Distributed System Summary

Don't expect everything here

Andy Chong Chin Shin

Contents

Chapter 1

Characterisation of Distributed Systems and System Models

Distributed System (DS) is a set of networked computers that communicate and coordinate their actions only by passing messages.
Internet is a very large distributed system that enables users to make use of services from everywhere

1.1 Fundamental Characteristics of DS

Concurrency: add more computers to increase capacity, and hence higher performance

Loosely coupled: no global clock and no global shared memory

Independent failures: any computer and subnetwork can fail at anytime, can be more fault-tolerant than stand-alone systems

1.2 Main Motivation of DS: Resource Sharing

Resources are shared by using services. Service is a distinct part of a computer system providing accesses to the managed resources

For resource sharing works, we must define:

- Content and format of resources
- Naming and address
- Communication infrastructures

1.3 Issues and Problems in DS

Heterogeneity: hardware and software components are not identical, such as networks, endian formats, operating systems and programming languages

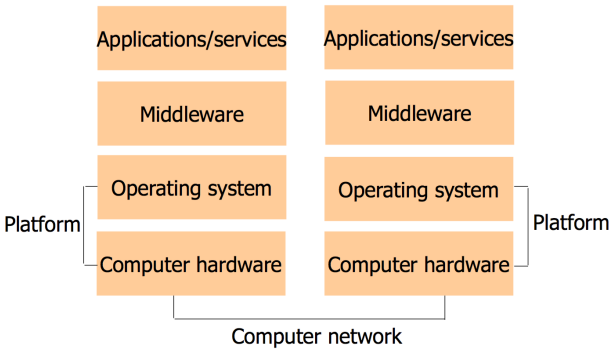
Scalability: remain effective when resources and users increase. This can be achieve by caching and replication of data and deploying multiple servers. We prefer decentralised design to avoid performance bottlenecks.

Failure handling: can be achieved by detecting, masking, tolerating, and recovery of failure

1.4 Architecture Models

Architectural model defines components, functions, placement of components, and relationship between components.

1.4.1 Software and Hardware Layers



Platform provides system's programming interface that facilitates communication and coordination between processes

Middleware masks heterogeneity and provide a convenient programming model. It provides generic services to application.

1.4.2 Client-server Model

M servers work together for N clients, so there is a division of work among M servers. These servers need partitioning of data or function and replication of data or function.

1.4.3 Peer-to-peer Model

All processes play similar roles as clients and servers. The interact cooperatively as peers to perform distributed computation

1.5 Fundamental Models

Fundamental model is formal description of common and intrinsic properties in architectural model. It makes exact claims by logical analysis and mathematical proof.

1.5.1 Interaction Model

Synchronous distributed system assumes upper and lower bounds on processing time, transmission time, and clock drift rate. We can infer properties, such as using timeouts to detect failures.

Asynchronous distributed system assumes no bounds on processing time, transmission time and clock drift rate.

1.5.2 Failure Model

Failure model defines the ways in which failures may occur

Omission failure: a process or communication channel fails to perform actions it is supposed to do.

Arbitrary / Byzantine Failure: arbitrarily omit intended processing steps and take unintended processing steps.

Chapter 2

Interprocess Communication

2.1 External Data Representation

External Data representation is an agreed standard for representation of data structures and primitive values. Data structure must be flattened before transmission and rebuilt on arrival.

Marshalling converts data items into the form suitable for transmission. Unmarshalling disassembles a message on arrival and restore data items

2.1.1 CORBA's Common Data Representation (CDR)

Big / little-endian handling: transmit in sender's ordering and the recipient translates

Example, flatten a `Person` struct, with `name` (string), `place` (string) and `year` (long) attributes, with value `{"Smith", "London", 1934}`:

Index in sequence of bytes	4 bytes	Description
0-3	5	length of string
4-7	`Smit`	
8-11	`h___`	padded to flush on word boundary
12-15	6	length of string
16-19	`Lond`	
20-23	`on__`	padded to flush on word boundary
24-27	1934	

Assumption: sender and recipient have common knowledge of the order and types of the data items

2.1.2 Java Object Serialisation

Assumption: the process doing deserialisation has no prior knowledge of the object types in the serialised form

Format: class information, types and names of instance variables, value of instance variables.

Each class is given a handle and no class is written more than once

Example, flatten a `Person` struct, with `name` (string), `place` (string) and `year` (long) attributes, with value `{"Smith", "London", 1934}`:

Person	8 bytes version number		h0 (class handle)
3 (number of fields)	int year	string name	string place
1934	5 Smith	6 London	h1 (object handle)

2.2 Client-Server Communication

Validity: the message reaches the destination

Integrity: the message received is identical to the one sent, and no message is delivered more than once

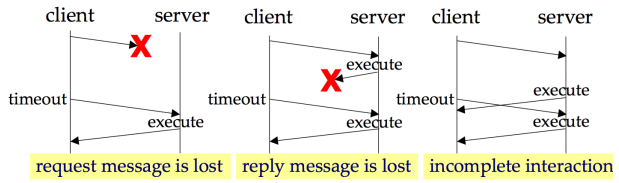
2.2.1 Request-Reply Protocol over UDP

Message is transmitted without acknowledgements or retires.

UDP is not a reliable communication service:

- Integrity: use checksum to detect corrupt packets
- Validity: Message validity is not guarantee because it suffers communication omission failures.

To build reliable request-reply protocols over UDP, client uses timeouts when waiting for server's reply and sends request repeatedly after timeout



2.2.2 Request-Reply Protocol over TCP

Message is transmitted with acknowledgement and retries. It is connection-oriented: a connection must be setup before any data are transferred.

