TRANSACTIONS: THE MODEL, SOME BUILDING BLOCKS, A "SOLUTION"

Professor Ken Birman
CS4414 Lecture 24

IDEA MAP FOR TODAY

Transaction model: a way to describe correct, consistent behavior when distributed programs concurrently access storage that could be spread over many machines.

Two-Phase commit: a central building block for a solution. Ensures that if any process commits, all do; otherwise it aborts.

Two-phase locking (similar name, totally different meaning!): A way to do read and write locking that, when combined with two-phase commit, ensures transactional serializability

How would we prove that a solution such as this really works?

Could it deadlock? What would we do if that happened?

What is something crashes but we want the system to be self-repairing?

TRANSACTION MODEL

A “model” is a descriptive formalism – a mathematical way to describe real-world things.

The transactional model starts by defining data object and processes. The idea is to have a very simple mathematical description that removes all the implementation details and leaves only the bare bones, but yet is still useful.

ROLE OF BEGIN AND COMMIT/ABORT?

Begin is a kind of a “curly brace”. But in fact it denotes the place where the transactional system initializes itself.

Commit is the way a successful transaction tells the runtime environment to save (make permanent) all its changes.

Abort tells the system to back the changes out.

ABORT IS USEFUL!

Suppose you were uncertain how to approach someone you really, really wanted to meet.

You could try different options. If they didn't work out, you just invoke "abort" and the world rewinds to how it was at the start.

Kind of like the movie "Palm Springs"

DATA AND PROCESSES

We model data as a set of variables, usually with alphabetical names such as X, Y, Z…

A transaction models an executing program that has begin/commit/abort blocks, inside of which it issues reads and writes to the variables.

SYNCHRONIZATION

We expect to have lots of concurrent processes running, so we need a way to avoid concurrency issues.

For this a transactional model introduces read locks and write locks. If you hold a read lock on X, you can only do reads. With a write lock, you can do both reads and writes.

EXAMPLE

Transaction 1:

```
Begin;
    ReadLock X;
    ReadLock Y;
    WriteLock Z;
    Z = X+Y;
Commit;
```

Transaction 2:

```
Begin;
    ReadLock Z;
    WriteLock X;
    WriteLock Y;
    X = Y-Z;
    Y = X+Z;
Commit;
```

Note: Running T_1 and then T_2 should leave $Z=X+Y$, and should swap X and Y relative to their initial values

WHY DOES OUR MODEL LOOK LIKE CODE?

The *real* program would be written in a language like C++

But the idea is to strip away everything except locking and data access operations.

So we still see a code-like structure, but now we think of it as a mathematical tool for describing our program.

CONCEPT: A STORAGE "EXECUTION TRACE"

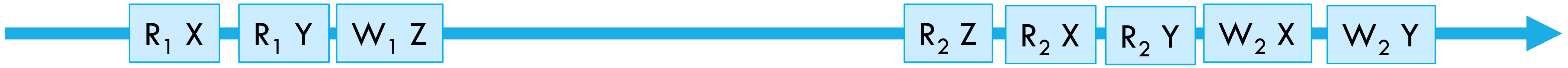
This is a time-line (left to right) showing the sequence of events as observed by the storage layer of the transactional system

Each read or write will be visible, but we don't show the locking requests (those are handled in a different layer, so they aren't part of the *storage* trace – they are part of a *concurrency control* execution trace)

EXAMPLE OF AN *EXECUTION TRACE*

```
Transaction 1:
    Begin;
        ReadLock X;
        ReadLock Y;
        WriteLock Z;
        Z = X+Y;
    Commit;
```

```
Transaction 2:
    Begin;
        ReadLock Z;
        WriteLock X;
        WriteLock Y;
        X = Y-Z;
        Y = X+Z;
    Commit;
```

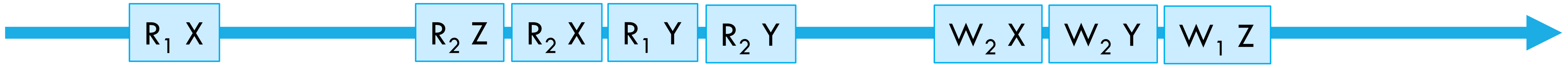


In this trace, time goes from left to right

EXAMPLE (INTERLEAVED)

```
Transaction 1:
    Begin;
        ReadLock X;
        ReadLock Y;
        WriteLock Z;
        Z = X+Y;
    Commit;
```

