Advanced Distributed Systems

> Author: Jose 胡冠洲 @ ShanghaiTech
>
> Teacher: [Prof. Tao Xie](#) from San Diego State
>
> Figures come from professor's slides, unless otherwise specified.

Distributed Systems Basics

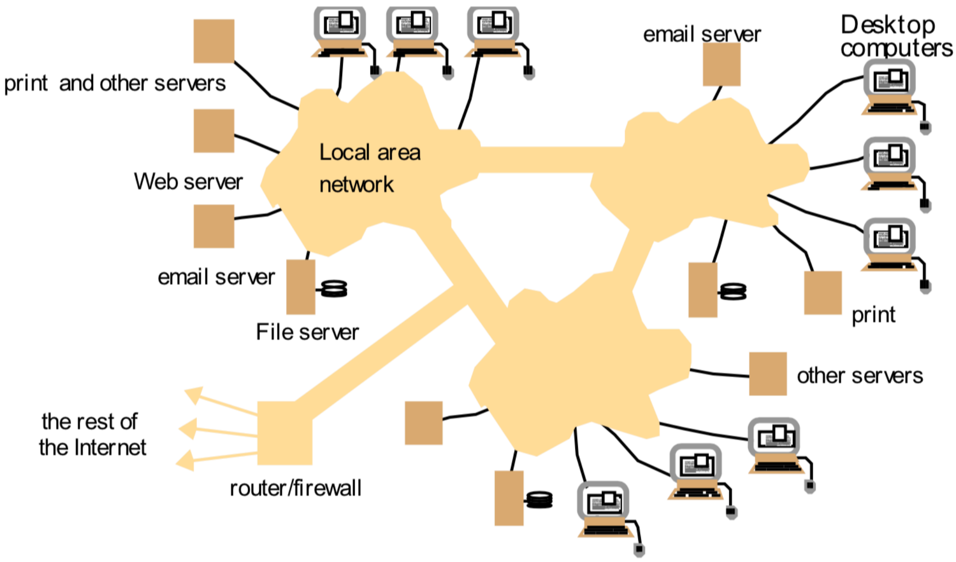
"Distributed Systems: Concepts & Design", Chapter 1

Characteristics of Distributed Systems

Definition: **Independent** components (may be far apart) working **collaboratively** by passing messages over computer networks.

1. Concurrency
2. Lack of a global clock
3. Independent failures

Compared to Parallel Computing (focusing on multiple homogeneous processors in ONE computer), distributed systems require a better understanding of: *Heterogeneity*, *Openness*, *Security*, *Scalability*, *Failure handling*, *Concurrency*, *Transparency*.



Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn. 4
© Pearson Education 2005

Challenges & Issues

- Heterogeneity: Different hardware, OS, network architecture, languages; VMs
- Openness: Standard published interfaces; RFC
- Security: Confidentiality + Integrity + Availability
- Scalability: Extensiveness, cost v.s. performance
- Fault Tolerance: Detecting faults, masking (hiding), ignoring, recovery; Dependence via Redundancy
- Concurrency issues
- Transparency:
  - *Access Transparency*
  - *Location Transparency*
  - *Concurrency Transparency*
  - *Replication Transparency*
  - *Failure Transparency*
  - *Mobility Transparency*
  - *Performance Transparency*
  - *Scaling Transparency*

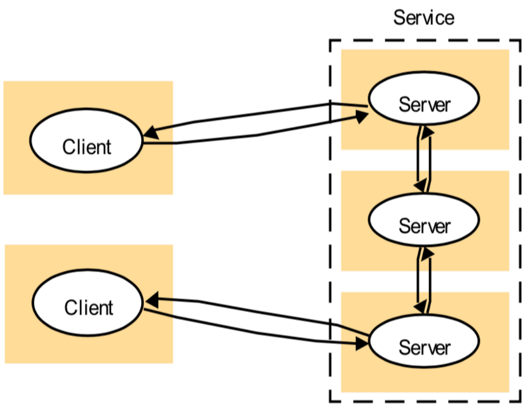
System Models

"Distributed Systems: Concepts & Design", Chapter 2

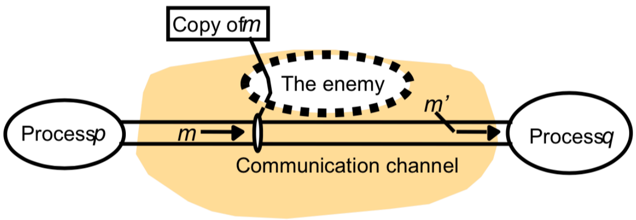
What is a Model

A description of a complex entity or process, simplified by ignoring certain details.

