID Properties
- Serializability and Recoverability
- Isolation Levels

# Basics of Transaction Processing

What is a transaction?

A composite software action that starts with a `begin` command, conducts several `read/write` operations on data and ends with either a `commit` or an `abort` command. E.g.

```
Begin, r(x), r(y), w(x,1), w(y,0), commit.
Begin, r(x), w(x,2), r(y), r(z), abort.
```

# Basics of Transaction Processing

ACID properties of a transaction:

- Atomicity
- Consistency
- Isolation
- Durability

# Basics of Transaction Processing

## Atomicity:

All-or-nothing property

```
Begin, r(x), r(y), w(x,1), w(y,0), commit.
Begin, r(x), r(y), w(x,2), w(y,1), r(z),
abort.
```

# Basics of Transaction Processing

## Consistency:

Any set of transactions executed concurrently will not leave the database in a inconsistent state.

```
Begin, r(x), r(y), w(x,1), w(y,0), commit.
Begin, r(x), r(z), w(x,5), w(y,10), commit.
```

# Basics of Transaction Processing

## Isolation:

Any transaction will not meddle in the execution of another, concurrently executing transaction.

```
Begin, r(x), r(y), w(x,1), w(y,0), abort.
                    Begin, r(x), r(z), w(x,5), w(y,10), commit.
```

# Basics of Transaction Processing

## Durability:

Persistence of data after a commit.

```
Begin, r(x), r(y), w(x,1), w(y,0), commit.
```

# Basics of Transaction Processing

Example : Transfer of money between two bank accounts

```
function Transfer ( from , to , amount ) {
  begin transaction ;
  if ( from.balance < amount ) {
    abort transaction ;
    return ;
  }
  from.balance -= amount ;
  to.balance   += amount ;
  commit transaction;
}
```

# Basics of Transaction Processing

ACID transactions are serializable:

When a set of transactions executes concurrently, their operations may be interleaved. The concurrent execution is modeled by a *history,* (which is nothing but a partial order of operations).

A history is *view-serializable* if there exists a possible serial execution of the operations such that in both executions (concurrent as well as serial), each transaction reads the same values and the final values of the database are the same.

A history is *conflict-serializable* if there exists a possible serial execution of the operations such that in both executions (concurrent as well as serial), the order of conflicting operations is the same.

# Basics of Transaction Processing

Recoverability:

In order to maintain correctness in the presence of failures, the execution histories need to be *recoverable* in addition to being serializable. Not all histories are recoverable.

- A history is recoverable if each transaction commits *after* the commitment of the other transactions from which it read.
- A recoverable history avoids cascading aborts if it reads values that are written only by committed transactions.

# Basics of Transaction Processing

Why is all this important?

Because even if the databases are replicated, the transactions must satisfy all these properties. Which makes it hard to replicate a database.

# Basics of Transaction Processing

Excellent references for Transaction Processing and Database Systems:

- Transaction Processing by Jim Gray and Andreas Reuter
- Concurrency control and recovery in database systems by Phil Bernstein et. al. (available online)

# Outline of the Talk

1. Introduction and Motivation
2. Problems with the Geographically Distributed Application model
3. Basics of transactions
4. Can rethinking application models be useful?
5. Conclusion and Q & A

# Optimistic Application models

We may not be able run the same applications unmodified on a geographically distributed platform and get the same level of performance and reliability

1. Can the usual techniques like caching and partitioning help?
2. Can we rethink the application model to make the servers stateless?
3. Can we rethink the application model to reduce the consistency needs? If the application's consistency requirement is low, then the CAP theorem cannot harm us. Can we trade off consistency at the expense of something else?

# Optimistic Application models

Caching data at the edge servers
(C. Mohan's DBCache project at IBM Almaden)

Since a good fraction of data is read-only and slow changing, caching it on the edge can give good hit rates for such data. E.g. If catalog data is cached at the edge, then browsing the catalog can happen almost completely at the edge.

