6.826 Principles of Computer Systems **Fall Semester, 1992**

Lecture 21: November 30, 1992

Lecturer: William Weihl *Scribe: Frank Ho*

1 Distributed Transactions

The transaction system, as we may recall, is used to guarantee the atomicity and serializability of actions. That is, we want to make sure that the effect is as if only one transaction is performed at any given time. In a distributed implementation, however, the transaction system is located at several different sites. It must be able to tolerate partial failures — failure of various sites of the system. In the implementation of a distributed system, the main issues we need to address are as follows:

1. partial failures
2. coordinating sites for synchronization

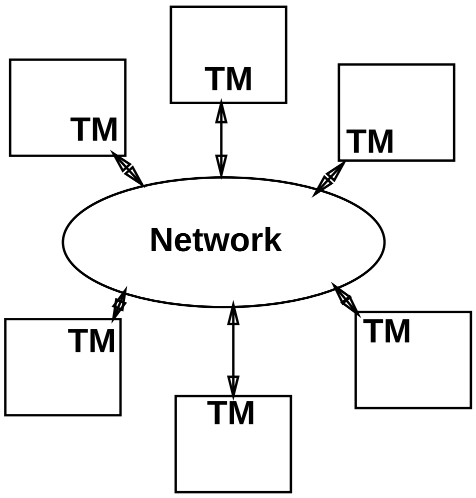

Figure 1: A distributed transaction system

2 Partial Failure

The distributed system must be able to provide service even if parts of the system are down. For a distributed transaction system, this requirement cannot be satisfied with the simple-minded implementation of a central lock manager. Consider the scenario in which the lock manager is the only agent that controls access to all shared objects in the system. We can see that there are two problems with this scheme. First, too much is dependent on the lock manager. Since all shared objects can only be accessed with the coordination of the central lock manager, the entire system fails if the lock manager fails. It therefore becomes a reliability bottleneck. Second, the system suffers from poor availability. The lock manager would be given too heavy a load if each shared object access in the system involves the lock manager. Thus it is also a performance bottleneck. A distributed system should alleviate the effect of a site failure, by spreading the management of transactions through the system.

In our model of the distributed transaction system, a *transaction manager* (TM) is associated with each site and maintains a local log for the stable state of the site. This log consists of a redo and an undo log. The network layer may lose or duplicate messages, but it cannot corrupt a message. Transactions may move to other sites by means of remote procedure calls. A transaction may, for example, start at site X and move through several sites in the system during its execution. We shall assume that a transaction commits at the site where it is started. Note that a transaction can do one of the following:

1. Commit successfully at all sites and persist

2. Abort, such that none of its modifications affect any other transactions

That is to say, every transaction either commits or aborts completely. We shall note that a transaction may abort at *any* time.

The issue here is to determine when a transaction can commit. It is unacceptable if some sites decide to commit a transaction and others decide to abort it. Therefore, we need a communication protocol that enforces *atomic commitment*, even if some sites fail while executing the protocol.

3 2-Phase Commit Protocol

The 2-phase commit protocol (2PC) is a scheme for ensuring atomic commitment. For each transaction, one site is chosen as the *coordinator*, which is responsible for controlling the commit procedure. We may assume that a transaction's coordinating site is the same site that it starts and commits. Other sites visited by this transaction are called the *participants*.

In the 2-phase commit protocol, a transaction's coordinator has to communicate with all its participants when it attempts to commit. First, in the *prepare* phase, the coordinator collects votes for commitment from the participants. It sends a vote request to each participant and waits for their replies. Each participant may either vote ‘yes’ or ‘no’. A participant votes ‘yes’ if it is ready to allow the transaction to commit; otherwise it votes ‘no’. The participant sends a ‘yes’ vote only if it has ensured that all modifications for the transaction are on stable storage and are therefore recoverable. This action requires a log force. Once a participant has sent the ‘yes’ vote, it may not change its mind unless told to do so.

If all the participants vote ‘yes’, then the coordinator decides to commit. Otherwise it must abort the transaction because some participants are not ready to commit. After the coordinator makes the decision, it sends the decision to all participants. This is the decision phase. When a participant receives the decision, it performs the updates associated with the transaction, and releases the locks.

4 Managing Failures

The 2-phase commit protocol needs to handle failures of the network, the participants, and the coordinator.