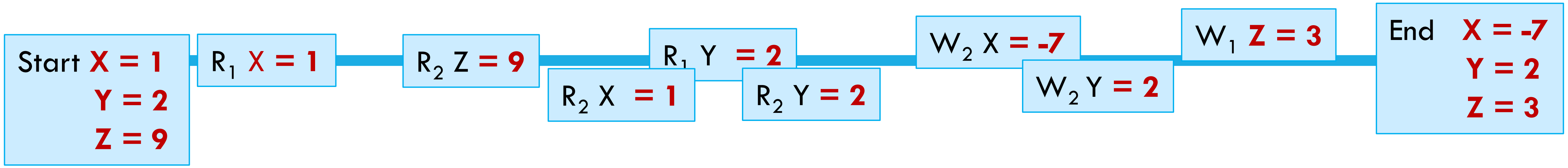
```
Transaction 2:
    Begin;
        ReadLock Z;
        WriteLock X;
        WriteLock Y;
        X = Y-Z;
        Y = X+Z;
    Commit;
```



EXAMPLE (INTERLEAVED)

```
Transaction 1:
    Begin;
        ReadLock X;
        ReadLock Y;
        WriteLock Z;
        Z = X+Y;
    Commit;
```

```
Transaction 2:
    Begin;
        ReadLock Z;
        WriteLock X;
        WriteLock Y;
        X = Y-Z;
        Y = X+Z;
    Commit;
```



DO THESE TRACES GIVE CORRECT RESULTS?

Suppose initially X=1, Y=2, Z=9

First trace:

T_1 leaves X=1, Y=2, Z=3

… then T_2 leaves X=2, Y=1, Z=3

DO THESE TRACES GIVE CORRECT RESULTS?

Suppose initially X=1, Y=2, Z=9

First trace:

T_1 leaves X=1, Y=2, Z=3

… then T_2 leaves X=2, Y=1, Z=3

Now consider trace 2 for X=1, Y=2, Z=9

Second trace:

They interleave, so we can't describe the output as if T_1 ran first, then T_2

… the outcome is X=-7, Y=2, Z=3

This can't happen if T_1 runs first, then T_2 or T_2 first followed by T_1

A FAMILIAR SITUATION! JUST LIKE CRITICAL SECTIONS WITH INTERFERENCE!

… And, if you work out some examples, you can easily confirm that although *some* interleavings are actually fine, this particular one leaves scrambled data.

We do want to allow "safe, correct" data-access interleaving:

- With lots of transactions (and some long, slow ones), a 1-by-1 approach would be impossibly slow
- Interleaving requests allows the code to run in parallel and speeds our system up, but it is important not to break the logic

WHY USE ABORT (IN CODE, NOT REAL LIFE)?

Suppose our code runs a big bank. Perhaps the customer’s credit limits would be exceeded by a request, but we can’t “verify account rules” until the end of the operation.

Abort undoes the effects – just as if the transaction never started

We also use abort to “clean up” if a crash disrupts a run

ACID MODEL, SERIALIZABILITY

Jim Gray and others proposed a simple set of rules to describe how transactions should behave: ACID

- **A**tomic: All or nothing.
- **C**onsistent: A correct transaction takes the data from one consistent state to another consistent state.
- **I**solation: If two transactions run at the same time, they should see one-another's pending (uncommitted) updates.
- **D**urability: Once committed, updates won't get lost.

ACID MODEL, SERIALIZABILITY

Serializability is another idea that Jim worked on in the early days of the model.

It says "Take a set of correct transactions. The transactional runtime can run them concurrently, interleaving reads and writes, as long as the outcome of the execution could have arisen in a serial (one by one) schedule (like "P ran first, then Q").

A BIT LIKE CRITICAL SECTIONS!

With critical sections we enforce that only one thing can run the protected block(s) of code at a time.

Transactions are using this concept but taking it a little further. P and Q can "simultaneously" access X and Y and Z. All we care about is the state at the end of the run when all the commit and abort operations are finished.

TWO-PHASE COMMIT

A “distributed protocol” aimed at solving a practical issue seen with transactions when data is spread over multiple servers.

Suppose that X and Y and Z are each held by different servers. When a transaction runs, it creates pending updates, X’, Y’, Z’. Commit makes these permanent… Abort would roll them back.

But how do we ensure “all or nothing” commit (or abort)?

FAILURES (CRASHES) MAKE IT HARD

Suppose server Y crashes and then restarts. The crash mangled transient update (Y’). Y can still abort but can no longer commit!

So, suppose T is trying to commit.

