Siyuan Liu

Distributed Systems

Quick Book

Contents

1 Distributed File Systems 1
- 1.1 Problems and Goals 1
- 1.2 Operations 2
  - 1.2.1 Unit of Transmission 2
  - 1.2.2 Implementation Idea 2
- 1.3 Coda 2
  - 1.3.1 Disconnected Operations: cache and write conflict 2
  - 1.3.2 Replication: Volume Storage Group 3
  - 1.3.3 Weakly Connected Mode 3
- 1.4 File System Interface 4
  - 1.4.1 MogileFS 4
  - 1.4.2 HDFS 4

2 Processor Allocation and Process Migration 5
- 2.1 Problems and Goals 5
- 2.2 Processor Allocation 5
  - 2.2.1 Approaches 5
- 2.3 Process Behavior and Scheduling 7
- 2.4 Process Migration 7

3 Log and Checkpoint 9
- 3.1 Problems and Goals 9
- 3.2 Sender-based Logging 9
- 3.3 Recovery from Failure 10
  - 3.3.1 Incarnation Numbers 11
  - 3.3.2 Checkpoint: consistency 11
- 3.4 Logging 12
  - 3.4.1 Kinds of Logging 12
  - 3.4.2 One Approach For Asychronous Logging: GDM 12
  - 3.4.3 Adaptive Logging 13

4 MapReduce and Hadoop 15
4.1 Problems and Goals 15

5 Anonymous Communication and Onion Routing 17
5.1 Problems and Goals 17
5.2 Peer-to-Peer 17
5.2.1 Distributed Hashing 18

6 Security 21
6.1 Cryptographic Techniques 21
6.1.1 Problems and Goals 21
6.1.2 Kinds of Cryptography 21
6.2 Key Distribution and Management 22
6.2.1 Key Distribution Center (KDC) - Kerberos 22

7 Limits and Troubles 25
7.1 Communication Failure: Two Armies Problem 25
7.2 Processor Failure: Byzantine Problem 25
7.3 CAP Conjecture 26

References 29

1

Distributed File Systems

1.1 Problems and Goals

When you want to have a file system and manage files, you need to consider following problems:

- Naming: allow users to find files with a human-friendly name.
- Accessing: create, delete, read, write, append
- Physical Allocation
- Security and Protection: ensure privacy
- Resource Administration: enforce quotas and implement priorities.

Besides the problems in mind when design a DFS (actually general file system), we have some goals for DFS. **The big difference isn't what it does but the environment in which it lives.** A distributed file system typically operates in an environment where the data may be spread out across many, many hosts on a network, and the users of the system may be equally distributed.

- Coordinate file systems on machines.
- Hide the existence of distributed file system from the user.

Why we need a DFS. Actually this problem can be generalized into a problem of distributed system.

- More storage
- More fault tolerance
- Users are distributed who need to access file system from many places.

1.2 Operations

1.2.1 Unit of Transmission

There is a lot of data movement across the network, **how much do we move one time?** There are two intuitive answers to this question: whole files and blocks.

- File: Only the users know how to use the data in one file. File systems do not need to get known how a file is organized.
- Block: Reduce the payload for one operation. When the first block of the file arrives, the user can start to operate it.

1.2.2 Implementation Idea

Caching

Cache is the same usage. Cache data and reduce access to the servers. There are two kinds of caching in distributed systems:

- Cache and Validate Approach: ask servers that if the data is newest. This approach is used in *NFS*.
- Callback: if servers have some modification for files, they inform the users. This approach is used in *AFS* and *Coda*.

1.3 Coda

1.3.1 Disconnected Operations: cache and write conflict

hoard daemon.

keep certain files in the client's cache, requesting them as necessary, just in case the client should later find itself unable to communicate with the server. Cache and validate. This ensure the users get access the data efficiently.

write conflict.

keep a version number. Before a client writes a file to the server, it checks the version of the file on the server.

- If that version number matches the version number of the file that the client read before the write, the client is safe and can send the new version of the file. The server can then increment the version number.
- If the version number has increased, the client missed a callback promise. Then users must take care of the conflicts.

1.3.2 Replication: Volume Storage Group

This part is discussed in the Replication chapter. There are some topics for this.