- Need query containment analyzers to determine if a given query can be answered from the cache.
- RW transactions, update everywhere, cache invalidations?

# Optimistic Application models

Making edge servers stateless

- Clients' session-state makes the servers stateful. If we take the per-session state away from the servers, they can be made less stateful.
- Home PCs are quite powerful. Can we extend the platform model to include the client machines as well? (Peer-to-peer systems do that)
- With client's state on the client machine, we reduce the size of the server state.

# Optimistic Application models

Partitioning data, instead of replicating:

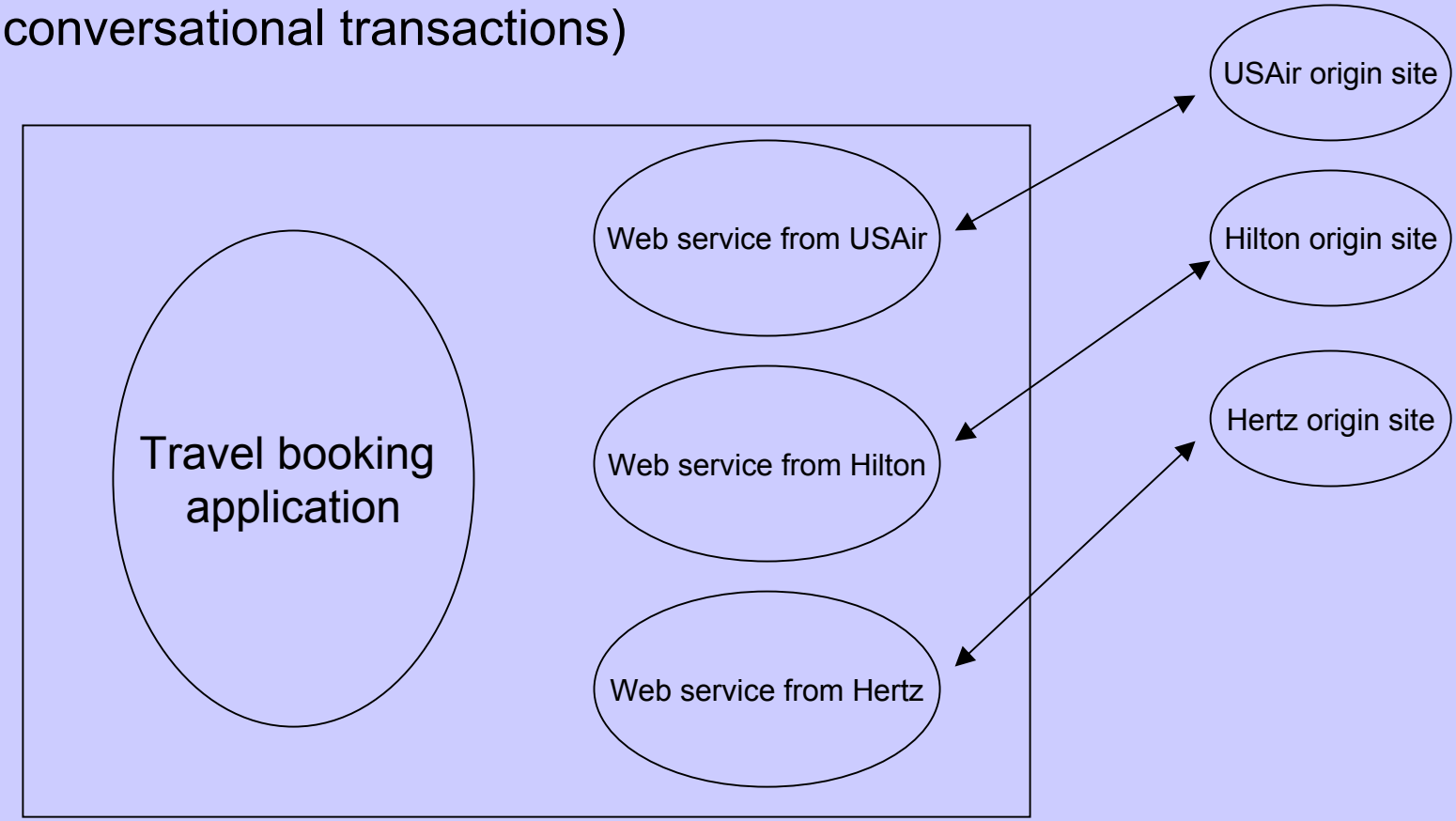
Each edge server *owns* a partition of data, whose local modification cannot have a global conflict. Makes the edge servers autonomous.

