6.826 Principles of Computer Systems	**Fall Semester, 1990**

Lecture 20: November 26, 1990

Lecturers: Butler Lampson and Bill Weihl *Scribes: David Chaiken and Doug DeAngelis*

Well, I guess this class won't be writing any operating systems... *Butler Lampson*

1 Administrative Information

Handout	Title
34	Problem Set #4 Solutions
—	Problem Set #4 Corrected

The failure on Problem Set #4 was generalized and wide-spread, so don't feel too bad if you did poorly.

2 Last Week's Leftovers

There are two issues regarding secure communication (discussed in Lecture 19) that need elaboration. The first involves the notation and semantics used to reason about secure channels, and the second is an observation about the context of secure channels.

2.1 Semantics of Secure Channels

The process of in-band authentication establishes a secure channel between two parties through the use of an authentication server. Section 6.1 of Lecture 19 examines a scenario in which principals A and B establish a secure channel. When reasoning about this scenario, the statement $K_{AS} \models K_{AB} \Rightarrow B$ has the meaning:

> "From the channel defined by the key K_{AS}, received a message stating that the channel defined by key K_{AB} speaks for principal B."

The syntax $K_{AS} \models K_{AB} \Rightarrow B$ may be replaced by another syntax, such as $\{K_{AB}, B\}_{K_{AS}}$. Although the latter syntax may more concretely represent the actual encoding of the message in a real system, both notations have the same meaning (semantics). Despite the jihads that are fought over syntax, formal reasoning is based on semantics, not on syntax.

2.2 The Context of Secure Channels

The communication network specifications discussed in class allow messages to be dropped and to be received multiple times. However in practice, it is a good assumption that such unfavorable events are rare. This assumption is *not* valid when reasoning about secure communication. In the context of secure channels, it is necessary to assume that an adversary (the "bad guy") will attempt to break the communication scheme.

In particular, an adversary can duplicate an arbitrary number of messages and repeat them at any time. Thus, certain arguments that are valid for a normal network are not valid for a secure network. For example, it is possible to argue that a message identifier may be reused after a given amount of time in a normal network. However, the activity of the adversary makes such an argument invalid for a secure network. While the adversary complicates the issues involving the generation and use of unique message identifiers, many of the concepts from the communication field carry over into the realm of security.

Communications	Security
Duplicate Receives	Replay Attack
Unique Identifier	Nonce

3 Distributed Transactions

The specification of a distributed transaction system is fundamentally the same as the specification of transactions in a centralized system. Both types of transaction system are used to guarantee the atomicity, permanence, and serializability of collections of operations [1]. However, a distributed implementation of a transaction system is spread over multiple sites, and can tolerate the failure of various combinations of the sites. The two fundamental issues that are inherent in a distributed system are

- coping with partial failures in the system, and
- coordinating sites for synchronization.

4 Recovery from Partial Failure

To identify the goals of a distributed system, examine a naive implementation of a transaction system. Appoint one site as a *lock manager*, which centralizes the access to all shared objects. When any site in the distributed system fails, reboot every site in the entire system. This first-cut implementation suffers from two fundamental problems. First, the lock manager is a performance and reliability bottleneck. Since all shared accesses must be coordinated

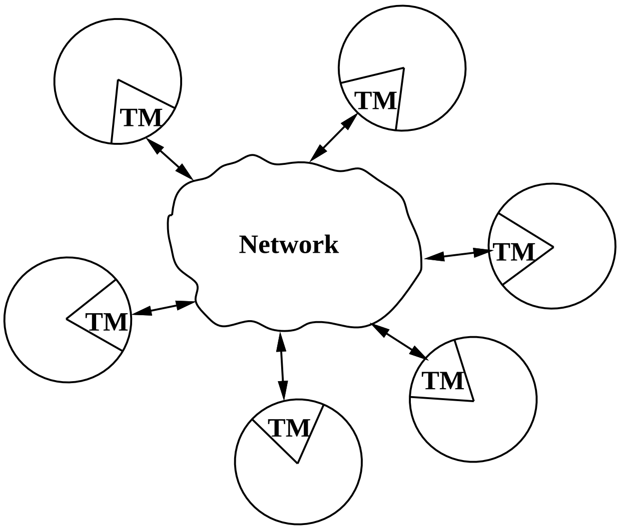

Figure 1: A distributed transaction system.