First, **how it requests a file?**

1. It asks all replicas for their version number.
2. It asks the replica with the greatest version number for the file.
3. If there is a conflict, the client can direct the servers to update or inform them of the conflict.

Second, **how it writes a file?**

1. The clients sends the file to all servers, along with the original CVV.
2. Each server increments its entry in the file's CVV and ACKS the client.
3. The client merges the entries from all the servers and sends the new CVV back to each server.

Third, **what if partition happens?** If one or more servers fail, the client cannot contact the servers. The collections of volume servers that the client can communicate with is known as *Available Volume Storage Group*.

If the network is partitioned, Coda will still work, but generate some conflicts and inconsistency. (As is said in Chapter Replication.)

If the partitioned or failed servers become accessible, the files need to be updated. The client needs to check VVV or CVV for conflicts. If there is a conflict, the client drops all callbacks in the volume, because the servers should have all the callbacks.

Conflicts

There are several kinds of conflicts appearing in this problem.

- read before write. When the client decide to write the updates to servers, it needs to check if the update reads before write. Otherwise, it may miss some callbacks. Situation could be that the client is broken down for a while and misses some callbacks.
- server consistency. After the updates written back to servers and the client getting the CVVs from servers, the client may find conflict in the CVVs. Situation could be that there is concurrent operation on the files.
- partitioned network recovered. Replication inconsistency conflict, users need to take actions.

1.3.3 Weakly Connected Mode

If a client finds itself with a limited connection to the servers, it will pick a server and send the update. This server will propagate the update to other servers.

1.4 File System Interface

There are some file systems that are designed at user level, rather than in the kernel.

1.4.1 MogileFS

Difference

Consider the file systems where files are not changed by users. Users are only allowed to upload and download the files.

Method

There are some key idea behind the implementation:

- RAID. Replication.
- Namespaces, rather than directory tree. In different applications. they can use a same MogileFS, but they can use different name.

Because there is little change happened, locks will be rarely used. It makes a lot of work efficient.

1.4.2 HDFS

Difference

HDFS supports the computation of Hadoop.

Idea

There are some key idea behind the implementation:

- The data needs to be very heavily distributed.
- The data needs to be local to each other, so the system requires location-awareness.
- The system needs to be implemented in a portable way at the user-level, to be operated at scale.

2

Processor Allocation and Process Migration

2.1 Problems and Goals

Scheduling a System

The discussion involves

- pick a processor to run a process
- move a process from one processor to another processor

Process Migration

The discussion involves

- decide a process that should be migrated.
- select a new host for the process
- migrate the resource of the process

The migration here is talking about process. The threads of one process should be migrated together. We do not discuss that the threads are dispatched on different processors.

2.2 Processor Allocation

You need to be careful when you want to migrate a process. There is a difference between *remote execution* and *processor allocation*. People may not know that their local processes have been migrated to a remote processor and executed remotely.

2.2.1 Approaches

A Centralized Approach: Up-Down

There are CPU consumers and suppliers existing in the network. We give credits to the suppliers and take credits from the consumers. The hosts with more credits can have high probability to acquire a CPU.

Hierarchical Approach

Goal

Hierarchical Approach can reduce the communication and information across the network in order to balance the load.

Method

There are workers on the leaves, and managers on the inner nodes. If a worker gets too much work or too little work, it will inform the manager above.

- The manager will use the information to try a shift between workers of it.
- The manager meets the limit of its quotas, then tell the directors who try to balance the load among the managers.

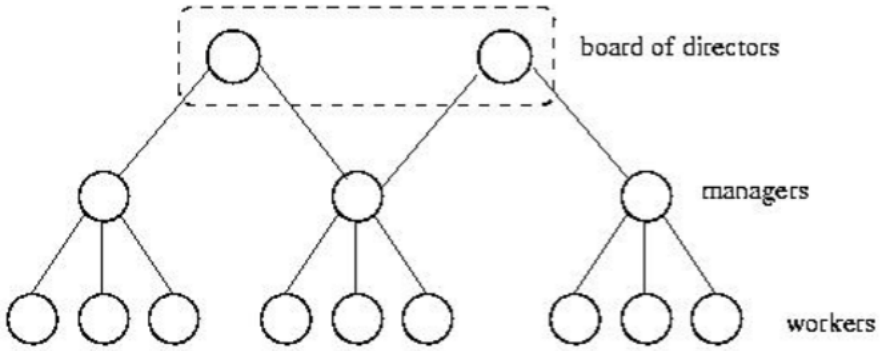

Fig. 2.1. Hierarchical Processor Management

Loads

We discussed the method for balance loads above that the workers gets too much work or too little work, it will inform managers. But how it communicate the information with managers and workers.

