DSM: Distributed Systems

Important Notes About These Notes

These notes were written by me, Mark Ormesher, during my revision for Computer Science exams at King's College London. I am publishing them here to help my classmates and other students in the years below me. To the best of my knowledge they are correct.

These notes are **not endorsed** by King's College London, the module lecturers, or any member of College staff. These notes are **not checked** by any qualified individual, and I offer no guarantee as to how accurate or complete these notes are. They are offered for free, as-is, and I accept no responsibility for how you choose to use them.

These notes were relevant for my year (2016/17) but **the content for your course may be different**. Check your lecture slides, syllabi, tutorial guides, etc.

These notes were produced for my own personal use (it's part of how I study and revise). That means that some annotations may by irrelevant to you, and **some topics might be skipped** entirely.

Feel free to **share** these notes, however please only share a link to the repo (see the link below), not individual files.

Notes are originally from **https://github.com/markormesher/CS-Notes**. All original work is and shall remain the copyright-protected work of Mark Ormesher. Any excerpts of other works, if present, are considered to be protected under a policy of fair use.

Contents

Characteristics of Distributed Systems

- Processes **share state** only via the exchange of **messages**.
- Processes commonly exist on **multiple hosts**, which are linked by a **network**.
  - The network is assumed to suffer from **latency** and occasional **message loss**.
- Processes use resources from each other, alert other hosts to state changes, etc.
- Processes **operate independently**.
  - There is no central authority, although some processes may serve as an authority on some tasks.
- Each host has an **independent clock**.
- Hosts **fail independently** of each other.

Middleware

Middleware is a layer of software designed to **hide the heterogeneity of hosts** and provide a **common, convenient programming model** to application developers. It can be used to alleviate the burdens a developer can face when building software to interact with numerous services, components, etc.

Different middleware frameworks emphasise different core requirements of distributed software, based on the set of concerns the middleware is designed to address.

Design Considerations

The design of a distributed system can be evaluated on various characteristics, each of which may be more or less important depending on the intended purpose of the system:

- Openness
- Concurrency
- **Scalability**
- **Fault tolerance**
- **Transparency**
- Security
- Management
- Performance
- Heterogeneity
- etc.

Bold topics will be focused on in particular in this module.

Openness

Openness describes the extent to which OS, network and service interfaces are based on **published specifications** and standards. Openness allows **systems from different vendors** to interact; without it, users are restricted to build around one particular vendor's products.

Note that **too many open standards** can be almost as bad as closed vendor standards, as users must then deal with all of them.

Concurrency

Any distributed system is by definition **multi-process**, and therefore concurrency must be considered. Additional, **multiple clients** may access a service, and **duplicated resources** may provide one functionality.

Issues such as **locking** and **synchronisation** must be addressed.

Scalability

An advantage of distributed systems is that they can **start small and gradually grow** with additional components being added as needed. For this to work, the system must handle **duplicated resources**.

Scalability is a measure of the extent to which a system facilitates scaling in this manner. For effective scaling, the following characteristics must be avoided:

- Single 'master' nodes or services, which can become overwhelmed and/or present single points of failure.
- Short, fixed names, which make future changes and expansions problematic.
- Performance bottlenecks, which can limit the effectiveness of scaling operations.

Scalability and State

Some resources have a notion of **state**, which is stored data that may alter future behaviour of the resource. Scaling by duplicating resources can duplicate state, so efforts are needed to ensure that **changes propagate** correctly and state is **kept consistent**.

Fault Tolerance

More components leads to more component failures, so this must be handled by distributed systems. Fault tolerance is a measure of **how well a system deals with failures**, usually measure by two characteristics:

- **Fault isolation**: failure of one component shouldn't affect operation of other components.
- **Fault masking**: the service should not be interrupted (or it should be restored) when a fault occurs.

Fault tolerance is a mixture of **hardware redundancy** and **software recovery**.

Measures of Failure Rate

- **Mean Time Before Failure (MTBF)**

  - Average time between failures.
  - Assuming a random distribution of failures the probability of a failure in a period of time can be calculated.

- **t-Fault Tolerance**
  - There must be more than t failed components before the service fails (we can suffer t failures before losing the service).
  - This can be more useful when designing the system.

Failure Modes

- **Failstop**: the process halts and remains halted. The fact that it is halted *can* be detected by or communicated to other processes.
- **Crash**: the same as failstop, but *cannot* be detected by other processes.
- **Crash+link**: a link fails for a period, losing some messages that it was carrying.
- **Receive omission**: a process fails to read some of the messages sent to it.
- **Send omission**: a process fails to write some of the messages required by the protocol.
- **General omission**: receive and/or send omission.
- **Byzantine failure**: the process behaves in a manner not permitted by the protocol.

Failstop failures are easiest to handle, because any dependants can use another service and maintenance software can take restorative actions.

Crashes can be dealt with like failstops in synchronous systems with a timeout period, but in an asynchronous system we can never tell the difference between a crashed system and a very slow one.

Transparency

Transparency is all about making the service itself **independent** from how it is provided. In a transparent system, the client does not need any specific details to use its services.

From a development point of view, this means there are some parameters that do not need to be passed with API calls.

There are different types of transparency:

- **Access Transparency**: the methods to access a resource do not vary according to the means of access (Java method call, SOAP request, etc.).
- **Location Transparency**: accessing a resource does not require its location to be given.
- **Concurrency Transparency**: access to a resource is not affected by the number of concurrent clients.

- **Replication Transparency**: methods to access a resource do not vary according to how many copies of the resource exist.
- **Failure Transparency**: services remain available regardless of partial resource failures.
- **Migration Transparency**: access to a resource does not change if the resource is moved between hosts.
- **Performance Transparency**: changes in resource performance do not alter the methods by which the resource is accessed.
- **Scaling Transparency**: scaling of a resource does not alter the methods by which the resource is accessed.

Security

Two main notions of security must be considered:

- Preventing clients from deliberately doing things they should not be able to do.
- Preventing actions which would accidentally hinder the correct functioning of the system.

Both of these are handled together, because we are concerned with preventing an action, not the intent behind it. Security falls into two main categories:

- **Identity Security**
  - **Authentication**: the client knows that the service is who it says it is, and vice versa.
  - **Confidentiality**: clients may be limited in terms of the information they can access.
  - **Access control**: clients may be limited in terms of the resources they can access.
- **Consistency Security**
  - **Integrity**: actions of a client cannot affect the overall integrity of a resource.
  - **Non-repudiation**: clients cannot deny that a communication or action took place.

Management

Management is the **supervision and control** of a system to make sure it satisfies requirements of its stakeholders.

A system should be divided into **domains**, each of which should have a system manager. Domains may be defined according to resource types, particular work groups, locations, etc.

There are some requirements for management of a distributed system:

- **Remote access**: changing the configuration of the system and monitoring its behaviour must be something that can be done remotely.

- **Programmatic interface**: configuration and management must be possible via a program interface.
- **Relevance**: only the necessary information should be presented.
- **Uniformity**: different resources should have similar styles of control.

Performance

There are several measures of performance:

- **Responsiveness**: operations should return results as quickly as possible.
- **Throughput**: as much processing as possible should occur throughout the system.
- **Load balancing**: work should be shared between hosts.

Heterogeneity

The uses and hosts of a service will have variations as the system grows, and this must be accommodated.

Note that heterogeneity and openness are not the same: the former is about the variation of resources; the latter is about allowing new hosts and resources to join the systems. Openness tends to allow for heterogeneity.

Other Considerations

- **Reliability**: the system should always be in working order.
- **Adaptability**: the system should be modifiable to allow for changes in use cases and technology.
- **Time-criticaliy**: when operations have time constraints, the system should meet these.

Reliable Broadcasts

Several services within a distributed system can require a mechanism to deliver a message to all processes within a set: a **multicast** or **broadcast**. To maintain **consistency**, it is important that every broadcast ultimately has one of these outcomes:

- **All** processes receive the message.
- **No** processes receive the message.

This is called **agreement**, and it is important for several reasons:

- Messages may not reach every node at the same time.
- Messages may be order-sensitive.
- Messaging can be affected by node failure.

Note: it is assumed that when a process broadcasts a message it delivers it to itself immediately, rather than sending it out over the network and waiting for it to (maybe) come back.

The primitive actions `send` and `receive` cannot guarantee agreement, so an explicit **reliable broadcast strategy and/or middleware** is needed.

Reliable broadcast middleware has the primitives:

- `broadcast(m)` - send message to all processes
- `deliver(m)` - indicate arrival of a message

Properties

- **Agreement**: if any process delivers m, then all *correct* processes eventually deliver m.
- **Validity**: if a *correct* process executes `broadcast(m)` then all *correct* processes will eventually deliver m.
- **Integrity**: `deliver(m)` occurs at most once at each *correct* process, and only if `broadcast(m)` occurred in some process.

A *correct* process is one that has not failed.

Uniform Broadcast

In **uniform** broadcasts, the agreement and integrity properties **include failed processes**, which is important because the system should allow failed processes to recover.

- **Uniform agreement**: if *any* process delivers m, all *correct* processes will eventually deliver m.

