Course summary
Distributed Systems, ID2201

1, Introduction

What is a ***distributed system***?

"*...one in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing*"

Why do we build distributed systems?

- Sharing of recourses;
- Data, computers and resources, users (clients) are geographically distributed;
- To improve/achieve performance, scalability, availability, fault tolerance

---

General Design Issues of Distributed Systems

- Quality
  - *Functional requirements* – what it does: functions, usage scenarios, use cases, APIs.
  - *Non-functional requirements* – how good it is: performance, scalability and elasticity, complexity, availability, fault-tolerance, consistency
- Communication latency
- Failures
- Replication and Consistency
- Dynamicity (in infrastructure, resources, workload, etc.)

4, Remote invocation

Idempotent operations

In computing, an idempotent operation is one that has no additional effect if it is called more than once with the same input parameters. For example, removing an item from a set can be considered an idempotent operation on the set. An idempotent operation can be repeated an arbitrary number of times and the result will be the same as if it had been done only once. In arithmetic, adding zero to a number is idempotent.

If operations are not idempotent, the server must make sure that the same request is not executed twice. Keep a history of all request and the replies. If a request is resent the same reply can be sent without re-execution.

At-most-once or At-least-once

at-most-once: the request has been executed once. Implemented using a history or simply not re-sending requests.

at-least-once: the request has been executed at least once. No need for a history, simply resend requests until a reply is received.

---

at-most-once delivery
means that for each message handed to the mechanism, that message is delivered zero or one times; in more casual terms it means that messages may be lost.

at-least-once delivery
means that for each message handed to the mechanism potentially multiple attempts are made at delivering it, such that at least one succeeds; again, in more casual terms this means that messages may be duplicated but not lost.

exactly-once delivery
means that for each message handed to the mechanism exactly one delivery is made to the recipient; the message can neither be lost nor duplicated.

Comparison

The first one is the cheapest—highest performance, least implementation overhead—because it can be done in a fire-and-forget fashion without keeping state at the sending end or in the transport mechanism. The second one requires retries to counter transport losses, which means keeping state at the sending end

and having an acknowledgement mechanism at the receiving end. The third is most expensive—and has consequently worst performance—because in addition to the second it requires state to be kept at the receiving end in order to filter out duplicate deliveries.

Pros and cons:

- **at-most-once without re-sending requests:**
  -simple to implement, not fault-tolerant

- **at-most-once with history:**
  -expensive to implement, fault-tolerant

- **at-least-once:**
  -simple to implement, fault-tolerant

RPC and RMI

Remote Procedure Call (RPC)

- Procedure interface; message passing implementation
- RPC allows calling procedures over a network

Remote Method Invocation (RMI)

- RMI is an object-oriented analog of RPC
- invokes objects' methods over a network

Remote Procedure Call

RPC is a mechanism that allows a program running on one computer (VM) to cause a procedure to be executed on another computer (VM) without the programmer needing to explicitly code for this.

Two processes involved:

- **Caller (RPC client)** is a **calling process** that initiates an RPC to a server.

- **Callee (RPC server)** is a **called process** that accepts the call.

Each RPC is executed in a **separate process (thread)** on the server side

An RPC is a synchronous operation.

- The caller is suspended until the results of the remote procedure are returned.

Java RMI (Remote Method Invocation)

- similar to RPC but:
  - we now invoke methods of remote objects
  - at-most-once semantics

- Objects can be passed as arguments, how should this be done?
  - by value
  - by reference

We can do either:

- A remote object is passed as a reference (by reference) i.e. it remains as at the original place where it was created.

- A serializable object is passed as a copy (by value) i.e. the object is duplicated.

Java RMI is a mechanism that allows a thread in one JVM to invoke a method on an object located in another JVM.

– Provides ***Java native ORB (Object Request Broker)***
The Java RMI facility allows applications or applets running on different JVMs, to interact with each other by invoking remote methods:

- – Remote reference (stub) is treated as local object.
- – Method invocation on the reference causes the method to be executed on the remote JVM.
- – Serialized arguments and return values are passed over network connections.
- – Uses ***Object streams*** to pass objects “by value”.

Location transparency: invoke a method on a stub like on a local object. Remote resources are accessed using location independent names.

Location awareness: the stub makes remote call across a network and returns a results via stack.

Locating Objects

How does a caller get a reference to a remote object, i.e. stub?

One approach is to use a ***Naming Service***:

• Associate a unique name with an object.
• Bind the name to the object at the Naming Service.

– The record typically includes name, class name, object reference (i.e. location information) and other information to create a stub.

• The client looks up the object by name in the Naming Service.

5, Indirect communication

Time and Space

In ***direct communication*** sender and receivers exist in the same time and know of each other.

In ***indirect communication*** we relax these requirements: sender and receiver are uncoupled (or decoupled).