1. Yell out. When a worker meets the situation, it yells out to others. If it is idle, it may cause *thundering herds* that everyone wants to use it which makes it busy immediately.
2. Receiver Initiated. The idle processor asks around. But this approach leads to heavy communications.
3. Sender Initiated. The busy processor asks around to reduce load.
4. Hybrid Approach. These try to balance the costs of the above two approaches. They only "yell out" if they are substantially overworked or under worked.

2.3 Process Behavior and Scheduling

In general-purpose systems, recently started jobs are likely to be short lived, whereas long-running jobs are likely to keep running for a very long time.

2.4 Process Migration

Some ideas are here:

- Virtual Machine can make the migration easier, but it arises some network problems, such as IP address.
- If we want to migrate processes, we need to build a recoverable, portable communication layer (do not hack with TCP). This layer allows suspend, update and resume programs.
- There are some systems working for this scheduling: HTCondor, TORQUE.

3

Log and Checkpoint

3.1 Problems and Goals

As for **recovery**, there are three ways

- Replication. It is true that replication can make the system more fault tolerant, but it is expensive and also require a lot of synchronization work.
- Checkpoint. If the system make changes (progress), we can make checkpoints to store the system states. But the problem for checkpoint is that the systems need a **freeze** when backing up. However, we can try to find a consistent recovery line, which will be discussed later.
- Log. The classic technique is **Write Ahead Logging**. Events need to be written in the log first before executed.

For cooperation between checkpoint and log, we use logs between checkpoints. Recovery involves checkpointing and logging. Checkpoints store the state of process, logging involves recording the operations that produced the current state.

3.2 Sender-based Logging

Sender based logging is very important in those cases where receivers are thin or unreliable. In other words, we would want to log messages on the sender if the receiver does not have the resources to maintain the logs, or if the receiver is likely to fail.

Ensure that the order in both senders and receivers are correct.

Senders can play back the logging in the same order in which they were dispatched.It is difficult to order the messages from senders in the receiver side.

To ensure this, we follow the protocol:

- The sender logs a message and send it.
- The receiver gets the message and ACKs it with the time local to the receiver.
- The sender adds this timestamp in the ACK into that log entry.

Under this protocol, we can resend the message with the order recorded by the senders when dispatching. At the same time, when messages arrive the receivers, they can also order the messages with the receivers' time stamp.

Ensure the sender get the timestamped ACKs from receivers (make sure logs are complete)

Require the sender to send a ACK-ACK to the receiver before send a following message.

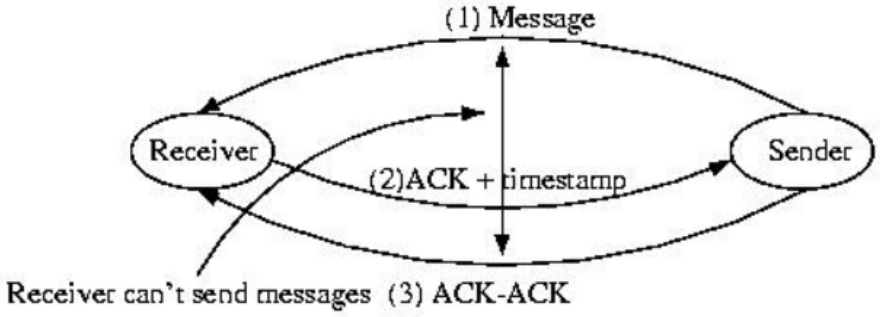

Fig. 3.1. ACK-ACK log.

Before the receiver gets the ACK-ACK for the first message, it cannot send messages to the sender. It make sure that we get the correct order in the receiver side.