TCP is a reliable communication service:

- Integrity: use checksums to detect corrupt packets; use sequence numbers to detect duplicate packets
- Validity: received packets are acknowledged; use timeouts and retransmission to deal with lost packets.

Reducing Overhead: reuse a single TCP connection to send multiple requests

2.2.3 Idempotent and Non-idempotent Operation

Idempotent operations: can be performed repeatedly with the same effect as if performed exactly once

Non-idempotent operations: if performed repeatedly have different effects from if performed exactly once

Chapter 3

Distributed Objects and Remove Invocation

Remote objects are objects that can receive remote invocations

Remote method invocation means method invocation between objects in different processes

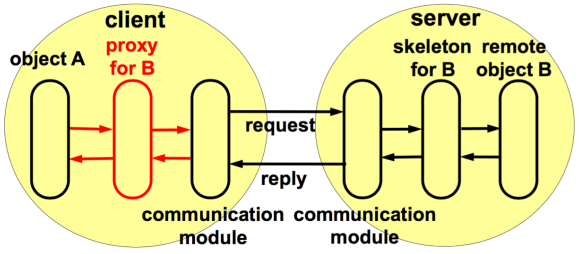
Remote object reference is used to uniquely identify an object throughout a distributed system

Remote interface: each remote object has a remote interface specifying the methods that can be invoked remotely

3.1 Invocation Semantics

Invocation Semantics	Fault Tolerance Measures		
	Retransmit request message	Duplicate filtering	Re-execute method or retransmit reply
Maybe	No	N/A	N/A
At-least-once	Yes	No	Re-execute method
At-most-once	Yes	Yes	Retransmit reply

3.2 Architecture of RMI



One proxy for each class of remote objects. It marshals arguments and unmarshals results

Communication modules transmit request and reply message between client and server

One skeleton for each class of remote objects. It unmarshals arguments, invokes the corresponding method in the remote object, and marshals results

Binder is a name service that maintains mappings from object names to remote object references.

3.3 Java RMI Architecture

Step 1: Design the remote interface

```
public interface City extends Remote {
  int addInformation(String cityName,
    String information)
    throws RemoteException;

  String getInformation(String cityName)
    throws RemoteException;
}
```

Step 2: Design the servant class to implement the methods specified in the remote interface

```
public class CityImpl
    extends UnicastRemoteObject
    implements City {
  public CityImpl() throws RemoteException {
    super();
  }
  int addInformation(...) ...  { ... }
  String getInformation(...) ...  { ... }
}
```

Step 3: Design the server class to create remote objects and register then in RMI registry

```
public class CityServer {
  public static void main(String args[]) {
    try {
      CityImpl aCityImpl = new CityImpl();
      Naming.rebind(
          "rmi://.../City",
```

```
          aCityImpl);
    } catch (Exception e) {...}
  }
}
```

Step 4: Design the client class to look up remote objects and access them

```
public class CityClient {
  public static void main(String args[]) {
    try {
      City aCityServer =
          (City) Naming.lookup("rmi://.../City");
      aCityServer.getInformation(...);
    }
    catch (RemoteException e) {...}
    catch (Exception e) {...}
  }
}
```

Step 5: Compile remote interface, generating proxies, skeletons, and compile source codes

Step 6: Start server, followed by clients

3.3.1 Implement Callback

Callback: the server informs the client when the information is updated

Client create a remote object that implements "Remote" interface, which contains a method for server to call

```
public interface Callback extends Remote {
  void theMethod() throws RemoteException;
}
```

```
public class CityClient {
  public static void main(String args[]) {
    try {
      City aCityServer =
          (City) Naming.lookup("rmi://.../City");
      Callback cb = new Callback();
      aCityServer.register(cb);
    } catch (RemoteException e) {
    }
  }
}
```

Server provides "register" and "deregister" operation for interested clients. When an event of interest occurs, server invokes the method in callback objects

```
public interface City implements Remote {
  void register(Callback cbObject)
      throws RemoteException;
  void deregister(Callback cbObject)
      throws RemoteException;
}
```

Chapter 4

Distributed File System

Distributed file system supports file accesses throughout an intranet. Requirements:

- Access transparency: clients should use the same interface for accesses to local and remote files
- Location transparency: client should see a uniform file name space
- File replication to improve performance and enhance fault tolerance
- Consistency maintenance of replications

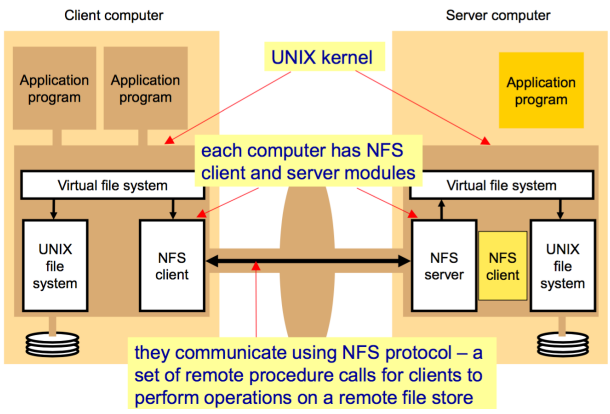
4.1 Stateless versus Stateful Servers

Stateful server remembers client's previous operations, so the requests are inter-dependent. It has heavier demand on server and harder to set up or restore on crashes.

Stateless server does not remember client's previous operations, so the request is independent of other requests. It is easier to set up and restore, less burden on server, but heavier demand on network

Stateless services are preferred for distributed file systems

4.2 Sun Network File System



Each computer can act as both a client and a server. File are accessed by file identifiers. File identifier is independent of the file name, created by the server hosting the file system and unique with respect to all files in the system.

4.2.1 Client-Server Communication

NFS server operations are stateless and idempotent, so at-least-once invocation semantics can be used.

