Distributed Transaction Management – 2003

Jyrki Nummenmaa
http://www.cs.uta.fi/~dtm
jyrki@cs.uta.fi

Concurrency control optimisations

Locking means preparing for the worst

- The basic locking techniques are seen as *pessimistic* – they prepare for the worst.
- However, in practice conflicts can be rare.
- For instance a bank can have a huge number of txns in a second. However, the number of accounts is much bigger, and most txns do not conflict with any other txns.

Drawbacks of locking

- Locking uses resources.
- The computational effort needed for locking does not in practice depend on the number of actual lock conflicts.
- Locking makes deadlock possible. Deadlock management requires further resources or damages performance.
- To avoid cascading rollbacks, locks are usually only released at the end of the txn. This further reduces concurrency.

Optimistic lock management

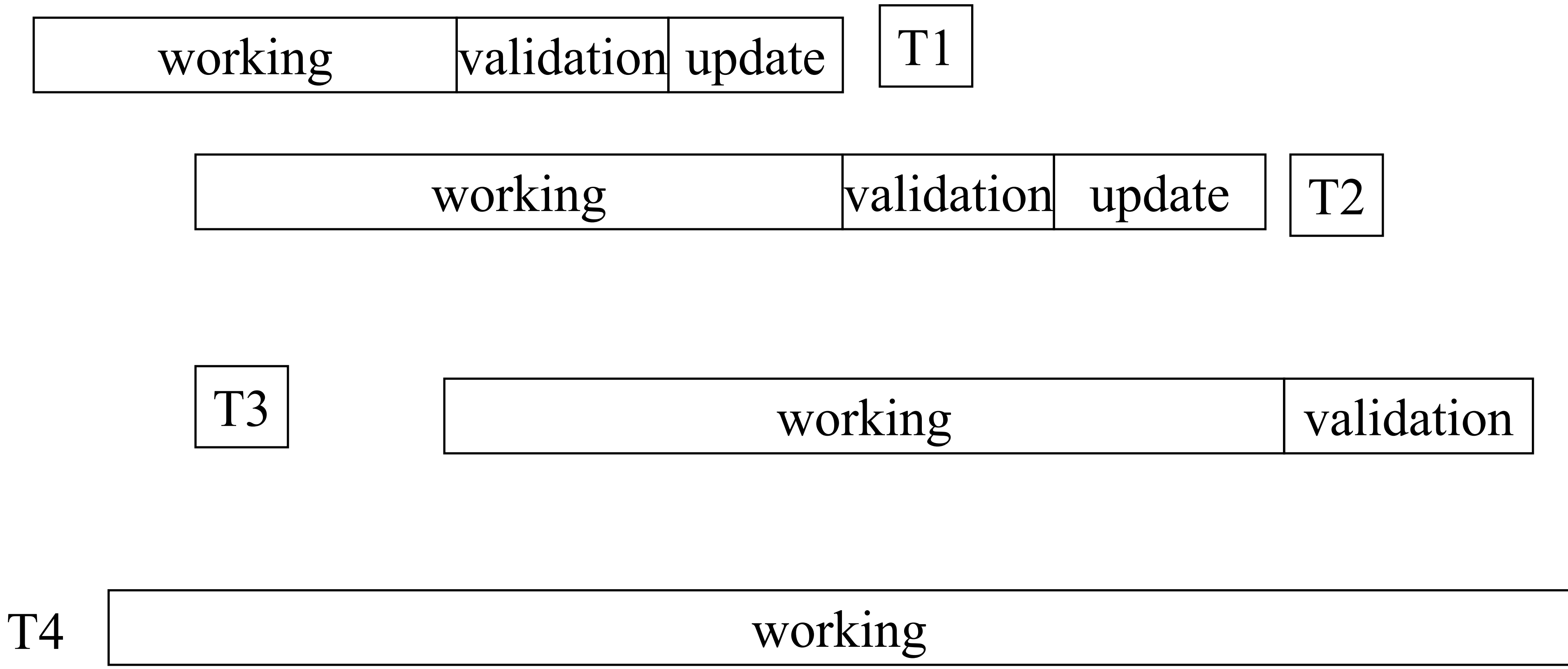
- By Kung and Robertson (1981)
- A txn will write all data into its private tentative copies of the data.
- All reads are performed either on its own tentative copies, if they exist, or on the (standard) last updated and committed value in the database. This means that no dirty data is being read, apart from the txns own data.
- The updates are validated and executed at the end of the txn.

Txn lifecycle

- The txn lifecycle contains three phases:
  1. Working phase
  2. Validation phase
  3. Update phase
- The txns are given txn ids, which also order them by age.
- Validation needs to be done in the order of ids (if a younger txn finishes its working phase earlier than some older txns, it must wait for the older txns to finish, before it can be validated).