E.g. in a bookseller application, an edge server is given a quota of books to sell. All transactions needed to sell these books are now local transactions, and have no global effect.

E.g. Web-services based travel-booking application
(HPL's conversational transactions)

# Optimistic Application models

Web-services based travel-booking application
(HPL's conversational transactions)



# Optimistic Application models

Two types of transactions:

- Pessimistic
  - The ones that won't ever be committed optimistically. E.g. transfer of money between two accounts
- Optimistic  or semi-optimistic
  - Some transactions that can be committed without much concern. E.g. A bookseller selling 1 copy of a Harry Potter book, when there is plenty of inventory
  - Some transactions can be optimistically committed, which might cause an inconsistency, but it can be tolerated or compensated. E.g. selling airline tickets

# Optimistic Application models

In each of the examples, we need to evaluate the optimistic transaction model with respect to:

- Preservation of ACID  properties end-to-end
- Global serializability and recoverablity of transactions
- Relation between isolation levels and optimism levels

# Outline of the Talk

1. Introduction and Motivation
2. Problems with the Geographically Distributed Application model
3. Basics of transactions
4. Can rethinking application models be useful?
5. Conclusion and Q & A

# Concluding Remarks

In this talk,

- I presented the issues in distributing transactional applications over geographically distributed server platform. Database replication and internet transactions are hard, and systems that need it are likely to offer low performance and low availability.
- The CAP theorem dictates what can be achieved and what can’t.
- We can engineer the application carefully to make it run on a geo-distributed platform
- We can rethink the application model, and come up with an optimistic one to beat the CAP theorem.

# Concluding Remarks

Interesting things to think about

- Can geo-distributed applications be equally reliable and have similar performance as the traditional applications running in a centralized server-farm environment ?
- Database caching and complexity of the query containment analyzers
- An end-to-end model for optimistic transactions in geographically distributed applications, and proving the properties of the transactions and TP system using this model

# Outline of the Talk

1. Introduction and Motivation
2. Problems with the Geographically Distributed Application model
3. Basics of transactions
4. Can rethinking application models be useful?
5. Conclusion and Q & A

Q&A

# Backup slides

# Basics of Transaction Processing

This was a textbook example of long-running transactions

The flat transaction model is not good for this type

- If a single transaction is used for the whole operation:
  - All account locks will be held till the commit
  - Any failure will cause the whole transaction to abort and all the work done will be lost.
- If a transaction is used for each interest-deposit
  - Too much per-transaction overhead

# Basics of Transaction Processing

Example 3: Travel booking.

1. Get source, destination, and travel dates from the customer.
2. Begin transaction.
3. Query airline company database, get list of flights, availability and price-list.
4. Query car-rental company database, get availability and price-list for cars.
5. Query hotel database, get availability and price-list for rooms.
6. Get customer's choice, and book tickets.
7. If all booked, commit transaction.
   else abort transaction.

# Basics of Transaction Processing

This was a textbook example of distributed / multidatabase transaction.

The flat transaction model is not good here

- Autonomous databases
- Commitment of the outer transaction depends on the commitment of the inner transactions
- If one of the transactions fails, it may be required to abort other, committed transactions.

# Basics of Transaction Processing

Advanced Transaction Models:

1. Flat Transactions with Savepoints
2. Chained Transactions
3. Sagas
4. Nested / Multi-level Transactions

# Basics of Transaction Processing

Flat Transactions with Savepoints:

Write state of the transaction into a savepoint periodically. So if there is a system failure, the transaction can be rolled back to the saved state and restarted from the saved state, instead of a complete transaction abort. Reduces the amount of lost work due to a failure.