Time and space uncoupling

Time uncoupling: a sender can send a message even if the receiver is still not available. The message is stored and picked up at a later moment.

Space uncoupling: a sender can send a message but does not know to whom it is sending nor if more than one, if anyone, will receive the message.

	Time coupled	**Time uncoupled**
Space coupled	Direct communication	Message storing systems
Space uncoupled	Broadcast	Group communication

Group communication

More than simple multicast:

- the group is well defined and managed
- ordered delivery of messages
- fault tolerant, delivery guarantees
- handles multiple senders

Broadcast vs Multicast

In a ***broadcast*** service, no one keeps track of who is listening, cf. radio broadcast, IP broadcast 192.168.1.255 etc.

In a ***multicast*** service, the sender is sending a message to a specific group, the system keeps track of who should receive the message cf. IP-multicast 239.1.1.1

IP-multicast is unreliable, does not keep track of members nor order of messages when we have several senders.

Ordering of messages

- ***FIFO order***: All messages are received in the order sent.
- ***Causal order***: If a message m2 is sent as a consequence of a message m1 (i.e. a process has seen m1 and then sends m2), then all members should see m1 before m2.
- ***Total order***: All members will see messages in exactly the same order.

Causal ordering does not strictly imply FIFO, a process can send m1 and then m2 but has not yet seen its own message m1.

Publish-subscribe

Processes *publish events*, not knowing if anyone is interested.

A process can *subscribe on events* of a given class.

Limited guarantees on ordering or reliability - scales well. Used when the flow of events is very high: trading platforms, news feeds etc.

Subscriptions

- ***Channel***: events are published to channel that processes can subscribe to.
- ***Topic (Subject)***: an event is published given one or more topics (#foo), if topics are structured in a hierarchy processes can choose to subscribe on a topic or a sub-topic.
- ***Content***: subscribers specify properties of the content, more general - harder to implement
- ***Type***: used by object oriented languages, subscribe on event of a particular class

---

Broker networks

A network of ***brokers*** that distribute events; clients connect to the brokers.

The ***network of brokers*** form an ***overlay network*** that can route events.

Given a broker network, how do we distribute events from publishers to subscribers?

Flooding

- send all published event to all nodes in the network
- matching is done by each node
- can be implemented using underlying network multicast

Simple but inefficient - events are distributed even if no one is subscribing.

Alternative - let the subscriptions flood the network and publishers keep track of subscribers.

Filtering

Let the brokers take a more active part in the publishing of events.

- a subscription is sent to the closest broker
- brokers share information about subscriptions

- a broker knows which neighboring brokers should be sent published events

Requires a more stable broker network

Advertisement

Let the publishers advertise that they will publish events of a particular class.

- publishers advertise event classes
- advertisements are propagated in the network
- subscribers contact publishers if they are interested

Can be combined with filtering

Message queues

A ***queue*** (normally FIFO) is an object that is independent of processes.

Processes can:

- send messages to a queue
- receive messages from a queue
- poll a queue
- be notified by a queue

More structured and reliable, compared to pub/sub systems.

Shared memory

Why not make it simple - if concurrent threads in a program can communicate using a shared memory why would it not be possible for distributed process to do the same?

A distributed shared memory - DSM.

Shared memory is mostly used in shared-memory multiprocessors, multi-core processors and computing clusters where all nodes are equal and run the same operating system.