Example

Q: What to check in each validation?



Txn validation

- Suppose we are validating a txn T against a txn T'.
- For this to be necessary, the lifetimes of T and T' must overlap.
- All of the following are not allowed:
  T reads data objects written by T'.
  T' reads data objects written by T.
  T and T' write same data objects.

Read sets and write sets

- To simplify checking, it is useful to maintain a read set (data objects read) and a write set (data objects written) for each txn.
- When validating T against T’, there is a conflict, if the write set of T overlaps either the write set or the read set of T’ or the write set of T’ overlaps either the read set or the write set of T.
- Suppose we write after validation. If we allow (on one site) only one txt to be in validation & write state at one time, then write data set conflicts can not occur and we do not need to compare a write set with a write set.

Backward validation

- Validation is processed in txn id order. Assume we are validating T.
- Therefore, all earlier txns' validation is done and, as a consequence, also their read operations are completed before the validation of T. Therefore, we only need to compare the read set of T with the write sets of these earlier txns.
- If we use backward validation and T has not read any data, no validation checks are needed.

Forward validation

- Now we compare the write set of the txn T to be validated against the read sets of all active txns.
- Notice that after validation it does not matter if the read sets of the active txns change.
- If T conflicts with some active txns, then as none of these has committed, we may roll back alternatively T or all conflicting active txns.

Backward validation algorithm

- StartTMax is the biggest txn id of any committed txn when T started
- FinishTMax is the biggest assigned txn id, when T entered validation.
-
```
boolean valid=true;
for (all txns T' from StartTMax+1 to
                              FinishTMax)
  if (read set of T intersects write set of T')
    valid = false; // and T will be rolled back
```
- Now the write sets of committed txns must be kept as long as overlapping txns are alive, to make validation possible.

Forward validation algorithm

- ActiveTMin is the smallest txn id of any active txn, when validation starts
- ActiveTMax is the greatest txn id of any active txn, when validation starts.
- ***boolean valid=true;***

```
boolean valid=true;
for (all txns T’ from ActiveTMin to
                              ActiveTMax)
  if (write set of T intersects read set of T’)
    valid = false; // some txn(s) are rolled
back
```

Comparison

- Backward validation requires us to store write sets of committed txns.
- Forward validation allows more flexibility on which txn to roll back. However, this may lead into some txn not making progress.
- Read sets are typically larger than write sets. Forward validation involves more large sets.

Synchronised validation

- We made a simplified assumption that only one txn is validated at one time.
- This is easily broken, if we have validations running on different servers.
- Synchronising the validations slows down processing.
- Suppose T completes before T’ on S1 and T’ before T on S2. Then, a straightforward validation requires T to be validated before T’ on S1 and T’ before T on S2. This deadlocks the system with synchronisation.

Parallel validation

- One option is to use global txn ids to order the validation on all servers.This, of course, will delay the validation and therefore commit of some txns.
- Another option is to validate first locally, and then check globally that the validation orders are the same on each servers.Now, of course, extra effort is needed for global validation and sorting out the problems.

Timestamp ordering

- In a way, timestamp ordering can be seen as an optimistic technique.
- In timestamping, operations are executed unless conflict rules forbid them.
- Conflict rules
  1. Ti may not write X, if some Tj, i<j, has read X.
  2. Ti may not read X, if some Tj, i<j, has written X.
- Rule number 2 can be relaxed, if old versions of X are kept, as then we just let Ti read an old version of X. If Rule 1 is triggered, a txn must be rolled back.
- Additionally, Ti may not write X, if some Tj, i<j, has written X. (Now we just skip the write.)

Example

- T1
  - Read X
  - Read Y
  - Write X
  - Read Z
  - Write Z
  - Commit
- T2
  - Read Y
  - Read Z
  - Write X
  - Write Y
  - Commit

Example –scheduled / 1

- T1.Read X
  T1.Read Y
  T1.Write X
  T1.Read Z
  T2.Read Y
  T1.Write Z
  T2.Read Z
  T2.Write X
  T2.Write Y
  T1: Commit
  T2: Commit

- This is a serializable schedule, equal to the serial schedule where T1 is executed first and T2 after T1.
- In all conflicting operation parts T1's operation comes first.

Example –scheduled / 2

- T1.Read X
  T1.Read Y
  T1.Write X
  T1.Read Z
  T2.Read Y
  T2.Read Z
  T2.Write X
  T2.Write Y
  T1.Write Z
  T1.Commit
  T2.Commit

- T1 reads X before T2 writes X. So T1 must be before T2.
- T2 reads Z, which is not written by T1, therefore T2 must be before T1.