NFS client operations need to supply an interface suitable for use by conventional application programs. The interface emulates UNIX file system semantics.

4.2.2 Mount Service

File systems have to be exported by the server and be mounted by a client before they can be accessed by the client.

At the server, a mount service process runs at user level on each server. A file `/etc/exports` contains the names of local file systems available for remote mounting and access list.

Client side use a modified UNIX mount command to request mounting, specifying remote host name, pathname of a directory in remote file system, and local name to be mounted. The command communicates with mount service process using PRC protocol. A table of mounted file systems is maintained at the client.

NFS does not enforce a single network-wide file name space. Clients can assign different local names to the same remote directory.

Pathnames are parsed, and their translation is performed in an iterative manner by clients. For example, if the root directory of a remote server is

mounted, access to `/bin/draw/readme` on the server requires three lookup request:

- `lookup(rootfh, "bin")` → `bin` file handler
- `lookup(binfh, "draw")` → `draw` file handler
- `lookup(drawfh, "readme")` → `readme` file handler

To improve efficiency, we using caching to take advantage of reference locality. For example, if `/bin/draw/install` is accessed soon after `/bin/draw/readme` on the same server, only one lookup request to the server is needed.

4.2.3 Client Caching

NFS client caches file data at block level to reduce communication with server. However, writing operation by a client do not result in immediate updating of cached copies of the same file in other clients. To maintain cache consistency, clients are responsible for polling the server to check the currency of cached data they hold.

NFS provides a close approximation to one-copy semantics:

- Each cache entry is tagged with two timestamps:
  1. The last validation time of cache, T_c
  2. The last modified time of cached file at server T_{mclient}
- Entry is considered valid when $T - T_c < t$, where t is a freshness interval and T is currect time
- When client want to access a cached file:
  - If $T - T_c < t$, read data from cache
  - If $T - T_c \geq t$, get T_{mserver} from server. If $T_{\text{mclient}} = T_{\text{mserver}}$, update T_c and T_{mclient} to current time. If $T_{\text{mclient}} < T_{\text{mserver}}$, send a new request to server to get new file.

4.2.4 Summary

NFS achieves access transparency because application use the same file operations for both local and remote files.

NFS does not achieve location transparency because different clients can mount the same server directory to different local directories.

4.3 Andrew File System

Andrew File System (AFS) are designed for scalability. AFS nodes are paritioned into two groups: (1) dedicated file servers and (2) a large number of clients.

Similar to NFS:

- Client can access to AFS files via normal UNIX file system operations.
- Venus clients and vice servers communicate using RPC
- Venus translates the pathnames into file identifiers using a step-by-step lookup
- File are accessed by file identifiers

Vice servers maintain a globally shared file name space. Clients have access to shared name space by means of a special local subdirectory `/afs`.

Shared files are grouped into volumes for ease of replication.

4.3.1 Server-side Replication

Each file is contained in exactly one logical volumne and each logical volume may have several physical volumes.

Each logical volume is associated with a Replicated Volume Identifier (RVID, location and replication-independent) and each physical volume is associated with a Volume Identifier (VID, location-independent).

Each shared file is identified by a unique file identifier (RVID + file handle).

Consistency for AFS server-side replication is read-one, write-all.

4.3.2 Whole-file Serving and Caching

Whole-file serving: entire files are transmitted to clients by AFS servers.

Whole-file caching: files transferred to a client are stored in a cache on local disk to satisfy future requests. When client close a file, the client transmits the updates to the server and retains the local copy.

AFS uses session update semantics to be more scalable. All other clients are able to see a modified file only after the file is closed by the client that modified it.

Callback mechanism is used to maintain cache consistency of client side. Vice server will issue a callback promise in *valid* state to Venus client when the server sends a fresh file copy to client. When there is a update of the file at server side, the server will inform all clients with valid callback promise and remove these clients from its callback list. When a client receives a callback from the server, the client sets the callback promise to *cancelled*.

However, if there is an update at server side before a client close a session / file, the callback has no effect on the file at currect session. If multiple clients write to a file concurrently, all update are silently lost except those of the last client closing the file.

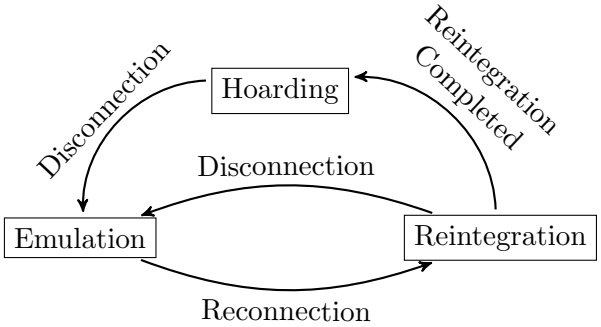
4.3.3 Coda File System

The main design goal of Coda file system is high availability. It allows clients to continue operation despite being disconnected from a server. Clients can use its locally cached copy of the files when disconnected from servers and reconcile later when the connection is established again.

To ensure client caches contain files that will be accessed during disconnection, we use hoarding technique, which fills the cache in advance with useful files:

- We first ask user to specify useful files in a hoard database. Then, priorities for each file are computed using hoard database and information on recent file accesses.
- We fetch files in priority to reach equilibrium, when all cached files have higher priorities than uncached files, cache is full, all uncached files have zero priority, or cached files are up-to-date.
- We do a hoard walk to periodically reorganize the cache to maintain equilibrium.

There are 3 states of Coda client with respect to a volume: hoarding, emulation, and reintegration.



When the connection is established again, updates made to files during disconnection are transferred to the server. In case of conflicts, we can use automatic conflict resolution or manual intervention to resolve conflict. However, conflicts are rare because most files are written by only one user at a time.

