The University *of* New South Wales School *of* Computer Science & Engineering

COMP9243 — Week 5 (18s1)

Ihor Kuz, Manuel M. T. Chakravarty & Gernot Heiser

Synchronisation and Coordination (Part 2)

Transactions

A *transaction* can be regarded as a set of server operations that are guaranteed to appear atomic in the presence of multiple clients and partial failure. The concept of a transaction originates from the database community as a mechanism to maintain the consistency of databases. Transaction management is build around two basic operations: `BeginTransaction`, `EndTransaction`. An `EndTransaction` operations causes the whole transaction to either `Commit` or `Abort`. For this discussion, the operations performed in a transaction are `Read` and `Write`.

Transactions have the *ACID* property:

Atomic: all-or-nothing, once committed the full transaction is performed, if aborted, there is no trace left;

Consistent: concurrent transactions will not produce inconsistent results;

Isolated: transactions do not interfere with each other, i.e., no intermediate state of a transaction is visible outside; (this is also called the serialisable property)

Durable: all-or-nothing property must hold even if server or hardware fails.

Flat vs Nested Transactions

If a significant amount of state is altered in a single transaction, an abort shortly before the transaction would have been completed can imply a significant wasted effort and, if this occurs repeatedly, seriously degrade the performance of a system. *Nested transactions* subdivide a complex transaction into many smaller *sub-transactions*. As a result, if a single sub-transaction fails, the work that went into the others is not necessarily wasted.

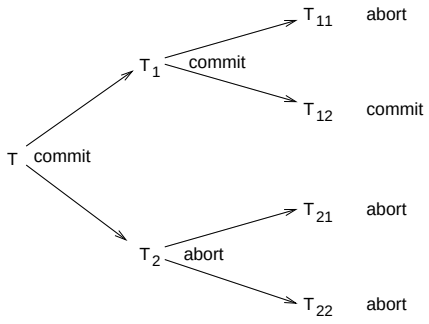

Figure 1: Nested transactions

Figure 1 displays such a hierarchical decomposition. Generally, a parent transaction may commit even if some child transactions abort. However, if a parent transactions aborts, it forces all child transactions to abort. As a result, child transactions can abort at any time, but they cannot commit until the parent transaction is ready to commit. An example of this is booking a trip that involves a number of separate flights. Because booking individual parts of the trip may be difficult enough, if any single flight cannot be booked it may still be desirable to commit the

rest of the transaction and try to find an alternative for the failed (or aborted) part later (as a separate transaction).

Transaction Atomicity Implementation

Two general strategies exist for the implementation of atomicity in transactions:

Private Workspace: all *tentative* operations are performed on a *shadow copy* of the server state, which is atomically swapped with the main copy on `Commit` or discarded on `Abort`.

Writeahed Log: updates are performed in-place, but all updates are logged and reverted when a transaction `Abort`s.

Concurrency in Transactions

It is often necessary to allow transactions to occur simultaneously (for example, to allow multiple travel agents to simultaneously reserve seats on the same flight). Due to the consistency and isolation properties of transactions concurrent transaction must not be allowed to interfere with each other. Concurrency control algorithms for transactions guarantee that multiple transactions can be executed simultaneously while providing a result that is the same as if they were executed one after another.

A key concept when discussing concurrency control for transactions is the serialisation of *conflicting operations*. Recall that conflicting operations are those operations that operate on the same data item and whose combined effects depend on the order they are executed in. We define a schedule of operations as an interleaving of the operations of concurrent transactions. A legal schedule is one that provides results that are the same as though the transactions were serialised (i.e., performed one after another). This leads to the concept of *serial equivalence*.

A schedule is serially equivalent if all conflicting operations are performed in the same order on all data items. For example, given two transactions T_1 and T_2 in a serially equivalent schedule, then of all the pairs of conflicting operations the first operation will be performed by T_1 and the second by T_2 (or vice versa: of all the pairs the first is performed by T_2 and the second by T_1).

There are three type of concurrency control algorithms for transactions: those using locking, those using timestamps, and those using optimistic algorithms.

Locking

The locking algorithms require that each transaction obtains a lock from a scheduler process before performing a read or a write operation. The scheduler is responsible for granting and releasing locks in such a way that legal schedules are produced. The most widely used locking approach is *two-phase locking* (2PL). In this approach a lock for a data item is granted to a process if no conflicting locks are held by other processes (otherwise the process requesting the lock blocks until the lock is available again). A lock is held by a process until the operation it was requested for has been completed. Furthermore once a process has released a lock, it can no longer request any new locks until its current transaction has been completed. This results in a *growing phase* of the transaction where locks are acquired, and a *shrinking phase* where locks are released.

