Architectures of Distributed Systems

Two main types of DS architecture:

Tightly Coupled
Highly integrated
May look as a single computer
Will share some aspect of the system (e.g memory)

DSM - Distributed shared memory. An abstraction providing the illusion of a single shared memory. Spares the programmer from handling message passing.
Issues with DSM:

- Machines are still connected via a network, so we have to consider latency between requests and completion.
- We need a method to keep track of the location of the shared data.
- How do we enable concurrent access?
- Data may need to be coherently replicated across multiple machines

Loosely Coupled
"Share nothing" systems

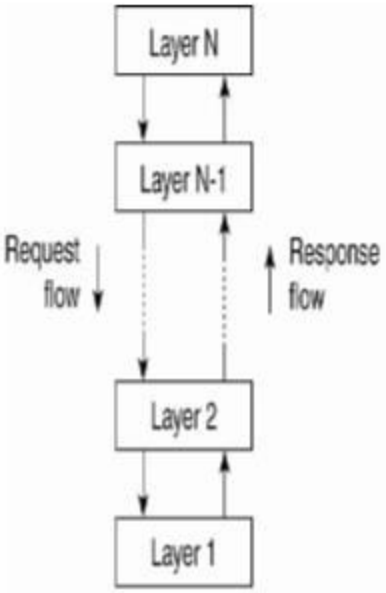
- Client-Server
- Peer-to-peer

Based on the logical organization of the components of distributed systems.

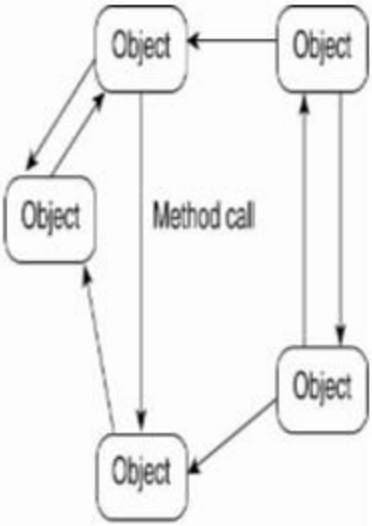
- Layered
- Object-Oriented
- Data-centered
- Event-based

Architectural Styles for Loosely-Coupled Systems

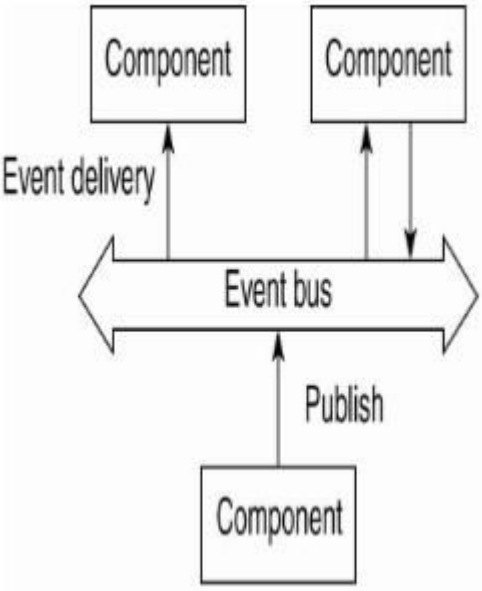
- **Layered** - Requests/Responses passed up/down the stack
- **OO** - RPC's. Performance restrictive, remote calls are quite network intensive (latency).
- **Event-based (publish-subscribe)** - Forum. When a component does something, publish to an event bus. Subscribed components receive notification of the event.
- **Shared-data space** - shared (persistent) data space



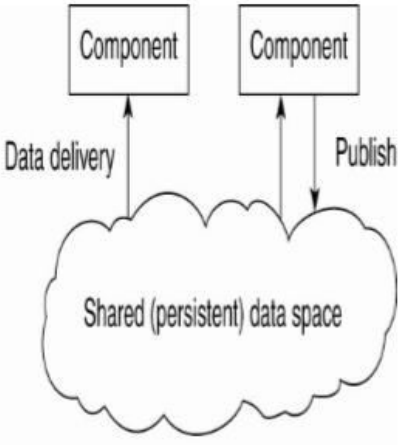
Layered



Object Oriented



Event Driven



Shared data space

Middleware

Lies between Applications and OS.

It is a software layer that provides abstraction to hide the heterogeneity of the underlying platforms (Networks, Languages, Hardware etc).

However, the end-to-end argument implies that the whole communication process cannot be fully abstracted away from the application layer. Some functions can only be correctly implemented with the knowledge and data contained in the application layer.

Client-Server Architecture

Clients invoke individual servers.
Invocation - issue request for some data on another computer
Server returns a result
Server may need to update local data
Asymmetrical
Tends to be centralized
Usually scales poorly - bottleneck at the server. Single point of failure.
No client-client communication
Great performance, easy to administrate, non-negligible cost

Server

Passive (slave)
Waits/listens for requests
Processes requests and sends replies
Can be stateless (keeps no information between requests) or stateful (remember information)

Client

Active (master)
Sends requests
Waits for and receives server replies

P2P Architecture

Equality between all peers
All hosts can be clients and servers
Truly distributed
Resource discovery is a challenge. Who has what file?
A large number of participants can exploit resources by sharing files between them.
Easy to deploy, low cost
difficult to administer and secure

Multiple Servers

Used by large companies where more than 1 server is needed to handle a large workload (google/facebook etc)
Multiple servers combine into a distributed **service**

Proxy Server
Connects servers, acts as an intermediate node in requests. Ask the proxy server to ask it's servers where a file is etc.

Mobile code
Mobile code is software that is transferred between systems (i.e via a network) and executed on the client without being installed. Applets are sent to the client, usually written in a scripting language such as JavaScript. The client then interacts with the applet, requiring little/no further input from the server.

Thin Client
Computation done at a remote location over a network. Very little functionality on the client end. Very cheap.

Factors Affecting Process Interaction
- Performance of communication channels (latency)
- Computer clocks and timing/concurrency
- Synchronous problems - Process execution speeds, message transmission delay, clock drift rate
- Asynchronous problems - All of the above, but in an unbounded arbitrary time period

We have to accept the uncertainty of the communications channel, we cannot eliminate it completely.
We have to deal with messages being lost due to unreliable comms channels, and also the potential for faulty components giving wrong answers.

Remember the two Generals problem? Solved via redundancy (sending lots of messages, small probability of none being received) This shows how we can **solve the problem of an unreliable communications channel**.

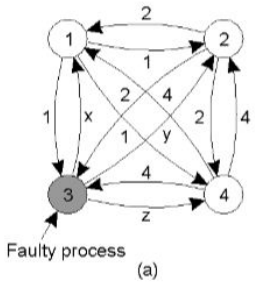
However, how do we handle the potential for faulty components?

Byzantine Generals problem - Lots of generals outside a village communicating by messenger (Assume we use the above strategy for pseudo-reliable communication). Some generals may be traitors. Must agree on a common plan of action, without interference from the traitor generals. There is a commanding officer who sends an order to each general, and the loyal Generals must choose whether to obey or disobey this order.
Solution - All generals send the message they received from the commander to all other generals. Combine their knowledge and ignore the minority.