- **Uniform integrity**: `deliver(m)` occurs at most once at each process, and only if `broadcast(m)` occurred in some process.

Message Ordering

Because of latency, reliably broadcast messages may arrive in **any order** and not within any particular period of time. Some applications require different levels of ordering restriction.

First In, First Out (FIFO) Ordering

This applies to messages **from a particular process**. A message $n + 1$ may only be delivered after message n has been delivered.

(Potential) Causal Ordering

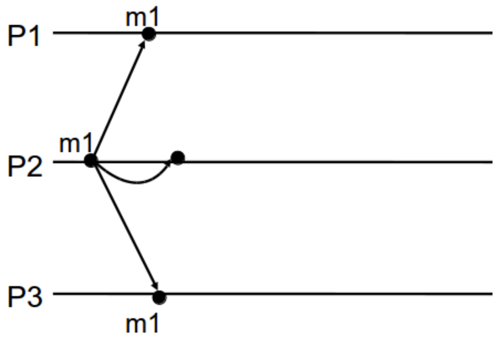
If a message B is broadcast from P1 after message A is delivered at P1, then at **all other processes** B must be delivered after A is delivered. The idea here is that receiving A may have caused B to be sent, so it's important that A must be delivered before B everywhere else.

Total Ordering/Atomic Broadcast

Delivery order is the **same at all processes**. The transmission of messages forms **atomic broadcasting**, which requires a guarantee that a message is delivered to all processes or to none at all, and all messages are delivered in the same order to all processes.

Interaction Models

We use interaction models to show the **broadcast and delivery of messages over time**. One line per process, one dot per event, and one arrow per message sent between two processes.



Note: messages delivered to the process that broadcast them are not usually shown.

Implementing Reliable Broadcasts

Implementation of reliable broadcasting is split between the **application** and **middleware**. The application can invoke a `broadcast(m)` function on the middleware, and the middleware can invoke a `deliver(m)` function on the application. The middleware builds its functionality from the `send` and `receive` primitives offered by lower levels of the system.

Note that middleware **does not** need to delivery messages it receive immediately.

Augmenting Messages

Extra information must be attached to each message to allow processes to **coordinate and reason with** the messages. An augmented message will contain:

- The message content itself (m).
- The sequence number of the message (s).
- The identifier of the transmitter (i).

The sequence number is local to the transmitter, so the sequence number *and* the transmitter ID form a **system-wide unique ID** for the message.

The primitive messaging functions are similarly augmented:

- `send(m, s, i) to p`
- `receive(m, s, i).`

Basic Algorithm

A very basic algorithm achieves reliability by re-broadcasting every message that is received. As viewed at each host's instance of the middleware, i is the host's ID, s is its sequence number (initially zero) and $recorded$ is an initially-empty set of messages that have been seen.

```
fun broadcast(m):
    for each processes p, do:
        send (m, s, i) to p
    s++
```

```
fun receive(m, s, i):
    if (recorded does not contain (m, s, i)):
        deliver(m)
        add (m, s, i) to recorded
        for each process p, do:
            send (m, s, i) to p
```

- **Achieving validity**: even if some links break down in the network, every path to a process will eventually be explored by some chain of re-broadcast. Note that this will not work in the case of a completely disconnected node, but such a node cannot be considered to be part of the system anyway.
- **Achieving agreement**: once a message is delivered at one process, it is re-transmitted to all other processes until every process has seen the delivery. |
  - This does not achieve **uniform agreement**, because a process may deliver and then fail before re-sending.
- **Achieving integrity**: the set of recorded messages prevents any message from being delivered more than once.

Uniform Algorithm

This algorithm differs by only delivering a message that is received after it has been successfully re-sent to all other processes. All other notation is the same.

```
fun receive(m, s, i):
    if (recorded does not contain (m, s, i)):
        add (m, s, i) to recorded
        for each process p, do:
            send (m, s, i) to p
        deliver(m)
```

With the `deliver(m)` line moved to *after* the re-sending block, delivery now only happens after the message has been sent to all processes. Because of validity, `send` guarantees an eventual `deliver` at all correct processes. Therefore, either all correct processes delivery the message or none do.

A sending process can **fail and recover** by checking the $recorded$ array for any non-delivered messages.

Uniform FIFO Algorithm

The previous algorithm can be extended by adding an array $next$ to track the next sequence number to deliver for each transmitter process. When a message is received out of order it is recorded but **not delivered**. When a message is received with the sequence number $next[i]$ for a transmitter, that message is delivered followed by any contiguous messages queued up for that transmitter.

```
fun receive(m, s, i):
    add (m, s, i) to recorded
    while (recorded contains (m2, next[i], i) for some m2:
        for each process p, do:
            send (m2, next[i], i) to p
        deliver(m2)
    next[i]++
```

Uniform Causal Algorithm

This algorithm adds an array of messages received since the last broadcast: $causes$.

```
fun broadcast(m):
    mc = concat(causes, m)
    causes = []
    for each processes p, do:
        send (mc, s, i) to p
    s++
```

```
fun receive(m, s, i):
    if (m is a concat of (mc, sc, ic) and m2):
        receive(mc, sc, ic)
        receive(m2, s, i)
    else:
        add (m, s, i) to causes

        // copy of FIFO:
        add (m, s, i) to recorded
        while (recorded contains (m2, next[i], i) for some m2:
            for each process p, do:
                send (m2, next[i], i) to p
            deliver(m2)
        next[i]++
```

It is okay to clear $causes$ on each broadcast, because the FIFO ordering will include all previous potential causes by transitivity.

Logical Clocks

The previous approaches to ordered, reliable broadcasts (*see more: Reliable Broadcasts, page 11*) were incomplete and/or very expensive (because of concatenation and repeats).

An obvious solution would be to **timestamp** each message, but that doesn't work in practise because each host in a distributed system has its own **independent clock** (*see more: Characteristics of Distributed Systems, page 6*).

A logical clock has **one 'tick' per event** such that larger values indicate later events in time. Ticks do not happen if no events happen to trigger them. Distributed logical clocks can be kept up to date by adding meta data to broadcast messages.

The clock value at process P_i is denoted C_i.

Synchronising Clocks

- When process P_i **broadcasts** a message:
    - Increment the clock, $C_i = C_i + 1$, because `broadcast` is an event.
    - Attach C_i as message meta data, sending (m, i, C_i).
- When process P_i **delivers** a message (m, j, C_j):
    - Increment the clock, $C_i = C_i + 1$, because `deliver` is an event.
    - Keep the maximum of the local clock value and one more than the received message's clock value: $C_i = max\{C_i, C_j + 1\}$.

Ordering Properties

Assuming delivery is done in order of clock values:

- **FIFO**: satisfied, because every message sent from a process has a **higher clock** value than all messages it has **previously sent**.
- **Causal**: satisfied, because every message sent from a process has a **higher clock** value than all messages it has **previously received**.
- **Total**: not satisfied yet, because multiple processes may broadcast a message with the same clock value. Total ordering can be enforced by using the sender ID to break ties on messages.

Causal ordering is normally adequate for most applications.

Clocks, Causation and Inconsistency

With logical clocks, the system knows that if $m1$ was delivered before $m2$ was broadcast (i.e. there is a potential causal connection) then $m1$ is guaranteed to have a lower clock value than

$m2$. The reverse is not true: if $m1$ has a lower clock value than $m2$ it doesn't guarantee that $m1$ has a causal connection with $m2$.

This is a problem:

- If the middleware knows that $m1$ and $m2$ are **causally connected** then **consistency can be maintained** by delivering $m1$ before $m2$.
- If the middleware knows that $m1$ and $m2$ are **not causally connected** it can inform the application layer that an **inconsistency** must be resolved.

The solution? *See more: Vector Clocks, page 18*.

Vector Clocks

With vector clocks each process P_i keeps **an array of logical clock values** - one for each process in the system.

Notation:

- C_i is the vector held by process P_i.
- $C_i[i]$ is P_i's own clock value.
- $C_i[j]$ is the process P_i's knowledge of P_j's clock value.

Algorithm

- Initially, $C_i[j] = 0$ for all j, because no process has any knowledge of the clock value of other processes.
- As before, $C_i[i]$ is incremented by P_i on each event.
- When broadcasting, P_i includes the whole vector C_i as message meta data.
- When P_i receives a clock vector V in a message it updates its own vector such that $C_i[j] = max\{C_i[v], V[j]\}$.

Comparing Vector Clocks

Given two vector clocks V_A and V_B:

- $V_A = V_B$ if and only if $V_A[i] = V_B[i]$ for all i.
- $V_A \leq V_B$ if and only if $V_A[i] \leq V_B[i]$ for all i.
- $V_A < V_B$ if and only if $V_A \leq V_B$ but $V_A \neq V_B$.
  - i.e. $V_A < V_B$ if at least one value in V_A is less than its counterpart in V_B and no value in V_A is higher than in V_B.

With these comparison rules we can determine message delivery ordering.

Identifying and Handling Inconsistency

Assuming two vector clocks V_A and V_B are attached to messages M_A and M_B...

- If $V_A < V_B$ then M_A **caused** M_B.
- If neither $V_A < V_B$ nor $V_B < V_A$ then the two messages were **concurrent and not causal**.