Example –scheduled / 2

- T1.Read X
  T1.Read Y
  T1.Write X
  T1.Read Z
  T2.Read Y
  T2.Read Z
  T2.Write X
  T2.Write Y
  T1.Write Z
  T1.Commit
  T2.Commit

- T1 reads X before T2 writes X. So T1 must be before T2.
- T2 reads Z, which is not written by T1, therefore T2 must be before T1.

Example – distributed, no replication

- S1: T.Read X
  S1: T.Read Y
  S1: T.Write X
  S2: T.Read Z
  S1: T'.Read Y
  S2: T'.Read Z
  S1: T'.Write X
  S1: T'.Write Y
  S2: T.Write Z
  T1.Commit
  T2.Commit

- Even though this is globally not serializable, the local subtransactions are, on
  Site S1: T,T'
  Site S2: T',T
- This means that distribution provides extra challenge (e.g. globally ordering ids will do the trick).

Avoiding restart/rollback with timestamps

- We can avoid restart/rollback by delaying the operations until preceding operations (in timestamp order) have been performed.
- How do we know in a distributed system, if more preceding writes are expected?
- If operations come in timestamp order from other servers, then by examining the timestamps we know what timestamps are still to be expected.

Avoiding restart/rollback with timestamps

- What if a server has got no operations to send to some other server for some time? We would not want this to block progress.
- Solution: send empty (null) operations with just timestamps either regularly or when the other servers request them. However, this increases network traffic.

Transaction classes

- A problem with the optimisations this far is that they delay unnecessarily, even when the data sets are not going to conflict.
- We may benefit from knowing in advance the data items each txn reads and writes.
- We say that txn T belongs to class C, if the write set of T is a subset of write set of C and the read set of T is a subset of read set of C.
- We identify at startup time a class with each server.
- Then, we only need to wait for operations from such servers that they may have conflicting operations based on their read and write sets.

Example – distributed

- S1: T1.Read X
  S1: T1.Read Y
  S1: T1.Write X
  S2: T1.Read Z
  S1: T2.Read Y
  S2: T2.Read Z
  -> Wait for S2!
  S1: T2.Write X
  S1: T2.Write Y
  S2: T1.Write Z
  T1.Commit
  T2.Commit
- Now the read set and write set of T1 both contain X and Y and the read set of T2 contains just X.

Integrated methods

- Let us consider integration of locking and timestamping.
- Suppose we want to use 2-phase locking to synchronise reads and writes and timestamping to synchronise writes.
- While we execute the txns, we take locks to stop reads and writes from conflicting.
- We need a way by which the locks and timestamps can interact. For this, it is possible to give timestamps for locks.

2PL and timestamps

- Each data item X has a lock timestamp Lts(X).
- When a txn locks a data item X, it receives Lts(X).
- When T has taken all locks, its timestamp ts(T) is made to be larger than that of any lock timestamp for its locks.
- When T releases a lock on X, X is made to be max(ts(T), Lts(X)).
- These timestamps are consistent with the order of transactions given by two-phase locking.
- The reads and writes are synchronised using locking while the txn is running (e.g. 1 lock to read, n to write).
- When writes are made permanent at the end, the writes are synchronised using timestamps by just ignoring the writes that are late.
- This way a writelock never conflicts with a writelock.

Optimising deadlock detection

- The methods proposed earlier require all of the waits-for graph to be transmitted from one server to another.
- Consider, again, a bank with a large number of txns, out of which very few are likely to have lock conflicts.
- It is known that a typical deadlock involves just two transactions.
- Under these circumstances, transporting huge waits-for graphs seems like a waste of resources.