Chapter 5

Peer-to-peer File Sharing Systems

All processes play similar roles and they interact cooperatively as peers to perform distributed computation. Every node provides some services that helps other nodes in the network to get services.

We will look at three unstructured P2P File Sharing, i.e. Napster, Gnutella, and KaZaA, and three structured DHT Systems, i.e. Distributed Hash Table Services, Consistent Hashing, and Chord.

5.1 Napster

Napster has a centralized directory server for centrolized search, but file downloads are done in a distributed way. The centralized server has difficult time keeping up with increasing traffic.

All clients will upload file list and IP address to Napster centralized directory. Users will search for files using the centralized directory. When users found the file they want, they pings hosts that have the file to look for the best transfer rate for downloading.

5.2 Gnutella

Gnutella uses a decentralized method to search for files. Each application instance has the responsibility to store files, serve files, route queries and respond to queries.

However, Gnutella use flooding strategy to search for files, and reverse path forwarding is used for responses. The actual downloading is done by point-to-point TCP. As a result, each query can generate huge amount of traffic. Users can limit the number of hops to travel by setting the TTL field.

To establish connection, we first connect to a bootstrap node to get IP addresses of existing Gnutella nodes. Then, we send join messages to some existing Gnutella nodes.

5.3 KaZaA

KaZaA provides powerful file search and transfer service without server infrastructure using hierarchical architecture. Frequent uploaders can get priority in server queue.

Nodes that have more connection bandwidth and are more available are designated as supernodes. Each supernode acts as a mini-Napster hub, tracking the contents and IP addresses of its children.

Connection: A list of potential supernodes are included within the software. New peer goes through the list until it finds an operational supernode.

Metadata: When a node connects to a supernode, it uploads its metadata, i.e. file name, file size, content hash, and file description.

Query: Node first sends keyword query to supernode. If supernode cannot get enough number of matches, it forwards query to subset of supernodes. Query results contain the ContentHash of the wanted file and a list of nodes holding the file.

Download: If file is found in multiple nodes, user can download the files parallelly.

5.4 Distributed Hash Table Services

In Distributed Hash Table (DHT), we want to assign particular nodes to hold particular contents and arrange neighbour relationship between nodes in a restrictive structure to facilitate query routing.

We use hash function to map files to unique identifiers. Each node is responsible for the files that hash within its range, it must know the location of the files (indirect) or physically store the files (direct).

Query must be efficiently rounted to the node whose range covers the file. This should be implemented in a fully distributed manner.

When nodes join / leave the system, we need bootstrap mechanism, repartition range space, and update routing information.

5.5 Consistent Hashing

We need to choose the range space of hash function as *a circle of identifiers* from 0 to $2^m - 1$. m must be large enought to avoid collision of two identifiers.

Each node is assigned the range space from the identifier of its counter-clockwise neighbour node on the identifier circle to its own identifier.

If each node maintains the identifier and IP address of its successor node, the complexity of routing information maintained at each node is $O(1)$ and the complexity of query routing is $O(N)$

If each node maintains the identifiers and IP addresses of all nodes, the complexity of routing information maintained at each node is $O(N)$ and the complexity of query routing is $O(1)$

5.6 Chord

Each node n maintains identifiers and IP addresses of m successors. The i-th entry in the finger table contains the first node that succeeds n by at least 2^{i-1} on the identifier circle.

In routing, each node n sends a query for key k to the node in entry:

$$\lfloor log_2(k-n) \rfloor + 1$$

The complexity of routing information maintained at each node is $O(log(N))$ and the complexity of query routing is $O(log(N))$

Joining and leaving process work like insertion and deletion in a double-linked list. When joining, the node need to copy the location information of all keys within its range from its sucessor, and then need to build the finger table and update all current finger tables. The complexity is $O(log^2(N))$.

Chapter 6

Name Services

Names identify resources in a *location-independent* fashion. Addresses identify resources in a *location-dependent* fashion.

Name service does translation between names and addresses. Name resolution is the conversion process from names to addresses.
Main requirements of name services:

- Scalability: to handle arbitrary number of names and to serve an arbitrary number of administrative organizations over long lifetime
- Reliability: name services should not fail frequently
- Performance: name resolution should have short response time

Name space is the collection of all valid names. Name space requires a syntatic definition. Name space can have hierarchical structure or flat structure, but hierarchical name space has better scalability than flat name space.

6.1 Name Resolution Process

Physically, the resolution process may involve more than on name server because the mapping may not be available at every name server.

Method 1: Iterative Client-controller Navigation. A client iteratively contacts a group of name servers to resolve a name. However, this method has high communication cost if name servers are located far away.

Method 2: Non-recursive Server-controller Navigation. A name server coordinates the name resolution process iteratively or multicast on behalf of a client.

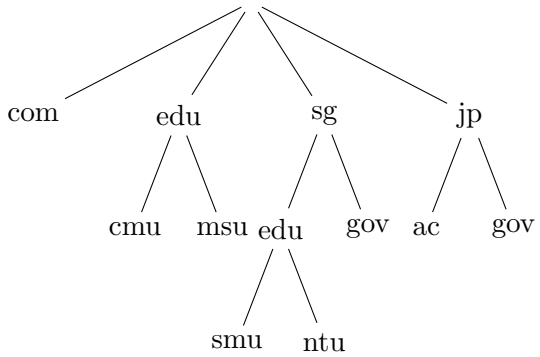
Method 3: Recursive Server-controller Navigation. If a name server does not have the name mapping, it contact another name server, and this procedure continues recursively until the anem is resolved. The caching results is more effective, but it put higher performance demand on each name server.