No solution is possible if n<=3f
where n is number of generals and f is number of traitors.
With 1 traitor, we need a total of 4 generals to isolate the taitor.

Agreement in Faulty Systems (1)

(see Tanenbaum, figure 8-5)



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

- The Byzantine generals problem for 3 loyal generals and 1 traitor.

a) The generals announce their troop strengths (in units of 1 kilosoldiers).
b) The vectors that each general assembles based on (a); this is propagated to all other generals.
c) The vectors that each general receives in step 3; there is an agreement about 1,2,4.

Could also use signed messages to make sure that messages from loyal generals cannot be forged

Mutual Exclusion

Mutual Exclusion in computer science is ensuring that two processes are never in their critical sections at the same time.
In a single machine we can implement mutual exclusion between processes using semaphores.

We can implement semaphores in a single machine by either inhibiting interrupts, or by using clever instructions such as test and set.

However, these techniques only work on singular machines which use a shared memory space. Physically distributed machines require a different approach.

Dedicated Server

Single point in charge of coordination. The simplest method for distributed mutual exclusion.
Election - Getting multiple distributed computers to agree who is in charge.

The server gives out tokens to clients in response to requests. The client releases this token on exit, although leaving token release purely in the hands of the client is a bad idea, as they could wait an undetermined amount of time before releasing. If all tokens are currently being held, new requests are queued to wait for a token to be released.

Although very simple, using a dedicated/centralized server has the downside of being a single point of failure, and potential bottleneck for the system.

An alternative is **election.**

Election

Requires a distinguished process to coordinate, and implement mutual exclusion.
From a fixed set of processes, we want to elect the process with the largest identifier.
Each process should know, and agree upon the result of the election.

Ring-based Election

Ring of processes with identifiers, arranged in a logical ring (each process knows which one comes next)
Assumes no failures, and that the system is asynchronous.
Initially every process is **not** a participant. The instigating process declares itself as a participant, and sends it's identifier in an election message to the next process.
The message is passed around the ring, and if a process has a higher identifier, it sends it's own identifier, until **the highest identified process receives its own identifier.**
Then we must declare who is in charge, by the elected process sending a message around the ring.

This method has a worst case scenario of 3n-1 messages being sent, if the elected process is right before the process which initiated the election.

A fairly large downside to this method is that it is not very failure tolerant at all, as if one process fails we have to rebuild the ring

Bully Algorithm

Every process is aware of the identifiers of other processes and can communicate with them.
Uses timeouts to detect failures (synchronous)

3 Types of Message

- An Election message - initiation
- Answer Message - Sent in response to an election message
- Co-ordinator message - sent to identify elected process

The initiating process sends an election message to all processes it is aware of which have a higher ID than itself.
All these processes then send answer messages back to the initiating process, informing it of their participance. They then broadcast election messages to all other processes with higher ID's
Once a process sends out an election message and doesn't get a reply, it must be the highest. It then broadcasts a co-ordinator message, informing everyone it is the elected process.

Timeouts are implemented to account for any potential crashes among the processes.

Middleware

A software layer to provide abstraction and mask details such as heterogeneity of the network, the details of the hardware, OS and underlying programming languages etc. Acts as a 'bridge' between the OS / the network / a database and an application.

Logical Clock

A logical clock is a method for capturing the 'happened-before' relation between events. It does not need to bear any relation to a physical clock, and is simply a monotonically increasing software counter. This means that its value may only go up, or remain the same - *never* decrease.

Each process p_i has it's own logical clock L_i. $L_i(e)$ denotes the 'time' an event occurred at a given process i. L(e) denotes the 'time' it occurred at whatever process the event happened.

In order to capture the happened-before relation → processes update their logical clocks as follows:

LC1: Before an event occurs increment the logical clock.

LC2: a) When a process sends a message it appends its' logical clock.
b) On receiving a message the process sets its' logical clock to the max of its current clock, and that of the clock sent in the message. After it has done this it increments this value for the occurrence of the event *receive*

If an event **e → e'**, then L(e) < L(e'). However, we cannot infer **e → e'** from knowing L(e) < L(e').

(*Adapted from TVS section 14.4 page 624, feel free to make suggestions or corrections*)

Vector Clock

Vector clocks were created as a means to overcome the limitation of logical clocks that dictates that we cannot infer **e** → **e'** from knowing L(e) < L(e').

Vector clocks for a system of N processes are N-ary integer tuples. Each process has it's own vector clock V_i used to attach timestamps to events in its own process.

Rules for updating the clocks:

VC1: Initialise $V_i[j] = 0$ for i,j in 0..N
VC2: Before an event **e** in process p_i: $V_i[i]$++ (including receiving a message)
VC3: p_i includes its' clock in each message it sends, denoted by *t*
VC4: When a message is received $V_i[j] = \max(V_i[j], t[j])$, for j = 0..N. This is called a **merge**

For a vector clock V_i, $V_i[i]$ is the number of events the p_i has timestamped.

And $V_i[j]$ (j !=i) is the number for events that have occurred at p_j that may have affected p_i. By this point P_j may have timestamped more events, but no message has been sent from p_j to p_i to indicate this.

Comparing vector clocks:

V = V' iff V[j] = V'[j] for j in 0..N
V <= V' iff V[j] <= V'[j] for j in 0..N
V < V' iff V <= V' ^ V != V'

e → **e'** implies V(e) < V(e')

IDL(Interface Description Language) - used to describe a software component's interface. IDLs describe an interface in a language-independent way, enabling communication between software components that do not share a language or even the same physical machine. IDLs offers bridge between two different systems. E.g. CORBA(Common Object Request Broker Architecture), Sun, DCE(Distributed Computing Environment).

Java RMI (rmiregistry) - a place for the server to register services it offers and a place for clients to query for those services. RMI stands for remote method invocation. Objects sent as arguments or results must be serializable.

Java Servlet - simply a class which responds to a particular type of network request (HTTP). Used to implement web applications. It extends the capabilities of a server.

Byzantine Failure - Any fault presenting different symptoms to different observers. Derived from the Byzantine General problem (attempting to coordinate an attack with unreliable communication / timing).

Parameter Marshalling - packing parameters into a message within RPC. (serialization). It is essential that different languages / data representation are handled.

Unmarshalling - the opposite of marshalling.

Stub - piece of code used to converting parameters passed during a Remote Procedure Call (RPC). The main idea of RPC is to allow the local computer(client) to remotely call a procedures on a remote computer(server).

Parallelism - is when tasks literally run at the same time. e.g. multicore processor or vectorised operations.

Concurrency - is when two tasks can start, run and complete in overlapping time periods. It doesn't necessary mean they will be both running on the same instance. e.g. multi-tasking on a single-core machine.

Eventual consistency - is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Thin Client - a user (computer) that relies heavily on a server for computational services. It's cheap, simple (from client POV), potential problems with single point of failure.

Replication - a process of propagating changes from one database to another database.

Kommentar [1]: I think this should be the duplication of a service for reasons of performance / redundancy

Caching - is the process of prefetching the frequently accessed data and storing it in close to the application.

URI - Uniform Resource Identifiers identify resources on the Web.

