6.826 Principles of Computer Systems	**Fall Semester, 1993**

Lecture 19: November 17, 1993

Lecturer: Bill Weihl *Scribes: Jan-Willem Maessen and Ron Weiss*[1]

1 Overview

Today's lecture discusses two issues of distributed systems:

- Consensus: Paxos Algorithm (see Handout 42, "Consensus")
- Distributed Transactions (see Handout 44, "Distributed Transactions")

2 Consensus: Paxos Algorithm

In the last lecture, the specification for a general consensus algorithm was described. Also, the Paxos algorithm was introduced. Today, we continue discussing the details of the Paxos consensus algorithm. Generally speaking, Paxos is run by a set of leader processes that guide a set of voter processes to achieve consensus on some value. It involves three phases of communication between a leader and each voter:

1. **querying** (one round-trip)—a leader chooses a ballot numbered n and sends a query to all processes requesting their voting record. The query also includes the current ballot number n. Each process tells the leader how it voted in all ballots numbered less than n and the value of each ballot in which it voted "yes". When the leader has collected information from at least a majority of the voters, it looks for the most recent earlier ballot in which any process voted "yes", and chooses that ballot's value d.

2. **polling** (one round-trip)—the leader asks every process to vote for the chosen ballot's value d. If the leader collects a majority of "yes" votes, then it knows the ballot has succeeded and accepts d as the consensus value. (On the other hand, if it collects a majority of "no" votes, then the ballot is said to be *dead*). Note that the majority which voted "yes" may be a different set from that which responded to phase 1 messages.

3. **outcome distribution** (half a round trip)—the leader sends the consensus value d, if any is obtained, to all processes.

Figure 1 illustrates the actions and the communication between a leader and a voter. A voter maintains a history of its votes in stable storage. In the case where it is asked to

[1]These notes are based on notes prepared by David Maw in Fall '92

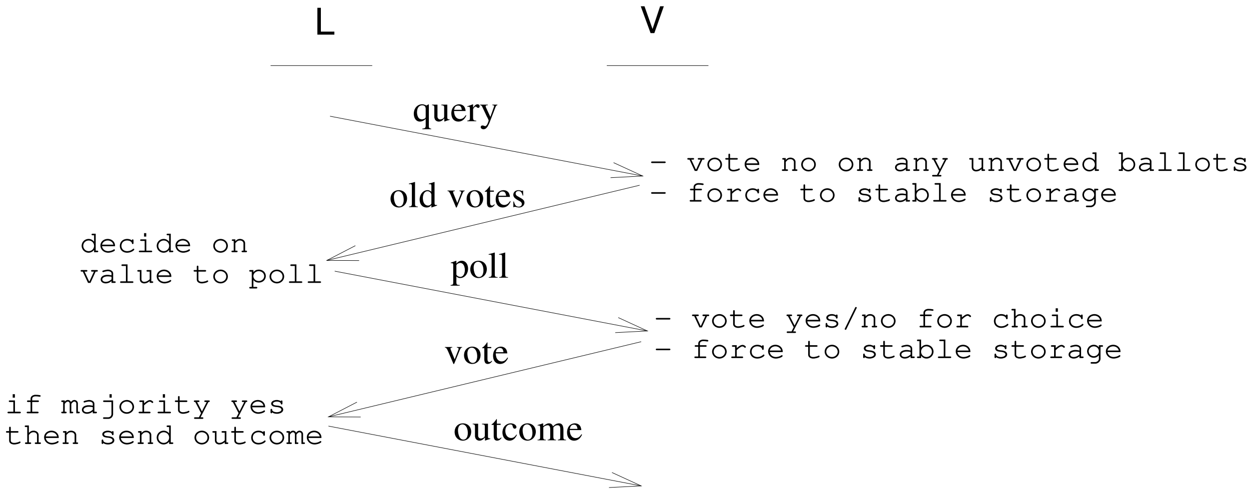

Figure 1: Leader and Voter in Paxos

reproduce its voting record during the query phase, and the current ballot number is n, it records a "no" vote for any ballot numbered less than n for which it has not voted yet. Before casting a vote in a ballot, the voter records its choice in stable storage. This guarantees consistency should the voter fail and later be asked to reproduce its voting record.