6.2 Domain Name System

To meet requirements of name services, DNS uses:

- Scalability: hierarchical name space
- Reliability: use replication
- Performance: use caching

6.2.1 Hierarchical Name Space



DNS uses a decentralized design, i.e no single server is responsible for all names. The entire name spaces is partitioned among a large group of name servers. Each server holds authoritative mappings for the names in one or more domains.

Each mapping contains domain name, time-to-live, class, type and value. The information held in DNS servers are:

- Mappings between names and IP addresses for the local domain
- Names and addresses of all name servers in local domain, sub-domains, and root name servers

Type of mapping:

- A-type record maps a computer name to its IP address

- CNAME-type record maps an alias to its canonical name
- NS-type record specifies authoritative name servers. Each NS-record is accompanied by an A-type record.
- MX-type record gives preferences and domain names of mail hosts

6.2.2 Replication for Reliability

Authoritative mappings of each domain must be held by at least two name servers, i.e. the primary server and secondary servers. Secondary servers periodically check with the primary server to see whether their stored mappings are up-to-date.

DNS uses a simple request-reply protocol between clients and name servers based on UDP. DNS primarily uses a combination of iterative client-controlled navigation and non-recursive server- controlled navigation.

6.2.3 Caching for Performance Improvement

Results of name resolution may be cached by clients and name servers. Clients and name servers always consult their caches before sending queries to other servers for name resolution.

However, caching introduces inconsistency problem. Query answers returned from cached mappings may be out-of-date. The consistency problem can be addressed by timeouts (time-to-live values).

Chapter 7

Time and Global States

Although one of the property of distributed system is no globally shared clock, many distributed applications still depend on timing.

7.1 Synchronizing Physical Clock

Computers have their own physical clocks that deviate from one another. Universal Coordinated Time (UTC) is commonly used reference clock.

Clock drift: clock count time at different rates.

Drift rate: the change in offset between a clock and a perfect reference clock per unit of time measured by the reference clock.

7.1.1 Cristian's Method

The algorithm:

1. A requests the time of B
2. B replies with the time value t
3. A records the total RTT, $T_{\text{round}} = t_2 - t_1$
4. A sets its clock to $t + T_{\text{round}}/2$

To calculate the accuracy of Cristian's Method, we need to consider 2 cases:

1. When the transmission time of reply message approaches 0, B's clock reading is t.
2. When the transmission time of reply message approaches T_{round}, B's clock reading is $t + T_{\text{round}}$.

When A receives the reply, the time reading on B should be in $[t, t + T_{\text{round}}]$. A is accurate within bound $T_{\text{round}}/2$

7.1.2 Berkeley Algorithm

The algorithm:

1. Master periodically polls slaves
2. Slaves send their clock readings back
3. Master evaluates the local times of slaves by observing the RTT
4. Master averages the values obtained and its own clock reading
5. Master sends the amount of adjustment to each slave

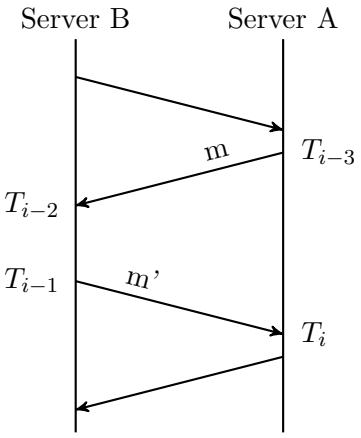
We use average clock reading because it cancels out individual clocks' tendencies to run fast or slow.

7.1.3 Network Time Protocol

Cristian's method and Berkeley algorithm use centralized design. Network Time Protocol (NTP) is used to distribute time information over the Internet.

NTP has a network of servers structured hierarchically into a synchronization subnet. This setup is fault-tolerance because servers can reconfigure themselves if someone becomes unreachable.

Analysis of Accuracy



Time between sending of m and receipt of m' is

$$T_i - T_{i-3}$$

Time between receipt of m and sending of m' is

$$T_{i-1} - T_{i-2}$$

The total transmission time of m and m' is

$$(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$$

The transmission time of m' is

$$[0, (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})]$$

If transmission time of m' approaches 0, when A receives m' B clock reading is

$$T_{i-1}$$

If transmission time of m' approaches $(T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$, when A receives m' B clock reading is

$$T_{i-1} + (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) = T_i - T_{i-3} + T_{i-2}$$

If A want to synchronize with B as accurately as possible, A should set its clock to

$$\frac{1}{2}(T_{i-1} + (T_i - T_{i-3} + T_{i-2})$$

The accuracy is

$$\pm\frac{1}{2}(T_{i-1} - (T_i - T_{i-3} + T_{i-2})$$

7.2 Causal Ordering and Logical Clocks

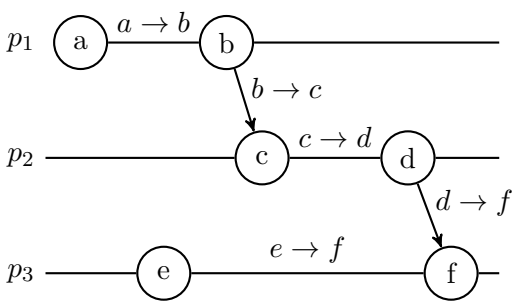
We refer the state of process p_i and as s_i, and there are $i = 1 \ldots N$ number of processes.

Process takes a series of actions, i.e. sending a message, receiving a message, or transforms the state of the process.

An event is the occurrence of a single action.