URL - Uniform Resource Locators - a subset of URI which give a location for a resource

URN - Uniform Resource Names. e.g. ISBN:0-201-62433-8

Publish-subscribe - messaging is where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Instead, published messages are characterized into classes, without knowledge of what, if any, subscribers there may be. Similarly, subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what, if any, publishers there are.

Little's Law - The long-term average number of customers in a stable system L is equal to the long-term average effective arrival rate, λ, multiplied by the average time a customer spends in the system, W, or expressed algebraically: $L = \lambda W$

Amdahl's Law - used to find the maximum expected improvement to an overall system when only part of the system is improved. It is used in parallel computing to predict the theoretical maximum speedup using multiple processors. No matter how many processors a system may use, they will reach a point in which the speed can't be improved.
Example: Program needs **20 hours** using a single processor core to execute, a particular portion of the program which takes **1 hour** cannot be parallelised, while the remaining **19 hours** can be parallelised, then regardless of how many processors are added to the system, the minimum execution time cannot be less than that critical **1 hour.** *Hence the speedup is limited to up most 20x.*

Speedup = 1 / ((1-P) + (P/N))

P = Parallel Portion of the Program **N** = Number of Parallel Parts

System - a set of connected parts.

w

Uuidgen - Universally unique identifier generator. In RPC, it is put into the header file and can be used to check that the client and server stubs are compatible. i.e. to protect against errors caused by updating one and not the other.

CORBA - Common Object Request Broker Architecture, an IDL

Caching - Storing a copy of some data in a location that is closer to the user

Replication - Storing a copy of some data and ensuring that update methods are incorporated

Kommentar [2]: Part of this isn't right. N is the number of processors serving requests. The maximal speedup regardless of processors is 1/1-P

Kommentar [3]: Not to be confused with Ciorba which is a acidic Romanian soup

Kommentar [4]: Superb

Availability of Replicated Service:

$$A = 1 - P(p_1 * p_2 * \ldots * p_n), \quad p_i = probability\ of\ replicated\ service\ i\ to\ fail$$

Probability of Failure:

$$p = \frac{t}{t+f}, \ t = time\ taken\ to\ repair\ the\ failure; \ f = average\ time\ between\ failures$$

Denial of service - Server reaction to a client request. It can be the result of repeated calls within a short period (perhaps indicating some kind of automation) or perhaps a call to perform restricted methods

REST - REpresentational State Transfer

WSRF - Web Services Resource Framework

LDAP - Lightweight Directory Access Protocol. Allows more simple directory lookups

Total / partial / causal order - descriptive of the nature of order that a system possesses. Lamport systems allow partial ordering, If there is a causal relationship between ALL events in the system then a total order can be established.

BASE - Basically Available, Soft State, Eventual Consistency (see below).

Eventual consistency - a consistency model used to achieve high availability that guarantees that if data is not updated then eventually all accesses to that data will return the last updated value. It must incorporate some method for conflict resolution, e.g. last writer wins, vector clocks.

Andrew File System - utilises a set of trusted servers to provide homogeneous, location transparent file name space to all workstations or users. It has security benefits and is highly scalable. Uses Kerberos (?) authentication. Clients cache files on local systems for speed if repeated access is made. Cache consistency is based on callback mechanism: server notes that a file has been cached and informs clients if the file is updated by other users. Callbacks are discarded after failures including time-outs and might need to be re-established. The file-locking nature means that AFS is not suitable for large shared databases or files shared between client systems.

Cloud computing - an implementation of shared resources, normally via internet enabled devices. Its aim is to maximise the effectiveness of shared resources, e.g. a global business could make resources available via the cloud to Far Eastern branches during their office hours. The same resources could then be made available to US or European branches in their operating hours. Cloud computing should reduce costs and allow users to focus on core business rather than IT obstacles. Concepts such as Service Oriented Architecture (SOA) require users to put problems into services, these can be solved independently and integrated to provide a solution.

Grid computing - each node in the system is set to perform a different task. Tend to be more dispersed than clusters. Can be quite large and often has general-purpose grid middleware. Examples include weather modeling, SETI@Home, LHC (Large Hadron Collider) Computing Grid.

Two-phase commit - a type of atomic commitment protocol (ACP). In normal execution: commit-request (voting) phase, a coordinator attempts to prepare all the transaction's participating processes, it requests *votes.* The next stage is the commit phase: the transaction either proceeds or is aborted based on the votes. The processes follow whatever is required locally and the transaction is deemed complete.

Multicast - one to many messages, follows an agreed protocol: reliable multicast ensures that messages are received either by all members or none. Ordered delivery is a refinement that ensures the order is as required. FIFO ordered guarantees that messages by the same sender are delivered in the order they were sent. Messages are causally related if one message is generated after the receipt of another.

Secure digest - a function utilised to encrypt data / documents prior to transmission over a network. A document is computed into a fixed-length digest, this is in turn encrypted with a private key and made available to intended users.

NTP: networking time protocol for clock synchronisation which works over packet-switched, variable latency networks.

Transaction - A set of operations that is either fully committed or aborted as a whole (Individual and Indivisible). If aborted, no operation in the set is executed.
Provides a guarantee that data will not be left in a corrupted state as a result of unforeseen circumstances.

Requires concurrency control and recovery mechanisms.

Concurrency Control
Two phase locking. Acquire lock, release lock.
transaction commits or aborts in the release stage

Disadvantage of locks
Significantly reduces the potential for concurrency
May result in deadlock

One way to reduce the impact on concurrency potential is to use **Optimistic concurrency control**. This means we let each transaction run its course, and then we evaluate the integrity of the transaction in the "commit transaction" phase, and abort if necessary.
This will increase the potential for concurrency, but at the expense of some redundancy.

Recovery
Backwards recovery - Bring the system from its current corrupted state back to a previous correct state. Requires checkpoints.
Forward Recovery - Bring the system to a correct new state from which it can continue to execute. It must know in advance which errors may occur so it can correct them.

Problems in Distributed Computing with spatially separated servers:
- One or more DBs could fail at any time
- 1Slow or faulty network
- How do we handle this?

Atomicity is extremely important for distributed transactions, as all databases must commit or abort a change. When we handle separate servers, we use **transactions.**

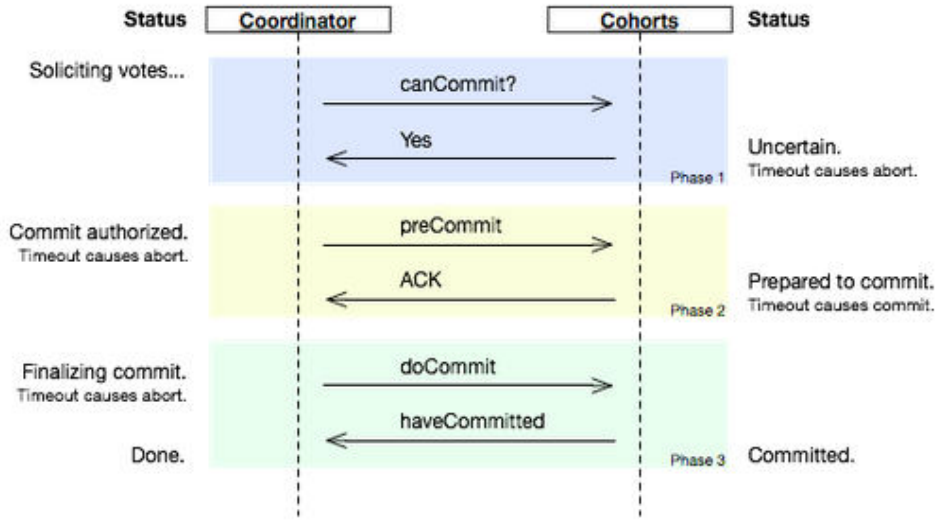
Distributed Commit Problem - An operation is performed by each member of a process group, or none at all.