- *Architectural* models: focusing on distribution & communication of data / tasks amongst physical nodes



- *Fundamental* models: focusing on conceptual issues to be solved



Architectural Models

Follow 3 steps to build an architectural model:

1. Functions of individual components
2. Placement of the components across a network
3. Interrelationships between the components

A distributed system usually expose to users **services**, which masks over heterogeneity of the underlying platform. **Servers** (a process that accepts requests from other processes) provide services, and **clients** access services.

- *Clients-Server* pattern - simple
    - 1 / more servers
    - 1 / multi-level servers; caches
    - Run code remotely v.s. Retrive code and run locally
- *Peer-to-Peer* (P2P) pattern - decentralization, improves scalability

Design requirement of distributed architectures:

- Performance: throughput, load balancing
- Quality of Service (**QoS**): latency
- Dependability: safety

Fundamental Models

Fundamental models describe basic issues (shared among different architectures).

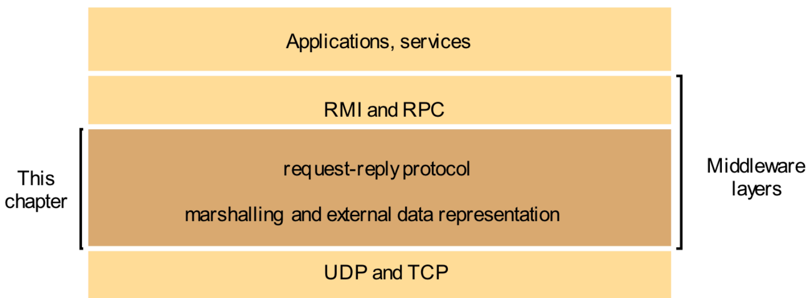
- Interaction model
    - Sequential v.s. Distributed timing / state
    - *Synchronous* v.s. *Asynchronous*
- Failure model
    - **Crashes** / Fail-stops
    - ***Omission* Failures**: failed to do what it is supposed to do (end up doing nothing); can recover by reapply
    - ***Timing* failures**: takes longer than time bound
    - ***Arbitrary* Failures** (*Byzantine*): inconsistent failures at arbitrary times
- Security model
    - Authorization: access rights
    - Authentication: identity
    - Denial of Service (**DoS**) attacks → Distributed DoS (**DDoS**)

Interprocess Communication (IPC)

"Distributed Systems: Concepts & Design", Chapter 4

Middleware Layer of Communication

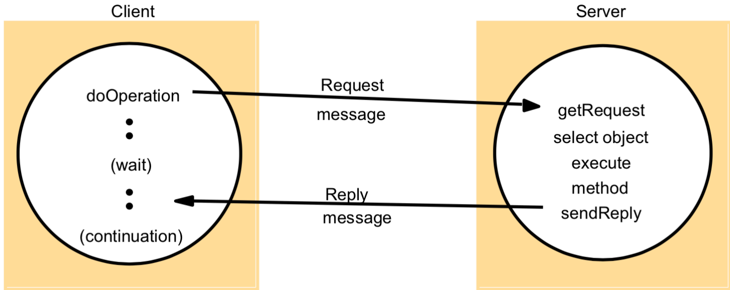
Layers:



- IPC relies on Remote Method Invocatons (**RMI**) / Remote Procedure Calls (**RPC**) / Events
- RMI / RPC / Events are implemented upon **Request-Reply** protocol
- Request-Reply protocol is built upon User Datagram Protocol (UDP) / Transmission Control Protocol (TCP, *streams*)
    - Defines external data representation, e.g. HTTP, XDR, Java object serialization
- UDP / TCP are provided by the operating system, they must act on proper hardware, with a message destination
    - Destination can be IP address + *Port* / multicast
    - Often utilized through the **Socket** programming API

Request-Reply Protocol

Request-Reply paradigm:



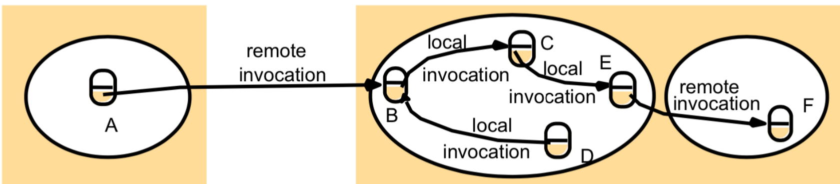
Involves *send* & *receive* operations, and can be synchronous / asynchronous. Some techniques can be used to protect / optimize this procedure:

- *Idempotent* operations: can be performed repeatedly with the same effect as performing once
- Cache history of replies to avoid re-execution on the same requests
- Timeout & Retrying
- ...