through a central site, the bandwidth of the system is limited by the bandwidth of the lock manager. When the lock manager fails, the entire system fails. Instead, a distributed transactions system should disperse the management of transactions. Second, the availability of the system as a whole is poor. Rather than propagating the effects of a site failure, a distributed system should attempt to isolate failure.

A model of a distributed transaction system is illustrated in Figure 1. A transaction manager (TM) resides at each site and maintains an independent log for the stable state at the site. Assume (for now) that when a site fails, it crashes completely (fail-stop). The network may lose or duplicate messages, but does not corrupt messages. A transaction can begin at any site, and can move to other sites via a parameter of a remote procedure call (RPC). A transaction starts at some site, and wanders around the distributed system through the network. To simplify the model, assume that a transaction commits at the site where it started. (This assumption about commits is not essential.)

There are two acceptable behaviors of a transaction.

1. A transaction can commit successfully at all sites and persist.

2. A transaction can abort, and none of its modifications can affect any transactions that commit.

The problem in a distributed system is determining when a transaction may commit. Since it is unacceptable for some sites to commit a transaction while others abort it, it is necessary to implement a communication protocol that enforces *atomic commitment*, even in the presence of site failures.

4.1 The 2-Phase Commit Protocol

The classic solution to the atomic commitment problem is the 2-phase commit protocol.[1] For each transaction, this protocol designates one site as a *coordinator* to control the commit procedure. Assume (for now) that a transaction's coordinator is the site where it starts and commits. The other sites that are involved in a transaction are called *participants*.

The 2-phase protocol is used when a transaction attempts to commit. In the first (*prepare*) phase of the protocol, the coordinator collects commitment votes. A participant votes "yes" if it thinks that it is okay to commit the transaction, otherwise it votes "no."[2] A participant sends a "yes" response only after ensuring that all modifications associated with the transaction are in stable storage and are recoverable. Thus, a "yes" vote requires a log force. After voting "yes," a participant may not change its mind and unilaterally abort the transaction.

Once the coordinator receives all of the votes, it decides whether or not to commit the transaction. If all of the participants vote "yes," then the coordinator decides to commit, otherwise it decides to abort the transaction. In the second phase of the protocol, the coordinator distributes the decision to the participants. When a participant receives the decision, it can release the locks and perform the updates associated with the transaction.

The above discussion describes how the 2-phase protocol achieves the goal of distributing transaction management. The protocol handles failures in the following ways:

1. **Network failures:** If the network loses a message, the coordinator retransmits the vote request until it receives a response. If a participant receives a duplicate vote request, it resends its vote. If a coordinator receives a duplicate response, it ignores the redundant vote.

2. **Participant fails after voting:** When a participant in a 2-phase protocol crashes, it can not simply abort all active transactions, as in a centralized system. If the participant votes to commit a transaction before failing, then after recovering, it must recover the transaction's state prior to the failure. To allow this type of recovery, a participant marks a transaction as *prepared* before sending a "yes" vote to the transaction's coordinator (hence the name of the *prepare* phase).

   The recovery is implemented by recording prepare records in the redo log. A participant must force a prepare record to stable storage before sending voting to commit a transaction. The redo procedure restores the locks associated with each of the prepared transactions, and leaves the transactions in an active state. The participant must be able to infer the locks associated with each transaction from the stable redo log.

[1] The 2-phase commit protocol should not be confused with the 2-phase locking scheme that is used to serialize committed transactions.