7.2.1 Causal Ordering

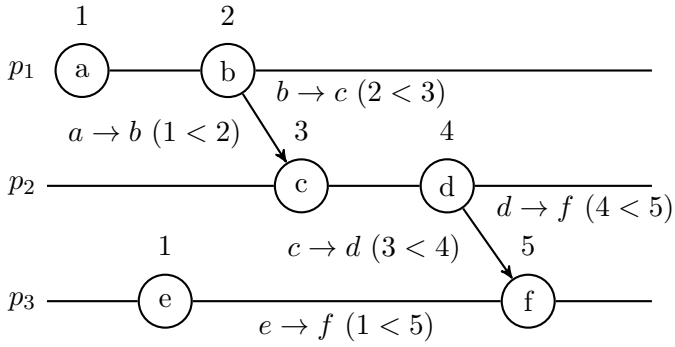
Events can be ordered based on cause-and-effect. Event within a single process p_i can be placed in a unique total ordering. The event of sending the message occurs before the event of receiving the message.



From the figure above, we can deduce that a happens before f, $a \to f$, and a is parallel to e, $e \| e$

7.2.2 Lamport's Logical Clocks

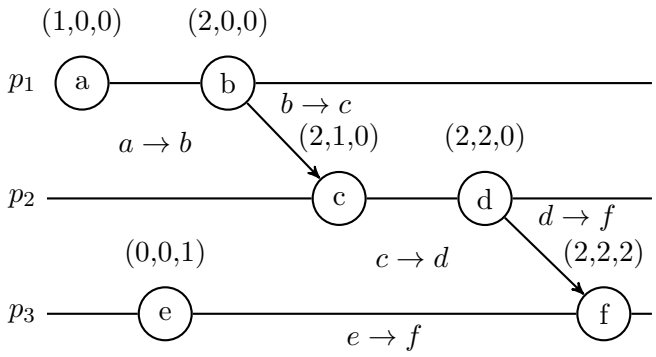
Logical clock is a monotonically increasing software counter whose value bears no particular relationship with real time.



From the diagram above, since $a \to f$, so $L(a) < L(e')$. This is always true, but the converse is not. Specifically, if $L(e) < L(e')$, we cannot infer that $e \to e'$.

7.2.3 Vector Clocks

A vector clock for N processes is an array of N integers. Each process has its own vector clock.



If $e \to e'$, then $V(e) \to V(e')$.
If $V(e) \to V(e')$, then $e \to e'$.
If $e \| e'$, then neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$

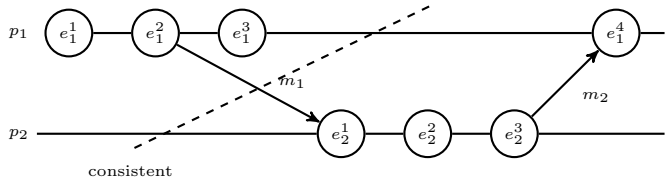
7.3 Global States

In many situations, we need to find out whether a particular property is true when a distributed system executes. We want to assemble a meaningful global state from local states recorded by different process at different real times.

The state of process p_i is the state immediately after its k-th event occurs. Global state is a set of states of all processes.

7.3.1 Consistent Cut

A consistent cut is a cut C, such that for all events $\forall e \in C$, $\exists f \in C, f \rightarrow e$. Otherwise, the cut is unconsistent.



7.3.2 Recording a Snapshot

To record a snapshot, we need to record the states of all processes and the state of communication channels.

Chandy and Lamport Algorithm

The algorithm uses a marker messages to prompt the receiver to save its own local state and determine which messages to include in the channel state.
The algorithm:

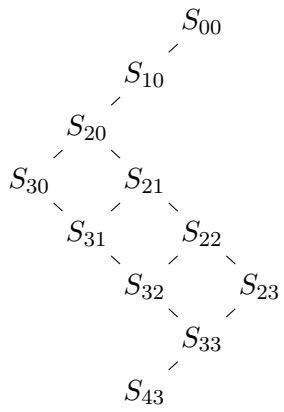
- Any process may initiate a snapshot recording at any time
- Each non-initiating process records its proces state on receiving the first marker message
- If p_j is the initiating process, p_j initiates the messages arriving on channel $c_{i \rightarrow j}$ between p_j initiates the snapshot recording and receives the marker message from p_i
- If p_j is non-initiating process and $c_{i \rightarrow j}$ is the channel through which p_j receives its first marker message, $c_{i \rightarrow j}$ state is recorded as an empty set
- If p_j is non-initiating process and $c_{i \rightarrow j}$ is not the channel through which p_j receives its first marker message, p_j records the message arriving on channel $c_{i \rightarrow j}$ between p_j receives the first marker message and the marker message from p_i
- Each process completes all recording activities when it has received a marker message from all other processes. Then, it may send the recorded states (local process state + channel states) to the initiating process.

7.4 Distributed Debugging

We want to establish what occurred during the execution of a distributed system and to check whether they are correct.

7.4.1 Lattice of Consistent Global States

Refering to the figure in section 7.3.1, we will get the following lattice of consistent global states. The lattice is constructed by listing all consistent global states starting from the initial state and advance the global state by one event at a time.



Each path from top-level state to the bottom level state in the lattice represents an ordering of all events that are consistent with causal ordering. Without global time, we cannot tell which one is the actual execution of the system among all possible executions.

In general, $< s_1^{c_i}, s_2^{c_j}, \ldots, s_N^{c_N} >$ is a consistent global state if $V(e_i^{c_i})[i] \geq V(e_j^{c_j})[i]$ for any i and j.

If there are only two processes, $S_{ij} =< s_1^i, s_2^j >$ is a consistent global state if $V(e_1^i)[1] \geq V(e_2^j)[1]$ and $V(e_2^j[2]) \geq V(e_1^i)[2]$

To check whether a constraint is broken during execution, we need to evaluate whether each global state satisfies the constraint:

- If all states satisfy the constraint, the constraint cannot be broken in the execution
- If there exists a state not satisfying the constraint, the constraint is possibly broken in the execution
- If every path from top to bottom passes through at least one state that does not satisfy the constraint, the constraint must be broken in the execution

Chapter 8

Coordination and Agreement

Problems and issues in coordination and agreement:

- Mutual exclusion: for processes to coordinate their accesses to shared resources
- Election: for processes to agree on a coordinator
- Consensus: for processes to agree on some value

8.1 Distributed Mutual Exclusion

Objective: to prevent interference when a set of processes access shared resources that require exclusive use.