Savepoints can be volatile or persistent. Volatile savepoints are lightweight, but do not survive system crashes.

# Basics of Transaction Processing

Chained Transactions:

Back-to-back transactions, in which transaction context and locks can be passed on from one transaction to the next.

```
Begin, r(x), w(x), chainWork, r(z), w(z), commit.
```

# Basics of Transaction Processing

Saga:

A saga is made up of a set of transactions T1..Tn, and a set of compensating Trnasactions C1..Cn, and guarantees the following:

- If all transactions succeed, then the result of running the saga is the same as running T1, T2, ..Tn sequentially,
- If any transaction Ti fails, then the result is equivalent to running T1, T2, .. Ti, Ci, … C2, C1.

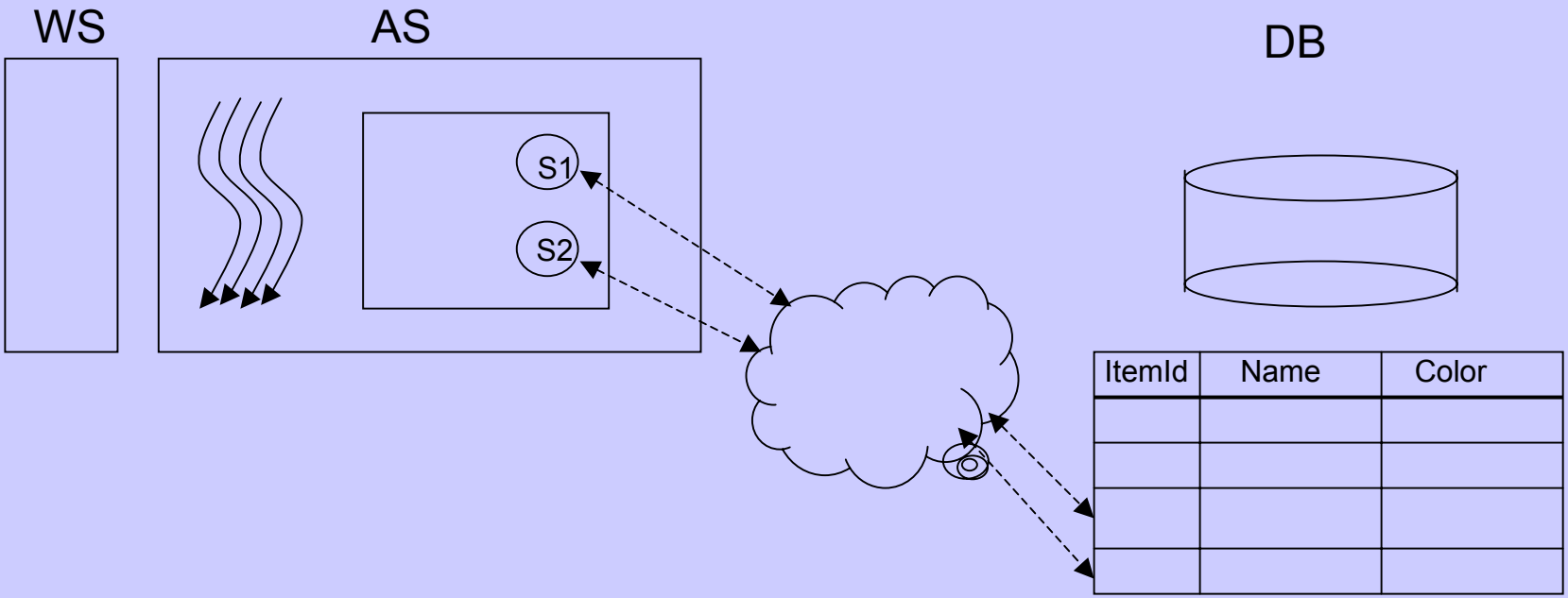
# Basics of Transaction Processing

Nested / Multi-level transactions

- Top-level transaction contains multiple sub-transactions
- Sub-transactions do not have the durability property
- Top level transaction commits only if all sub-transactions commit, and only after the commit of the top-level, the output of sub-transactions becomes durable.
- If the top-level transaction aborts, the actions of the committed sub-transactions are undone