[2] Implementations of the 2-phase protocol can make provisions for transactions that are pending. For example, a participant may wait a certain amount of time before responding to a vote request. A participant might also reply with a "try again later" message.

3. **Participant fails before voting:** Since it is usually impossible for a coordinator to tell the difference between communication failure and participant failure, the coordinator must time-out after long voting delays. Upon a time-out, a coordinator aborts the delayed transaction. However, this policy is not enough to guarantee correct behavior after participant failure. Consider the following scenario:

    1. Transaction T visits participant X
    2. T leaves X and executes at a different site
    3. X crashes
    4. X recovers
    5. T attempts to commit

    In general, it is difficult to detect this scenario. To ensure correct behavior, X can abort the transaction when it recovers and then vote "no" when the coordinator attempts to commit T.

4. **Coordinator fails before decision:** The coordinator aborts pending transactions during recovery. To successfully abort a transaction after a crash, the transaction's participants must be recorded in stable storage.

5. **Coordinator fails after decision:** If the coordinator fails after making a decision, the decision must persist. This implies that the coordinator must write the decision to the redo log and force the log before transmitting the decision to any participants.

There are two types of log forces that are required during a 2-phase commit protocol. First, the participants must make sure that all of the transaction's updates are logged in stable storage. This force is analogous to the ForceAll() in the sequential implementation of transactions. Second, the coordinator must record the decision in stable storage. This action is analogous to writing the commit record to the stable log in the sequential implementation.

4.2 The 3-Phase Commit Protocol

The two-phase commit protocol is termed a *blocking commit protocol*, because it blocks a participant from taking action on a transaction while the coordinator of that transaction is down. The time span of concern starts at the time that the coordinator gets all the votes and ends when the coordinator finishes distributing the decision. If the coordinator fails during this window of vulnerability, the participant continues to hold the locks associated with that transaction. Participants have no autonomous method for deciding the fate of pending transactions.

It would be quite desirable to allow participants to release the locks associated with a transaction when the coordinator fails. By assuming that sites can detect the difference

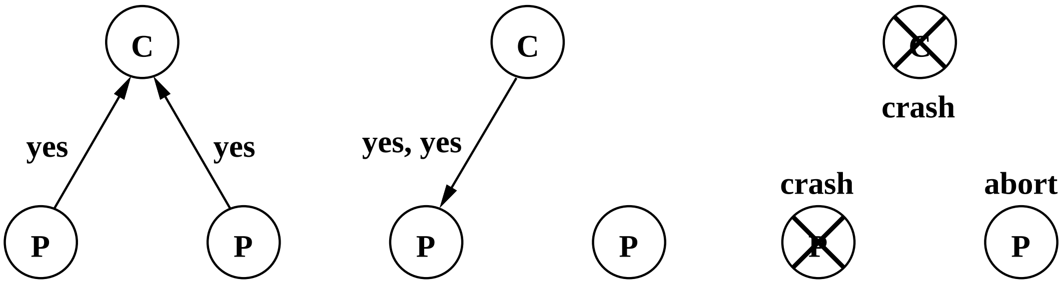

Figure 2: 3-phase protocol scenario. The transaction is aborted, even though both participants vote "yes."

between communication failures and failures of other sites, it is possible to implement a non-blocking scheme. Unfortunately, it is extremely difficult (if not impossible) to differentiate between network partitions and site failures. Thus, schemes that assume such a mechanism have more theoretical than practical interest.

Nevertheless, the 3-phase commit protocol has been proposed to implement a non-blocking scheme. Camelot [1], the CMU distributed transaction processing system, offers the 3-phase protocol as an optional feature. The protocol consists of the following three phases:

1. collect votes
2. distribute votes
3. distribute decision

This protocol returns a degree of autonomy to the participants. In the typical case, the coordinator makes the commit/abort decision, and eventually distributes the decision. However, the transaction can be aborted, even if all of the participants vote "yes." Figure 2 shows such a scenario. In the first phase, the two participants in a transaction vote to commit. During the second phase, the coordinator distributes the votes to one participant. Before the vote distribution phase is complete, both the coordinator and the participant that received the votes crash. The other participant detects that all of the sites that know the votes have crashed. (Recall that the 3-phase protocol requires a crash detection mechanism.) The remaining participant must abort the transaction in order to ensure correct behavior without blocking.