4.1 Network Failures

The network may lose or duplicate messages. If the coordinator thinks that a vote request message has been lost (perhaps because a participant has not voted after a long time since the request was sent), it retransmits the vote request, but timeouts after a number of trials and decides to abort the transaction. If a participant receives a duplicate vote request, it sends the vote again. The coordinator can simply ignore a duplicate vote. Retransmissions are also used in the decision phase to accommodate network failures.

4.2 Participant Failures

A participant may fail during any part of the commit protocol. That is, it may fail either before or after voting. Different measures are taken to handle each case.

4.2.1 Participant fails before voting

If the participant fails before voting, it might lose updates, locks, and other information about the transaction. If it does not find any record of the transaction that the coordinator is asking it to vote on, it votes to abort the transaction.

On the other hand, it may not vote ‘yes’ merely because it has a record. The following scenario illustrates the problem:

1. Transaction T visits participant X

2. T goes to another site
3. X crashes and forgets about T
4. X recovers
5. T visits X again, so that X has a new record for T

In this case, it is important for the participant to know that it has lost part of the transaction's records. Several schemes can be used to achieve this :

- The participant logs a record to stable storage *each* time transaction T visits it.
- The participant maintains a *crash count*. The transaction maintains a record of the crash count of each participant at the time it visits it. If it sees that the crash count for any participant has increased since it last visited it, the transaction is aborted.
- The participant marks itself on the transaction if it *thinks* that this is the first time the transaction visits it. The transaction knows that a participant has crashed if it has marked more than once on the transaction.

4.2.2 Participant fails after voting

There are two cases when a participant fails after voting. If the participant voted to abort, there is little that needs to be done on recovery. However, if the participant voted to commit, then it must recover the transaction's state prior to failure. In particular, it must reapply the locks held by the transaction. To this end, the participant must have forced all information about the transaction to the disk and marked the transaction as *prepared* before voting yes to the coordinator. The record thus forced is called the *prepare record*.

4.3 Coordinator Failures

The 2-phase commit protocol requires that the coordinator distribute consistent information to the participants, even if it crashes during the protocol.

If the coordinator fails before it decides upon the transaction, it may not have any record of the transaction on recovery. Note that once a participant has prepared a transaction, it expects to hear from the coordinator about the decision. If it does not receive the decision in a while, it starts to query the coordinator. If a coordinator that has no record of the transaction receives such a query, it can simply ask the participant to abort. In a later section, we explore alternatives to this scheme.

When the coordinator decides upon the transaction, it forces a commit/abort record to the disk. This is the decisive action when the transaction actually gets committed or aborted. The coordinator then sends the decision to all participants, and to the client, if any.

If the coordinator crashes before some participant has received the decision, the participant will be *blocked* until the coordinator recovers. Thus, 2-phase commit is not guaranteed to terminate if the coordinator fails and does not recover. Normally, it would recover, and retrieve the decision about the transaction from stable storage, if it made any.

Protocols like 3-phase commit and Paxos try to avoid the problem of participants blocked on a failed coordinator by bringing up new "leaders", which try to resume the work of an earlier leader that has supposedly failed. But these protocols do not guarantee termination either, they only make it more likely.

5 Discarding Information

When can the coordinator discard the commit/abort record for a transaction? A simple answer is: when all participants have received the decision and recorded it stably, so that no participant will ever again ask the coordinator for the result — even after a crash. Thus, when a participant receives the decision, it records it stably and sends back an ack to the coordinator. When the coordinator has received acks from all participants, it may forget all information about the transaction. For this, the coordinator must have stably recorded the set of participants along with the decision record; otherwise, if it crashed and recovered, it would have no way to get acks from the participants, or to even know that it has received acks from all participants. Also, note that a stable write is required even to "forget" the information — if logging is used, an "EndTransaction" record needs to be appended. All records of a transaction ending with such a record can be truncated (discarded).

The above protocol involves four rounds of messages between the coordinator and the participants. This is the unoptimized version of 2-phase commit, often said to be the *Presumed-Nothing* scheme. We describe two ways to reducing the number of message rounds under certain circumstances.