---

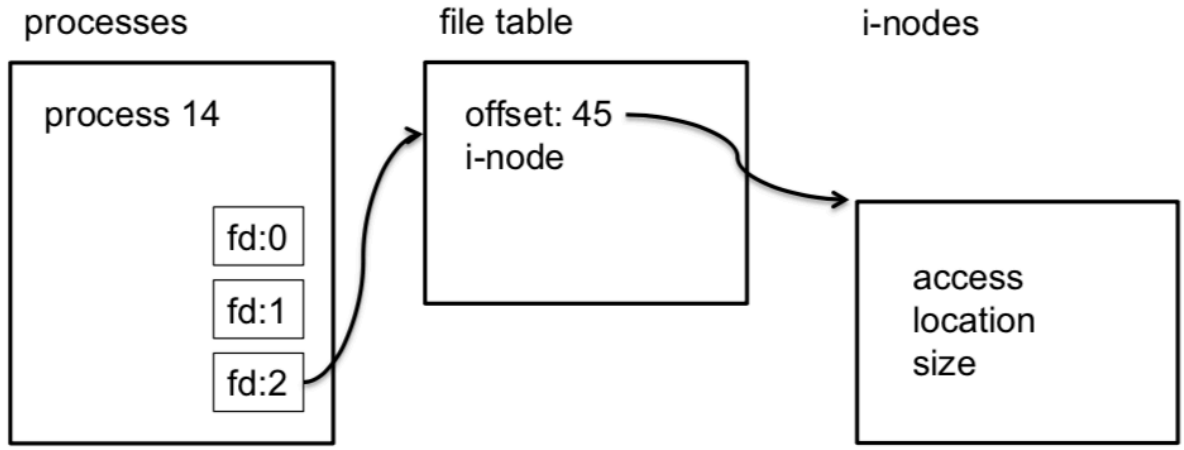
6. Distributed file systems

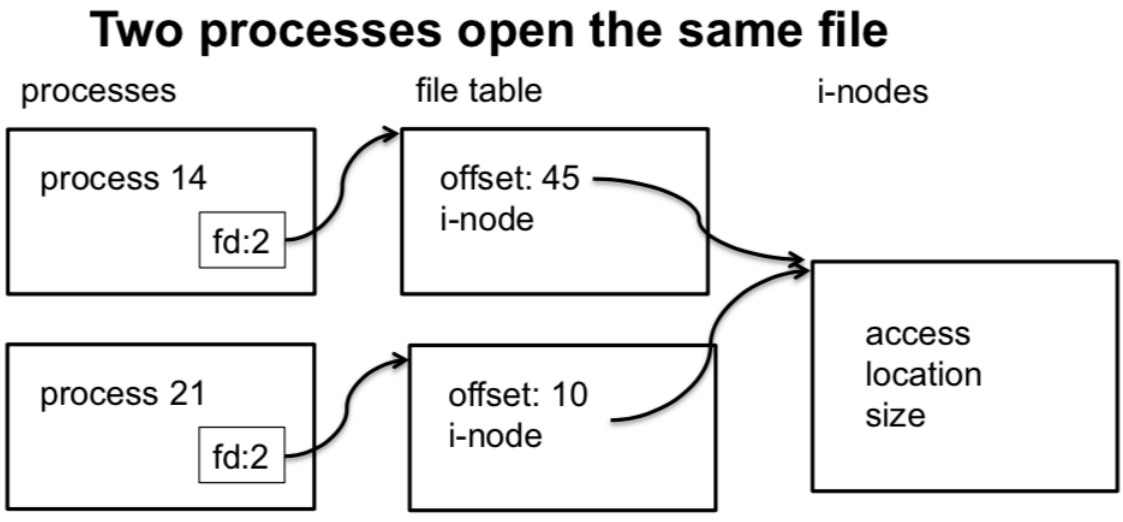
Descriptors, table entries and i-nodes

A process holds a set of open ***file descriptors***, each descriptor holds a pointer to a table of open files.

The file table entries holds a ***position*** and a pointer to an ***inode*** (index node).

The inode holds information about where file blocks are allocated.





Two processes that open the same file will have independent file table entries.

All threads in a process (or even if process is *forked*) share the same file descriptors and thus share the file table entry.

6. Name Services

What's a name service

A service that provides information about remote resources given a name.

Resolving:

A name is ***resolved***, resulting in information about an object, often the address so that one can access the object.

Address:

An ***address***, at one level, could be a name on a lower level.

---

7. Time and Clocks

In an asynchronous system clocks cannot be completely trusted.

Nodes will not be completely synchronized.

We still need to:

- talk about before and after
- order events
- agree on order

Logical time

All events in one process are ordered.

The sending of a message occurs before the receiving of the message.

Events in a distributed system are partially ordered.

The order is called ***happened before***.

Logical time gives us a tool to talk about ordering without having to synchronize clocks.

Lamport clock

One counter per process:

- initially set to 0
- each process increment only its own clock
- sent messages are tagged with time stamp

Receiving a message:

• set the clock to the greatest of the internal clock and the time stamp of the message

---

If *e1* **happened before** *e2* then the time stamp of *e1* is less than the time stamp of *e2*. ***e1 happen-before e2 → L(e1) < L(e2).***

Vector clock

A ***vector*** with one counter per process:

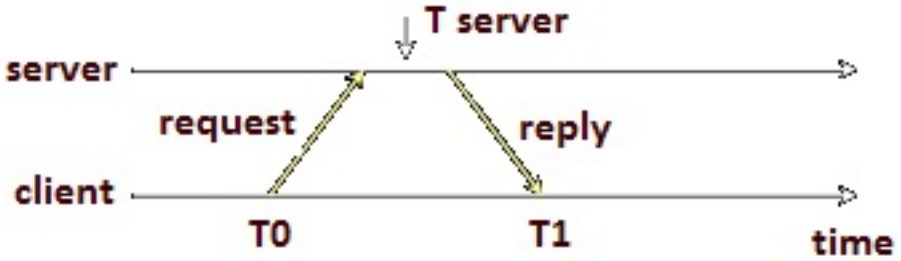
- initially set to <0,....>
- each process increments only its own index
- sent messages are tagged with a vector

Receiving a message:

• **merge** the internal clock and the time stamp of the message

---

Cristian's Algorithm is a clock synchronization algorithm that is used to synchronize time with a time server by client processes.



1) The process on the client machine sends the request for fetching clock time(time at server) to the Clock Server at time T_0.

