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

Lecture 19: November 23, 1992

Lecturer: Butler Lampson *Scribe: David Maw*

1 Overview

Today's lecture discuss two issues of distributed systems:

- Consensus: Paxos Algorithm
- Security

Note: Handout 43 "Consensus" and handout 44 "Authentication in Distributed Systems: Theory and Practice" contain details on Paxos consensus algorithm and security issues respectively.

2 Consensus: Paxos Algorithm

In last lecture, the specifications for general consensus algorithm was described. Today, a specific consensus algorithm, Paxos algorithm is discussed. Generally speaking, the algorithm is run by a set of leader processes that guide a set of voter processes to achieve consensus. It involves three phases of communication between a leader and each voter:

1. information collection (one round-trip) — a leader chooses a ballot numbered n and sends to all processes. 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". The leader looks for the most recent earlier ballot in which any process voted "yes", and chooses that ballot's value d.

2. vote collection (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. result distribution (half a round trip) — the leader sends the consensus value d, if any obtained, to all processes.

A voter votes "no" on a ballot n only if it has already seen another ballot with a number larger than n.

2.1 An example

For instance, consider 3 voter processes: `a`, `b`, and `c`, and history of votes as follows:

Ballots	Value	Yes	No
1	d_1	`a`	`bc`
2	d_2	`b`	`ac`
3	*nil*		`abc`
4	d_1	`ac`	`b`
...			

This history tells that process `a` voted "yes" on ballot 1 before it saw the information-collecting (*IC*) message for ballot 2. Also, process `a` voted "no" on ballots 2 and 3 because it saw the *IC* message for ballot 4 (IC_4) before the vote-collecting (*VC*) messages for ballots 2 and 3 (VC_2 and VC_3). A possible sequence of messages that process `a` saw was:

$$IC_1,\ VC_1,\ IC_2,\ IC_3,\ IC_4,\ VC_2,\ VC_3,\ VC_4,\ ...$$

In this case, as long as two processes voted "yes" on a ballot, a majority could be reached. The history shows that both processes `a` and `c` saw message VC_4 before any *IC* message with a number larger than 4; therefore, they both voted "yes" and a majority was determined with the value of ballot 4, d_1, being the consensus value of the algorithm.

As for ballot 3, the leader has not yet reached the stage of fixing a value for its ballot, but the three voters have already decided to vote no because of having seen IC_4.

Note that the leader of ballot 4 might have chosen either d_1 or d_2, depending upon the particular majority set it heard from in response to its IC_4 message. If it heard from only `a` and `c`, then it would choose d_1. If it heard from `b` among others, then it would choose d_2. Such non-determinism in the choice for d exists only if consensus has not already been reached.

2.2 Consensus is Stable

Consider the above example again, with the exception that process `c` voted "yes", stored its vote in its stable storage, sent out its vote to the leader and crashed, and the vote message got lost. The consensus was reached at this point with both processes `a` and `c` voting "yes" on ballot 4, and the value of ballot 4, d_1, was the consensus value.

However, the leader could not see a majority of votes, and thus could not identify d_1 as the consensus value. The abstraction function and the invariants of the Paxos algorithm do not guarantee that the leader knows of the consensus result right after it is determined. They only ensure that once the consensus is reached, it will not be changed.

In the case above, the leader will be unable to obtain consensus on ballot 4, so after some delay, the same or a different leader will decide to start ballot 5. In response to IC_5, process `a` would reply that ballot 4 was the most recent earlier ballot in which it voted "yes"; while process `b` would reply with ballot numbered 2, and process `c`, if recovered, would reply with ballot numbered 4. It can be seen that the only possible value for ballot 5 is that of ballot 4, because a majority will contain at least one of a or c. This is true in general of any consensus value, because the majority that voted yes for the consensus will intersect with the majority that responded to the new IC message.

One complication in this algorithm is that it may not terminate if there are always too many impatient leaders. If there are several leaders, the one running the biggest ballot will eventually succeed. However, if leaders keep starting bigger ballots in alternation, none may ever succeed. As an example, a voter may see the following sequence of messages:

$$IC_i,\ IC_{i+1},\ VC_i,\ IC_{i+2},\ VC_{i+1},\ IC_{i+3},\ VC_{i+2},\ ...$$

which will cause the voter to vote no on all ballots.

3 Security

The security issue in computer systems is only briefly discussed this year. Please refer to handout 44 and lecture notes of last year for more details[1].

3.1 Model: Access Control

There are two popular security models in computer systems: information flow and access control. Only the access control model is discussed in today's lecture. Figure 1 shows the components that make up an Access Control Model of computer system security. The model consists of an *agent* (subject) and a *resource* (object). The subject issues *requests* that access or change information at the object. In real life, some `[Subject, Object, Request]` examples are: (1) `[User, File, Read/Write requests]` in a file server setting, and (2) `[Customer, Teller Machine, Withdrawal/Deposit requests]` in a banking system.

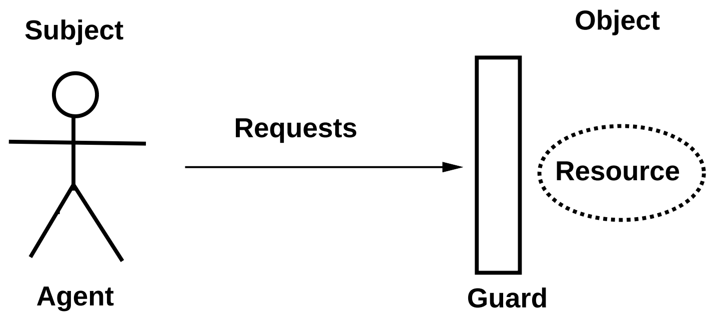

Figure 1: Access control model components.