Marshalling & *Unmarshalling*: convert internal data to / from standard external transferring format.

RMI & RPC

Local v.s. Remote invocation pattern:



RMI needs:

- *Message structure*: defines how a message is composed

  Example: `| Message type | Request ID | Remote object reference |`

- *Remote object reference*: an identifier throughout the distributed system to access the remote object

  Example: `| IP address | Port # | Time | Object ID | Remote interface |`

- *Remote interface*: specifying which methods can be invoked remotely, and their signatures

  Example: implemented using Interface Definition Languages (*IDL*)

Three possible effects of RMI:

1. The state of the receiver may get changed
2. A new object may be instantiated
3. Further chained invocations of other methods

Three fault-tolerance semantics choices:

Retransmit Request Message on Failure?	Duplicate Request Filtering?	Re-execute / Retransmit Reply on Duplication?	Invocation Semantics
No	\	\	*Maybe*
Yes	No	Re-execute the procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

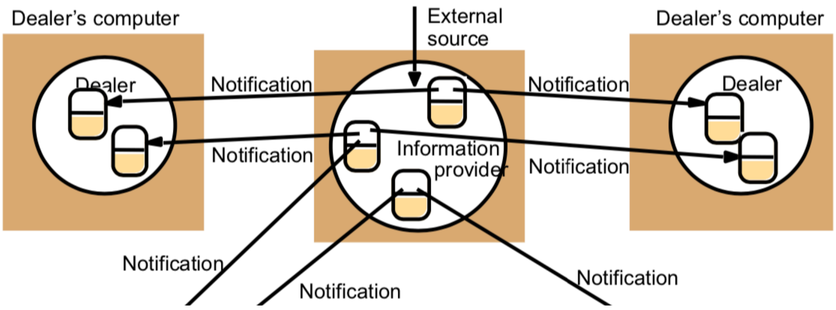
- *Maybe* is suitable when occasional failed invocations are acceptable
- *At-least-once* is suitable for idempotent invocations
- *At-most-once* is completely fault tolerant

> A **Proxy** on the client side plays the role of a local object gate as to the invoking object, it marshalls the request and sends it downwards & unmarshalls the received result. (In RPCs, called a **Stub**)
>
> A **Binder** is a table that maps remote object textual names to remote object references / maps remote procedures to local ports. Servers register into the binder, and clients lookup this table.

To void polling, often uses **Callbacks** (Server calls a remote call-back method on client call-back object on reply).

Events & Notifications

Events & Notifications pattern:



Components of this pattern:

- *Object of Interest*: the object that experiences changes of state
- *Event*: occurs at the end of the state change of an Object of Interest
- *Notification*: a marshalled message containing information about an Event
- *Subscriber*: an object that is interested in a certain type of Events; will receive Notifications of that type of Events
- *Publisher*: the object responsible for generating Notifications
  - Can be the Object of Interest, or
  - An *Observer Object* (*Proxy*): an object gate, like proxy in RMI, but at the server side

Distributed File Systems

"Distributed Systems: Concepts & Design", Chapter 12

FS & DB Terms

Filesystem (FS):

- Local FS [Kernel-level]: part of the operating system kernel, which wraps over naked storage devices (HDD, SSD, ...) and provides POSIX API for upper applications to use these devices; e.g. NTFS (Win), Ext4, XFS, BtrFS, ... (Linux), HFS, APFS (Apple)
- Virtual FS (VFS) [Kernel-level]: an OS module that supports interactions with multiple devices formatted as different FS formats; e.g. Linux VFS supports reading and writing over Ext4 and also a plugged-in FAT-32 USB drive (may sometimes be considered as part of the local FS)
- FS Middleware [User-level]: a middle layer between user applications and the local FS, which often provides parallel / distributed optimizations for IO scheduling to improve FS performance; e.g. GPFS, Lustre, Sun NFS, OrangeFS (PVFS), Ceph, ...
    - Can be integrated into the OS kernel as well, like NFS protocol integrated in Linux kernels
    - Can run as FUSE (Filesystem in User SpacE)
    - Can be implemented as an application-level dynamic library; may require user applications to get re-compiled with the designed APIs instead of POSIX