If two non-causal messages create an **inconsistency** this can now be identified and resolved.

Resolving inconsistencies is an **application-specific** process, so the inconsistent messages are handed to the application layer which will execute an appropriate **merge** or **reconciliation** routine. The application will then broadcast a fixed state to all hosts, which will be causally dependent on all of the conflicting messages.

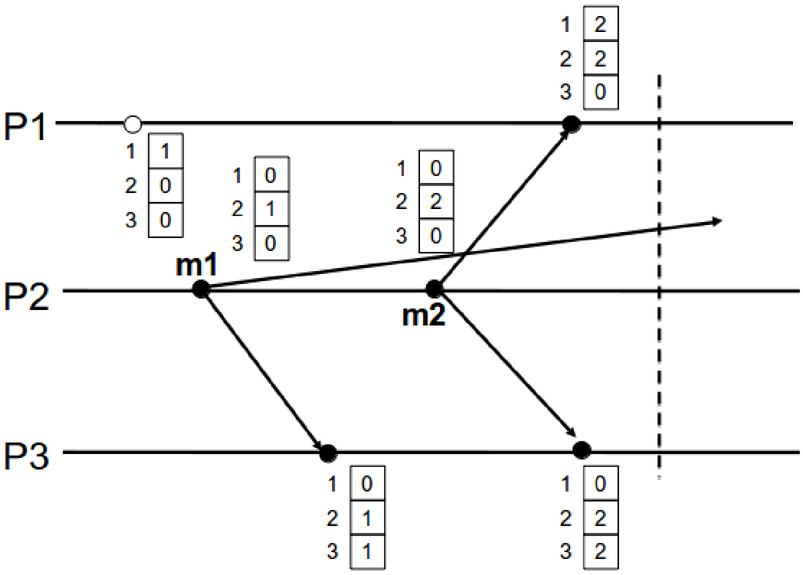
Message Stability

A message M is **stable** at P_i if P_i knows that it is not waiting to receive any other message that should be delivered before M (i.e. M is now ready to deliver).

This is **easier said than done**: with logical/vector clocks messages can be delivered in order, but there is no way to know if messages exist with lower clock values that haven't arrived yet. This is because clock ticks may be caused by events other than message broadcasts.

Vector clocks can determine stability in **some** cases: if P_i receives a message M from P_j with vector clock V, but $C_i[k] < V[k]$ for some other process P_k, then P_i knows that P_j has seen some message M' from P_k that P_i hasn't seen yet and that M was caused by M', so therefore M is not yet stable.

This does not work in every case though. In the diagram below, P_1 needs some way to know that M_2 is not stable:



This can be resolved by adding a small amount of meta data to each message:

- When sending a message, a process includes **its own clock value from the last message** it broadcast.
- When receiving a message, a process knows which message it should have already received and transitively knows whether there are any messages to wait for before delivering the one just received.

  - Hosts can essentially build up a linked-list of received messages for each sender, treating the ‘root’ part of the list as stable and considering any non-contiguous messages as unstable until the gaps are filled in.

Replica Management

Types of Data

Different distribution strategies are more suited for different types of data. Data can be classified by its **type** and **longevity**.

- Types of data:
  - **Content**: application data that is visible to the user, or is analysed and transformed into data for the end user.
  - **User information**: information about users, such as billing details, authentication data, etc.
  - **Application**: executable code, configurations, logs, etc.
- Longevity of data:
  - **Persistent**: data that characterises and shapes the service over time.
  - **Working**: data that is used internally; usually possible to recover following a loss, but needs to be low-latency.

Distribution of Data

Running a service in a distributed manner means replicating hardware and/or **distributing data**. There are different ways to distribute data:

- **Fragmentation**: data is *split* between hosts.
- **Migration**: data *moves* between hosts over time.
- **Duplication**: multiple *copies* of data are maintained at different hosts.
  - **Replication**: copies are maintained in a proactive manner.
  - **Caching**: copies are maintained in a reactive manner.

Duplication is more complex than fragmentation, because mechanisms for **consistency** must be implemented.

Database Partitioning

There are two main forms of database partitioning: **vertical** and **horizontal**.

- **Vertical partitioning**: different columns are stored on different hosts.
  - *Where* a query is handled depends on the *type* of data being requested.
- **Horizontal partitioning**: different rows are stored on different hosts.
  - *Where* a query is handled depends on the *entity* being requested.

Note: this is not to be confused with vertical and horizontal scaling!

Sharding

Sharding involves **horizontal partitioning** across hosts plus **replication** of related data, so that one host can resolve entire queries. It essentially breaks one large database into **multiple small databases** that can operate somewhat independently, sharing the load of the application.

Sharding is **effective for read-only** operations or **temporary transaction** data, because in these cases there is no need to maintain consistency between shards.

For persistent, writable data the system must ensure that a user is always **served by the same shard** for different transactions, or that **shards are synchronised** between transactions. This means that sharding is less effective for persistent content that is shared between clients, because synchronisation between transactions is needed.

Why Replicate Data?

From a client's point of view, every replica is identical; they should not need to know which replica they are working with. So why have replicas?

- **Fault tolerance and availability**: if one replica fails, other replicas can still serve requests.
- **Load balancing**: if multiple replicas can handle a given query then the load can be spread across them, improving response times for each client.
  - This can be taken further with **location-aware replication**: if clients requests can be routed to replicas that are physically close to them, latency will be reduced. This is the idea employed by content delivery networks (CDNs).

Replication Issues

Replication issues arise from the concept of **state**: stateless services have very few (if any) replication issues. For example, replication in a CDN just requires more compatible servers to run the service.

Stateful applications have more issues, because **state must be kept consistent** between replicas. Updates must be broadcast to all replicas, and some updates may be received in different orders and/or after delays.

Replication Objectives

- **Transparency**: the action of issuing a request should be the same, regardless of how many replicas are implemented or functioning. *See more: Transparency, page 8*.
- **Consistency**: the result of a request should be the same, regardless of which replica served it.

Update Models

- **Synchronous updates**: access to each replica will give identical results.
- **Asynchronous updates**: each record is identical, but the ordering and presence of records may be different between replicas.
- **Causal updates**: similar to asynchronous, but with partial ordering imposed when a potential causal relationship exists.

The **synchronous** update model is most desirable from a user's point of view and essential where **data consistency** is paramount. However, it may be very expensive to implement.

For some applications, weaker update models may be tolerated. This would allow **communication with replicas and propagation of updates to be disconnected**: responses from replicas can be immediate, and changes can be propagated when suitable.

General Replication Model

- Clients communicate with the service via **front-ends**.
- Services are implemented by a set of **replica managers** (RMs).
- Requests are **reads** if they do not change the state of the replica, or **writes/updates** if they do.

Replica Manager (RM) Connectivity

- Each RM fails independently of others.
- Each pair of RMs has its own link, which fails independently of other links.

Client Requests

Requests to a service from a client have an **ordering**. The notation R_{ci} will be used to denote a request with the **sequence** number i from the **client** c. Clients are enumerated as $A, B, C, ...$ and sequence numbers follow $1, 2, 3, ...$

E.g. R_{B3} is the 3rd request from client B.

Replica State Changes

The **state** of a resource replica at any point in time may be represented as s.

For an RM, $R_{A1}(s)$ indicates the state reached after processing request R_{A1}, starting from state s.

Operations can be nested for successive changes: $R_{A2}(R_{A1}(s))$ represents the state reached when R_{A1} is applied to s, followed by R_{A2}.

Pending and Stable Requests

An RM can be in one of the following states, with regards to a request R_{ai}:

- **Initial**: R_{ai} has not been received yet.
- **Pending**: R_{ai} has been received, but it is not yet known to be stable (*see more: Message Stability, page 19*).
- **Stable**: R_{ai} is known to be stable.
- **Processed**: R_{ai} has been processed and any changes have been committed.

Consistency Rules

Each RM has the same initial state, and depending on the application data, one of the following holds:

- **FIFO ordering**: for any given client c, R_{ci} is processed before $R_{c(i+1)}$.
- **Causal ordering**: if R_{ai} is a potential cause of R_{bj}, for any a, b, i and j, then R_{ai} is processed before R_{bj}.
- **Total ordering**: every RM processes requests in the same order.

Relaxing Consistency Rules

Read requests do not alter state, so if R_{ai} is a read request then $R_{ai}(s) = s$. Read requests can be send to just **one RM** if the RM has only failstop failures, and the read does not cause other requests.

Some requests **commute** such that $R_{a1}(R_{b2}(s)) = R_{b2}(R_{ai}(s))$. This is the case when both requests are **reads** or both **write to disjoint parts of the data**.

Requests of these types do not need to be ordered, and so some consistency constraints can be relaxed.

Maintaining Consistency

The first consistency rule (same initial state) is trivial, but the other consistency rules (FIFO, causal and total orderings) require both of the following:

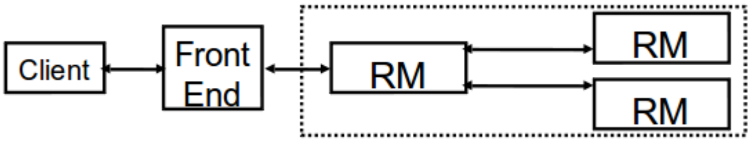
- **Agreement** between all non-faulty RMs about the requests that have been sent.
- **Ordering** of all requests according to the selected rule.