Requirements for Mutual Exclusion:

1. Safety: at most one process may execute in the critical section at a time
2. Liveness: requests to enter and exit the critical section eventually succeed, no deadlock and starvation.

8.1.1 Central Server Algorithm

Token: permission to enter the critical section

To enter critical section, a process sends a request to the server and awaits a reply from it. If no other process has the token, the server replies immediately and granting the token. If there is another process has the token, the server queues the request. The process enters the critical section once it receives the token.

To exit a critical section, a process sends the token back to the server. The server will get the next process from the queue and grat the token.

The algorithm satisfies safety and liveness requirements, but the server is a single point of failure and can become a performance bottleneck.

Performance:

1. Bandwidth consumption: entering critical section takes two message (request and grant), existing critical section takes one message (release)
2. Client delay: Entering critical section delays the process by two message transmissions
3. Synchronization delay: two message transmissions (release message to server and grant message to next process)

8.1.2 Ring-based Algorithm

The algorithm:

- A token is circulated in clockwise direction around the ring.
- On receiving token, if the process
  - does not require to enter critical section, it forwards the token
  - does require to enter critical section, it retains the token and enters. The process sends the token to neigbour to exit critical section.

Performance:

- Bandwidth consumption: if every process constantly wants to enter a critical section, each token pass results in one entry and exit.
- Client delay: between 0 to N message transmissions
- Synchronization delay: between 1 and N message transmissions

8.1.3 Ricart and Agrawala Algorithm

This algorithm requires a total ordering of all events. Each process is in either `RELEASED`, `WANTED`, or `HELD` states.

The algorithm:

- To enter critical section, a process change to `WANTED` state and send request messages to all other processes. The process can enter critical section (`HELD` state) only when all other processes have replied to the requests.
- When a process receives a request message, if the receiver is in `HELD` state, it queue the request. If the receiver is in `WANTED` state, it compares the timestamp of the request with its own request. If the request has smaller timestamp, reply. Else, queue the request.
- When the process exits critical section, change to `RELEASED` state and reply to all queued requests.

Performance:

- Bandwidth consumption: Entering critical section takes $N-1$ requests and $N-1$ replies.
- Client delay: entering critical section delays the process by 2 message transmission if all requests are sent simultaneously
- Synchronization delay: 1 message transmission

8.2 Election

Objective: to choose unique process (coordinator) to play a particular role.

Different processes may call elections concurrently, so we need to ensure a unique coordinator is elected. More processes may fail during election, so we need to ensure that non-crashed process is elected eventually.

We assume that each process has a unique identifier, and we require the coordinator with the largest identifier to be chosen.

Turnaround time: the number of serialized message transmission times between the initiation and termination of an election.

8.2.1 Ring-based Algorithm

Arrage processes in a logical ring. A process is participant if it engaged in some elections, otherwise, it is non-participant.

The algorithm:

- A process initiate an election by marking itself as participant, placing its identifier in the election message and send it to its neigbour.
- When a process receives election message, it compares the identifier in the message with its identifier.
  - If arrived identifier is greater, forward the message
  - If arrived identifier is smaller and the receiver is non-participant, substitude with own identifier and forward the message
  - If arrived identifier is smaller and the receiver is participant, drop the message
  - Once the process forwards the message, it marks itself as participant
  - If the arrived identifier is equal, the process become coordinator. It marks itself as non-participant and sends elected message to neighbour. All other processes will marks themselve as non-participant, records the coordinator, and forwards the message.

Performance:

- Bandwidth consumption: when the anticlockwise neighbour has the largest identifier, it takes $N-1$ messages to reach the neigbour, then N more messages to complete the circuit, then N elected messages. In total, we need $3N-1$ messages
- Turnaround time: $3N-1$ messages

8.2.2 Bully Algorithm

This algorithm allows process to crash during an election. It assumes asynchronous system and us timeouts to detect process crash. If no response arrives within time $T = 2T_{\text{trans}} + T_{\text{process}}$, then intended recipient must have crashed.

We assume that each process knows which processes have larger identifiers and it can communicate with all such processes.

The algorithm:

- Any process that detects the coordinator crashes can start an election. When the process begins an election:
  - The process sends election messages to all processes with larger identifiers.
  - If no answer message responds within time T, the process wins the election and becomes coordinator. The process announces the result of election by seding coordinator messages to all processes.
  - If some process responds within answer message, the process waits for coordinator message to arrive from new coordinator
- When a process receives an election message, the process sends back answer message and starts an election
- If a crashed process comes back up, and it knows that it has the largest identifier, it announces that it is coordinator by sending coordinator messages.

Performance:

- Bandwidth consumption:
  - Best case: the process with the second largest identifier detects the crash, 1 election message and $N-2$ coordinator messages.
  - Worst case: the process with the smallest identifier detects the crash, we need $O(N^2)$ messages.
- Turnaround time: Best case: $T + 1$ message transmission

8.3 Consensus Problem

Objectives: for processes to agree on a value even in the presence of failtures after one or more processes have proposed the value.

Requirements of Consensus Problem:

- Termination: each correct process sets its decision variable eventually
- Agreement: the decision values of all correct processes are the same
- Integrity: if the correct processes all proposed the same value, then any correct process in the decided state has chosen that value

8.3.1 Consensus in Synchronous System

We assume that up to f of the N processes may crash.