5.1 Presumed-Abort

If the coordinator decides to abort a transaction, it need not force any decision record. When a participant receives an abort decision, it need not send an ack. If the coordinator crashes, losing the decision, and is queried on recovery for the decision, it finds no record of the transaction, and can presume that it aborted. If the decision is commit, however, the coordinator must force the decision, and the participants must send an ack back, so that the coordinator may garbage collect the decision record after it has received all acks.

The drawback with Presumed-Abort is that it optimizes the uncommon case. The common case is commit, which still takes four rounds of messages.

5.2 Presumed-Commit

The strategy here is that if the coordinator is queried by a prepared participant about a transaction the coordinator has no record of, it presumes that the transaction must have committed. This allows the coordinator to forget about a committed transaction without receiving acks from the participants.

However, the scheme as described above faces a serious problem: what if the coordinator had crashed before deciding a transaction, and is queried about it on recovery? It will be wrong to reply that the transaction committed. To fix this, the coordinator has to force a prepare record itself before sending out prepare messages to the participants. This record contains the set of participants and marks the transaction as being in phase one. Later, if the coordinator commits the transaction, it forces an "EndTransaction" record that nullifies the earlier record. If the coordinator crashes before committing, it will detect the prepare record and may choose to either abort the transaction or continue with it. If it crashes after the commit, then on recovery it will find no valid prepare record of the transaction, and will therefore respond to queries with a commit decision.

There is another caveat to be observed. If the coordinator is queried by an *unprepared* participant about a transaction the coordinator has no record of, then it must respond with "abort" instead of "commit". Such a situation arises when the coordinator failed *before* initiating the 2-phase protocol, and therefore the proper decision is "abort". From this point of view, the protocol described above is really Presumed-Abort to begin with, and switches to Presumed-Commit after the coordinator forces the initial prepare record.

Although this scheme requires only 3 rounds of messages for commits, it has an overcoming drawback: the coordinator has to force a record before initiating phase one. This is especially bad because the extra forced diskwrite comes before the coordinator can respond to the client with the decision (at the end of phase one). In Presumed-Nothing or Presumed-Abort, there were only two forced writes before the coordinator could respond to the client. Now, there are three. Because of this, Presumed-Commit has never been used in practice.

Recently, however, Butler Lampson and David Lomet have come up with a scheme that optimizes the message passing for the commit case without requiring the extra forced diskwrite.

5.3 Read-only Transactions

Further optimizations are possible for a read-only transaction. If a transaction is locally read-only at a participant, the participant does not really care whether the transaction commits or aborts. Therefore, when it receives a prepare message, it need not force a prepare record, and may release the locks immediately. Further, the coordinator does not need to involve such a participant in phase two.

If the transaction is read-only at *all* participants, then:

1. phase two can be completely avoided.

2. no disk write is necessary.

6 Synchronization

If different sites in a distributed transaction system use different mechanisms for synchronization, then even if each site achieves local serialization, the system may still lack global serialization. For example, if one site uses 2-phase locking whereas another uses multi-version timestamps (MVTS), then the two sites may order transactions in mutually incompatible orders. Recall that strict 2-phase locking serializes in commit order, while timestamping serializes in the order the transactions began.

6.1 Homogeneous Systems

One solution is to assume that all the participating transaction managers use the same serialization scheme. This will, however, limit the distributed system's ability to use different schemes.

6.2 System Specification for Synchronization

Another solution is to allow heterogenous schemes provided they satisfy a certain specification for correct synchronization behavior. Such a specification is tricky to write, and has not been provided in the handout. A rough sketch is attempted below.

We can view the 'system' as the collection of all transaction managers. Since each transaction manager handles synchronization locally, we must address how the transaction managers could agree on a compatible synchronization order.

In 2-phase locking, the *locking point order* is the basis of serialization. Locking point order is defined to be any point between the last lock acquire action and the first lock release action. The problem is that each transaction manager has no global knowledge of the locks held by other transaction managers. Nonetheless, each transaction manager should be able to use its local knowledge to deduce something about the global lock point order. The operation *Precedes* is defined as follows:

- **Precedes(x) :** For any two transactions T_1, T_2, and transaction manager X, T_1 precedes T_2 at X if X sees the commit of T_1 before T_2 executes any operation at X.

The transaction manager must ensure that committed transactions are serializable in *all* orders consistent with Precedes(x). Each transaction manager may use different synchronization mechanisms as long as it satisfies the above.