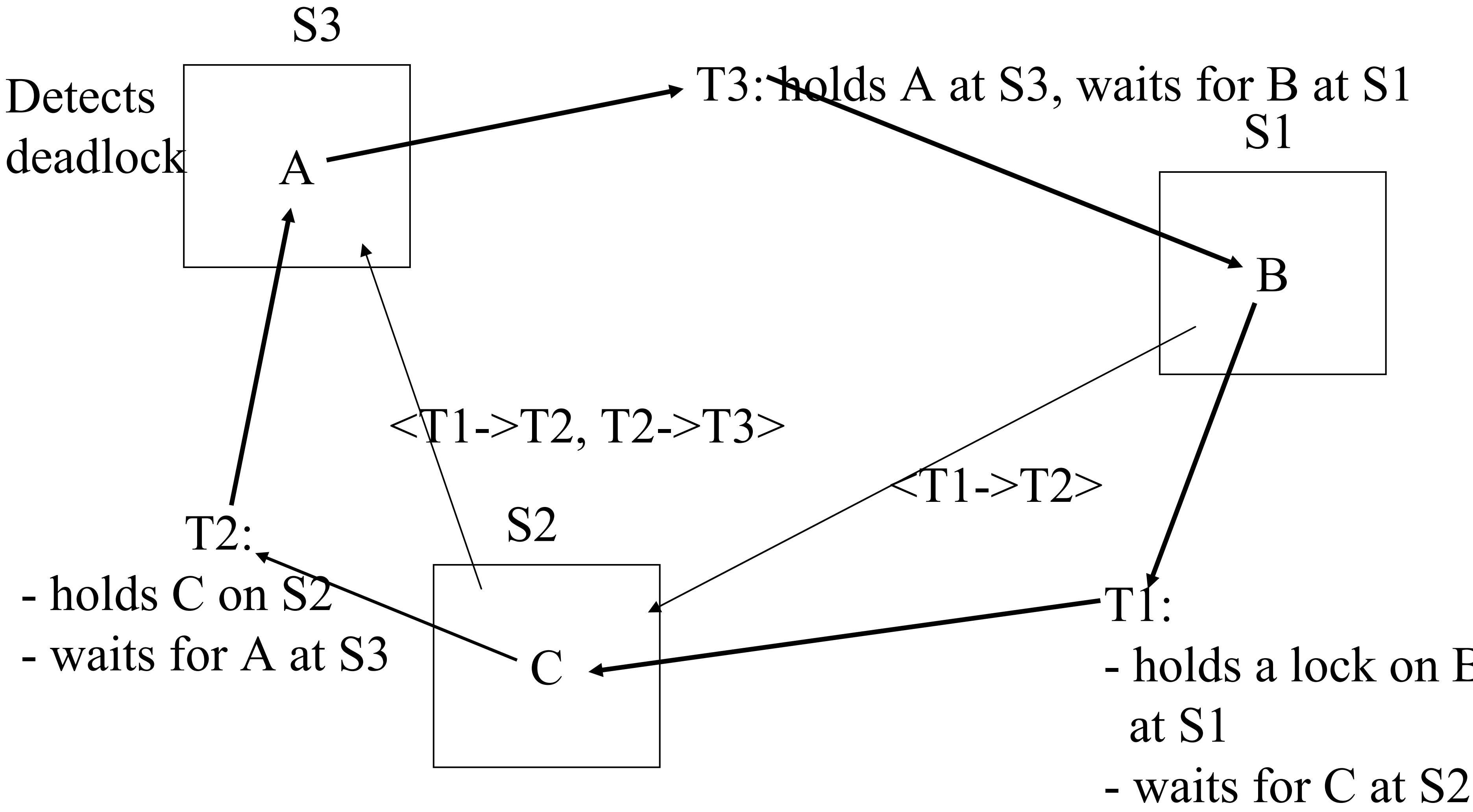
Chandy-Misra-Haas Algorithm

- The idea is to just chase the edges in the waits-for graph, which are suspected in participating a cycle.
- A suspected edge on server S is such that some txn T waits for U, and U waits for a data item held at some other server S'.
  - Local waits do not initiate such suspicion.
  - If T is not waiting for some *waiting* txn U, it can not participate in a deadlock.

Initiating the deadlock detection

- If we suspect that T->U is part of a waits-for cycle, we send a probe with graph <T->U> to the server, where U waits.
- If U waits for V, then U->V is added to the graph. This way, the graph grows (now < T->U, U->V >) and is transmitted further.
- Because of shared locks, a txn may in fact wait for several txns. All these waits imply potential edges in the graph.
- In fact, the probe could be given to one (coordinating) local txn, which then forwards it.

Example


S3
Detects deadlock
A
T3: holds A at S3, waits for B at S1
S1
B
<T1->T2, T2->T3>
<T1->T2>
S2
T2:
- holds C on S2
- waits for A at S3
C
T1:
- holds a lock on B
at S1
- waits for C at S2

Detecting deadlocks

- If at some point it is found out that the graph contains a cycle, it indicates a deadlock.
- It may happen that the same deadlock is detected on several servers.
- To minimize the number of rolled back txns (and to guarantee progress) it is a good idea to roll back the youngest txn.

Storing probes

- When a server receives a probe message, it may be that there are no edges by which the probe graph is grown and sent further.
- However, it may be that such a new wait is initiated that it should be added to the graph.
- Then the probe messages are also sent forward.

Phantom deadlocks

- If 2-phase locking is not used, txns releasing locks may cause phantom deadlocks to be detected.
- Some txn may participate in another deadlock and be aborted simultaneously as some other txn is aborted to solve another deadlock, although one txn might have ben used both aborts.
- In fact in all deadlock detection schemes the following may happen:
  - It may be that one of the txns in the cycle is aborted for some local reason on some of the sites, and this information is not available soon enough to cancel finding a deadlock.

2PC Implementation

- We will have a look at the example implementation (and it will be made available after the lecture).