TWO-PHASE COMMIT

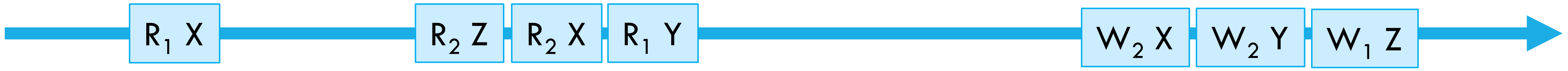
1. T says to X, Y and Z: are you able to commit?
2. X and Y and X **must first log X' and Y' and Z' on disk.** This is to ensure that even with a crash, they are still prepared to commit.
3. Then each replies: "I'm prepared to commit!"
4. T can commit if all three are prepared… but should abort if any doesn't respond or replies that it "must abort".
5. T also logs its decision, so if Y is down when T commits, later Y can find out what it should do. *We call this an* ***outcomes log.***
6. *Step 5 assumes the log is highly available, but there are ways to ensure this.*

PROBLEM SOLVED!

With two-phase commit, either all of the servers (eventually) commit and install the update, or all of them abort.

A crashed server will reboot with the update still pending, but won’t have lost it. So by checking the outcomes log, it learns that the transaction committed, and then it finalizes the outcome before resuming participation in the system. “Automatic repair”!

WHAT ABOUT LOCKING?

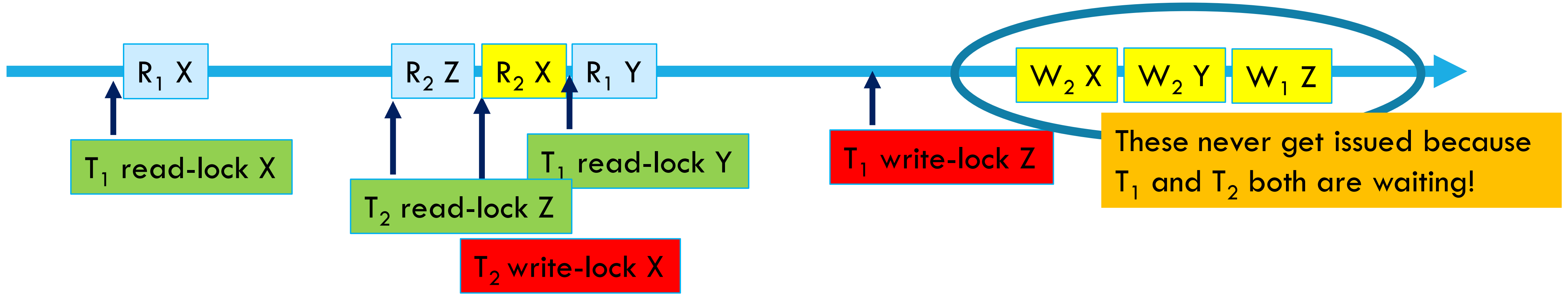


T_1 has a read lock on X, and wants a write lock on Z.

But T_2 has a write lock on Z, and is waiting for a read lock on X.

Deadlock! The red lock operations never complete… the yellow data reads and writes never actually occur!

WHAT ABOUT LOCKING?

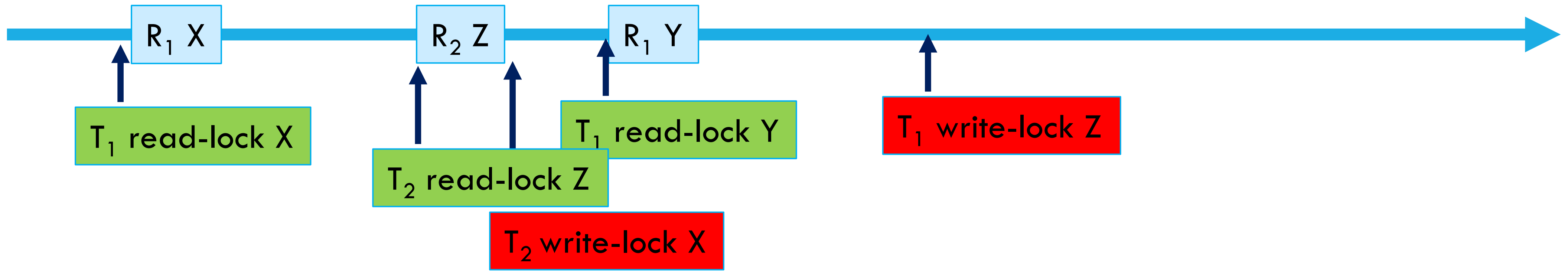


T_1 has a read lock on X, and wants a write lock on Z.