Database (DB):

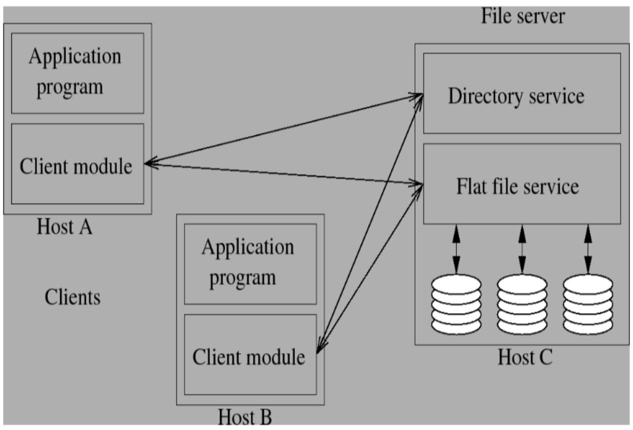
- Application-level DB [User-level]: similar to FS Middleware, provides a new layer of data management over the local FS to unify data format and provide in-memory caching; e.g. MySQL
- Naked-device DB [Kernel-level]: databases that directly operate on naked devices to improve performance, started supported by Oracle (no longer popular)
- Object Storage (Key-Value Store, KV Store) [Kernel-level / User-level]: databases built upon new object storage devices (or somehow simulated using traditional disks), guided by *NoSQL* design; e.g. Redis, Oracle NoSQL

Distributed FS we discuss here are mostly FS Middleware operating over multiple distributed nodes connected over a network.

File Service Design

Distributed FS have many different kinds of design and implementation, each having its own characteristics and suited workloads.

A typical distributed file service model:



Typical caching techniques for distributed FS:

- Server-side caching:
    - Read-ahead
    - Delayed-write / Write-back v.s. Write-through v.s. Commit
- Client-side caching (Collective IO)

Peer-to-Peer Systems (P2P)

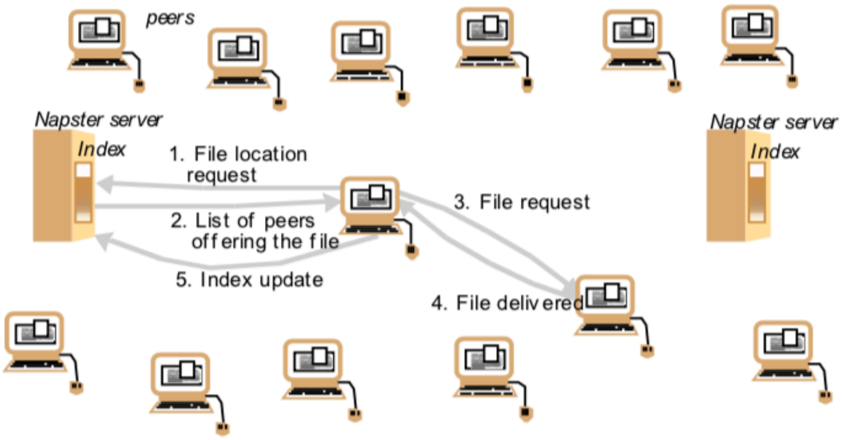
"Distributed Systems: Concepts & Design", Chapter 10

Overview of P2P Systems

Purpose: Utilize data and computing resources available in personal computers on the Internet. Requires *Scalability*, *Reliability*, and *Security*.

Classification:

- Pure **P2P**: Peers act as equals; No central managing server; No central router
- Hybrid P2P: Has a central server that keeps info and responds requests



Key aspect of a P2P system is a set of algorithms for the placement and subsequent retrieval of information objects.

Examples of P2P systems:

- Music exchange services: Napster, ...
- File sharing: Freenet, Gnutella, Bit Torrent, ...
- *Overlay Networks* middleware running on top of the Internet (routing not necessarily specified by IP address): Pastry, Tapestry, ...