Distributed Transaction Protocol
1. A coordinator is chosen
2. Ask all servers (participants) if they are ok to execute a transaction, if yes, join.
3. Use two-phase commit. Ready to commit, ack, do commit, ack.
4. If any replies come back negative, abort the whole transaction.

What if the coordinator fails?

Three-Phase commit protocol. Multicast to all other participants.

Three Phase commit - We add an upper bound (timeout) to the amount of time before a transaction commits or aborts. See diagram below for a visualization.



Deadlock

A deadlock occurs when there is a cycle in the “wait-for” graph of transactions for locks.
Resolved by killing a process. Multiple resolution techniques.
Kill youngest/ Kill one participating in multiple cycles/ kill lowest priority.

Distributed Deadlock

The locks are held in different physical places. The problem with this is that the wait-for graph may not be apparent for a single server.

One solution to this is to have a coordinator to which each server forwards its wait-for graph.

However, centralized coordination is not ideal in a distributed system. We can also have a problem of phantom deadlocks with this, where the graphs may become out of date, causing the centralized server to see deadlocks that do not exist.

As an alternative to using a centralized deadlock checker, we can use **Edge Chasing**
We send a probe to the server that holds the lock U “T->U”
This server then forwards, and adds to the message any locks that are blocked
If a transaction repeats in the message chain, then we know deadlock is detected.
“T->U->V->W->X->T”

Partial Failures - one or more machines in the DS fails
Goal = to **Tolerate faults**. We should still be in the position to use the other machines in the DS in the event of a partial failure.

Requirements for Dependability

- Availability - The probability that the system operates correctly at a given time
- Reliability - The system can run a long time without failure
- Safety - Consequences of failure should be small to the whole system
- Maintainability - The system can be repaired easily in the event of a failure

Types of Failure

- Crash - Server halts (e.g power failure)
- Omission failure - Server fails to respond/receive/send messages
- Response Failures - Incorrect response
- Timing Failures - Server fails to respond within a given time
- Arbitrary Failures - A component may produce output it shouldn't have produced, which may not be detected. Arbitrary responses at arbitrary times.

Failure Masking using redundancy - Two Generals Problem

- Physical Redundancy - Think jets with 4 engines can fly with 3,
- Time Redundancy - Actions can be performed again and again
- Information Redundancy - Send extra bits when transmitting information to allow recovery

Redundancy is an effective method of failure tolerance, but has some drawbacks.
We must make sure redundancy is consistent, for example, if we update a piece of data which has replicas, we must update all copies.
However, the main tradeoff is Redundancy vs Cost (money/bandwidth/memory etc)

Transactions

Transactions are an implementation of failure tolerant transfer of data. They are an individual, indivisible operation with the ability to recover from many types of failures. Transactions adhere to 4 basic principles to ensure fault tolerance.

ACID Transaction principles

Atomicity
Consistency
Isolation
Durability

Atomicity is very important in DS. All operations must complete or none of them complete. This is to ensure that we do not leave our system in a corrupted/inconsistent state in the event of a failure.

An application should never violate a database's integrity constraints. Therefore, we must ensure that any changes made are **consistent** with any defined rules.

We must also ensure **isolated** execution, so that concurrent processes are not interfering with one another and making critical changes at the same time.
One way to achieve this is to produce a serial schedule, executing one transfer at a time with no interleaving.
This is a simple solution but is very slow and scales poorly.

The final criteria of a transaction is to ensure **Durability**. That is, ensure that any updates are persistent once committed.

Implementing Transactions

There are two main ways to implement the functionality of a transaction.

- Using **concurrency control** algorithms. This ensures that the execution is equivalent to a serial execution.
- Using **recovery** algorithms to ensure durability. Replaying the actions of a committed transaction, and undoing changes made by aborted transactions.

Transaction concurrency control

Two-Phase locking - Has **acquire** and **release** locking phase

Acquire - Get read/write lock before reading/writing. Read locks conflict with write locks, and write locks conflict with read and write locks.
Release - In this phase, the transaction terminates, either by a commit or by an abort.

Load balancing - spread load evenly between available resources. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource

Distributed System - A computing platform built with many computers that:

- Operate concurrently
- Are physically distributed (Have their own failure modes)
- Linked by a network
- Have independent clocks

A distributed system is a collection of independent computers which **appears to users as a single coherent system**. The hardware and software components of a networked DS communicate and coordinate using purely message passing.

Trade-offs in Distributed Computing

- Engineering is never an exact science
- Constraints/Optimizations
- Unreasonable Demands
- Imperfections & trade-offs
- Requires flexibility

Kommentar [1]: So trade-offs are some of the trade-offs in DC? □

Kommentar [2]: yes

Kommentar [3]: u wot m8

Consequences of Distributing a System

- Concurrent execution of processes - synchronization, deadlock
- No global clock - co-ordination purely through message passing. There is no singley correct time. Introduces the need for message passing to enable concurrency
- No global state - Each node exists with no knowledge of the global state of the system
- Nodes may fail independently
  - Network Failures - Leave running systems in isolation
  - System Failure - May not be immediately known

Why? People can be distributed, requiring hardware to be physically close to them. These people need to work together, leading to the need to share information between them. Hardware can also be shared, and operate in parallel to increase computing power, and achieve more efficient resource usage.

Fallacies of Distributed Systems

Fallacy 1: Network is reliable

Hardware could fail at any time. Switches fail on a regular basis, and have a defined mean time between failures.

Fallacy 2: Latency is Zero

The time taken to transfer data between two points is not negligible. Minimum RTT is determined by the speed of light, but there are other factors which impact on how achievable this time is.

Fallacy 3: Bandwidth is infinite
There is a limit to throughput (bp/s). Bandwidth limitations are increasing, allowing for more data to be transferred per unit time, but so does the amount of data we are attempting to transfer.

Fallacy 4: The Network is secure
Some security may need to be implemented at the application level.

Fallacy 5: Topology doesn't change
In a real world situation, servers/clients can be added/removed arbitrarily, causing a constantly changing topology. Due to this, it is sensible to not rely on a specific route or endpoint, as these may not even exist at the time of transmission.

Fallacy 6: There is one administrator
Networks can be huge, different administrators of varying skill can be in charge of different sections of networks.

Fallacy 7: Transport cost is zero
Time costs - Application --> Transport layer is not free, data needs to be serialised
Money costs - Leasing bandwidth, powering servers etc

Fallacy 8: Network is homogeneous
There can be many types of devices on a network, operating in different ways. Interoperability is required. This can be achieved using standards (e.g XML)

N.B. If you ever see CDK(4 / 5) it means Coulouris et al & TVS means Tannenbaum and Van Steen

Lecture 1 Introduction to Distributed Computing:
Links:
http://blogs.msdn.com/b/oldnewthing/archive/2006/04/07/570801.aspx
http://www.rgoarchitects.com/Files/fallacies.pdf

Reading:
- Coulouris, 1.1 to 1.3 (pages 1-15)
- Tanenbaum 1.1 and 1.3

Lecture2 A few words about Parallel Computing:
Q1:

Kommentar [1]: Don't assume this is correct!