3.3 Recovery from Failure

Recovery involves sending lots of history messages. **Duplicate messages** are the messages sent to other normal systems. **Orphan messages** are that after a rollback some systems may receive the messages that the recovering system does not remember sending. Rollback to a fail system may causes another system to rollback, which is known as **cascading rollbacks**. Eventually the systems will reach a state where they can move forward together, which is known as **recovery line**. After a rollback, a system may duplicate output, or request the input again, which is called **studdering**.

3.3.1 Incarnation Numbers

Incarnation is the period between checkpoints. Rebooting a system or restarting a cooperating process results in a new incarnation. We can number these incarnations. This number can be used to eliminate duplicate messages.

When a system is reincarnated, it sends a message to the cooperating systems informing them of the new incarnation number. The incarnation number is also send out with all messages. Therefore, the receivers can determine whether or not it is a duplicate message.

How the receiver handles the incarnation numbers?

- If the incarnation number of the message is less than the expected number, the message is a duplicate, so it should be discarded.
- If the incarnation number is in the message is greater than the expected number, the sender is recovering, so block accepting messages, until it informs us about its new incarnation number. (Maybe because the sender gets the recovery incarnation number first from the sender.)
- If they are the same, accept the message.

3.3.2 Checkpoint: consistency

The checkpoints of all the servers are unnecessary consistent. Therefore, we need to find a maximum recovery line.

Interval Dependency Graph (IDG)

The graph is constructed by creating a node for each interval, and then connecting subsequent intervals on the same processor by constructing an edge from a predecessor to its successor. Then an edge is draw from each interval during which one or more messages were received to the interval or intervals during which the message(s) was or were sent.

Where to store the graph? Each processor keeps the nodes and edges that are associated with it.

How to find the recovery line? If there are some lost intervals, the other servers rollback to a interval that is independent of the lost intervals. And this rollback continues until there is a recovery line established.

Coordniated Checkpoints

We can decrease the rollbacks happened in the whole system, by coordinating checkpoints. Actually we only discussed how to use IDG, but not how to implicitly build a IDG. Here is the discussion.

There are two methods:

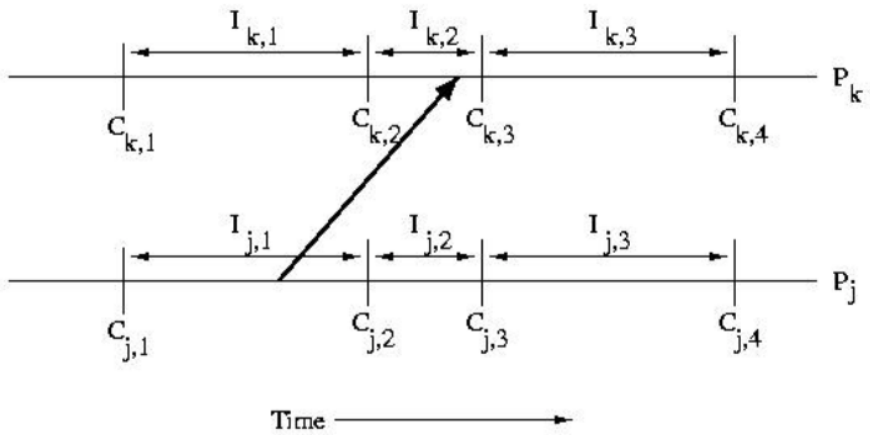

Fig. 3.2. IDG

- Record message sequence number (the sender information). If the receiver gets a message, it send back a message to tell the senders to check if they checkpointed since the last time they sent the message. If not, the senders need to checkpoint in order to satisfy the dependency.
- Synchronized clock. Each processor creates a checkpoint every T units of time.

3.4 Logging

3.4.1 Kinds of Logging

- Synchronous Logging. Logging before execution. It is expensive and slow.
- Asychronous Logging. Occasionally write the logs. Some messages may be logged, while others may not be logged.

3.4.2 One Approach For Asychronous Logging: GDM

First, the system maintains a **Global Dependency Matrix (GDM)**, each processor has a vector recording the interval numbers of processors it knows.

We can check if the matrix is consistent. The method is similar as before. You can check if the number on the processor's self is the greatest.

The method to find a new recovery line:

1. Get the previous recovery line and new updates after the recovery line.
2. Check for each update, if it can hold a consistent state for the system. If so, update the recovery line and go on checking.