The Access Control Model assumes that all objects are protected by *guards* that grant or deny access to the object. The guard associated with an object, called the access control list (ACL), maintains a sequence of principals and the operations each of them is allowed to perform.

[1]Notes on this section make use of two lecture notes from Fall '91: Lecture 22, scribed by Kah-Kay Sung, and Lecture 23, scribed by David Waitzman

The security issues are more complicated in a distributed system than in a single local machine. There are some problems specific to distributed systems, for example,

1. Requests between *subject* and *object* may pass through a long path with many intermediate machines.

2. Transport mechanism is unreliable.

A secure distributed system based on the Access Control Model must therefore have a secure *channel*. A channel is a physical path between a subject and an object, where data flows back and forth between the two ends. A channel is "secure" if we know who is on each side of the channel.

3.2 Channel

Channels provide a means for principals to make statements or for one principal to communicate with another. Secure channels provide:

1. **Multiplexing**: one channel to represent many channels

2. **Authentication**: determination of the true identity of whoever is making the request. For example,

   (a) A channel C may `speak for` a person (*e.g. bwl*). This is denoted as $C \Longrightarrow bwl$, and means that if C `says` something, then *bwl* `says` it too.

   (b) A member M of a group G may `speak for` the group. [$M \Longrightarrow G$]

3. **Secrecy**: known receiver.

4. **Integrity**: known sender.

3.3 Encryption

Data encryption is typically needed for channels to be secure. If you believe that P is the only sender that knows the encryption key, and will use it to send messages, then you can use encryption to ensure integrity.

All methods of encryption have some common properties. They require the sender to encrypt the message using some key $\mathtt{K}^{-1}$ and transmit the encrypted message, which the receiver is supposed to decrypt using a possibly different key $\mathtt{K}$. Encryption and decryption must satisfy the following relationship.

$$\begin{aligned} \mathtt{Enc}(\mathtt{K}^{-1}, \mathtt{m}) &= \mathtt{x} \\ \mathtt{Dec}(\mathtt{K}, \mathtt{x}) &= \mathtt{m} \\ \mathtt{Dec}(\mathtt{K}, \mathtt{Enc}(\mathtt{K}^{-1}, \mathtt{m})) &= \mathtt{m} \end{aligned}$$

Further, it is desirable that a message not encrypted using K^{-1} be caught upon decryption as fake. In general, it is possible for the decryption algorithm to output something random when given something random as input. To avoid this, the legitimate sender appends the checksum to the message *before* encrypting it. Upon decrypting, if the included checksum does not match the computed checksum, the message is discarded.

The keys K and K^{-1} may or may not be the same. There are two kinds of encryption:

- **shared** $(K = K^{-1})$, is a common choice for pairwise communication.
- **public** $(K \neq K^{-1})$, is used in two ways. If the encryption key K^{-1} is private but the decryption key K is public, only the authorized senders can send a message, but any one can read it. This provides integrity and can be used for broadcasts. The scheme is often referred as *digital signature*. On the other hand, a public encryption key and a private decryption key provides secrecy and can be used in collecting information (dubbed convergecast). We will concern ourselves with only the former usage, because that is the one useful in access control.

3.4 Certification Authority

A formula like 'K_a says "S"' means that there is an encrypted message which, when decrypted using K_a, conveys "S". Since the decryption succeeded, the reader can be sure that it was sent by someone that knew K_a^{-1}. A statement like above is said to be a *certificate* for "S" made by K_a.

Consider this: P says "read file foo". We could make P some key K_{bwl}, and have file foo's access control list (ACL) say "K_{bwl} can read foo". The problem is that this is not a humanly readable ACL, and it is hard to change K_{bwl} since we will need to update the ACL. Therefore, we want a key $K_{bwl} \Rightarrow bwl$, and an ACL that says "bwl can read foo" with P being bwl. So, we need a way for the file server to learn things like $K_{bwl} \Rightarrow bwl$.

We have the **handoff rule**: If $A' \Rightarrow A$ and A' says $B \Rightarrow A$, then $B \Rightarrow A$

Suppose $K_{ca} \Rightarrow anybody$; that is, there is a widely known public key K_{ca} of a certification authority, CA, that can be trusted to speak on behalf of all. Since CA knows $K_{bwl} \Rightarrow bwl$, it can generate the certificate K_{ca} says $K_{bwl} \Rightarrow bwl$. Because of the handoff rule, we will believe $K_{bwl} \Rightarrow bwl$.

To communicate with each other, A and B need to ask the CA for the certificates K_{ca} **says** $K_b \Rightarrow B$ and K_{ca} **says** $K_a \Rightarrow A$ respectively. For public-key systems, these certificates can be stored insecurely — accessible to everyone. The case with shared-key systems is different. We want $K_{a/ca}$ says $K_{a/b} \Rightarrow B$ and $K_{b/ca}$ says $K_{a/b} \Rightarrow A$, where $K_{x/y}$ denotes the shared key between Y and X. In this case, it is imperative to protect the contents of the message from others because the key $K_{a/b}$ is used for both encryption and decryption. The certificates are in fact protected because they have been encrypted using the shared-key $K_{a/ca}$ or $K_{a/ca}$.