Two options are made available to a client to retrieve some information from a remote server:

•The first option allows the client to retrieve a 100MB file (containing all the relevant information) at once.

•The second option requires the client to make 10 remote requests to the server. Each remote request will return a file of 1MB (all 10 files are needed by the client to retrieve the relevant information).

If the latency of the network link between the client and the server is 1sec and the available bandwidth is 100MB/sec which option would be faster? What if the latency is 1msec? What do you conclude?

100MB at once:
- 1s latency, 2s (1s latency, 1s transfer)
- **1ms latency**, 1.001s (1ms latency, 1s transfer)

10 * 1MB:
- 1s latency, 10.1s (10 * 1s latency, 0.01s * 10 transfer)
- **1ms latency**, 0.11s (10 * 0.001s latency, 0.01s * 10 transfer)

Reading:
- Coulouris4, Coulouris5, skim through Chapter 1.
- Tanenbaum, Section 1.2 (all of Chapter 1 covered)
- Read about Amdahl's law!

Watching:

Recommended, but not required:
https://www.youtube.com/watch?v=yjPBkvYh-ss

Lecture 3 Models and Architectures:

- Read "Web Services are not Distributed Objects", Internet Computing, Nov-Dec. 2003, http://www.allthingsdistributed.com/historical/archives/000343.html
- Lots of research in the context of building parallel computers – see Chapter 18 in Coulouris *et al* book.
- end-to-end argument: See Coulouris pages 33-34 and http://www.reed.com/dpr/locus/Papers/EndtoEnd.html
- Coulouris (4th or 5th edition), Chapter 2; Tanenbaum, Sec. 2.1, 2.2 (skim through the rest of Chapter 2)
- Read the text associated with the figures from Coulouris *et al* and Tanenbaum *et al* textbooks that are shown in this handout.
    - *Unsure how best to reference these figures*

Lecture 4 RPC+RMI:

- Coulouris et al: Chapter 5 (but see Chapter 4 for "marshalling")
- Tannenbaum: Section 4.2 (see 10.3.4 for RMI)
- Remote Procedure Call (1984 – Birrell & Nelson)
- Remote object references - (Coulouris 4.3.4)

Kommentar [2]: Probably beyond scope?

Lecture 7 RPC+RMI - exercises:

Q1:

Given the following C code:

```
void doIt1 (int *p) {
   *p += 1 ;
   return ;
}
void doIt2 (int *a, int *b) {
   *a += 1 ;
   *b += 2 ;
   return ;
}
int main (int n, char **args) {
   int i = 1 ;
   doIt1 (&i) ;
   printf ("%i\n, i) ;
   doIt2 (&i, &i) ;
   printf ("%i\n", i) ;
}
```

What values would you expect to be printed for i normally?

What values would be printed if `doIt1` and `doIt2` were called using RPC instead of just being local functions?

Assume that the RPC knew, from the IDL, that the parameters were just "inout" references to ints so copy/restore is used.

.

Single machine:

> 2

> 5

Distributed:

> 2

> 3

or

> 2

> 4

Q2:

- Where does the need for at-least-once and at-most-once semantics come from? Why can't we have exactly-once semantics?

The problem is how to deal with a suspected server crash. Exactly once is impossible because you don't know if the server has crashed before or after executing the requested operation.

Q3:

- Consider a client/server system based on RPC, and assume the server is replicated for performance. Sketch an RPC-based solution for hiding replication of the server from the client.

Simply take a client stub that replicates the call to the respective servers and execute these calls in parallel

Q4:

- Traditional RPC mechanisms cannot handle pointers. What is the problem and how can it be addressed?

Pointers refer to a memory location that is a local to the caller and meaningless to the recipient.

This can be addressed by using copy/restore semantics instead of direct referencing

Kommentar [3]: ?

Q5:

- Explain what is in the runtime library in the Figure of slide 4.

The runtime library contains calls to the underlying transport-level interface or routines for converting data structures to host-independent representations.

Kommentar [4]: Rizos' answer

Machine / OS specific code to be called from the client / server implementation, e.g. socket handling / TCP & UDP and memory management etc

Kommentar [5]: ??

Kommentar [6]: This is what I wrote before I realised there were answers

Q6:

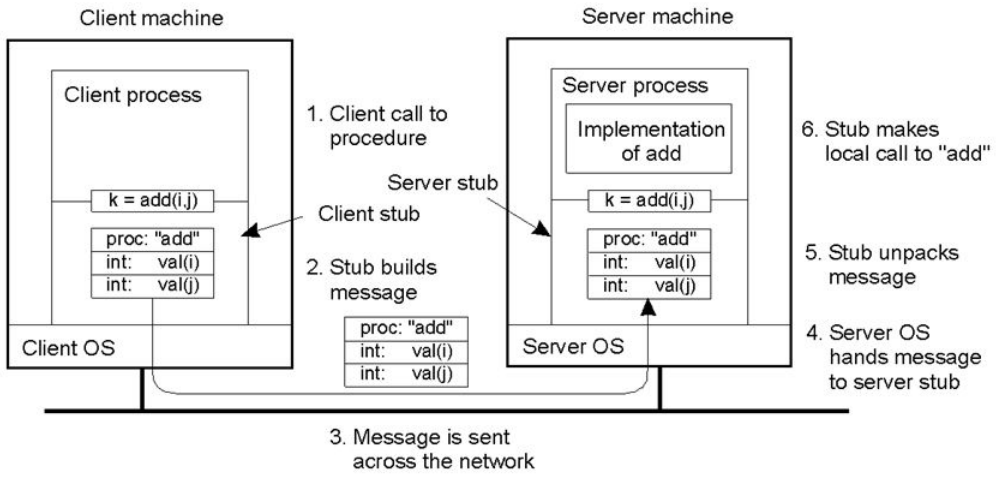
- Executing an RPC requires that a client can contact a server. How does it find the contact point for a server, and what does that contact point consist of?

Contact a name server; the contact point will consist of an IP address and port number.

Q7:

- Explain the principal operation of a remote procedure call (RPC).

The client process makes a remote procedure call. The client stub marshalls the parameters and procedure name into a serial form which may be transferred over the network.



It then sends this information via the name server to a server, whose stub unmarshalls this request and executes it on their implementation of the procedure.

We can assure that the interface is properly defined by using a UUID generated when the client and server stubs / implementations were compiled and linked.

After executing the procedure the server stub marshalls the information to return, sends it over the network back to the client stub, who in turn unmarshalls it and returns it to the original calling client code.

Q8:

- An RPC to a replicated server can be made highly transparent to caller and callee with respect to access, replication, and failure transparency. How?

Place the replicated call inside the client-side stub from where it can simply be executed in parallel to the servers.

Q9:

- What is the major disadvantage of using RPCs in comparison to messaging as in message-queuing systems?