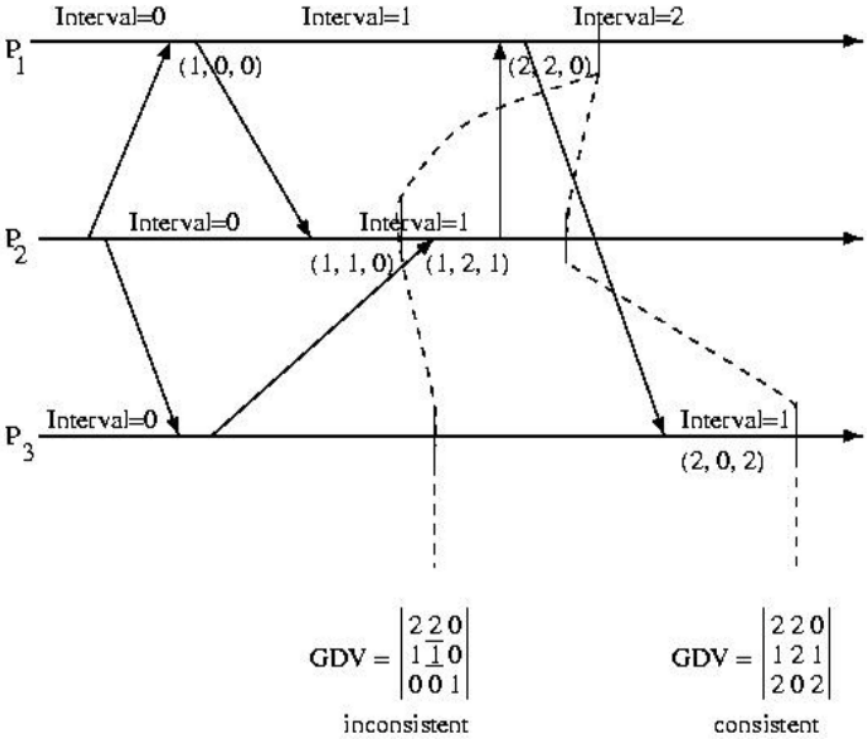

Fig. 3.3. GDM

3.4.3 Adaptive Logging

We do not need to log every message. It only needs to log those messages that have originated from processors that have taken checkpoints more recently than it has. Before there are new updates existing on the senders. If the receiver is ahead, it won't worry about it and do not need to checkpoint.

4

MapReduce and Hadoop

4.1 Problems and Goals

- How many maps
- How many reduces
- Locality, replication
- Combining Maps and Reduces: combiner only reduce the size of results from Maps, and the results are written in an intermediate file.
- Worker failure. When a Map worker dies, it needs to be re-executed from scratch. The reason for this is the results are stored on the Worker's local disk and are now inaccessible to Reduces. But, should a Reduce Worker fail, its results remain available in the global file system.
- Master failure. The master isn't scaled up. It is just one central Master. Like your desktop. Failures are years apart. And, checkpointing things will waste tons of time. Instead, if a computation times out, the program can just restart the computation a new, perhaps after checking the status of and with the Master.
- How many Map-Reduce phases is optimal? You can say it is one. But in reality, it is not possible and also not necessary.

5

Anonymous Communication and Onion Routing

5.1 Problems and Goals

For safety, we need a technique for routing that protect the identity of the sender and the receiver.

Ideas

1. Nodes that do the type of hop-to-hop routing.
2. Keys that are needed to decrypt the message (payload, not the header)
3. New routes. Although we want to utilize the route with keys for a while, we need to switch to a new route.

Weakness

Attackers can find out the hot spots in the network at certain times. **Why?** Because if the network is on client/server mode, the traffic will concentrated around the server, which makes it a hot spot. Distributing the servers may help, but as long as clients out numbers the servers, there are always hot spots. The solution to this imbalance is a peer-to-peer architecture.

5.2 Peer-to-Peer

Why we want to transfer to peer-to-peer mode?

Because client-server mode make the system more vulnerable to attackers.

Challenge from Peer-to-Peer

Naming the hosts, Finding the hosts, Trusting the hosts.

Solution to the Challenges

For naming and finding the hosts:

1. distributed directory service.
2. directory service distributed on selected peers.
3. distributed hashing.

For trust,we need to find some stable hosts for hashing or something else.