Overlay Networks

Before overlay networks, P2P uses **Flooding**-Style Networks (using Time-To-Live limits, **TTL**):

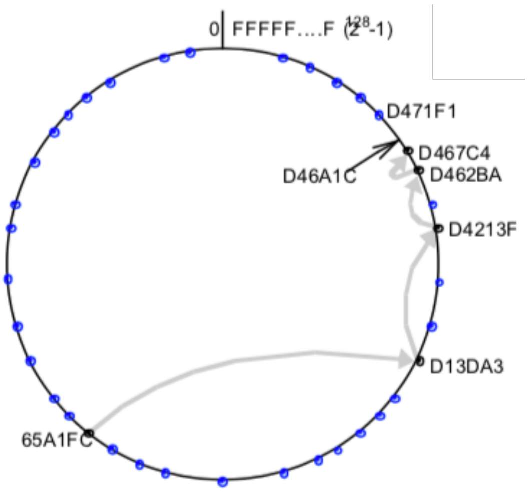
1. Send query to all known neighbors;
2. Each neighbor checks to see whether they can reply (by matching keys);
3. If match, then reply; If not, ++hop_count, and forward to its neighbors;
4. If hop_count passed the TTL limit, stops forwarding.

[**Distributed Hashing**] **Consistent Hashing** and its classic implementation *Chord*: a *Ring*-based Overlay Network, nodes are linked in a circle, each having only two neighbors. A middleware layer takes the responsibility for routing request from any client to a host that holds the object to which the request is addressed (specified by a GUID). Nodes may:

- Join and leave the network
- Create / remove objects

A request on an overlay network is guaranteed to be handled.

A *Routing Table* can help reduce the number of hops of a message (by pushing a message to a node with longer common GUID prefix, instead of just searching in its leaf set, i.e., doing binary search. [READ HERE](#)).



Time & Global State

"Distributed Systems: Concepts & Design", Chapter 14

Distributed Timing

Physical computer clocks (meaning the clock signal for electric circuits) are circuit oscillations at a well-defined frequency. Every designed number of oscillations trigger a timer interrupt. In a distribuetd system, timing is mostly *ambiguous*. Timing in different machines cannot guarantee to run at the same frequency.

Algorithms for keeping *absolute* times for distributed systems:

- Centralized timing for intranets:
  - **Christian's Algorithm**: For a cluster with one time server that has access to WWV time (UTC)
  - **Berkeley UNIX Algorithm**: For a cluster with no WWV access, time server polls other machines local times, computes an average, and tells all machines to adjust according to that
- Decentrailzied timing for intranets
- Internet time synchronization

Algorithms for keeping a *logical* time (not necessarily correct, but all machines agree):

- *Lamport Timestamps*: All machines agree on events' occurrence order, if they have message passing dependency
  - If A is the event of machine 1 sends a message, and B is the event of machine 2 receives it, we say A happens before B ($A \to B$)
  - If A and B happen in different machines and do not have dependency, we do not care about who happens first; we say they are cocurrent ($A \parallel B$)
  - **Lamport's Clock Algorithm** ensures if $A \to B$, then all machines agree to $C(A) < C(B)$. [READ HERE](#) and refer to "Lec11.pdf".

Global State

Global state of a cluster can be recorded by a **Snapshot**, reflecting a state in which the distributed system might have been.

- $H(i)$ is the History of node i, which is a vector of events $< e_{i,0}, e_{i,1}, \dots >$
- Global history H is the union of all individual histories $H(0) \cup H(1) \cup \dots$
- A **Cut** of is a subset of the global history. A cut is *consistent* iff for every event it contains, it also contains all events happened before that event
    - Can contain a msg send event without the corresponding receive event
    - CANNOT contain a receive event without the corresponding send event

Chandy-Lamport Algorithm: for recording a consistent global state. READ HERE and refer to "Lec12.pdf".

Coordination & Agreement

"Distributed Systems: Concepts & Design", Chapter 15

Distributed Mutual Exclusion

Without shared variable, how can we achieve **Mutual Exclusion** only by message passing? Mutual exclusion requirements: 1. safety (mutual exclusiveness), 2. liveness (bounded waiting), 3. $\rightarrow$ ordering.