The main disadvantage is the synchronous and non-persistent nature of RPCs – when a fault occurs it has to be dealt immediately.

Q10:

- Give two compelling arguments why an RPC can never provide the same semantics as a local procedure call.

Access transparency cannot be achieved as it is impossible to pass pointers to local data structures. Also, masking failures (as we operate across a network) is a major issue.

Q11:

- What is the main difference between a remote method invocation (RMI) and an RPC?

RMI provides a system-wide object reference mechanism, hence objects can be referenced from a remote machine.

Q12:

- Give a typical example of client-side software that is needed in a distributed system.

Stubs would be required to encapsulate the behaviour needed to support operating in a distributed manner. For instance looking up an authenticating with a remote host via a name server, handling replication to multiple servers and tolerating failures of the system in a way that is safe, reliable and transparent to the client.

Lecture 8: Name and Directory Servers

CDK4: Chapter 9

CDK5: Chapter 13

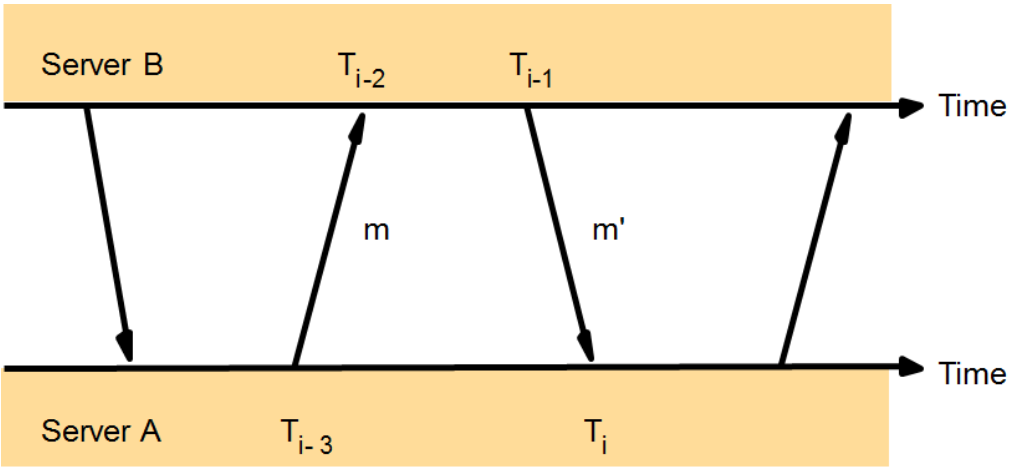
TVS: Chapter 5

Lecture 9: **Time & Clocks**

CDK4: Sections 11.1 – 11.4

CDK5: Sections 14.1 – 14.4

TVS: Sections 6.1 – 6.2



Q1:

- Use this information to estimate the offset between the two clocks, o, from the equations (where t, t' are transmission times for m, m' resp.), and a d, delay, total transmission time of the two messages.

$\mathbf{T}_{i-2} = \mathbf{T}_{i-3} + t - o$
$\mathbf{T}_i = \mathbf{T}_{i-1} + t' + o$

$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
$o_i = (T_{i-2} - T_{i-3} + T_i - T_{i-1}) / 2$
$o = o_i + (t' - t) / 2$

Lecture 10: Coordination & Agreement

CDK4: Chapter 12 (esp. Section 12.3)

CDK5: Chapter 15 (esp. Section 15.3)

TVS: Chapter 6 (esp. Section 6.5)

Q1: (2012, Q3d)

In a system containing 7 computers, identified by the integers 1-7, the coordinator is chosen by the Bully algorithm to be the live one with the highest identifier. Assume for this part that all messages are delivered promptly, and that the computers and the network are entirely reliable. At a certain point in time the coordinator (computer 7) and the computer with the second-highest identifier (computer 6) crash. How many messages in total are sent if the computer with identifier 1 is the computer discovering the crash and triggering an election? You need to count all three types of messages that the algorithm sends. You can assume that computers with identifiers 6 and 7 remain crashed during the election.

Elections:
6+5+4+3+2 = 20

Answers:
4+3+2+1 = 10

Coordinator:
4

Total:
34

Lecture 11: Fault Tolerance - Transactions

- Tanenbaum *et al*: Sections 1.3.2, 8.1-8.3 (weak on transactions).
- Coulouris *et al (4th ed)*: Sections 2.3.2, 13.1, 13.2.
- Coulouris *et al (5th ed)*: Sections 2.4.2, 16.1, 16.2.

Lecture 12: Distributed Transactions

- Coulouris4, Chapter 14
- Coulouris5, Chapter 17
- Tanenbaum, Sections 8.4-8.6 (too detailed in parts and not transaction-focused)

Lecture 13: Byzantine Fault Tolerance

- CDK4, page 501, and Section 12.5.3 (pages 504-507)
- CDK5, Section 15.5.3
- Tanenbaum *et al*, pp. 332-335

Binding an RPC Client to a server

We don't want to hardwire a server's machine name and port number into a client. We use a known directory server to locate the server, and a local daemon on said server to find the correct port to use.

The server registers its endpoint with a local daemon, so an entry exists for the process in the Daemon's "endpoint table".
The server then registers its service with a directory server.
The client can then query the directory server to find the location of the server.
The client can then query the daemon at this address to find the port number used for RPC.
RPC is now setup and communication can be performed.

Names

Pure names - contain no information about the item they are associated with.
Other names may tell you something about an item or something about it's location
An address is an extreme example of a **non-pure** name

URL translated through DNS lookup to a IP address.
IP address translated to a network address (MAC Address)
URL can also contain the port number, which can be used to access the correct part of the file system on the server

URI - Uniform Resource Identifier. "A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource." - Tim Berners-Lee

URL - Uniform Resource Locator. Subset of a URI which gives the location of a resource. "A URI can be further classified as a locator, a name, or both. The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location")." - Tim Berners-Lee

URN - Uniform Resource Name. Local name of the resource (file path?)

For an analogy,
URI = Person's name and address.
URL = Person's address
URN = Person's name