# Introduction and Motivation

## Geographically Distributed Setting



# Introduction and Motivation

For a making a web-application geo-distributable, the strong coupling between AS and DB should be eliminated

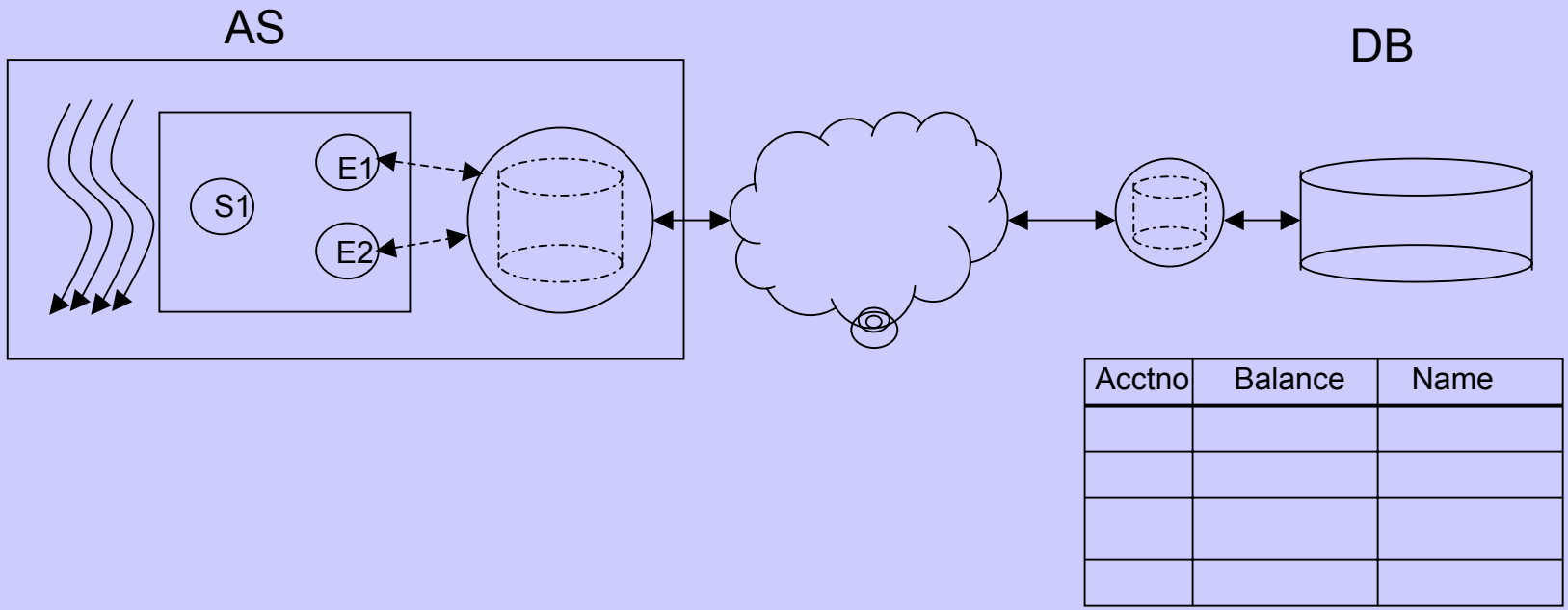
One way to do it:

Relaxing applications' consistency requirements and using optimistic transactions

# Introduction and Motivation

Quick Preview:

An architecture for geo-distributed optimistic transactions



# Introduction and Motivation

Related work:

- HP Labs' *conversational* transactions for e-services
- Distributed file systems (such as AFS)
- Shared memory multiprocessor systems and their consistency models