While this approach results in legal schedules, it can also result in deadlock when conflicting locks are requested in reverse orders. This problem can be solved either by detecting and breaking deadlocks or by adding timeouts to the locks (when a lock times out then the transaction holding the lock is aborted). Another problem is that 2PC can lead to *cascaded aborts*. If a transaction (T_1) reads the results of a write of another transaction (T_2) that is subsequently aborted, then the first transaction (T_1) will also have to be aborted. The solution to this problem is called *strict two-phase locking* and allow locks to be released only at commit or abort time.

Timestamp Ordering

A different approach to creating legal schedules is to timestamp all operations and ensure that operations are ordered according to their timestamps. In this approach each transaction receives a unique timestamp and each operation receives its transaction's timestamp. Each data item also has three timestamps - the timestamp of the last committed write, the timestamp of the last read, and the timestamp of the last tentative (noncommitted) write. Before executing a write operation the scheduler ensures that the operation's time stamp is both greater than the data item's write timestamp and greater than or equal to the data item's read timestamp. For read operations the operation's time stamp must be greater than the data item's write timestamps (both committed and tentative). When scheduling conflicting operations the operation with a lower timestamp is always executed first.

Optimistic Control

Both locking and timestamping incur significant overhead. The optimistic approach to concurrency control assumes that no conflicts will occur, and therefore only tries to detect and resolve conflicts at commit time. In this approach a transaction is split into three phases, a *working phase* (using shadow copies), a *validation phase*, and an *update phase*. In the working phase operations are carried out on shadow copies with no attempt to detect or order conflicting operations. In the validation phase the scheduler attempts to detect conflicts with other transactions that were in progress during the working phase. If conflicts are detected then one of the conflicting transactions are aborted. In the update phase, assuming that the transaction was not aborted, all the updates made on the shadow copy are made permanent.

Distributed Transactions

In contrast to transactions in the sequential database world, transactions in a distributed setting are complicated because a single transaction will usually involve multiple servers. Multiple servers may involve multiple services and files stored on different servers. To ensure the atomicity of transactions, all servers involved must agree whether to `Commit` or `Abort`. Moreover, the use of multiple servers and services may require *nested transactions*, where a transaction is implemented by way of multiple other transactions, each of which can independently `Commit` or `Abort`.

Transactions that span multiple hosts include one host that acts as the *coordinator*, which is the host that handles the initial `BeginTransaction`. This coordinator maintains a list of *workers*, which are the other servers involved in the transaction. Each worker must be aware of the identity of the coordinator. The responsibility for ensuring the atomicity of an entire transaction lies with the coordinator, which needs to rely on a *distributed commit protocol.*

Two Phase Commit

This protocol ensures that a transaction commits only when all workers are ready to commit, which, for example, corresponds to validation in optimistic concurrency. As a result a distributed commit protocol requires at least two phases:

1. *Voting phase:* all workers vote on commit; then the coordinator decides whether to commit or abort.

2. *Completion phase:* all workers commit or abort according to the decision of the coordinator.

This basic protocol is called *two-phase commit (2PC)* [LS76].

Figure 2 displays the state transition diagram implemented by the coordinator in the 2PC protocol. It starts by sending `CanCommit` messages to all workers, which answer with either a *yes* or *abort* message. If *yes* messages are received from all workers, the coordinator sends a `DoCommit` message to all workers, which then implement the `Commit`. However, if any of the workers replies