2) The Clock Server listens to the request made by the client process and returns the response in form of clock server time.

3) The client process fetches the response from the Clock Server at time T_1 and calculates the synchronized client clock time using the formula given below.

$$T_{client} = T_{server} + \frac{(T_1 - T_0)}{2}$$

T_{client} refers to the synchronized clock time,

T_{server} refers to the clock time returned by the server,

T_0 refers to the time at which request was sent by the client process,

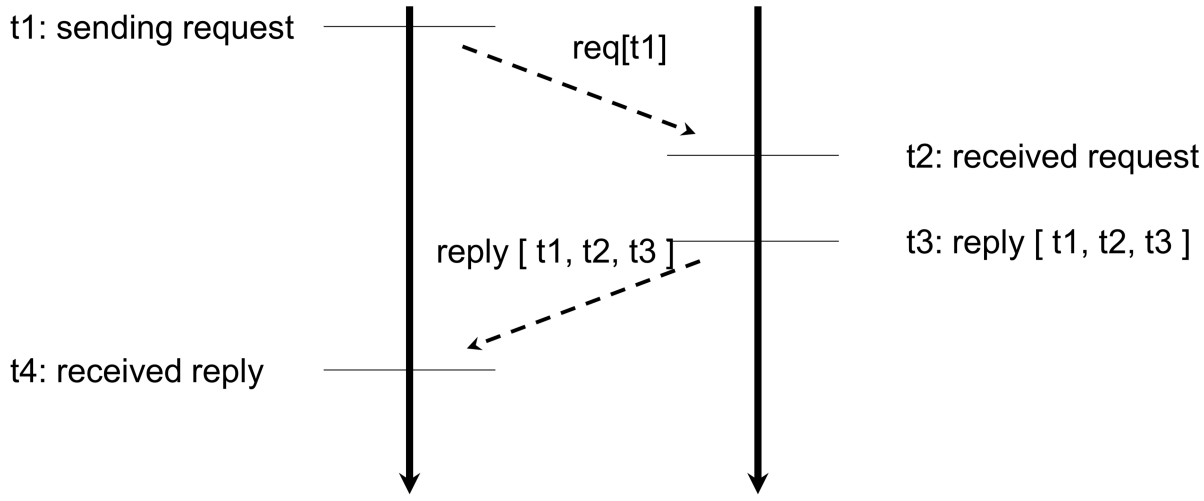
T_1 refers to the time at which response was received by the client process

---

Network Time Protocol (NTP)

- An architecture targeting reliability and wide area networks.
- A hierarchy of servers: stratum-1 connected to external sources.
- Fault tolerant: servers can be degraded to lower stratum if external source is lost, client can connect to secondary servers.
- Several synchronization protocols: LAN multicast, request reply and synchronous.

NTP



Similar to Christian's but with better estimate of delay.
Stateless, no need to record *r*.

Berkeley algorithm

Used to synchronize a network of nodes.

- – send requests to all nodes
- – collect it and calculate an *average* time T
- – send out individual deltas to each node

Algorithm

1) An individual node is chosen as the master node from a pool nodes in the network. This node is the main node in the network which acts as a master and rest of the nodes act as slaves. Master node is chosen using an election process/leader election algorithm.

2) Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

---

8. Global state

Time is very much related to the notion of ***global state***.

If we cannot agree on a time, how should we agree on a global state?

The ***history*** of a process is a sequence of events: <p0, p1, ..pn>

The ***state*** of a process is a description of the process after (before) an event.

A state corresponds to a ***finite prefix of the process's history***

A ***cut*** is a subset in the global history up to a specific event in each history.

Consistent cuts

For each event *e* in the cut:

- if *f* happened before *e* then
- *f* is also in the cut.

Consistent global state

A ***consistent cut*** corresponds to a ***consistent global state***.