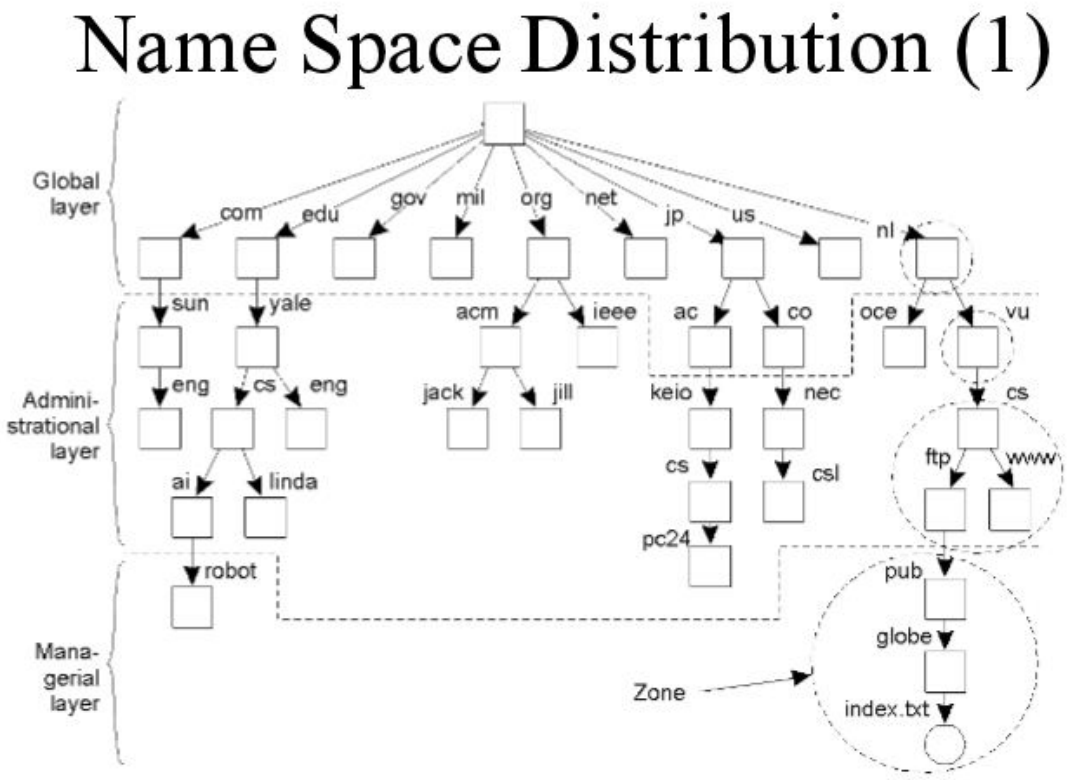
Name Resolution

When a name is “resolved”, it is translated into data about that item. (Non-pure) Names are bound to attributes, such as an address

Namespaces can be flat (number/string) or hierarchic.
Flat is useful for a small number of entries.
Hierarchic - each part of the name is resolved in a different context (allows for distribution of responsibility)

Domain Names

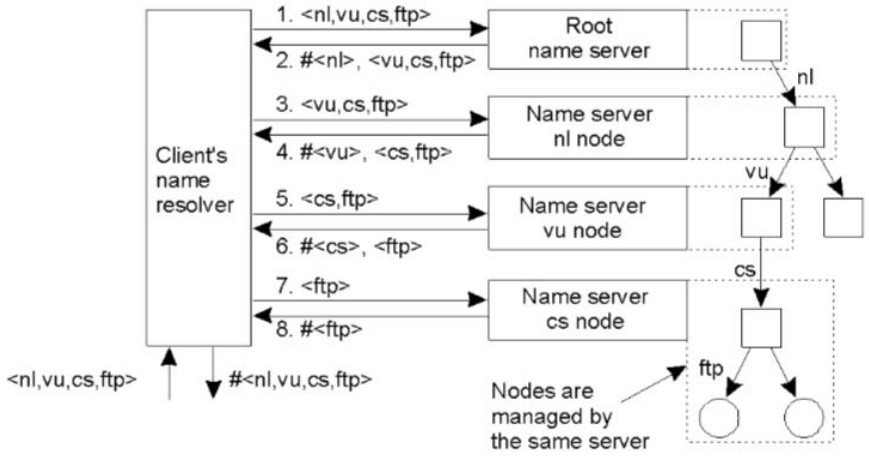
Domain Name System - names computers across the internet.
There is a huge amount of data spread across the internet, so we partition it by domain.



TVS: Fig. 5-13. An example partitioning of the DNS name space, including Internet-accessible files, into three layers.

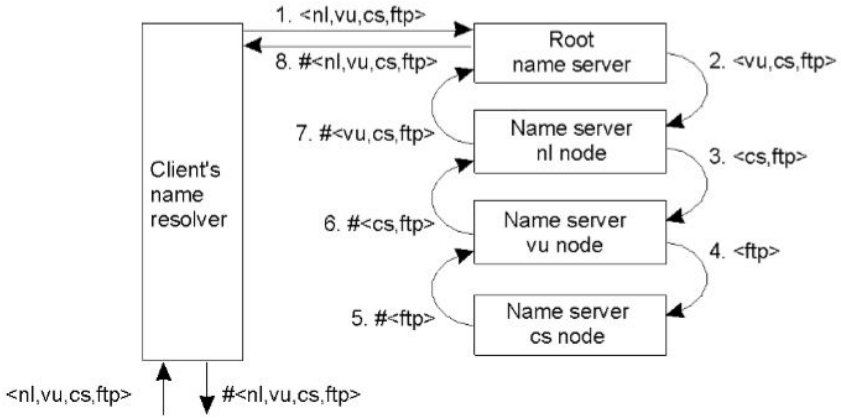
A client has a DNS Resolver, which is used to make queries to known DNS servers to resolve an address
Clients can use iterative name resolution. Get information from each step in the hierarchy to eventually find the correct node.
Clients can also use recursive name resolution. Information about each layer is passed between the layers. Client only communicates with top layer.
Recursive puts more burden on the servers, however it makes caching much more effective.

Implementation of Name Resolution (1)



TVS: Fig 5-15 Iterative Name Resolution

Implementation of Name Resolution (2)



TVS: Fig 5-16: recursive name resolution.

Zones

DNS **data** is divided into zones. Each zone contains attribute data for its associated domain, such as Zone management data, which dictates the lifetime of cached items etc.
Below is a table showing the most important types of resources held by a node in a zone.

The DNS Name Space

Type of record	Associated entity	Description
SOA	Zone	Holds information on the represented zone
A	Host	Contains an IP address of the host this node represents
MX	Domain	Refers to a mail server to handle mail addressed to this node
SRV	Domain	Refers to a server handling a specific service
NS	Zone	Refers to a name server that implements the represented zone
PTR	Node	Symbolic link with the primary name of the represented node
CNAME	Host	Contains the canonical name of a host
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any entity-specific information considered useful

TVS: Fig. 5-19 .The most important types of resource records forming the contents of nodes in the DNS name space.

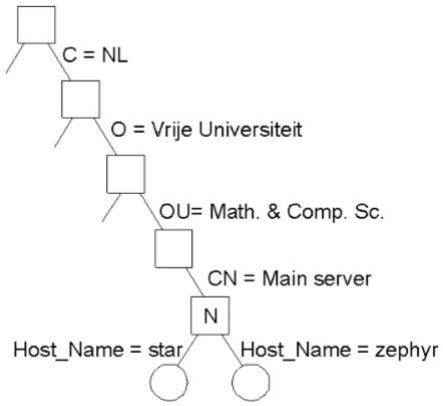
Name server - takes a name, and returns attributes of the names object
A directory server takes attribute values and returns sets of attributes of objects with those attribute values.

LDAP - Lightweight Directory Access Protocol is a simple, widely adopted directory lookup mechanism. Directory trees are requested by Directory User Agents (clients) and returned by Directory Service Agents.

Below is an example of an LDAP directory entry:

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main server
Mail_Servers	–	130.37.24.6, 192.31.231.42,192.31.231.66
FTP_Server	–	130.37.21.11
WWW_Server	–	130.37.21.11

TVS: Fig. 5-22. A simple example of an LDAP directory entry using LDAP naming conventions.



If enough information is provided, we can navigate the Directory Information Tree as specified by the LDAP entry. This allows us to also discover other attributes/resources at levels defined in the DIT.