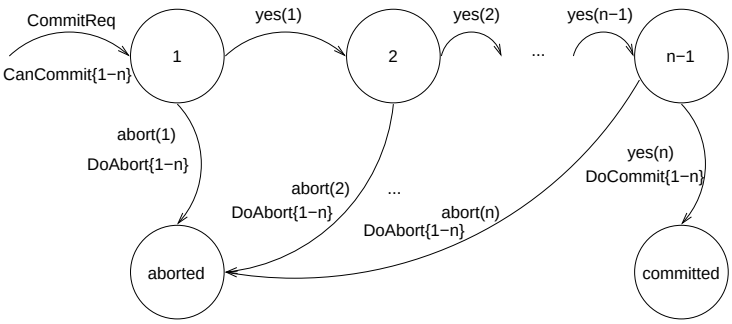

Figure 2: State transition diagram of simple 2PC coordinator

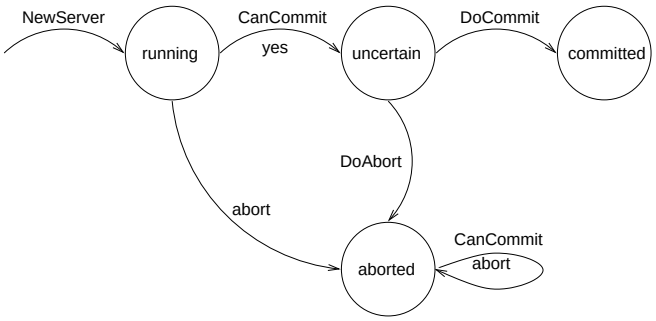

Figure 3: State transition diagram of simple 2PC worker

with *abort* instead of *yes*, the coordinator sends a `DoAbort` message to all workers to trigger a collective `Abort`.

The behaviour of the coordinator is complemented by that of the workers, of which the state transition diagram is displayed in Figure 3. A worker on receiving a `CanCommit` message answers with either a *yes* or *abort* message, depending on the workers internal state. Afterward, it commits or aborts depending on the instructions from the coordinator. The only extra case is where the worker is in an abort cycle and immediately answers a `CanCommit` with an *abort*.

The simple 2PC protocol has two main limitations:

- Once a node has voted with *yes*, it cannot retract this decision—for example, if it subsequently crashes. Hence, each node has to ensure that, after having sent a *yes*, it can eventually complete the transaction (possibly by logging the required state update to persistent storage).
- If the coordinator crashes, it blocks all workers. This can be avoided by moving to a decentralised three-phase commit protocol [Ske81].

Distributed Nested Transactions

Distributed nested transactions are realised by letting sub-transactions commit provisionally, whereby they report a *provisional commit list* that contains all provisionally committed sub-transactions to the parent. If the parent aborts, it aborts all transactions on the provisional commit list. Otherwise, if the parent is ready to commit, it lets all sub-transactions commit. The actual transition from provisional to final commit needs to be via a 2PC protocol, as a worker may crash after it has already provisionally committed. Essentially, when a worker receives a `CanCommit` message, there are two alternatives:

1. If it has no recollection of the sub-transactions involved in the committing transaction, it votes *abort*, as it must have recently crashed.

2. Otherwise, it saves the information about the provisionally committed sub-transaction to a persistent store and votes *yes*.

Coordination and Multicast

Recall that group communication provides a model of communication whereby a process can send a single message to a group of other processes. When such a message is sent to a predefined group of recipients (as opposed to all nodes on the network), we refer to the communication as *multicast*. Since a multicast is sent out to a specific group, it is important to have agreement on the membership of that group. We distinguish between static group membership, where membership does not change at runtime, and dynamic membership, where the group membership may change. Likewise we distinguish between open and closed groups. In an open group anyone can send a message to the group, while in a closed group only group members can send messages to the group.

Besides group membership there are two other key properties for multicast: reliability and ordering. With regards to reliability we are interested in message delivery guarantees in the face of failure. There are two delivery guarantees that a reliable multicast can provide: the guarantee that if the message is delivered to any member of the group it is guaranteed to be delivered to all members of the group; and the slightly weaker guarantee that if a message is delivered to any non-failed member of the group it will be delivered to all non-failed members of the group. We will further discuss multicast reliability in a future lecture.

With regards to ordering we are interested in guarantees about the order in which messages are delivered to group members. This requires synchronisation between the group members in order to ensure that everyone delivers received messages in an appropriate order. We will look at four typical multicast ordering guarantees: basic (no guarantee), FIFO, causal, and total order.

Before discussing the different ordering guarantees and their implementations, we introduce the basic conceptual model of operation for multicast. A multicast sender performs a single multicast send operation `msend(g,m)` specifying the recipient group and the message to send. This operation is provided by a multicast middleware. When it is invoked, it eventually results in the invocation of an underlying unicast `send(m)` operation (as provided by the underlying OS) to each member of the group. At each recipient, when a message arrives, the OS delivers the message to the multicast middleware. The middleware is responsible for reordering received messages to comply with the required ordering model and then delivering (`mdeliver(m)`) the message to the recipient process. Only after a message has been mdelivered is it available to the recipient and do we say that it has been successfully delivered.

Note that multicast implementations do not typically follow this model directly. For example, unicasting the message to every member in the group is inefficient, and solutions using multicast trees are often used to reduce the amount of actual communication performed. Nevertheless the model is still a correct generalisation of such a system. For the following we also assume a static and closed group. We will discuss how to deal with dynamic groups in a future lecture together with reliable multicast. An open group can easily be implemented in terms of a closed group by having the sender send (i.e., unicast) its message to a single member of the group who then multicasts the message on to the group members.