Two main approaches exist for achieving agreement and ordering: **primary backup** and **state machine**.

Primary Backup Approach

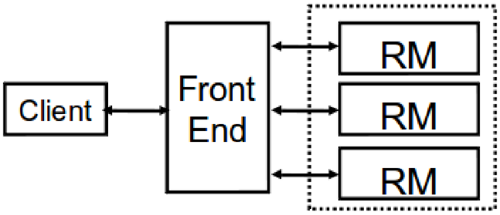
All clients/front-ends communicate with **one RM**, called the **primary**. Other RMs are **backups**, which the primary updates.

This system is simple to construct, but introduces a single point of failure if proper failover procedures are not in place.



State Machine Approach

This system treats **all RMs as equal** and clients **communicate with all RMs**, via the front-end(s). This system is **more robust** than the primary backup approach.



Achieving Consistency in Replication

Primary Backup Requirements

- **Agreement**
  - Front-ends agree with a primary by a standard **handshake** protocol.
  - The primary agrees with backups by **handshaking**.
- **Ordering**
  - The primary controls the processing of requests and so determines their order.

State Machine Requirements

- **Agreement**
  - Each front-end must be able to broadcast requests to all non-fault RMs.
  - Each front-end must know that the RMs have received a request (i.e. reliable broadcasts).
- **Ordering**
  - A **unique identifier** is assigned to each request, enabling the same ordering to be maintained at each RM.

Replica-Generated IDs

Replicas can and do broadcast between themselves to maintain consistency, but the set of clients can be vast and rapidly-changing, so **clients should not be required to communicate with each other** to establish message ordering (as would be the case for vector clocks). Replica-generated IDs allow message ordering to be established without requiring all clients to communicate with each other. They are generated by the following process:

1. A client broadcasts a request R_{Ai} to all RMs.
2. Each RM RM_n generates a **candidate** identifier, C_{ni}.
    - Values chosen must exceed the identified of any other request already accepted.
3. The client chooses the **maximum** candidate identifier as a unique request identifier.
4. The client informs all RMs of the choice.

Stability

For a request R_{Ai} to be stable at RM_n, the following must be true:

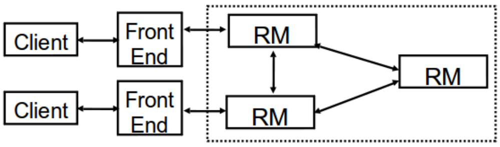
- RM_n must have been informed of the unique ID for R_{Ai}.
- There must be no other request for which RM_n has generated a lower candidate ID but has not been informed of the chosen unique ID.

Using this approach, each process knows a minimum value that each request can have: the candidate ID it proposed.

State Machine Improvements

The replica-generated ID system may sometimes lead clients to choose **duplicate IDs** for their requests, which is a problem. This can be avoided by ensuring that **different RMs don't propose the same numbers** (in a system of n RMs, each could start from a different offset and increment their candidate IDs by n).

The system also requires **additional complexity** in the clients and **increased communication overhead** to handle the two-phase approach. Instead, a **server-side model** is common: clients send their requests to one RM, which handles the rest of the request identification issues on the client's behalf.



Where feasible, the server-side model can make use of **vector clocks** because clients are no longer involved. However, clients should still monitor their local RM to make sure requests are distributed as expected.

Handling Failure in Replication

Failover in Primary Backup Systems

Primary backup systems suffer a **single point of failure**: the primary RM. If the primary RM fails then an outage occurs and requests will fail until a backup RM notices that the primary is dead and takes over. This is called a **failover**, a process which has certain properties:

- Failover can **take time** - during this period there is an outage and requests will fail.
- A **backup must be selected** to take over - this is achieved by consensus, discussed later (*see more: Consensus, page 31*).
- For backups to notice that the primary is dead, it must fail with a **failstop** or be part of a synchronous system (*see more: Failure Modes, page 8*).
  - Timeouts aren't ideal: too short and a slow-but-functional primary will be replaced; too long and the failover will take too long.

Handling Primary Change

When a failover occurs, all further requests must be **sent to the new primary**. It's also possible that some requests were lost when the original primary failed, so front-ends must detect outages and re-transmit non-acknowledged requests.

Properties of Primary Backup Systems

- At any one time there must be **at most one primary** RM.
- **Failover time** is the period for which there is no primary RM.
- Each front-end keeps a pointer to the RM that it currently believes to be the primary RM.
- Any request to a backup RM should be failed and ignored.

Maintaining Consistency

The usual process for a primary RM is the following:

1. Receive the request R_{Ai} from a client.
2. Update it's own state s to $R_{Ai}(s)$.
3. Broadcast R_{ai} to all backup managers (via reliable broadcast).
4. Acknowledges success of R_{Ai} to the client.

The ordering of these steps ensures that if the primary RM crashes at some point in the middle of the process, the client or front-end will know that the request must be re-sent. The time taken by the whole process is called the **blocking time**, because it's how long a client must wait for confirmation that a request has been processed.

The true system state is only guaranteed to be held by the primary RM.

How Much Replication is Needed?

CAP Theorem

The CAP theorem argues that where **partitions** occur in a network (i.e. messages not being communicated between some parts), it is only possible to have **consistency** (every successful read gives up-to-date data) or **availability** (every read gets a response), but not both.

A distributed system cannot have all of **c**onsistency, **a**vailability and **p**artition tolerance at the same time.

Recap: Failure Measurements

- **Mean Time Before Failure (MTBF)**
  - Average time between failures.
  - Assuming a random distribution of failures the probability of a failure in a period of time can be calculated.
- **t-Fault Tolerance**
  - There must be more than t failed components before the service fails (we can suffer t failures before losing the service).
  - This can be more useful when designing the system.

Recap: Failure Modes

- **Failstop**: the process halts and remains halted. The fact that it is halted *can* be detected by or communicated to other processes.
- **Crash**: the same as failstop, but *cannot* be detected by other processes.
- **Crash+link**: a link fails for a period, losing some messages that it was carrying.
- **Receive omission**: a process fails to read some of the messages sent to it.
- **Send omission**: a process fails to write some of the messages required by the protocol.
- **General omission**: receive and/or send omission.
- **Byzantine failure**: the process behaves in a manner not permitted by the protocol.

Meaning of t-Faults

In the case of **crash**, **failstop** or **Byzantine failures**, t-faults refer to the number of hosts (RMs in this case) that can crash.

In the case of **crash+link**, **receive**, **send** or **general** omissions, t-faults refer to either the number of hosts that may crash or the number of links that may fail (i.e. the network has partitioned).

Reaching t-Fault Tolerance

The minimum amount of replication will be called the **lower bound** on replication. The minimum can be useful for two reasons:

- A protocol which claims to need less than the minimum must contain some flaw.
- A protocol which requires more than the minimum is sub-optimal, and therefore may be worth improving.

State Machine Replication: Non-Byzantine Failures

The front-end(s) must look at the results of a request as an ensemble, so minimally **we need one RM to be failure-free**. Therefore, we need the number of RMs to be **greater than** t.

State Machine Replication: Byzantine Failures

Responses will be sent from all RMs, but some may be invalid. The front-ends must take a **majority vote** on responses, so therefore the number of RMs needed must be **greater than** $2t$.

Primary Backup Replication: Send Omissions, Failstop and Crash Failures

As with state machine replication, the number of RMs must be **greater than** t. In the worst case only one RM will be left: it will know (from lack of communication from other RMs) that it is alone, and must serve as the primary.

Primary Backup Replication: Byzantine Failures

A Byzantine failure of the primary RM can cause a total failure of the primary backup protocol: **there is no lower bound on** t.

Primary Backup Replication: Crash+Link Failures

In a crash+link failure, RMs may remain active but cannot know whether others have failed, because of broken links. If all connections between the primary and the backups are broken, each one may believe itself to be alone and nominate itself as the primary. This creates **multiple primaries**.

For t-fault tolerance we need more than t backups, so **more than** $t+1$ **RMs** (primary plus t).

Primary Backup Replication: Receive Omissions

The worst case scenario is as follows:

- Consider three groups of RMs: A, B and C.
- All in A fail to receive messages from all in B and C.
- All in B fail to receive messages from all in A and C.
- Each of A and B now has no contact with others, so they do not know whether they have crashed.
- Each elects a new primary (or contains the original).
- **Multiple primaries** now exist.

For t-fault tolerance we need to ensure that there are more than $t/2$ RMs in any one subset. In total, there must be **more than** $3t/2$ RMs.

Primary Backup Replication: General Omissions

The worst case scenario is as follows:

- Consider two groups of RMs: A and B.
- All in A fail to send/receive messages to/from all in B.
- A now has no contact with B, so B may have crashed.
- B contains a primary, and A elects a new one.
- **Multiple primaries** now exist.

For t-fault tolerance we need to ensure that there are more than t RMs in any one subset. In total, there must be **more than** $2t$ RMs.