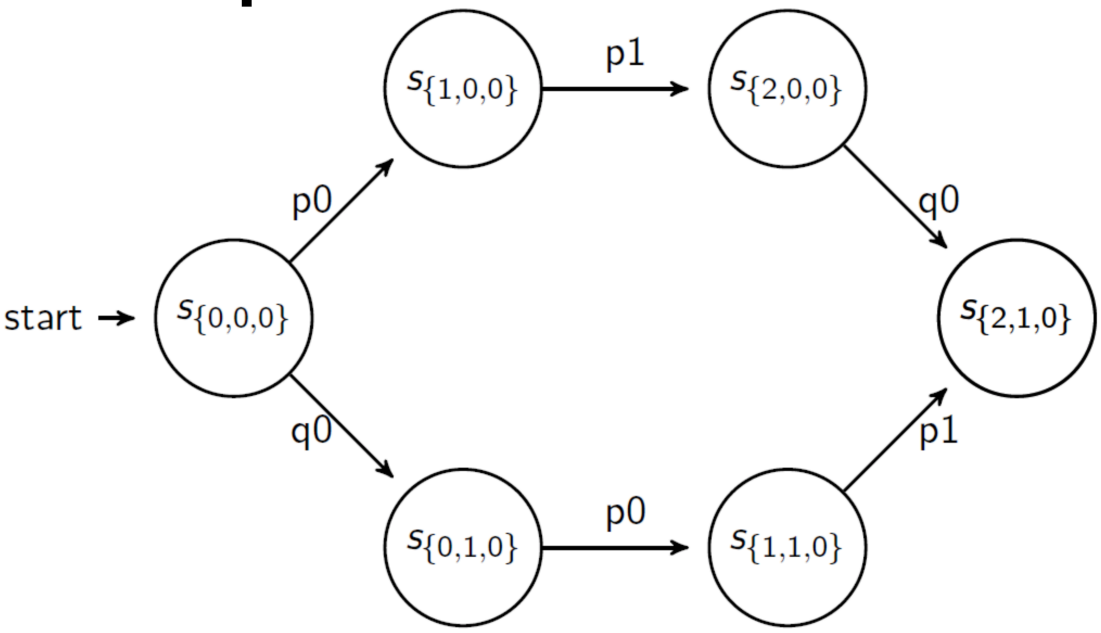
- it is a ***possible state*** without contradictions
- it is ***consistent with*** the ***actual execution***
- the actual execution might not have passed through the state, even though it's consistent

---

Linearization

- A ***run*** is a total ordering of all events in a global history that is consistent with each local history.
- A ***linearization*** or ***consistent run*** is a run that describes transitions between ***consistent global states***.
- A state *S'* is ***reachable*** from state *S* if there is a linearization from *S* to *S'*.

Possible paths



Each path is a consistent run, a linearization, one of which the execution actually took.

Global state predicate

A global state predicate is a property that is true or false for a global state.

- ***Safety*** - a predicate is never (or always) true in any state.
- ***Liveness*** - a predicate that eventually evaluates to true.

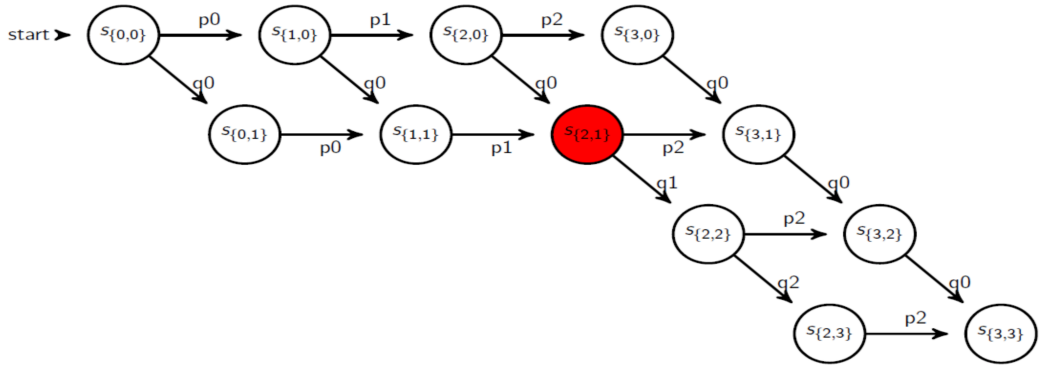
Stable and non-stable

We differentiate between:

- ***Stable***: if a predicate is true it remains true for all reachable states
- ***Non-stable***: if a predicate can become true and then later become false

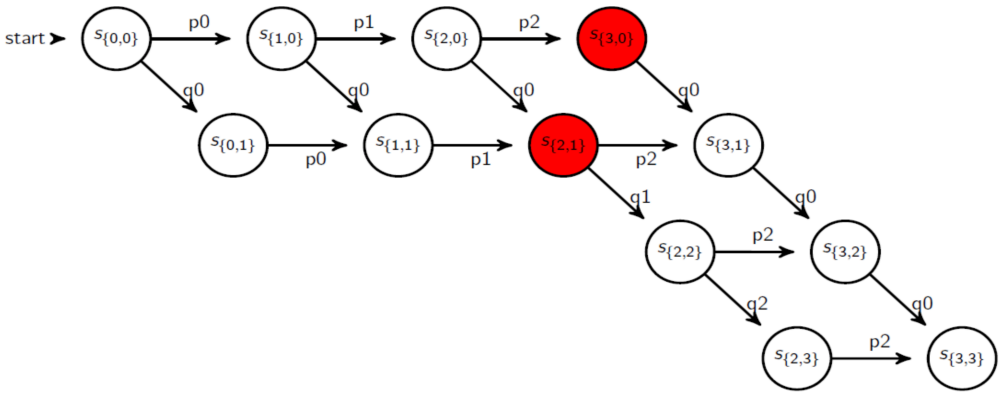
---

Possibly true



If a predicate is true in a consistent global state of the lattice, then it is *possibly true* in the execution.