Evaluating a distribteud mutual exclusion algorithm:

- *Bandwidth*: number of messages sent in each `enter()` or `exit()` operation
- *Client delay*: waiting time at each `enter()` or `exit()`
- *Synchronization delay*: time between `exit()` and next process's `enter()`

Different algorithms to handle this:

1. **Central Server Algorithm**: use a central *token* server which queues entry requests, dequeue one request and gives the token to that client
    - Can violate "$\rightarrow$ ordering"
    - `enter()` takes 2 messages (request + receive token), `exit()` takes 1 message (return token)
    - Client delay depends on size of queue
    - Synchronization delay is 2 messages (return token + next one receives token)
2. **Ring-Based Algorithm**: peers arranged in a ring, enter when token is received and exit by passing on the token to next neighbor
    - Can violate "$\rightarrow$ ordering"
    - `enter()` has to wait for token to come, `exit()` takes 1 message (pass on token)
    - Client delay & Synchronization delay similar to entry
3. **Ricart-Agrawala Algorithm**: multicasting + Lamport timestamping. READ HERE and refer to "Lec13.pdf".

Election Algorithms

How can we elect a unique process out of all distributed nodes? Election requirements: 1. safety (result either none or a process with largest identifier), 2. liveness (each process either picks the result or crashes).

Different algorithms to handle this:

1. **Ring-Based Election** (Chang-Roberts): READ HERE and refer to "Lec14.pdf".
2. **Bully Algorithm** (Garcia-Molina): can handle process failures. READ HERE and refer to "Lec14.pdf".

Multicast Algorithms

Multicast is a sending scheme where we send a message to a group of nodes, and the message gets replicated only at path divergence.

I really don't understand the meaning and necessity of *reliable* / *ordered* multicast, so ~~omitted~~ here.

Consensus Algorithms

How do a group of process agree to the same single value? That is called a **Consensus**. Consensus algorithms have to meet the following requirements:

- *Agreement*: Decided value of all correct processes is the same.
- *Integrity*: If all correct processes proposed the same value, then they must decide on that value.

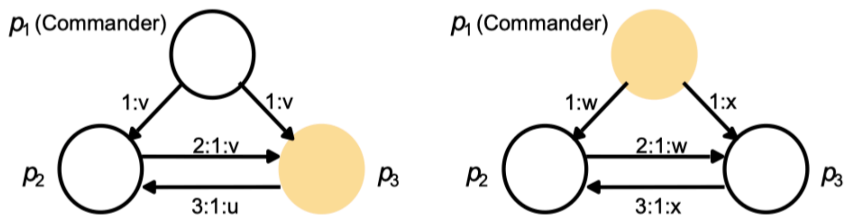
BASE CASE - When processes cannot fail & communication cannot fail, the problem is easy: each processor broadcast its proposed value $\rightarrow$ waits until collected all N values $\rightarrow$ choose the majority one.

HARDER - When communication is reliable (no uninteded message modifications) but processes can fail:

1. Two-Phase Commit (**2PC**), Three-Phase Commit (**3PC**): not 100% guaranteed, READ HERE.
2. **Paxos** $\rightarrow$ Multi-Paxos $\rightarrow$ **Raft**: higher availability. READ HERE & my blog.

EVEN HARDER - When message content can be wrong (modified / erroneous), i.e., **Byzantine Generals Problem**: Some of the processes can be faulty. If it is the commander (i.e., it is broadcasting), it may give out different values to different processes; If it is a general, it may relay a wrong value to others. (Message faults causesd by unreliable communication links also count.)

- No solution when total number of processes $N \leq 3f$, where f is the number of faulty processes. A correct processor cannot tell who is falty:



Faulty processes are shown coloured

- For more information, read Lamport's Paper & READ HERE, and refer to "Lec16.pdf".
- It is important for any modern real-time distributed system to be highly *Byzantine Fault Tolerant* (**BFT**); but, it's impossible at least for now to ensure 100% correctness, given that even the electric circuits can fail.

Transactions Interface

"Distributed Systems: Concepts & Design", Chapter 16

Definition of Transactions

A **Transaction** (事务) is *a sequence of operations* on a server that satisfy the **"ACID" rules** of databases:

- **Atomicity**: a transaction is either finished completely or not done at all; cannot end up in the middle (o.w., do *rollback*)

  - For example, a failed file write may: a) write nothing at all, b) write a wrong value but checksums are used so that readers will detect that.
    - These are omission failures, and
    - The system must provide a failure model and corresponding recovery techniques
  - But, this file write CANNOT write to the wrong block. That will be an arbitrary failure.
- **Consistency**: a transaction must bring the whole system from one valid state to another ("valid" is defined by a set of preset constraints, like some object's value must be ≥ 0); this is not the same term as in other contexts
- **Isolation**: allow concorrent transactions to act on the same object, w/o causing inatomicity / inconsistency
- **Durability**: whenever a transaction is done, its modification to objects are permanent and will not be lost even at system crashes, so that a crashed system can recover from the permanent storage

Providing a transaction *interface* to user applications can make the underneath concorrency control transparent to them. Users can simplify their application code logic by triggering transactions w/o worrying about violating the 4 rules.