Consensus

Choosing a new primary during failover in a primary backup system (*see more: Failover in Primary Backup Systems, page 27*) is an instance of the more general problem of **consensus**.

A consensus between processes involves them all accepting a single value based on values they propose.

- `propose(v)` occurs at a process which wants the consensus value to be v.
- `decide(v)` occurs at a process which decides the consensus value is v.
- A special value, `FAIL`, is to be agreed upon when consensus cannot otherwise be achieved.

Consensus Protocol Properties

- **Termination**: every correct process will eventually decide on a value.
- **Validity**: if all proposals are for v then all correct processes will eventually decide v.
- **Agreement**: if one correct process decides v then all correct processes will eventually decide v.
- **Integrity**: every correct process decides at most one v, and either $v =$ `FAIL` or some process must have proposed v.r

Consensus via Total Ordering

Achieving consensus is **trivial with total/atomic ordering**: every process will deliver messages in the same order, so every message can `decide` on the first value it receives and ignore all subsequent values.

Consensus with Raft

Raft is a popular algorithm for **primary election** that uses **logical clocks**. The primary RM sends a regular **heartbeat** message to other RMs to declare that it is alive. When the primary crashes, RMs are aware because the heartbeat will stop (they can detect this with timeouts), at which point **majority voting** is used to elect a new primary.

Each RM can be in one of three states at any time: **leader**, **follower** or **candidate**.

Voting Terms

A distributed logical clock, called the **term** (as in 'term in office'), is used by the algorithm and attached to every message.

When a backup RM believes the primary has crashed, it increases its term by 1 (a clock event), becomes a **candidate** and broadcasts a request to be elected.

RMs vote for the first candidate whose request they received (per term). If an RM receives a majority of votes for the current term, it will become a **leader**, inform all RMs that it is the leader, and start sending out a heartbeat.

Failed Elections

It's possible that multiple backup RMs will detect a dead primary before receiving proposals from others, and all become candidates. RMs will vote for the first proposal they received, but that **may not achieve majority**. In this event, an RM will timeout again, increase the term, and start a new round of voting.

RMs ignore heartbeats and voting messages with earlier terms (lower clock values), allowing them to keep trying until an election is successful in picking a new primary.

RMs have different, random timeout periods to help minimise failed elections.

Distributed Transactions

A major concern in concurrency control is the notion of performing tasks that **indivisible**. Such tasks are called **transactions** and are generally considered in the context of a read or write operation applied to some data source.

ACID Properties

Transactions must satisfy all of the following:

- **Atomicity**: the entire task is applied, or none of it is.
- **Consistency**: consistent input leads to consistent output (if the data source is consistent at the start, it must be consistent at the end).
- **Isolation**: transactions should be independent of each other.
- **Durability**: once completed, a transactions effects will not be 'forgotten'.

Maintaining Atomicity and Durability

Atomicity and durability can only hold for **certain t-fault tolerances** and can be achieved by keeping copies of actions in a log file. This allows partially-completed transactions to be 'undone' or completed transactions to be 'redone' (in the event of data loss).

Maintaining Consistency

Transactions must be programmed in such a way that a **consistent state will not be broken** by doing, redoing or undoing the actions of a transaction.

When the above property cannot be trusted, **consistency constraints** must be used: check the database consistency at the end of each transaction and abort if consistency is violated.

Maintaining Isolation

Isolation can be achieved trivially by **serialising** all transactions, but this is slow.

With concurrent transactions, some **concurrency control mechanism** must be applied to ensure that transactions are **serialisable**. This means that performing them in parallel is certain to produce the same effect as performing them in serial. When this cannot be guaranteed, some transactions may need to be queued.

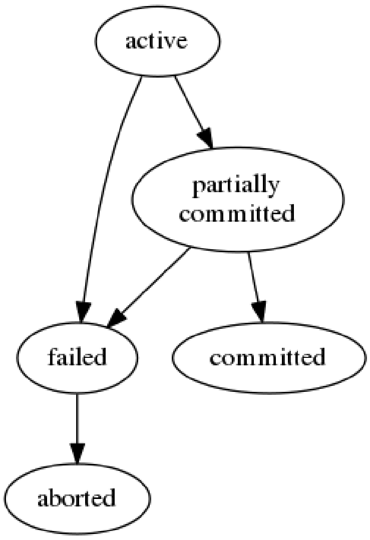
Specifying Transactions

A transaction forms a program executed by a **transaction manager**; often this program is specified by SQL.

Technically, the program specifies a **transaction type**, each execution of which is called a **transaction**. Unless the distinction is important, this detail is ignored.

Transaction State

- **Active**: the initial state, which the transaction stays in whilst it is executing.
- **Partially committed**: after the final statement has been executed.
- **Failed**: when it is discovered that normal execution cannot proceed.
- **Aborted**: after the transaction has been rolled back and the database has been restored to its state prior to the transaction.
- **Committed**: after successful completion.



The one of the final two states must always be reached to ensure **consistency**.

Example Transactions

The following example transactions will be used throughout this section:

- **T1**: transfer £100 from one account to another.
  - `UPDATE accounts SET balance = balance — 100 WHERE id = 123;`
  - `UPDATE accounts SET balance = balance + 100 WHERE id = 789;`
- **T2**: debit £10 from all accounts.
  - `UPDATE accounts SET balance = balance — 10;`
- **T3**: sum the balance of all accounts.
  - `SELECT SUM(balance) FROM accounts;`

Nested Transactions

Nested transactions **break up** one transaction into sub-transactions. Nested transactions are said to be **in the scope of the parent transaction**. Each individual nested transaction **follows ACID principles**, as does the parent.

When a parent transaction fails and aborts, all nested transactions will also do so. When a nested transaction aborts, the parent may either abort or take some alternative action (depending on the scenario).

Benefits

- An **increase of throughput** may be seen, as nested transactions may be executed concurrently and smaller transaction scopes are less likely to overlap and cause queuing.
- **More situations can be handled** because parent transactions can choose different actions when children fail.
- It may be **easier to handle fragmented data**, because different nested transactions can be sent to different hosts.

Examples

These examples list a few options for applying nested transactions to the previously-seen example transactions. Other options exist - these are just a few examples.

See more: Example Transactions, page 34.

- **T1**: transfer £100 from one account to another.
  - Use a nested transaction for each of the two update statements.
- **T2**: debit £10 from all accounts.
  - Use a nested transaction for blocks of n account IDs.
  - Use nested transactions to each customer time-zone (logic: active accounts will mostly fall into daytime time-zones, so night-time time-zones shouldn't need to be queued).
- **T3**: sum the balance of all accounts.
  - *Same as above.*

Transaction Managers

In a centralised DBMS, **transaction execution is controlled by a transaction manager** to ensure that ACID properties are maintained.

In a distributed environment, two 'levels' of management exist:

- **Local** maintenance of ACID properties is managed by **Local Transaction Managers (LTMs)**.
- A **Global Transaction Manager (GTM)** distributes requests to LTMs and coordinates their execution.

Several GTMs are used to avoid creating a single point of failure.

Concurrency Control

Concurrency control is the **maintenance of the serialisability** of transaction execution.

Reminder: transactions are serialisable if performing them in parallel is certain to produce the same effect as performing them in serial.

Notation

We model the execution of any transaction as a series of **read** and **write** operations, which are expressed with the following notation:

- $rA(o)$ - read from object o in transaction A.
- $wA(o)$ - write to object o in transaction A.

Assumptions

- Any object is read by a transaction before being written to by the same transaction.
- Each transaction only reads from and writes to an object at most once.
- Every operation can be uniquely categorised as a read or a write.

Serial Execution

Isolation can be easily maintained by **removing concurrency completely** an executing every transaction sequentially. This would give extremely **poor performance**. Transactions must be **interleaved** to better utilise the resources and distributed nature of a system.

Serial Equivalence

Transaction operations can be **interleaved in many ways**, but for a given interleaving of two transactions to meet ACID properties the interleaving must have exactly the same effect as executing the two transactions serially.

Such a sequence is said to be **serially equivalent**, and the concurrent execution of the sequence is said to be **serialisable**.

Lost Update Problems

An interleaving that is **not serially equivalent** may result in a **lost update**. This means that the effect of one update will be overwritten by another.

For example, consider the following two transactions:

- A: `UPDATE accounts SET balance = balance + 50 WHERE id = 123;`
- B: `UPDATE accounts SET balance = balance + 100 WHERE id = 123;`

Serially, the actions $rA(acc123)$, $wA(acc123)$, $rB(acc123)$, $wB(acc123)$ should add £150 to the account. However, the following interleaving loses one of the updates:

	Assume a start balance of £0
$rA(acc123)$	A reads the balance of £0
$rB(acc123)$	B reads the balance of £0
$wA(acc123)$	A writes a balance of £0 + £50 = £50
$wB(acc123)$	B writes a balance of £0 + £100 = £100
	The balance is now £100

Formally, a lost update occurs when **two writes are made to an object without a read in between by the writing transaction**.

Inconsistent Retrieval Problems

An interleaving that is **not serially equivalent** may result in **inconsistent retrieval**. This means that the effects of a transaction that is mid-execution will cause inaccurate data to be read by another transaction.