But T_2 has a write lock on Z, and is waiting for a read lock on X.

Deadlock! The red lock operations never complete… the yellow data reads and writes never actually occur!

WHAT ABOUT LOCKING?



T_1 has a read lock on X, and wants a write lock on Z.

But T_2 has a write lock on Z, and is waiting for a read lock on X.

Deadlock! The red lock operations never complete… the yellow data reads and writes never actually occur!

NOTICE THE WAIT-FOR CYCLE

T_1 waits for T_2. T_2 waits for T_1.

Any deadlock involves a cycle of this kind. A solution that cannot form lock cycles will be free of deadlocks.

For example: pre-agree on a locking order, like "you must get your locks in alphabetical order".

HOW WOULD THIS WORK?

In our sample transactions, the code for T_1 has no problems: it locks X, then Y, then Z.

The code was written as if T_2 would first lock Z, then X, then Y. This breaks the new rule!

- T_2 will need to be redesigned to ask for locks in X, Y, Z order
- Otherwise, at runtime, T_2 will get a "lock order exception"

PRACTICAL CONSEQUENCE?

T_2 would need to know what locks it will need from the moment it starts – it can't just walk through our storage system and get locks as it runs into objects.

Some transactions could definitely be written to anticipate future locking needs, but often this would be infeasible.

So the rule isn't always practical, but if it can be done, it works.

MORE LIMITATIONS

It turns out that ordered locking is not quite enough.

We also need the rule that a process always asks for the strongest form of locking it will ever need. So if T_1 wants to read X now, it can’t later try to upgrade its lock to a write lock. It needs a write lock “from the start” if it *might* update X.

Our examples didn’t need this form of “lock upgrade” but random fragments of code might not know, in advance, if they will read X now and try to update X later. So again, this might be tricky.

TWO-PHASE LOCKING WITH ORDERED LOCKS

This is a name for the rule that:

- Transactions get their locks in the proper order
- … and can never release a lock before the commit point, so they can't acquire, release, re-acquire
- There is a phase when locks are accumulated, then commit (or abort), then locks are released.
- Don't let the similarity of the name confuse you: this is not related to two-phase commit.

ADD IT ALL UP AND… IT WORKS!

Many modern computer systems use transactions.

Very easy to understand, simple coding style. For many applications, the basic rules aren’t too hard to follow.

Gives a basic and robust way to handle failures

GOOD THINGS ABOUT TRANSACTIONS

They are an easy model to understand.

Many packages implement the model.

Database systems like MySQL, Oracle, etc. have transactions "baked in." You talk to the database via a query/update API, and they handle everything.

TRANSACTIONS *ON MEMORY OBJECTS*

One big area of research involved “transactional memory”

The idea was that a language like C++ could support transactional objects, where the methods would execute as atomic actions.

All needed mechanisms (locking, versions, commit, abort) built in

"STM" VERSION OF TRANSACTIONAL MEMORY

A very famous idea used a hardware accelerator to try and speed up the costly steps, like checking to see if a commit can be done safely, maintaining versions and rolling back if needed

The other big effort used "software transactional memory" models, where the logic was entirely inserted by the compiler and the STL (or equivalent).

HOW DID (S)TM DO IN THE REAL WORLD?

People quickly found that although it is super easy to code this way, performance was sometimes unexpectedly terrible.

With lots of threads, aborts and retries became very common.

Over a few years, enthusiasm faded and by now, you don’t see a lot of use of the hardware, or even the STM version.

BROADER ISSUES WITH TRANSACTIONS

They bring a lot of "mechanism" that can be really costly if you didn't actually need so much infrastructure.

A common concern is that if a query or update almost never conflicts with other queries and updates, the overheads of locking and two-phase commit can be larger than the "work" you are actually doing.

RATE OF DEADLOCK, ABORT, RETRY

Many transactional systems can’t be sure the user’s code is deadlock free, so they check for deadlock, sense wait-for cycles and automatically abort some of the waiting transactions.

Then those are automatically restarted.

But if you do this, you sometimes see “exponential” numbers of retries, as a function of how many transactions are running.

JIM GRAY ON THIS PHENOMENON

Jim Gray studied this issue.

In cloud computing (cs5412) we will learn about his findings.

But in brief, he concluded that it is inherent to this form of synchronization and not at all easy to avoid.

MORE CONCERNS THAT ARE RAISED

With a non-transactional key-value storage system we get massive scalability mostly because every shard is running totally independently.   Then we add replication to make our shards fault-tolerant.