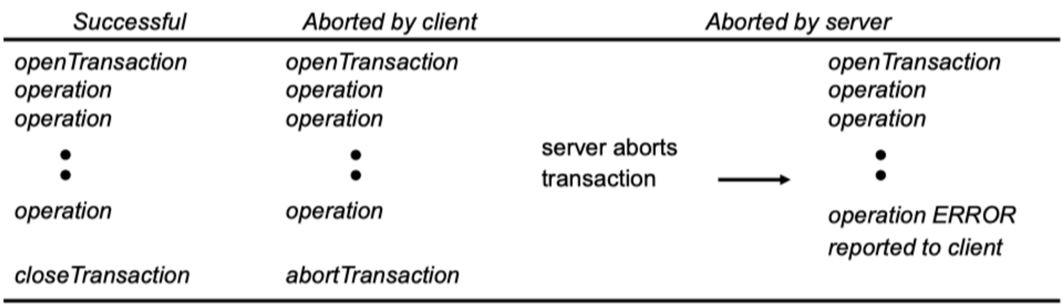
Example on a banking account:

```
# Operations:
acc.withdraw(n)
acc.deposit(n)

# A Sample Transaction T:
def transaction_T(a, b, c):
    a.withdraw(100)
    b.deposit(100)    # A transfers $100 to B.
    c.withdraw(200)
    d.withdraw(200)   # C transfers $200 to D.
```

Life History of a Transaction

The life history of a transaction must be one of the following situations:



- Operations are first done on volatile memory
- `close` = `commit`: flush the corresponding objects in volatile meory downto permanent storage
- `abort`: discard the changes in volatile memory

Concurrency Control

First assumption: an **operation** itself must be atomic (called *synchronized operations*). Without this property, we cannot discuss the concurrency issues of transactions.

Problems that might happen w/o proper concurrency control on transactions:

1. **Lost update**: two transactions both read the old value

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

2. **Inconsistent retrieval**: a transaction reads an intermediate value of another transaction

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	:	

Correct concurrency of transactions should be **serial equivalent**: the result it the same as if they are processed serially. A *serially equivalent interleaving* is an interleaving of operations that makes the combined effect serial equivalent. Same effect here means:

- Read operations return the same value
- Object states are the same in the end

Operations that can conflict:

- `read(i)` and `write(i, x)` on the same object `i`
- `write(i, x)` and `write(i, y)` on the same object `i`

These operations MUST happen in the original order of transactions. Tools like DAG dependency graphs can help.

Recoverability from Aborts

If a transaction *aborts*, the server must make sure that other transactions do not see any of its effects.

Problems that might happen w/o proper recoverability from aborts:

1. **Dirty read**: a transaction reads a modified value written by an aborted transaction

Transaction *T*:		**Transaction *U*:**	
a.getBalance() *a.setBalance(balance + 10)*		*a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	$100	•*U* reads *A*'s balance (which was set by *T*) and then commits	
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
T subsequently aborts.		*commit transaction*	
abort transaction			

2. **Premature write**: in a system that uses *before images*, it records the snapshot before a transaction and restore to the before image if that transaction aborts; if another transaction's write is done before the abortion, that write will get lost after restoration

Methods to ensure recoverability:

- *Delayed commit*: a commit is **delayed** until the commitment / abortion of other concurrent transactions are observed
    - If T aborts then U must also abort
    - May cause potential *cascading aborts*
- *Delayed read*: any read operation on `i` is delayed until other concurrent transactions who will apply writes on `i` are committed / aborted
    - This ensures only reading objects written by committed transactions, thus avoids cascading aborts
- *Delayed write*: any write operation on `i` is delayed until other concurrent transactions who will apply writes on `i` are committed / aborted
    - This prevents premature writes with before images
    - A system is said to implement **strict execution** if both read & write are delayed

Nested Transactions

Nested Transactions are transactions composed of subtransactions recursively. To a *parent*, a subtransaction is atomic, while transactions at the same level can actually run concurrently just like over *flat* transactions.