# Outline of the Talk

1. Introduction and Motivation
2. **Basics of Transaction Processing**
3. An architecture for optimistic transactions
4. Concluding remarks
5. Q & A

# Basics of Transaction Processing

## Underlying database operations:

Client sends:

```
Begin, r(x), w(x,500), w(y,550), Commit.
```

Database does:

```
Begin,
readLock(x), read(x),
writeLock(x), updateLog(x), write(x,500),
writeLock(y), updateLog(y), write(y,550),
prepareToCommit, unlock(x), unlock(y), commit.
```

This was a textbook example of  a flat transaction with strict 2-phase locking and operation logging.

# Basics of Transaction Processing

More properties of transactions:

- Transaction Histories
- Serializability
- Recoverability
- Isolation Levels

# Basics of Transaction Processing

Transaction Histories:

When a set of transactions executes concurrently, their operations may be interleaved. The concurrent execution is modeled by a *history,* (which is nothing but a partial order of operations).

```
T1 = r1(x) -> w1(x) -> c1 ;
T2 = r2(x) -> w2(y) -> w2(x) -> c2 ;
T3 = r3(y) -> w3(x) -> w3(y) -> w3(z) -> c3 ;
```

# Basics of Transaction Processing

A complete history for these transactions is:

```
                    r2(x) -> w2(y) -> w2(x) -> c2 ;
                     |         |
H1 = r3(y) -> w3(x) -> w3(y) -> w3(z) -> c3 ;
                     |
     r1(x) -> w1(x) -> c1 ;
```

# Basics of Transaction Processing

A history is *view-serializable* if there exists a possible serial execution of the operations such that in both executions (concurrent as well as serial), each transaction reads the same values and the final values of the database are the same.

A history is *conflict-serializable* if there exists a possible serial execution of the operations such that in both executions (concurrent as well as serial), the order of conflicting operations is the same.

# Basics of Transaction Processing

Recoverable histories

In order to maintain correctness in the presence of failures, the execution histories need to be *recoverable* in addition to being serializable.

- Recoverable histories
- Recoverable histories which avoid cascading aborts
- Strict histories

# Basics of Transaction Processing

A history H is recoverable if whenever Tj reads from Ti, where

- both Ti and Tj $\in$ H
- and Cj $\in$ H
- and Cj < Ci

Intuitively, a history is recoverable if each transaction commits *after* the commitment of the other transactions from which it read.

# Basics of Transaction Processing

A history H is said to avoid cascading aborts if whenever Tj reads from Ti, where

- both Ti and Tj $\in$ H
- and Cj < Ri(x)

Intuitively, a history avoids cascading aborts if it reads values that are written only by committed transactions.

# Basics of Transaction Processing

A history H is said to be strict if
whenever $W_j(x) < O_i(x)$,

either $Abort(j) < O_i(x)$ or $Commit(j) < O_i(x)$

Intuitively, a history is strict if no item is read or written till the previous transaction that wrote that item terminates either by committing or aborting.

# Basics of Transaction Processing

## Isolation levels

Different name for locking

Bad dependencies between transactions:

- Lost update
  - write-write sequence
- Dirty Read
  - T1 reads an object previously written by T2, and T2 modifies it again
- Unrepeatable read
  - T1 reads an object twice and gets two different values

# Basics of Transaction Processing

- Isolation level 3 (repeatable read)
  - True isolation
  - No lost updates and reads are repeatable
- Isolation level 2 (cursor stability)
  - Guards against w->w and w->r dependencies, but ignores r->w dependency
  - No lost updates and no dirty reads
- Isolation level 1 (browse isolation)
  - Disables w->r isolation
  - No lost updates
- Isolation level 0 (anarchy)
  - Does not overwrite other's dirty data if the other is level 1 or more.

# Basics of Transaction Processing

Why is all this important?

Because any optimistic TP system we design should provide guarantees that it can support some (or all) of these properties of transactions.