5.2.1 Distributed Hashing

Objective

A consistent hashing scheme is one that makes the hash value independent of the table size.

Chord Protocol

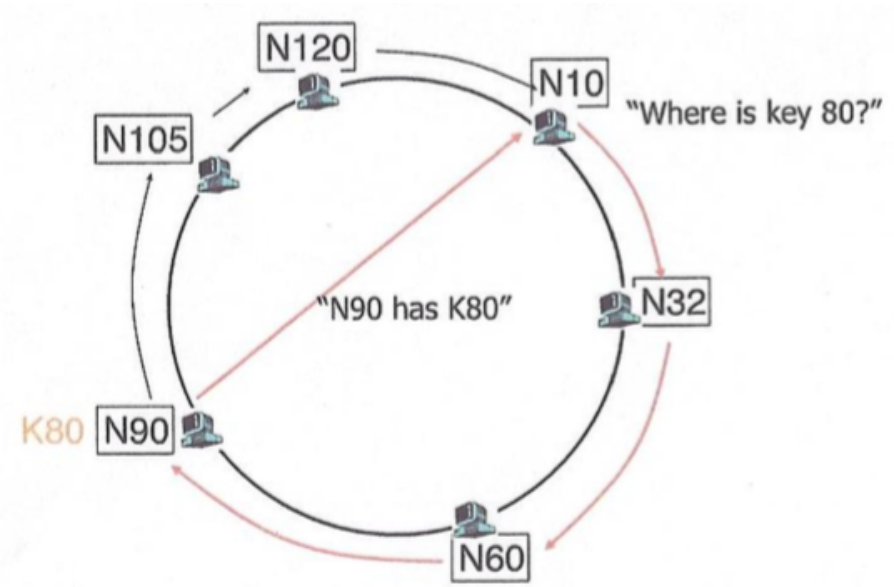

Fig. 5.1. Chord Protocol

Structure

Given m bit keys, there are 2^m logical positions on the ring. Each host can occupy one position.

Find Hosts

Each host maintains a finger table for successors. They are pointers to the nodes at 2^i away from it. The table makes it possible for a binary search on the network.

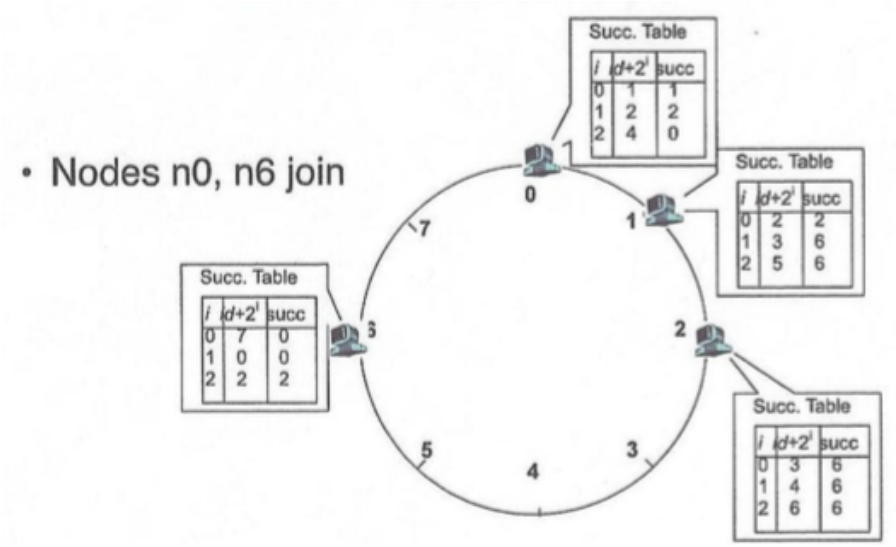

Fig. 5.2. Chord Protocol with Suc Table

A Node to Join

In order for a node to join,

1. It is added to an unrepresented position.
2. It gets its portion of the keys from its successor.

6

Security

6.1 Cryptographic Techniques

6.1.1 Problems and Goals

- Authentication: who am I talking to?
- Confidentiality: is my data hidden?
- Integrity: is my data changed?