4.3 Crash Counts

If a transaction manager is asked for a vote, it must vote no if it has no record of the transaction for any reason. It can not vote yes based solely on the fact that it has a record, however. Consider the following scenario:

1. Transaction T visits participant X
2. T leaves X and executes at a different site
3. X crashes
4. X recovers
5. T visits X again

Here, the participant must have a record of the fact that it crashed at some point after the transaction first visited it. The following ways of accomplishing this were suggested.

- The participant logs a "T was here" record at each visitation (this implies a log force).
- Each node keeps a count of the number of times it has crashed. The transaction can then record the crash count of each node at the time it visits it and will know to abort if the crash count of any participant increases.
- The participant marks itself on the transaction if it believes that this is the first time it has been visited. If any site marks a transaction multiple times, then it must have crashed and the transaction should be aborted.

4.4 Comparing Distributed and Centralized Transactions

These are the fundamental differences between the DTP systems described and the sequential systems discussed earlier in the term:

1. Two log forces are involved.
2. A node can not just simply decide to abort after a crash.
3. The coordinator (or someone) needs to remember the decision on the fate of a transaction.

5 Managing Synchronization

Consider two sites in a distributed transaction processing system, one of which achieves serialization using two-phase locking (2PL), the other of which uses multi-version time stamps (MVTS). Now consider the scenario in Figure 3.

Locally to each site, things are fine. But from the point of view of the system, the site doing 2PL would have expected the value read in T1 on site Y to be that which was written in T2 on site Y because it considers the committed transaction to have occurred first. Site

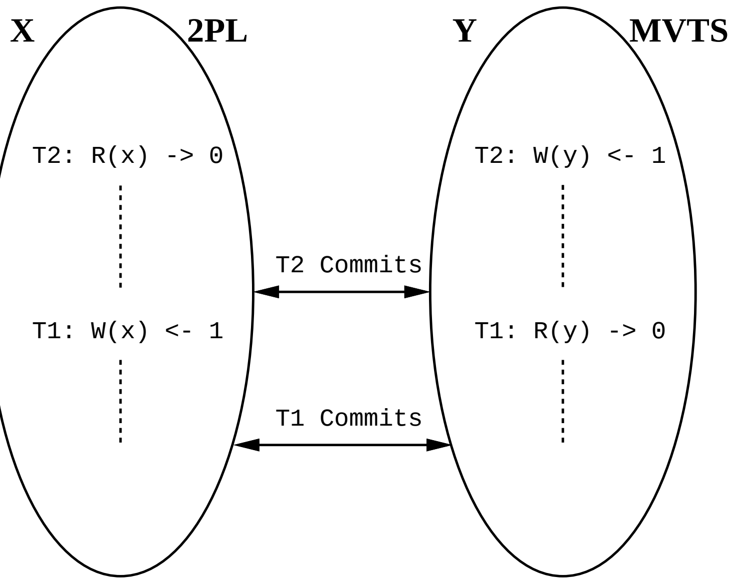

Figure 3: Effect of multiple serialization techniques.

Y, however, is serializing based on time stamps, so it is simply assuring that the value read while servicing T1 is *not* affected by T2.

One solution to this is simply to assure that all the participating transaction managers have the same impression of the serialization being done. This, of course, decreases the ability to use heterogeneous systems which may be made by different vendors.

References

[1] Alfred Z. Spector. *Distributed Transaction Processing and The Camelot System.* Technical Report CMU-CS-87-100, Carnegie-Mellon University, January 1987. Also appears in *Distributed Operating Systems*, Edited by Paker *et.al.*, 1987.