In deciding the value to use for polling, the leader examines all the voter records that it received. In order for the leader to proceed with a ballot, the leader blocks until it hears from a majority of the voters. A ballot is *dead* if a majority of all the voters has voted "no". If the voter records indicate that all the previous ballots are *dead*, than the leader can choose any value that is allowed. Otherwise, the leader has to choose the value that received the most recent "yes" vote that the leader knows about.

2.1 Some Examples

Example 1

For example, consider a scenario with two leaders L_1 and L_2, and three voter processes, V_1, V_2, and V_3. The following table shows the actions taken by the leader and the voters' responses. Blank entries represent idle, or no action noticeable by others.

L_1	V_1	V_2	V_3	L_2
query (ballot 1)	ret {}	ret {}	ret {}	
choose a				
poll (1, a)	yes(1, a)			
				query (ballot 2)

Notice that V_2 and V_3 haven't responded to the polling phase of ballot 1. Then, L_2 begins querying the voters about ballot 2.

L_1	V_1	V_2	V_3	L_2
query (ballot 1)	ret {}	ret {}	ret {}	
choose a				
poll (1, a)	yes(1, a)	no(1, a)	no(1, a)	
		ret record	ret record	query (ballot 2)
				choose b

Since V_2 and V_3 did not initially respond to ballot 1, when asked by L_2 about their vote history, they cast a "no" vote for ballot 1. L_2 hears from V_2 and V_3, and notices that ballot 1 is *dead*. It is therefore free to choose a new value for polling.

Example 2

Consider a scenario that begins the same as Example 1. However, this time V_3 does not respond to the query from L_2.

L_1	V_1	V_2	V_3	L_2
query (ballot 1)	ret {}	ret {}	ret {}	
choose a				
poll (1, a)	yes(1, a)	no(1, a)		
	ret record	ret record		query (ballot 2)
				choose a

Now, L_2 must choose a, because there is a ballot with a "yes" vote for a, that L_2 cannot prove is *dead*.

2.2 Picking ballot numbers

A leader must guarantee that the ballot number it picks is distinct from all other ballot numbers picked by other leaders. This problem can be solved by by having each ballot number consist of the leader identifier and a ballot number. For best behavior, the ballot number chosen should be higher than any previously active ballot. If the ballot number is not the highest so far, then the voters have already decided to vote "no" for this ballot. Therefore, the leader may need an extra round to pick a large enough ballot number by polling the voters' records. Picking a higher ballot number insures that progress will be made.

2.3 Picking a leader

In the Paxos algorithm, a timeout mechanism is used to detect that there is no active leader. Then, the nodes broadcast their identifiers. The node with the lowest identifier is chosen as the leader. Note that this process can yield more than one leader in the case of network partition or delayed response. The algorithm converges because it is likely that after some time, only one leader will be chosen for a sufficiently long duration.

The consensus problem is guaranteed to be solved by one leader that is able to communicate with a majority of the voters and does not crash throughout the entire protocol. It is not guaranteed to be solved if there is more than one leader during any phase of the protocol. A result by Fischer, Lynch and Patterson states that there is no consensus algorithm that is guaranteed to terminate in an asynchronous system with perfect links and even one faulty process.

2.4 Storage Optimization

The leaders are only interested in the last "yes" vote of a voter, and the subsequent "no" votes. "Yes" votes before the last "yes" vote will be ignored by the leader since the leader will choose a value that either depends on the last "yes" seen, or all the ballots are dead and therefore the leader can choose any allowed value. Thus, a voter needs to store only information about the latest "yes" vote and the number of subsequent "no" votes. As a farther communication optimization, the voter can only transmit the latest "yes" vote, and the leader will assume that any later vote is a "no". However, if the leader is trying to choose a ballot number, the voter must also transmit the number of subsequent "no" votes.

3 Distributed Transactions

The specification of the distributed transactions is the same as the spec for atomic transactions (Handout 44, pages 1–2). However, the implementation is different. The architecture (see Figure 2) consists of nodes (sites) connected to each other via a network. The network is unreliable and the sites can crash unpredictably. However, we do not assume byzantine failures.