6.1.2 Kinds of Cryptography

Table 6.1. Compare between two Cryptography

	Symmetric	Asymmetric
Shared Secret	Yes	No
Speed	Fast	Slow

Symmetric Cryptography

Confidentiality: One-time Pad (OTP)

- Key is as long as the message.
- One-time key.
- Two kinds: stream cipher and block cipher.

Integrity: Hash Message Authentication Code (HMAC)

Hashing for Cryptography

Properties:

- Consistent: same input, same output.
- One-way: have Y, cannot get X.
- Collision resistant: different input, different output.

1. Sender creates a hashed message (MAC) and attach it with the content message;
2. Receiver computes the MAC with message and verify the integrity of the message.

Authentication: HMAC and Nonce
Much similar with the verification in integrity.

Asymmetric Cryptography

Instead of shared keys, each person has a key pair: public key and private key.

The public key of a holder will be used by others, and private key will be used by itself.

The authentication will be against the holder of a key, so that the message should be first signed by the private key of the holder. Then the receiver can verify it with public key.

6.2 Key Distribution and Management

Why use a central key server?

Each pair of hosts have a pair of keys $-> n^2$ keys

6.2.1 Key Distribution Center (KDC) - Kerberos

What is the setting?

- Alice and Bob know own symmetric keys for KDC (KDC holds different symmetric keys with each registered user).
- Alice and Bob ask the KDC to get a symmetric key for communication.

How two hosts communicate with KDC?

1. By using K_{A-KDC}, Alice asks KDC to generate a symmetric key for communication with Bob.
2. KDC generates R_1 for the communication, and send $(R1, K_{B-KDC})$ to Alice
3. Alice use $(R1, K_{B-KDC})$ to communicate with Bob.
4. Bob knows using R_1 as the session key to communicate with Alice.

Why use KDC?

- Centralized trust and failure.
- KDC can expose session keys to others

Kerberos: authentication before using service

The client authenticates itself to the Authentication Server (AS) which forwards the username to a key distribution center (KDC). The KDC issues a ticket-granting ticket (TGT), which is time stamped, encrypts it using the user's password and returns the encrypted result to the user's workstation. This is done infrequently, typically at user logon; the TGT expires at some point, though may be transparently renewed by the user's session manager while they are logged in.

When the client needs to communicate with another node ("principal" in Kerberos parlance) the client sends the TGT to the ticket-granting service (TGS), which usually shares the same host as the KDC. After verifying the TGT is valid and the user is permitted to access the requested service, the TGS issues a ticket and session keys, which are returned to the client. The client then sends the ticket to the service server (SS) along with its service request.

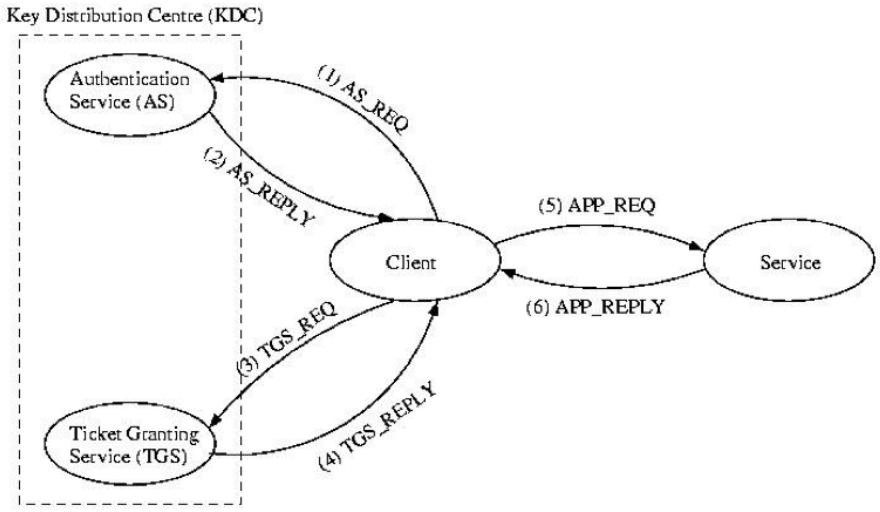

Fig. 6.1. Kerberos

How the procedure works:

1. Client asks AS for authentication: AS issues a TGT and session key to the client. TGT cannot be read by client, it is for TGS. Session key is used for later encryption.