In basic multicast there are no ordering guarantees. Messages may be delivered in any order. The stricter orderings that we discuss next can all be described and implemented making use of basic multicast to send messages to all group members. This means that any suitable basic-multicast implementation, such as IP multicast, can be used to implement the following.

FIFO multicast guarantees that message order is maintained per sender. This means that messages from a single sender are always mdelivered in the same order as they were sent. FIFO multicast can be implemented by giving each group member a local send counter and a vector of the counters received from the other group members. The send counter is included with each

message sent, while the receive vector is used to delay delivery of messages until all previous messages from a particular sender have been delivered.

The causal message delivery guarantee requires that order is maintained between causally related sends. Recall that a message B is causally related to a message A (A happens before B) if A is received at the sender before B is sent. In this case all recipients must have causally related messages delivered in the happened before order. There is no ordering requirement for concurrent messages. Causal multicast can be implemented using a vector clock at each group member. This vector is sent with each message and is used to track causal relationships. A message is queued at a receiver until all previous (i.e., happened before) messages have been successfully delivered.

Finally, totally-ordered multicast guarantees that all messages will be delivered in the same order at all group members. There are various approaches to implementing totally-ordered multicast. We briefly present two: a sequencer-based approach, and an agreement-based approach. The sequencer-based approach requires the involvement of a centralised sequencer (which can be either a separate node, or one of the group members). In this case, all messages are assigned a sequence number by the sequencer, with all recipients delivering messages in the assigned sequence order.

The agreement-based approach is decentralised, and requires all group members to vote on a sequence number for each message. In this approach each process keeps a local sequence number counter. After receiving a message each process replies to the sender with a proposed sequence number (the value of its counter). The sender chooses the largest proposed sequence number and informs all the group members. Each group member then assigns the sequence number to the message and sets its counter to the maximum of the counter's current value and the sequence number. Messages are delivered in their sequence number order.

Totally-ordered multicast is often combined with reliable multicast to provide *atomic multicast*. Note that most practical totally-ordered multicast implementations are based on some form of the sequencer-based approach, typically with optimisations applied to prevent overloading a single node.

Coordination and Elections

Various algorithms require a set of peer processes to elect a leader or coordinator. In the presence of failure, it can be necessary to determine a new leader if the present one fails to respond. Provided that all processes have a unique identification number, leader election can be reduced to finding the non-crashed process with the highest identifier. Any algorithm to determine this process needs to meet the following two requirements:

1. *Safety:* A process either doesn't know the coordinator or it knows the identifier of the process with largest identifier.
2. *Liveness:* Eventually, a process crashes or knows the coordinator.

Bully Algorithm

The following algorithm was proposed by Garcia-Molina [GM82] and uses three types of messages:

- `Election:` Announce election
- `Answer:` Response to an election
- `Coordinator:` Elected coordinator announces itself

A process begins an election when it notices through a timeout that the coordinator has failed or receives an `Election` message. When starting an election, a process sends `Election` message to all higher-numbered processes. If it receives no `Answer` within a predetermined time bound, the process that started the election decides that it must be coordinator and sends a `Coordinator` message to all other processes. If an `Answer` arrives, the process that triggered an election, waits a pre-determined period of time for a `Coordinator` message. A process that receives an

`Election` message can immediately announce that it is the coordinator if it knows that it is the highest numbered process. Otherwise, it itself starts a sub-election by sending `Election` message to higher-numbered processes. This algorithm is called the bully algorithm because the highest numbered process will always be the coordinator.

Ring Algorithm

An alternative to the bully algorithm is to use a ring algorithm [CR79]. In this approach all processes are ordered in a logical ring and each process knows the structure of the ring. There are only two types of messages involved: `Election` and `Coordinator`. A process starts an election when it notices that the current coordinator has failed (e.g., because requests to it have timed out). An election is started by sending an `Election` message to the first neighbour on the ring. The `Election` message contains the node's process identifier and is forwarded on around the ring, with each process adding its own identifier to the message. When the `Election` message reaches the originator, the election is complete. Based on the contents of the message that originator process determines the highest numbered process and sends out a `Coordinator` message specifying this process as the winner of the election.

References

[CR79] E. G. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 22(5):281–283, 1979.

[GM82] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1), January 1982.

[LS76] Butler Lampson and H. Sturgis. Crash recovery in a distributed system. Working paper, Xerox PARC, Ca, USA, Ca, USA, 1976.

[Ske81] D. Skeen. Nonblocking commit protocols. In *SIGMOD International Conference on Management of Data*, 1981.