But with transactions, the execution on one shard becomes linked to things happening on other shards. We no longer have such an easy path to scalability.

ACTUAL EXPERIENCE?

In fact, teams that try to run full transactional infrastructures over distributed key-value storage have run into scalability issues.

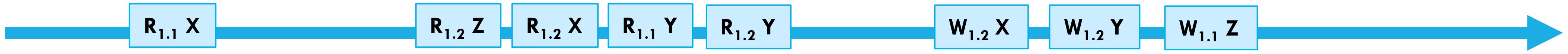
The key-value layer itself is just as scalable as ever.

But the transactional components get very sluggish and the system quickly comes to a halt.

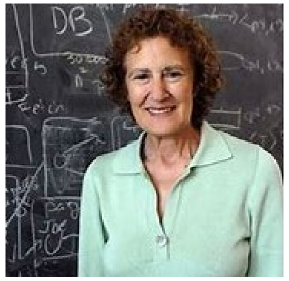
SUBTRANSACTIONS

Another common complaint is that real systems are modular

Suppose that component A talks to component B (perhaps B is an STL library, for example). What if *both* try to run transactions?



HOW THIS CAN WORK


Barbara Liskov

Luiba Shrira

Elliot Moss

The idea was explored by Barbara Lislov, Luiba Shrira and others at MIT, with Barbara's student Elliot Moss.

When A calls begin, this starts a transaction, maybe T_1.

Now, when B calls begin, we consider it to be a nested subtransaction running inside the context created by A: $T_{1.1}$

$T_{n.m}$ means step n of this process was a subtransaction, and within it this is the m'th sub-subtransaction, etc

THE DETAILS

Each lock request is understood to occur in the "scope" defined by the parent transaction. Locks are "inherited"

For example, if B requests a lock on X in transaction $T_{1.1}$, but then commits, T_1 inherits the lock – it isn't fully released.

$T_{1.2}$ can acquire this lock, but some other transaction, T_2, must wait until T_1 either commits or aborts.

EXTENDING TWO-PHASE COMMIT

Moss, Liskov and Shrira also showed that you need to track every server that any subtransaction ever talked to.

We “inherit” this list of servers up to the top level.

Then the top-level transaction, when it commits or aborts, must include all of the servers on this commit list.

MORE DETAILS

Total ordering (for lock acquisition) turns out to be very hard.

When your code for A called B, you had no idea what B would do.   And B was coded as part of a library: it has no idea what A was doing. How can we ensure a deadlock-free lock ordering?

Nobody ever really solved this puzzle!

CRASHES CREATE WEIRD ISSUES TOO

Suppose that A used a remote procedure call to talk to B, like if B was part of a key-value storage server but A was executing on some other machine and talking to it over the network.

Now, if A crashes, B might be still doing work on behalf of A, or holding locks and uncommitted data, etc.

THIS LEADS TO "ORPHAN TERMINATION"

The idea is to identify orphan transactions and terminate them

No need to check the commit log: they always abort. If the transaction leader died while running two-phase commit, the child transactions would know

The terminator. Killing orphans.

ULTIMATELY, NESTED TRANSACTIONS BECOME STANDARD BUT "UNPOPULAR"

Today it is easy to find products that support nested transactions

So you can definitely use this model if you wish

But the costs are quite high, and people rarely use these features in production code that cares about performance

UNDER THE HOOD?

Inside the slow transactional systems you would see a lot of lock waiting, and a lot of aborts.

When transactions abort they often need to be reissued (restarted from scratch). So the data layer is working hard yet nothing useful is happening.

This is like a form of livelock. It causes extremely high overheads.

ANOTHER OVERHEAD ISSUE

Applications with threads can also create serious issues for transactional systems.

Should each thread be viewed as a separate subtransaction, or should they be considered to be distinct “top level” transactions?

It turns out both answers lead to costly, problematic logic

CONSEQUENCE?

In some ways, the world has split.

Database users and big-data platforms often do use transactions, but they are more and more “read mostly”, with updates often occurring when the queries can temporarily pause

Many large distributed systems just don’t use transactions, at all

IDEAS PEOPLE HAVE PROPOSED

MSR FaRM Team: Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro

The nice aspect is the simplicity of the model… So researchers have tried to invent new ways to implement transactional key-value stores that won't have these scalability issues.

Some exciting recent work was done at Microsoft. They used a form of hardware accelerator (RDMA). We will discuss this solution, FaRM, in our next lecture. Microsoft Bing uses it.