Advantages:

- Additional parallelism
- More robust: a subtransaction failure can be properly handled by its parent, w/o restarting the whole transanction all over

> Strict execution can be implemented using *strict two-phase locking*. Locks, *deadlocks* and *starvations* in transactions are ~~omitted~~ here. Already covered in details in the OS lectures.
>
> **Distributed Transactions**, where a transaction accesses objects across multiple servers, are also ~~omitted~~ here. Refer to Chapter 17 of the book and "Lec{20,21}.pdf".

Replications

"Distributed Systems: Concepts & Design", Chapter 18

Replication = Maintencance of copies of data at multiple computers. Can provide:

- Performance enhancement (rare)
- Fault tolerance
- High availability

Replication Transparency

Replication of data should be **transparent** to users, which means that they appear as one single logical object, and different online users (disregard those disconnected copies) should see a *consistent* value.

Each *logical* object is implemented by a collection of *physial* copies, called **replicas**. All replicas on one device are managed by a *replica manager* (RM). The *frontend* (FE) talks to RMs and provides transapency to clients. Five phases in performing a request:

1. Issue request through FE
2. Coordination among RMs
    - Whether to apply / not
    - Its ordering relative to other requests (FIFO / Causal / Total-ordering)
3. Execution
4. Agreement: RMs reach a consensus on the effect of the request
5. Response back to FE

Often implemented using dynamic **Process Groups**.

Fault-Tolerant Services

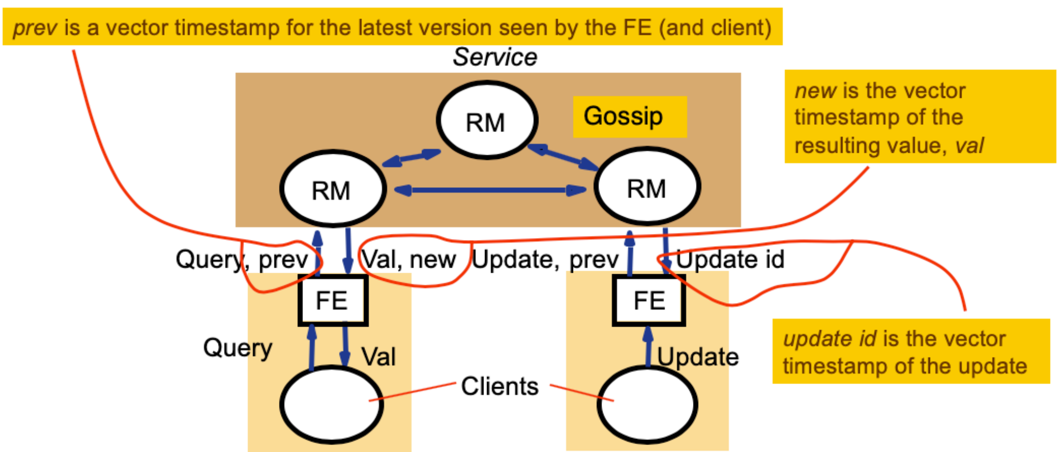
Replciations enable **Fault-Tolerant Services**, where a service / data is still available even if up to f processes failed. Different models to do that:

- Passive (*primary-backup*) model: at any time there's a primary RM and others are backups
    - Needs $f+1$ RMs to achieve f-degree fault-tolerance
    - CANNOT handle Byzantine failures
- Active model: RMs all play the same role, and communicate with each other and compare the replicas it receives
    - Can use $2f+1$ RMs to handle up to f Byzantine failures
    - May have better performance
    - Needs a good consensus algorithm

Highly Available Services

High Availability is a different goal from fault tolerance. We aim to give clients quick responses for as much of the time as possible, even if some results do not conform sequential consistency. (E.g., a disconnected client may accept some inconsistency and will fix that later.) Updates between RMs are propagated *lazily*, i.e., they have less agreement to enable shorter response time.

The **Gossip Architecture** is a way to provide high availability:



- Two types of operations: *Query* (read-only) and *Update* (will modify)

- FE sends operations to any chosen available RM, guaranteeing:
  - Each client gets consistent values *over time*, using vector timestamps
  - RMs eventually receive all updates, but they agree only by lazy gossips, so consistency is relaxed. Thus, a client may observe stale data at certain timepoint

Security

"Distributed Systems: Concepts & Design", Chapter 11

Interesting names used in security protocols: Alice, Bob, Carol, Dave, Eve, Mallory, Sara ;)

This section should be found in Cryptology and Computer Security, so ~~omitted~~ here.