For example, consider the following transactions (A and B are the nested transactions of example T1; C is example T3):

- A: `UPDATE accounts SET balance = balance — 100 WHERE id = 123;`
- B: `UPDATE accounts SET balance = balance + 100 WHERE id = 789;`
- C: `SELECT SUM(balance) FROM accounts;`

Assume both accounts start with £100, and all others have £0.

Serially, the actions $rA(acc123)$, $wA(acc123)$, $rB(acc123)$, $wB(acc123)$, $rC(acc123)$, $rC(acc789)$ should yield £200. However, the following interleaving produces an inconsistent retrieval:

	Assume a start balance of £100 for both
$rC(acc123)$	C reads the balance of £100
$rA(acc123)$	A reads the balance of £100
$wA(acc123)$	A writes a balance of £100 - £100 = £0
$rB(acc789)$	B reads the balance of £10
$wB(acc789)$	B writes a balance of £100 + £100 = £200
$rC(acc789)$	C reads the balance of £200
	The sum is now £300

The sum is inaccurate because the £100 that was transferred was 'seen' twice by the summing transaction.

Formally, an inconsistent retrieval occurs when **a transaction writes to some object(s) while another transaction is in the middle of reading it/them**.

View Equivalence

View equivalence between sequences of actions means that the **relevant state of database objects** is as expected for any given action.

A sequence of actions interleaved from multiple transactions is **serialisable if it is view equivalent to a serial execution** of those transactions.

Two sequences $S1$ and $S2$ are view equivalent if they obey the following rules:

- For each data item Q, if T_i reads the initial value of Q in $S1$ it must also do so in $S2$.
- For each data item Q, if T_i reads the value of Q that was written by T_j in $S1$, it must also do so in $S2$.
- For each data item Q, the transaction that executes the final $write(Q)$ in $S1$ must also execute the final $write(Q)$ in $S2$ (if such an action exists).

Enforcing Serialisability

There are multiple ways to enforce serialisability rules across a distributed system:

- **Timestamping**
  - Add a timestamp to each object and transaction.
  - One write, set the object's timestamp to the transaction's timestamp.
  - Transactions may only access objects with earlier timestamps.
- **Optimistic Control**
  - Allow transactions to do whatever they need to do.
  - Check for rule violations at commit time.
- **Two-Phase Locking**

Two-Phase Locking (2PL)

This locking system prevents read or write commands that would cause serialisability rules to be violated. It works by **locking** objects, preventing operations that should not be permitted. Locks can **restrict reading and/or writing**.

Type of Lock

- **Read locks** are placed on objects whenever a process wishes to read its contents.
  - Can be obtained by any number of processes.
  - Cannot exist if a write lock exists on the object.
- **Write locks** are placed on objects whenever a process wishes to change its contents.
  - Can only be obtained by one process at a time.
  - Can only be obtained if no read locks are held (unless held by the writing process).

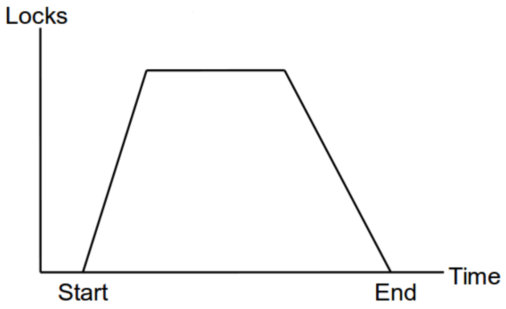
The Two Phases

Transaction execution follows two phases:

- **Growing phase**
  - Operations that require additional locks to be obtained.
  - Transactions may obtain locks and may not release them.
- **Shrinking phase**
  - Operations that do not require additional locks to be obtained.
  - Transactions may release locks and may not obtain them.

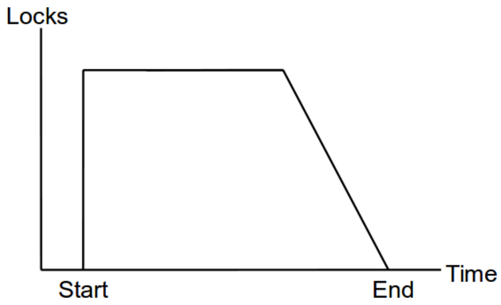
Locking Strategy: Standard 2PL

Locks are gradually obtained as necessary and gradually released when no longer required.



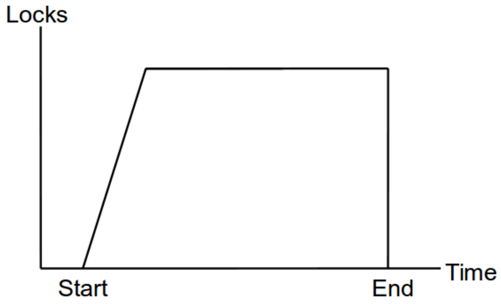
Locking Strategy: Conservative 2PL

All necessary locks are obtained at the start and then gradually released when no longer required.



Locking Strategy: Strict 2PL

Locks are gradually obtained as necessary and then all locks are released at the end. This is most common.



When to Lock?

The choice of locking strategy will affect the throughput of the system. In practise, **strict locking** is usually followed:

- Locks are obtained gradually as required.
- The growing phase covers most of the transaction.
- All locks are held until the commit or abort.

Conflicts

Conflicts occur in a transaction when it needs to obtain a lock on an object which the locking rules refuse. Usually this means the transaction needs to **wait** until the other transaction ends and releases the locks.

Conflicts: Deadlock

Certain combinations of conflicts and waiting can lead to **deadlock**: two transactions may each be waiting for the other to release a lock that it wants to acquire. In this situation neither can proceed.

A local transaction manager may detect deadlocks by constructing a **waits-for** graph, showing for each transaction which other transaction(s) they are waiting for. A **cycle** in such a graph indicates a deadlock.

Deadlocks may be **resolved** by aborting one or more of the deadlocked transactions and re-trying it later.

Conflicts: Livelock

Instead of simply waiting (potentially forever), transactions can **react to lock conflicts** by taking some action or changing state.

For example, a transaction that encounters a lock conflict might release the locks it already holds, pause for a moment, then re-try. This could allow other transaction to acquire the necessary locks and continue, clearing the way for the first transaction to try again.

If multiple transactions do this they can end up continually reacting to each other's locks by releasing their own, meaning neither of them will make progress - this is **livelock**. Livelock can potentially be avoided by introducing variance in the delays and/or approaches of different transactions.

Conflicts: Starvation

Some objects can be in **higher demand** than others. A given transaction may find that some other transaction always obtains a lock on some object first, which would **starve** the first transaction.

This can be solved by giving each transaction a **priority** that is used to control the order in which locks requests are considered. Transactions that are delayed have their priority gradually increase, so they are eventually able to execute.

Issues in Distributed Locking

To use 2PL on distributed data the following issues must be considered:

- How should locks be applied?
    - With **fragmented** data the lock can be held by the server holding the fragment - simple.
    - With **replicated** data, is it enough to replicate the locks? How else might it work?

- How should conflicts (deadlock, livelock and starvation) be handled across distributed LTMs?
  - For **livelock** and **starvation** we can still use priorities.
  - For **deadlock** we need distributed waits-for graphs.
- How should the phases by synchronised?

Locking Replicated Data

With the **primary backup model**, locking replicated data is easy: locks are applied to and held by the primary, as discussed previously. In the event of failover, all locks are lost and all in-progress transactions are aborted.

With the **state machine model**, things get harder:

- We could nominate one RM to be responsible for locking for a given set of objects, but this creates a single point of failure (in much the same way as the primary backup model).
- For robustness and fault-tolerance, we must **distribute** the locks.

Distributed Locking

Applying all locks to all RMs is very expensive, so we can use two facts to **reduce the amount of locking required**:

- Only write locks cause conflicts.
- We only need a conflict to be detected on $t + 1$ hosts in a t-fault tolerant system.

Therefore, for one object replicated over n hosts with *no* failures:

- If we have k read locks on the object...
- ...we need to apply $n - k + 1$ write locks to guarantee a conflict.
- This gives a total of $k + n - k + 1 = n + 1$ locks applied to n hosts, so via the **pigeonhole principle**[1] a conflict must occur.

Similarly, for one object replicated over n hosts with t failures:

- If we have k read locks on the object...
- ...we need to apply $n - k + 1 + t$ write locks to guarantee a conflict.
- The $+t$ means that, in the worst case, write locks could be applied to all t failing hosts and there would *still* be enough to detect a conflict.

[1] If n items are put into m containers, with $n > m$, then at least one container contains more than one item.

Distributed Waits-For Graphs

With fragmented data, deadlock may occur between transactions operating on separate hosts. Waits-for graphs can be used to detect this, but must be maintained on a **system-global scale**.

One approach is to hold the graph on one central server, but this **delays detection**, **reduces scalability** and introduces a **single point of failure**.

The alternative is to use distributed waits-for graphs:

- A **local waits-for graph** is stored, with any transactions outside the local server being marked as an `EXT` node.
- Local deadlocks can be detected quickly, as before.
- When requests are received, the local manager sends its waits-for graph to the current **coordinator node**, timestamped so that the coordinator can determine the most recent information.
- The coordinator will construct the **global waits-for graph** and detects cycles as before.