Definitely true



If we cannot find a path from the initial state to the final state without reaching a state for which a predicate is true then the predicate is *definitely true* during the execution.

Snapshot - Chandy and Lamport

A node initiates a snapshot when it receives a *marker*.

- Record the local state and
- send a *marker* on all out going channels.
- Record all incoming messages on each channel, until you receive a marker.
- When the last channel is closed you have a local and a set of messages.

Ask one node to initiate the snapshot, collect all local states and messages and construct a global state.

Snapshot

- Allows us to collect a global state during execution.
- Only allows us to determine stable predicates.

---

9. Coordination

Coordination in a distributed system:

- no fixed coordinator
- no shared memory
- failure of nodes and networks

The hardest problem is often knowing who is alive.

Failure detectors

How do we detect that a process has crashed and how reliable can the result be?

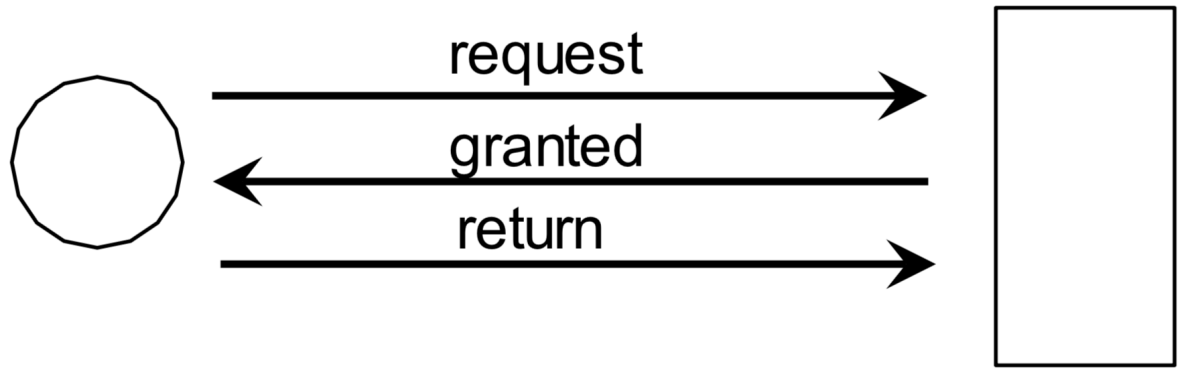
- unreliable: result in *unsuspected* or *suspected* failure
- reliable: result in unsuspected or *failed*

Reliable detectors are only possible in synchronous systems.

Mutual exclusion algorithms

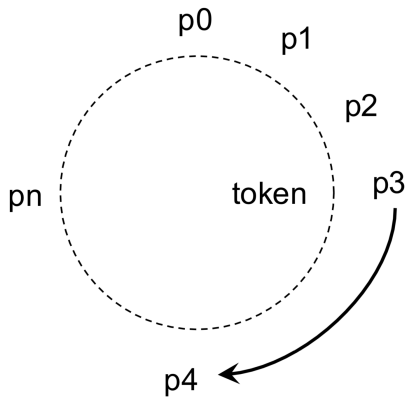
A central server

- *request* a token from the server
- *wait* for a token that grants access
- *enter* critical section and execute in it
- *exit* critical section and *return the token*



A ring based approach

Pass a token around the ring



- pass a token around
- before entering the critical section - remove the token
- when leaving the critical section - release the token

A distributed approach

Why not complicate things?

To request entry:

- *ask* all other nodes for permission
- *wait* for all replies (save all requests from other nodes)
- *enter* the critical section
- *leave* the critical section (give permission to saved request)

otherwise:

- *give* permission to anyone

Ricart and Agrawala

A request contains a *Lamport time stamp* and a *process identifier*. Request can be ordered based on the time stamp and, if time stamps are equal, the process identifier.

When you're waiting for permissions and receive a request from another node:

- if the request is *smaller*, then give permission
- otherwise, save request

Maekawa's Algorithm for Mutual Exclusion in Distributed System

With Maekawa's algorithm, a site does not request permission from every other site but from a subset of sites which is called quorum.

To request entry:

- *ask* all nodes your quorum for permission
- *wait* for all to vote for you:
  - queue requests from other nodes
- *enter* the critical section
- *leave* the critical section:
  - return all votes
  - vote for the first request if any in the queue

otherwise:

- if you have not voted:
  - vote for the first node to send a request
- if you have voted:
  - wait for your vote to return, queue requests from other nodes
  - when your vote is returned, vote for the first request if any in the queue

---

Election

The bully algorithm

Electing a new leader when the current leader has died.

- assumes we have *reliable failure detectors*
- all nodes know the nodes with higher priority

Assume we give priority to the nodes with lower process identifiers.

This algorithm applies to system where every process can send a message to every other process in the system.