In the absence of process crash, each process p_i multicasts its proposed value v_i to all other process and waits until it has collected all N values and then evaluates a function to set its decision variable.

When processes crash, the processes may not actually send values to all other processes. So not all processes receive the same set of values.

To solve the problem, we run the algorithm for $f+1$ rounds. At the end of the last round, all correct processes that survice must have received the same set of values.

8.3.2 Consensus in Asynchronous System

No algorithm can guarantee to reach consensus in asynchronous systems even if there is only one faulty process

Chapter 9

Replication and Consistency

Data replication means maintaining copies of data at multiple computers to improve performance, scalability, reliability and fault-tolerance. However, we need to pay the price of consistency of replicas.

9.1 Data-Centric Consistency Models

To provide a system wide consistent view of data objets when concurrent processes simultaneously update data objects.
Target scenario: concurrent processes may simultaneously update data objects.

9.1.1 Strict Consistency

Any read operation on a data object returns a value corresponding to the result of the most recent write operation on the object. All wirtes are instaneously visible to all clients.

In the absence of a global clock in distributed system, it is difficult to assign a unique timestamp to each operation that corresponds to actual global time. It is also difficult to define precisely which write operation is the most recent.

9.1.2 Sequential Consistency

The result of any execution is the same as if the operations by all clients were executed in some sequential order. The operation of each individual client appear in this sequence in the order specified by its program.

9.1.3 Causal Consistency

Causal consistency makes distinction between events that are potentially causally related and those that are not. A read is causally related to the write that provided the data the read got.

Writes that are potentially causally related must be seen by all clients in the same order. Concurrent writes may be seen in different orders by different clients.

9.1.4 FIFO Consistency

Writes done by a single client are seen by other clients in the order in which they were issued. Writes from different clients may be seen in different orders by different clients.

9.2 Consistency Protocols

A consistency protocol describes an actual implementation of a specific consistency model.

9.2.1 Primary-Based Protocols

For each data object, one replica is designated as the primary replica and the remaining replicas are called backups / slaves.

Remote-Write Protocol

Implementation of sequential consistency. The algorithm:

- Write operations are forwarded to the primary replica. The primary replica performs the update and then forwards the update to the backup replicas.
- Each backup replica performs the update as well and sends an acknowledgement back to the primary replica
- Finally, the primary replica sends an acknowledgement back to the initiating client.

Write operations are blocking, read operations can be carried out at any replica.

Local-Write Protocol

The algorithm:

- The primary replica is migrated to the replica that receive client request.
- As soon as the primary replica has updated its local copy, it returns an acknowledgement to the client
- Then, the primary replica tells the backup replicas to perform the update

Advantage: Multiple successive write operations can be carried out locally.

9.2.2 Active Replication

In active replication, all replicas play the same role and write operations are carried out at all replicas. Operations are sent to all replicas via totally ordered multicast. All replicas start in the same state and perform the same operations in the same order.

Write operations are blocking, and read operation can be carried out at any replica.

Active replication achieves sequential consistency. It can tolerate up to f byzantine failture using $2f + 1$ replicas.

9.2.3 Gifford's Quorum-Based Protocol

A network partition separates a group of replicas into two or more subgroups in such a way that replicas of one subgroup can communicate with one another, but replicas of different subgroups cannot communicate with one another.

A quorum is a subgroup of replicas whose size gives it the right to perform operations.

In this protocol, update operation may be performed by one subset of replicas only. When a replica is updated, its version number is changed accordingly. Operations are applied only to replicas with the latest version number.

Operation:

- Each read operation must obtain a read quorum $\geq R$ replicas before it can read
- Each write operation must obtain a write quorum $\geq W$ replicas before it can write

R and W are set such that (T is the total number of all replicas):

$$W > \frac{T}{2}$$

$$R + W > T$$

These constraints ensure that any pair of read and write quorums or two write quorums contain common replicas.

To perform read operation, client must contact at least R replicas and the replicas must contain at least one up-to-date replica. The read can be done in any up-to-date replica in the read quorum.

To perform write operation, client must contact at least W replicas and the replicas must contain at least one up-to-date replica. If there are out-of-date replicas in the set, replace them with the up-to-date copy. The write operation is applied to all replicas in the write quorum, and version number is incremented.

9.3 Client-Centric Consistency Models

Target scenario: lack of simultaneous updates and most operations involve reading data.

Eventually consistency: if no update takes place for a long time, all replicas will gradually become consistent. Eventually consistent works fine as long as clients always access the same replica, but problems arise when different replicas are accessed.

Client-centric consistency provides guarantees for a single client concerning the consistency of its accesses to data objects.

9.3.1 Monotonic Reads

If a client reads the value of a data object x, any successive read operation on x by that client will always return that same value or a more recent value.

9.3.2 Monotonic Writes

A write operation by a client on a data object x is completed before any successive write operation on x by the same client.

9.3.3 Read Your Writes

The effect of a write operation by a client on a data object x will always be seen by a successive read operation on x by the same client

9.3.4 Writes Follow Reads

A write operation by a client on a data object x following a previous read operation on x by the same

client is guaranteed to take place on the same or a more recent value of x that was read

9.3.5 Implementation of Monotonic Reads Consistency

Each write operation is assigned a globally unique identifier. Each client keeps track of two sets of write operations:

- Read set has identifiers of writes relevant to the read operations performed by the clinet
- Write set has identifiers of writes performed by the client

The algorithm:

- When a client performs read at a replica, the replica check whether all the write operations in the read set have been taken place locally.
- If not, the replica contacts other replicas to update itself.
- After the read operation, the write operation that have taken place at the selected replica and which are relevant to the read operation are added to the client's read set.