2. Client uses the TGT to ask TGS for a service ticket. At the same time, use session key to communicate with TGS. The service ticket also cannot be read by client.
3. Client uses the service ticket to request service servers. At the same time, use session key to communicate with service server.

7

Limits and Troubles

7.1 Communication Failure: Two Armies Problem

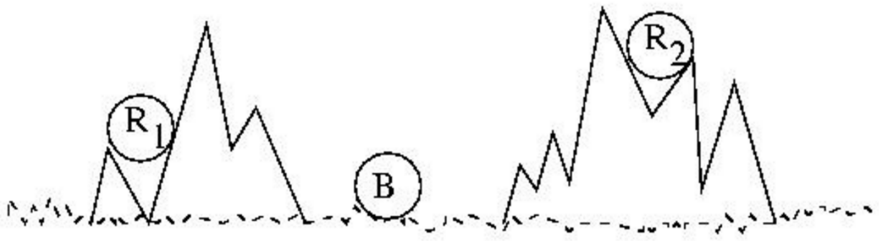

Fig. 7.1. Two Armies

1. Attack at the same time.
2. Fake messages.

The moral of this story is that there is no solution to this probem if the communications medium is unreliable. Please note that I said medium, not protocol.

7.2 Processor Failure: Byzantine Problem

1. Traitor

If the disloyal general tells the same lie to all of the generals, they will each agree to the wrong value. For this reason, based on this pedagogical story, undetectible errors (faulty hardware, not faulty communications) are known as Byzantine Errors by computer scientists.

The moral of this story is that failures can be expensive to detect – sometimes impossible. This means that distributed agreement may not be expensive – or impossible. Sometimes we have to pay the price, sometimes we don't. The successful design and implementation of distributed systems depending on knowing what we can do.

7.3 CAP Conjecture

- Consistency
- Availability
- Partition Tolerance

The CAP Conjecture is that we can build systems that guarantee up to two of these properties – but not necessarily all three.

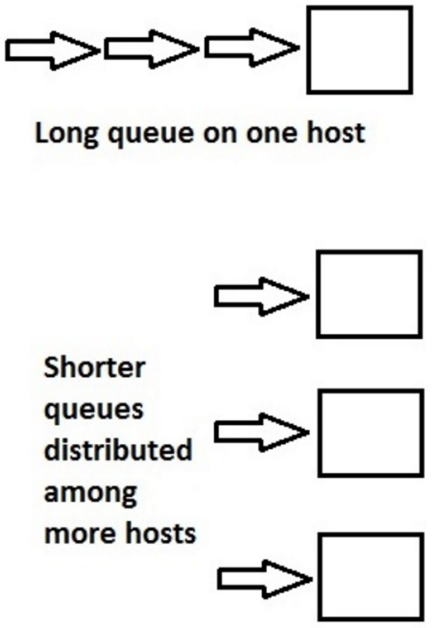

Fig. 7.2. AP Case

The picture above is nice, because we have availability. And, in the event of a partitioning, the reachable nodes can still respond, so we have partition

tolerance. The problem, though, is that we've lost consistency. Each host is operating independently, so the values can diverge if updates differ. This is the "AP"/"PA" case.

To add back consistency, we'll need to have communications among the hosts such that they can sync values: But, notice what has happened. We

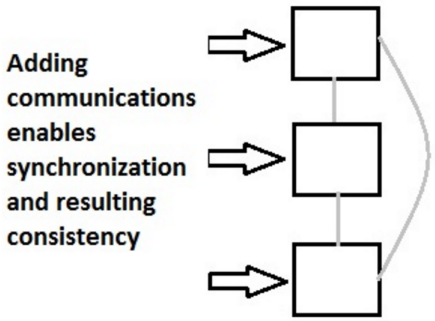

Fig. 7.3. Take consistency back.

gained consistency through communication. If we break that communication, we're back where we started. So, we now have consistency and availability, but not also partition tolerance. This is the "CA"/"AC" case.

What about the "PC"/"CP" case? How can we have consistency and partition tolerance without availability, at least in any meaningful way? One answer, which is I think a good example, is that we enable reads, but not updates. Now we have sacraficed the avaialability of writes in order to ensure maintaining consistency in light of a partitioning.

References

1. Gregory Kesden's lectures at UCSD