This means that communication only happens during requests, and only for the parts of data touched by that request.

Distributed 2PL

It is not enough to run two-phase locking on each server. 2PL ensures transactions are locally serialisable, but will not work if one server starts **shrinking** whilst another is **growing**.

- We could have a protocol to ensure that a global transaction manager **synchronises the growing and shrinking phases** of all local transaction managers, but this would be complex and expensive.
- In practise, we make locking **strict** (hold all locks until the transaction ends) and use **atomic commitment** to ensure locks are all released at the same time.

Atomic Commitment in Distributed Systems

ACID properties require transactions to be durable when they commit.

- In a centralised system this amounts to making sure all updates are logged and that the 'transaction complete' action is atomic.
- In a distributed system we must also ensure that **all servers involved in the transaction** agree to commit, or agree to abort.

Atomic commitment generally requires the following protocol:

- One **coordinator process**.
- Several **server processes** which are executing the transaction.
- When the coordinator process decides to end the transaction:
  - The coordinator process asks the server processes to **vote** if they are able to commit the transaction.
  - The server processes may vote to **commit or abort**.
  - The coordinator process instructs all server processes to commit if all of them voted to do so (or abort otherwise).

Two-Phase Commit (2PC) Protocol

Execution

- Phase One
  - The coordinator transmits `C—PREPARE` to all servers, telling them that the transaction should now commit.
  - A server may reply with `C—READY` if it is ready to commit. After this point it **may not abort** the transaction.
  - A server may reply with `C—REFUSE` if it is unable to commit, and then abort the transaction.
- Phase Two
  - If the coordinator receives `C—READY` from **all servers** it transmits `C—COMMIT` to all servers, each of which then commits.
  - If the coordinator receives `C—REFUSE` from **any server** it transmits `C—ROLLBACK` to all servers, each of which then aborts.
  - After committing or aborting, each server sends `C—ACK` back to the coordinator.

Robustness

With no crashes and no network errors, 2PC is reliable and robust.

Some failures can be easily handled with **timeouts**. For example, if the coordinator doesn't receive a reply after `C—PREPARE` from one or more servers, it can tell all of the servers to abort (via `C—ROLLBACK`) or request some servers to restart (via `C—RESTART`).

If the **coordinator fails** after asking for a vote:

- All server will time-out waiting for `C—COMMIT` or `C—ROLLBACK`.
- An election will establish a new coordinator.
- The new coordinator will resume/restart the transaction.

Disadvantages

- **Blocking**: processes will block whilst waiting for messages. Locks are held by blocked processes, so other competing processes will have to wait for locks to be released.
- **Conservative bias**: the protocol is biased towards aborting.

Blocking

If a server S has sent C—READY, it **must** commit if it then receives C—COMMIT. There are two options to handle when the coordinator crashes:

- The coordinator may send C—PREPARE and then crash. S knows that another server might have replied with C—REFUSE, so the correct behaviour would be for S to **abort**.
- The coordinator may send C—COMMIT to all other servers and then crash. Those servers might have committed their transactions, so the correct behaviour would be for S to **commit**.

Once a server has voted to take part in a commit, it does not know the result of the vote until the command to commit arrives. It must **block during this uncertain state**.

By the time it has timed-out after not receiving the vote result, all others may have received C—COMMIT, performed a commit, or failed and become unreachable. Eventually one of the failed servers will execute a recovery, but this may take a great deal of time.

Three-Phase Commit (3PC) Protocol

3PC adds an extra layer to 2PC to create a **non-blocking protocol**. The extra phase obtains and distributes the result of the vote before sending out the command to commit. There are two new messages: C—PRECOMMIT and C—PRECOMMIT—ACK. If the vote times out, servers do not have to commit.

Execution

- Phase One
  - *identical to 2PC*
- Phase Two
  - If the coordinator receives C—READY from **all servers** it transmits C—PRECOMMIT to all servers, each of which replies with C—PRECOMMIT—ACK.
  - If the coordinator receives C—REFUSE from **any server** it transmits C—ROLLBACK to all servers, each of which then aborts.
- Phase Three
  - If the coordinator receives C—PRECOMMIT—ACK from **all servers** it transmits C—COMMIT to all servers, each of which then commits.

  - If the coordinator does not receive `C—PRECOMMIT—ACK` from a it transmits `C—ROLLBACK` to all servers, each of which then aborts.

Avoiding Blocking

The vote result is sent to all servers, and is confirmed to have arrived at all of them before a command to commit is sent out.

If a server misses a `C—PRECOMMIT` it will time-out and contact some other server to get the result of the vote. If all other server have failed, the server can abort because none of the other servers could have committed.

Performance

3PC **increases throughput in the face of failures** and **decreases abort-bias** compared to 2PC, but involves **more communication** and **more situations to handle** than 2PC.

Distributed File Systems

Naming

Names are data items used to **refer to resources**, in order to **access** them, **specify** their use, **describe** them, look up **details** about them, etc. Names may be **typed**, depending on the object they refer to.

Naming Terms

- **Namespace**: the range of names supported by a schema or system.
- **Naming domain**: name space with a single authority.
- **Naming scheme**: the structure or syntax from which names are constructed.

Naming Characteristics: Size

- Fixed-size
  - Examples: IP addresses, memory addresses, phone numbers (sort of).
  - Pro: easier to handle.
  - Con: finite range.
- Infinite
  - Example: email addresses (sort of).
  - Pro: allows for indefinite range.
  - Pro: allows integration of name spaces.
  - Con: can be harder to deal with.

Naming Characteristics: Presentation

- User-oriented: formatted in a way that users can understand and use.
  - Examples: ‘google.com’, ‘print_server’.
- Machine-oriented
  - Example: 216.58.198.110.

Naming Characteristics: Purity

- A pure name has no structure to it.
  - It can only be used to compare equality with another name.
- An impure name has some structure.
  - A valid name is something about the object.

  - Examples: IP addresses (network and host IDs), absolute file paths (which give location), room numbers like ‘S4.01’.
  - Can be physically- or organisationally-oriented:
    - Physically-oriented: some physical layout is encoded in the name (such as ‘room S4.01’).
    - Organisationally-oriented: some information about how objects and resources are organised is included (such as file paths).

Naming Characteristics: Scope

- Global scope: the name alone gives the type of object it defines.
  - Example: google.com, a book’s ISBN.
- Namespace-specific: the same name may be used in another namespace for a different object.
  - Example: ‘room 4-12’ has relevance in one building, but not worldwide.

Naming Characteristics: Context

Considering paths within a naming hierarchy:

- Context-dependent: a path may resolve to a different identifier depending on the context.
  - Example: phone extension numbers.
- Absolute: a path resolves to the same identifier, regardless of context.
  - Example: a full postal address.

The context name together with the object name forms a unique object identifier.

Associating Objects

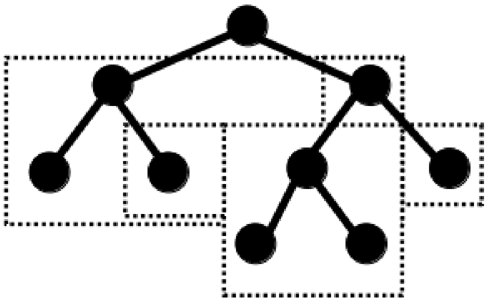
Objects can be associated with other objects in various ways, for example a domain name with an IP address.

- An association between different names is called a **binding**.
- A list of bindings forms a **directory**.
- Access to directories is provided by one or more **name servers** (NSs).

Association Resolution

In principle, every request for resolution of a name results in a network request to a name server. On a busy system this produces **a lot of traffic** and limits **scalability** and **fault tolerance**. Two approaches are used to reduce load on the network and the name server:

- **Caches** are used on local hosts to store the results of lookups.
  - Caches are **shared** by all processes and managed by a stand-alone user agent.
- Directories are spread across multiple name servers, by means of **replication and/or fragmentation**.
- **Partitioning** (fragmenting) the directory will mean using a hierarchy of names, which results in a tree of **zones**.



Access Modes

- **Client iterative navigation**: the client makes requests to a set of name servers in sequence, until one is able to respond to the request.
- **Server recursive navigation**: the client makes a request to one name server, which will spread that request out to other name servers until one is able to respond, then the response is passed back along to the first name server and, ultimately, the client.
  - This is how DNS works.
- **Server multicasting**: the client makes a request to one name server, which follows an approach similar to client iterative navigation with its neighbours until a response is found.

In all cases, the first name server reached may be able to respond, and the request would go no further.

File System Characteristics

A file system provides **permanent storage of data** in a manner that survives system crashes. The basic features are:

- File, modelled as arrays of bytes and identified by names.
- Primitive file operations, like `open`, `read`, `write` and `close`.

Centralised Execution

The name `stuff.txt` is used to locate a record containing attributes of the file and its location on disk. These attributes must record several things:

- The **permission** of a user to read the file.
- The **state** of the file being opened.
- A **cursor** to store a user's read position within the file.