Algorithm – Suppose process P sends a message to the coordinator.

1. If coordinator does not respond to P within a time interval T, then it is assumed that coordinator has failed.

2. Now process P sends election message to every process with high priority number.

3. It waits for responses, if no one responds for time interval T then process P elects itself as a coordinator.

4. Then it sends a message to all lower priority number processes that it is elected as their new coordinator.

5. However, if an answer is received within time T from any other process Q,

    - (I) Process P again waits for time interval T' to receive another message from Q that it has been elected as coordinator.

    - (II) If Q doesn't responds within time interval T' then it is assumed to have failed and algorithm is restarted.

Group communication

Reliable multicast

When receiving a message, forward it to all nodes.

Watch out for duplicates.

A lot of messages!

Reliable multicast often implemented by detecting failed nodes and then fix the problem.

---

10. Transactions and currency control

Atomic operations

Even if we have a distributed system that provides atomic operations we sometimes want to group a sequence of operations in a transaction where:

- either all are executed or
- none is executed
- even if a node crash

Recoverable objects: a server can store information in persistent memory (the file system) and can recover objects when restarted.

ACID

- ***Atomic*** - either all or nothing
- ***Consistent*** - the server should be left in a consistent state
- ***Isolation*** - total order of transactions
- ***Durability*** - persistent, once acknowledged

Serial equivalence

Used in distributed system technology to describe the effect that a number of parallel or nested transactions have on the state of such a system. When such transactions are carried out their effect is said to be serially equivalent if the state of the system is the same as if they were carried out sequentially.

Two transactions are ***serially equivalent*** if, and only if, *all pairs of conflicting operations* of the two transactions are executed in *the same order at all of the objects they both access*.

A **dirty read** (aka uncommitted dependency) occurs when a transaction is allowed to **read** data that has been modified by another running transaction and not yet committed.

Recoverability from aborts

In order to recover from an aborting transaction: a transaction must not commit if it has done a *dirty read*.

In order to avoid cascading aborts we should suspend when (before) we read a dirty value.

Dirty read

- To be ***recoverable*** a transaction must suspend its commit operation if it has performed a dirty read.
- If a transaction aborts, any suspended transaction must be aborted.
- To prevent cascading aborts, a transaction could be prevented from performing a read operation of a non-committed value.
    - Once the value is committed or the previous transaction aborts the execution can continue.
    - We will restrict concurrency.

Also write operations must be delayed in order to be able to recover from an aborting transaction.

Strict execution

- In general, both read and write operations must be delayed until all previous transactions containing write operations have been aborted or committed.
- ***Strict execution*** enforces isolation, no visible effects until commit.

To increase concurrency while preserving serial equivalence, three methods are used.

- ***locking***: simple but dangerous
- ***optimistic***: large overhead if many conflicts
- ***timestamp***: ok, if time would be simple

Locks

Idea - lock all objects to prevent other transaction to read from or write to the same objects.

To guarantee serial equivalence we require ***two phase locking***:

• lock objects in any order,
• release locks in any order,
• commit

We are not allowed to take a new lock if a lock has been released.

Does not handle the problem with dirty read and premature write.

Two-version locking

Similar idea but now with: read, write and commit locks.

- A read lock is allowed unless a commit lock is taken.
- One write lock is allowed if no commit lock is taken (i.e. even if read locks are taken)
- Written values are held local to the transaction and are not visible before commit.
- A write lock can be promoted to a commit lock if there are no read locks.
- When a transaction commits it tries to promote write locks to commit locks.

Disadvantages of locking

- Locking is an overhead not present in a non-concurrent system. You're paying even if there is no conflict.

- There is always the risk of deadlock or the locking scheme is so restricted that it prevents concurrency.
- To avoid cascading aborts, locks must be held to the end of the transaction.

Optimistic concurrency control

- Perform transaction in a copy of an object, hoping that no other transaction will interfere.
- When performing a commit operation *the validity* is controlled.
- If transaction is *valid*, the values written to permanent storage.
- A transaction passes three phases:

    1. Working
    2. Validation: if passed, commit
    3. Update

    Validation and update are a critical section

Validation

Uses the read-write conflict rules: read-write sets of two overlapping transactions must be disjoint

Tv	Ti	Rule
write	read	1. Ti must not read objects written by Tv.
read	write	2. Tv must not read objects written by Ti.
write	write	3. Ti must not write objects written by Tv and Tv must not write objects written by Ti.

Transaction is assigned a transaction number in its validation phase: a transaction finishes its working phase after all transactions with lower numbers

---

Backward validation of Tv:

- read operations of earlier overlapping transactions (performed before validation of Tv) cannot be affected by the writes of Tv. The validation checks Tv's read set against write sets of earlier transactions, failing if there is any conflict;

Forward validation of Tv:

- write set of Tv is compared with the read sets of all overlapping active transactions;
- differently from backward validation, in forward validation there are choices of which transaction to abort (Tv or any of the conflicting active transactions);

Optimistic - pros and cons

Works well if there are no conflicts.