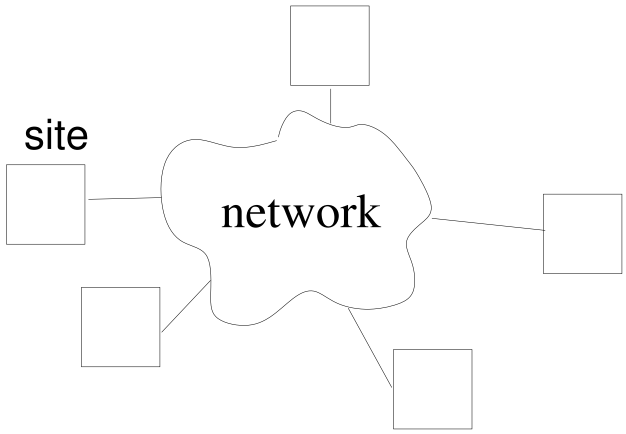

Figure 2: Distributed Transactions Architecture

There is a *Transaction Manager* module at each site. RPC is used for doing work at other sites. Normally, a site sends the transaction number associated with the action in the

RPC request. We need to guarantee the following:

- The transactions must be recoverable and atomic. They must either be committed at all the sites, or aborted at all of them.
- Sites must choose compatible serializations of the transactions.

3.1 Recoverability

The distributed transactions problems is essentially a consensus problem. The transaction should be committed if and only if **all** sites agree to commit, otherwise the transaction must be aborted. If any of the sites aborts, then all the other cooperating sites must also abort. Note that the names of the participants show up in the spec, since all participants must be accounted for when deciding whether to commit a transaction. We assume that each site implements concurrent transactions with logging and caching. To implement distributed transactions, we use the Paxos consensus module with the modification that agreement requires a unanimous vote.

The algorithm consists of the following phases:

- The coordinator (*i.e.*, leader) sends a *prepare* message to the participants.
- The participants respond with a commit vote (*i.e.*, "yes") if they were able to log the transaction to stable storage (either the log or the base). This is different from the normal atomic commit in the sense that the participant might be asked in the future to *undo* this transaction. At this point, the participant is ready to either commit or abort the transaction, even if it crashes at any point after preparing.
- The coordinator chooses either commit or abort as the ballot value for Paxos. The choice is commit only if all participants have responded with a commit/yes. Now, run the Paxos consensus algorithm.

If a participant crashes before it is prepared, then it will respond with an abort/no for all the active transactions. If the coordinator crashes during the protocol, the participants will timeout, decide to abort the transaction, choose another coordinator, and this coordinator will globally abort this transaction. If a participant crashes after it prepares, but before it has received the outcome, it should ask the coordinator for the outcome. It can still commit the transaction, because it was logged stably. During recovery, the participant should not abort the transaction if it has prepared, because the participant may later be required to commit it. Thus, an undo operation should not abort all active transactions, but rather only the ones which are neither prepared nor committed.

3.2 Two-Phase Commit

The usual implementation of the distributed transaction consensus involves the coordinator logging the outcome of the prepare phase stably. The algorithm involves:

- A similar prepare phase as before, with the modification that this phase is also used for determining the outcome of the consensus algorithm.
- The participants respond to the prepare message as before.
- The coordinator chooses the outcome based entirely on the responses from the participants. If all agreed on commit, then the coordinator stably logs commit as the outcome of the modified consensus algorithm. Otherwise, it stably logs abort. If the coordinator crashes after the force of the outcome to stable storage, once the coordinator recovers, it can continue with the operation.
- The coordinator broadcasts the outcome to all the participants.
- The participants acknowledge the outcome message.

The problem with the above *two-phase commit* algorithm is that it is not fault tolerant. The coordinator is the only source of the decision after the participants are prepared and before all have acknowledged the outcome. If the coordinator has crashed, no progress can be made to resolve the outcome of this transaction, until the coordinator recovers.

Some have argued that this is not an important deficiency, because the availability of the coordinator is a more serious problem than the availability of commit/abort decisions on a given set of active transactions should the coordinator crash. However, with current replication technology, availability is much improved, and the above reasoning no longer applies.

To resolve this situation, we use Paxos to improve the fault-tolerance of the system by allowing a new coordinator to make a decision. The naive adaptation of Paxos requires four phases. However, we can optimize by combining the prepare and query for old votes messages, and thus reducing the number of phases to three. This solution is more robust than the two-phase commit, and only incurs a cost of one extra round of communication. If a new leader is chosen, it will ask the participants to prepare a second time, but since they have already logged the fact that they are prepared, they will simply respond with the record of previous votes.