Naive Read Implementation

During a read operation, the following could happen:

- The user asks the file system for a byte from the file.
- The file system commands the disk to read that byte, and blocks the user process while it waits for that to happen.
- The disk reads the byte, loads it into a buffer and informs the file system.
- The file system puts the process on the queue of ready processes.

This is extremely inefficient: each byte read causes the disk to seek and the user process to be blocked and unblocked.

Buffered Block Reading

In a more realistic implementation, data is read from the disk in fixed-size **blocks**. On each disk request, the process above is followed, but for a block of bytes rather than just one. Using blocks of n bytes, the disk is only touched for every $\frac{1}{n}$ bytes read.

Many systems will also store recently-read blocks in main memory, allowing them to be returned immediately.

File Naming

- **Symbolic names**: given by a user process to identify a file.
  - Usually impure.
  - Example: `/home/markormesher/.bashrc` refers to a file `.bashrc`, which is in the directory `markormesher`, which is in the directory `home`, which is at the root of the file system.
- **Binary names**: used inside the file system to identify a file's physical location.
  - Usually a pure name, like a UNIX inode.
  - Usually a disk address, or record containing the disk address of several file fragments.

Services

- **File Service**
  - This service handles basic operations like reading and writing of data from and to files.
  - It works with binary names.
  - It deals with the physical layout of data.
- **Directory Service**
  - This service handles the structuring of files into a user-oriented catalogue.
  - It works with symbolic names.
  - It provides a map between symbolic names and binary names.

Linking Files to Directories

- **Hard links**: the binary name of a file is inside the directory.
  - Pro: files can be found rapidly.
- **Symbolic links**: the symbolic name of a file is inside the directory.
  - Con: must be passed to the directory service to resolve a binary name.
  - Pro: not limited by the binary name space of a particular file system.

Using Hard Links

- At least one hard link must point to every binary name that is valid in the file system.
- More than one hard link can exist for a given binary name.
- The exception is the binary name of the root directory: either the file itself or its binary name must be at a predefined location on disk.

Using Symbolic Links

- The file linked to doesn't necessarily have to exist.
- During the lifetime of the link, the file can be deleted and replaced, without changes being required in the directory containing the link.
  - This is contrary to hard links, which require that a file is only deleted when the last hard link pointing to it is deleted.

Distributing File Systems

The majority of accesses are **sequential** reads of **small files** by **single processes**, so a distributed file system should optimise for this use case. It should still, however, support concurrent reads, large files, and random access.

In a distributed system, we may have **several file and directory services**, allowing files to be accessed in different ways according to different user systems. For each, we must specify:

- How files should appear to users.
- What primitives should exist for dealing with files.

There may be **many file server processes**, i.e. one for each host serving files in the system. Each will fetch blocks to and from a device, such as a disk.

File System State

In a centralised system, some notion of **state** is maintained for files, describing which processes are accessing them. Some sort of **open** and **close** commands must exist to administer this state.

The state may also record information about **who** is reading/writing with a file, in order to perform locking, and a **cursor** to store the read position that some process has reached.

Stateless File Systems

Maintaining state in a centralised system is relatively easy, but in a distributed system with date split between clients and server, maintaining consistency can be difficult. It is sometimes easier to use a stateless system, where each request is entirely self-contained and provides all information necessary to handle it:

- The identification of the user (for authentication and permissions checking).
- The identification of the file to operate one.
- The position of the file at which to operate (i.e. the cursor).
- The operation (primitive) to perform.
- Any other required metadata.

This approach has several advantages:

- Nothing is lost when either the server or the client crashes: when the server fails and is restored, the client can carry on as before; when the client crashes, the server doesn't need to know and doesn't care.
- The server does not need to maintain information about active users.

- There will be no ‘too many files open’ issues.
- No explicit open and close actions are needed, reducing latency and removing the issue of abandoned handles.

Caching Data

Caching can be used to improve performance and reduce latencies. As mentioned earlier, most file systems will keep recently-read blocks in memory - this is a cache. Caching is also used for DNS resolution.

For a distributed system, a network separates the client and the file server. Files are cached on the client’s side of the network to reduce latency and traffic.

Caching can cause **consistency problems** if multiple users access the same files, because each may be working on their own separate, cached copy of the file. One solution to this is **write-through**: any write to a cache is also copied as a write on the file server. However, this can still leave hosts with **stale** versions of files in their cache is they don’t update their cached copy. A better solution is the Unix Model.

Unix Model Caching

In a Unix file system, all changes by one user become visible to all other users immediately, because **local caches are refreshed on every read**. This is implemented with timestamps:

- On read, the block’s local timestamp is sent to the file system, and compared with the timestamp held there. If the file system’s timestamp is higher than the one held locally, the data is downloaded and the cache is updated.
- On write, the file system’s timestamp is increased.

This reduces the amount of data sent by only updating when changes have happened, but will still introduce some latency.

Cache Update Semantics

- **Unix** semantics
  - This is ideal, because a distributed file system can be used in exactly the same way as a centralised one.
  - This method is expensive.
- **Session** semantics
  - Changes are made to the local copy only.
  - Changes are written to the file system on close.
  - The last client to close will overwrite any other changes.

- **Don't care** semantics
  - Some updates may be lost unpredictably.
  - This may be okay, if speed is favoured over consistency.
  - If consistency is critical, an DBMS should be used anyway.

Distributed Naming

In a distributed system, we must integrate **at least two systems into one namespace**. This could be achieved by including the server name in the name space, such as 'machine:symbolic-file-name'. This destroys **location transparency**, which in turn would create migration issues.

Mount Points

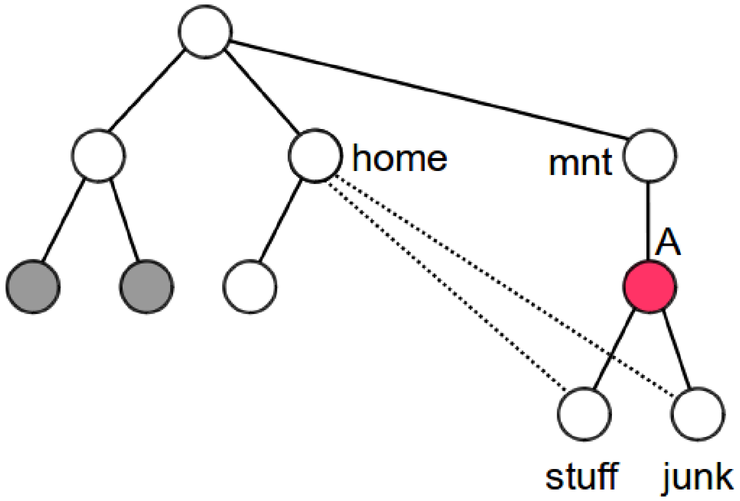
An alternative is to combine the symbolic naming spaces by **mounting** server directories as directories of the client. This requires **cooperation** between the hosts:

- On one host, the file system needs to have a **mount point**.
- On the other host, the file systems needs to have a directory that is **exported**.

However, this can also remove **location transparency**: moving a directory from one server to another means that the user must use a different file name. We can also access the same file by two different paths based on where it is accessed from: if a directory from host A is exported and mounted on host B, a file on A might be read from `/home/stuff`, whereas the same file on B might reside at `/mnt/A/stuff`.

Achieving Location Transparency

Location transparency can be achieved by combining methods: an external file system is **mounted** into a specified directory (such as `/mnt` on Linux), and its directories are then **symbolically linked** into the client's directory structure.

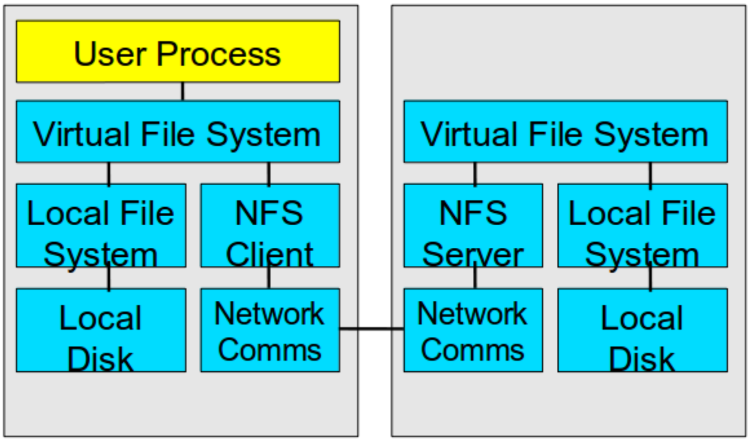


Virtual File Systems

A Unix file system is **stateful**, because files must be opened and closed; a distributed file system is usually **stateless**, for the reasons mentioned earlier. Therefore, for a client to access both local and remote files, they must use two completely different protocols: there is **no access transparency**. This can be avoided using a virtual file system layer.

A virtual file system sits atop **multiple different file systems** and provides a uniform access pattern:

- Operations on local file systems are simply passed through to them.
- Operations on remote file systems are processed by the virtual file system layer, which maintains state information and processes requests using the remote file system's stateless protocol.


User Process
Virtual File System
Local File System
NFS Client
Local Disk
Network Comms
Virtual File System
NFS Server
Local File System
Network Comms
Local Disk