- Backward validation: simpler to implement, need to save all write operations
- Forward validation: moving target, flexible if not successful

---

Timestamp ordering

Each transaction is given a *time stamp* when started.

Operations are validated when performed:

- writing only if no later transaction has read or written
- reading only if no later transaction has written

Timestamp ordering implementation

Each objects keep a list of *tentative*, not committed, versions of the value.

- Write operations can be inserted in the right order, no fear for deadlocks.

- Read operations wait for tentative values to be committed.
- If an operation ***arrives too late*** the transaction is aborted.

Too late:

1. Tc must not write an object that has been read by any Ti such that Ti > Tc.
2. Tc must not write an object that has been written by any Ti where Ti >Tc.
3. Tc must not read an object that has been written by any Ti where Ti > Tc.

Timestamp ordering

- consistency is checked when the operation is performed
- commit is always successful
- an operation can suspend or arrive too late
- read operations will succeed, suspend or arrive too late
- write operations will succeed or arrive too late
- multiversion timestamp can improve performance

Transactions group sequences of operations into a ACID operation.

- Atomic: all or nothing
- Consistent: leave the server in a consistent state
- Isolation: same result as having executed in sequence
- Durability: safe even if server crashes
- problem is how to increase concurrency
- need to preserve serial equivalence
- aborting transactions is a problem
- how do we maximize concurrency

Implementations: locking, optimistic concurrency control, timestamps

---

• Locks are used to order transactions that access the same objects according to the order of arrival of their operations at the objects.

• Optimistic concurrency control allows transactions to proceed until they are ready to commit, whereupon a check is made to see whether they have performed conflicting operations on objects.

• Timestamp ordering uses timestamps to order transactions that access the same objects according to their starting times.

---

11. Distributed transactions

Problem

- Several independent transaction servers should be coordinated in one transaction.
- How do we coordinate operations to guarantee serial equivalence?

Two-phase commit

- phase one (voting): ask participants to vote for commit or abort
  - if voting for commit, one has to be able to commit even after a node crash
  - if anyone aborts all must abort

- phase two (completion): inform all participants of the result

---

Consensus

Two-phase commit is a consensus protocol but:

- all servers must vote
- if any server wants to abort then we abort

---

12. Replication

Why replicate?

Performance

- latency
- throughput

Availability
• service respond despite crashes

Fault tolerance
• service consistent despite failures

A replicated service should, to the users, look like a non-replicated service.

What do we mean by “look like”?

- linearizable
- sequential consistency
- causal consistency
- eventual consistency

Linearizable

A replicated service is said to be ***linearizable*** if for any execution there is some interleaving of operations that:

- meets the specification of a non-replicated service
- matches the real time order of operations in the real execution

> *All operations seem to have happened: atomically,* ***at the correct time****, one after the other.*
>
> *A register that provides linearizability is called* ***an atomic register***

Sequential consistency

A replicated service is said to be ***sequential consistent*** if for any execution there is some interleaving of operations that:

- meets the specification of a non-replicated service
- matches the *program order* of operations in the real execution

---

Eventual consistency

There exist a total order that will eventually be visible to all.

Passive and active replication

- ***Passive replication***: one primary server and several backup servers
- ***Active replication***: servers on equal term

Passive replication – Pros and cons

Pros

- All operations passes through a primary that linearize operations.
- Works even if execution is non-deterministic

Cons

- Delivering state change can be costly.
- Replicas under-utilized.
- View-synchrony and leader election could be expensive.

Active replication - consistency

Sequential consistency:

- All replicas execute the same sequence of operations.
- All replicas produce the same answer.

Linearizability:

- Total order multicast does not guarantee real-time order.
- Linearizability not guaranteed if front-end acknowledge operation before it has been processed by replicas.

Active replication – Pros and cons

Pros

- No need to send state changes.
- No need to change existing servers.
- Read request could possibly be sent directly to replicas.
- Could survive Byzantine failures.

Cons:

- Requires total order multicast.
- Requires deterministic execution.

---

13. Distributed Hash Tables

• Large scale data bases
– hundreds of servers

• High churn rate
– servers will come and go

• Benefits

- fault tolerant
- high performance
- self-administrating

Design issues:

• Identify : how to uniquely identify an object

• Store: how to distribute objects among servers

• Route: how to find an object

Key distribution – direct map

Direct map of keys to identifiers (buckets) gives a non-uniform (uneven) distribution of keys among buckets

---

Key distribution – hashing keys

A cryptographic hash function gives a uniform (even) distribution of the keys among buckets.

Stabilization

Stabilization is run periodically: allow nodes to be inserted concurrently.

Inserted node will take over responsibility for part of a segments

Responsibility: From your predecessor to your number