Challenges/goals for distributed systems

- Heterogeneity
- Openness
- Security
- Scalability
- Failure Handling
- Concurrency
- Transparency

Latency is not zero

Lower latency and low bandwidth is better for lots of smaller data transactions. Higher latency and high bandwidth is better for a smaller number of larger data transactions.
Once the connection has been established, data keeps flowing, this is why we only need to take latency into account once.
Latency and bandwidth are interdependent. Latency = speed, bandwidth = capacity/throughput.

Increasing Performance

Many situations (especially in science) where fast response time is desired.
We can increase execution time by using more computers, working in parallel.

Different machines may operate on different data. Ideally, this speeds up execution by a factor equal to the number of processors

Speedup = sequential timead of attempting to split between this many processors will actually decrease performance.

Parallel vs Distributed

Parallel - Multiple CPU's on same Computer, emphasis on performance
Distributed - Multiple networked come / parallel time
where sequential time = Time without improvement
parallel time = Time with improvement

If a lot of cores are used, eventually the overhputers

Kommentar [1]: What?

Not all applications can be parallelized

If proportion of application that can be parallelized, is 0<x<1, speedup from parallel execution is 1/(1-x)

Running time "tp", of a program on p CPU's, which when run on one machine has running time ts, will be equal to
tp = to + ts*(1-x + x/p)
where "to" is overhead added to enable parallelization.
and (1-x + x/p) is the fraction of the program that can not be parallelized plus the fraction that can, split between p processors

Amdahl's Law

Amdahl's law states that the speedup of a system is strictly limited by the section of the code that is sequential.

Parallelization [edit]

In the case of parallelization, Amdahl's law states that if *P* is the proportion of a program that can be made parallel (i.e., benefit from parallelization), and (1 − *P*) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using *N* processors is

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$

In the limit, as *N* tends to infinity, the maximum speedup tends to 1 / (1 − *P*). In practice, performance to price ratio falls rapidly as *N* is increased once there is even a small component of (1 − *P*).

As an example, if *P* is 90%, then (1 − *P*) is 10%, and the problem can be sped up by a maximum of a factor of 10, no matter how large the value of *N* used. For this reason, parallel computing is only useful for either small numbers of processors, or problems with very high values of *P*: so-called embarrassingly parallel problems. A great part of the craft of parallel programming consists of attempting to reduce the component (1 – *P*) to the smallest possible value.

P can be estimated by using the measured speedup (*SU*) on a specific number of processors (*NP*) using

$$P_{\text{estimated}} = \frac{\frac{1}{SU} - 1}{\frac{1}{NP} - 1}.$$

P estimated in this way can then be used in Amdahl's law to predict speedup for a different number of processors.

- This is a **collaborative** revision document for answering past papers, feel free to join in.
- Remember that you may make comments on items by selecting them and choosing comment from the menu. Alternately **ctrl + alt+ m.**
- **There is a chat for people viewing the page. Press the speech bubble in the top right to access.**

Kommentar [1]: Like so

I suggest adding a voting system where a question is answered two times so we can see which one is better for the exam. For example add '+' after the answer?

Past paper links (relevancy ∝ recency):
2007-08
2008-09
2009-10
2010-11
2011-12
2012-13 - Feedback
2013-14 - Feedback
http://studentnet.cs.manchester.ac.uk/assessment/exam_papers/UG_sem2_2013/COMP28112.pdf

2013-14

Question 1.

a) Explain briefly why the lack of homogeneity is a challenge when developing distributed systems.

(2 marks)

ans1) Machines tend to run with different operating systems and software, so interoperability will be necessary. Similarly, it will be necessary to use standard technologies, for example, XML for communication. This may slow down a system.

ans2) Due to different architectures / software on systems targeted middleware must be developed, and technologies such as IDL must be used in order to allow the programmer to use a common interface despite the heterogeneity of systems. This is a large amount of development overhead to even perform simple tasks in a distributed manner.

b) Explain briefly why the assumption “latency is zero” is considered a common fallacy in distributed computing.

(2 marks)

ans) Information over a network can only travel at a finite speed (maximum speed = speed of light), so data transfer will inevitably take some time between nodes in a distributed system. This can be problematic for time-critical systems which rely on accurate timing points, program synchronisation etc. It also makes it impossible to have a universal notion of time.

c) Explain briefly what publish-subscribe messaging is.

(2 marks)

ans) Publish-subscribe messaging is an abstract architectural style of loose-coupling (maybe also tightly-coupling?), that determines how messages are sent and received between components in a distributed system.

ans2) Senders of messages ‘publish’ a message without specifying specifically who will receive it. Messages about or concerning a certain thing are categorised. Clients subscribe to a specific category to receive all messages contained within it.

d) In a distributed system, what is the purpose of an IDL?

(2 marks)

ans) An IDL is to ensure that a common interface exists for making remote procedure calls, which is up to date, and is respected by each interacting machine. By linking against headers generated by the IDL machines are free to call procedures implemented on different machines transparently. The use of UUID to identify which version of the interface is used prevents conflicts when machines are updated. I.E. attempting to use different versions of the interface will cause an error.

Kommentar [2]: What does this mean?

Kommentar [3]: The architecture of the DS tends to share nothing, similar to a client-server relationship. Whereas tightly-coupled refers to a highly integrated DS that appears to be a single machine.

Kommentar [4]: OK great ty

Kommentar [5]: where does publish-subscribe messaging come from? which lecture?

Kommentar [6]: Lecture 3

Kommentar [7]: Thank you

Kommentar [8]: Is this not referring to lecture 17? "Sender publishes (sends) a message on a topic rather than a destination
Subscriber subscribe to topics
Similar to the idea of mailing lists"

Kommentar [9]: Yes, ^ this is the right answer (missed it by accident), thanks

Kommentar [10]: This might be too verbose

Kommentar [11]: _Marked as resolved_

Kommentar [12]: _Re-opened_

e) In the context of RPC, what is copy-restore and what is it used for?

(2 marks)

ans) Naturally across networks parameters may not be passed by reference, instead copy-restore is used to attempt to emulate this behaviour. The value of dereferenced pointers is copied and marshalled into the message to be sent to make the RPC. Upon being returned this modified copy is restored to its original location.

f) What must a server do to provide at most once semantics to its clients?

(2 marks)

ans) At most once semantics means that one message is sent and you wait until either a response is received or timeout is reached. f

g) Explain briefly what failures are known as Byzantine failures.

(2 marks)

ans) A byzantine failure is a system wide failure caused by the failure of one or more of it's components which causes the the system to enter an unknown state. Specifically, the failing components will send incorrect information to the functioning ones, who will treat it as accurate and act on it. In the worst cases, individual components will receive different messages from the failed one, resulting in them having a different local understanding of the system']
overall state.

h) In the context of data replication, explain briefly what eventual consistency is.

(2 marks)

ans) Given a sufficiently long period of time over which no changes are sent, all updates are expected to propagate, and eventually all replicas will be consistent.

i) When using Java RMI, what is the purpose of the rmiregistry?

(2 marks)

ans) The rmiregistry allows a server to make remote objects available to clients. The client interrogates the registry by using a string which is bound to the desired remote object. The client is required to know the server name and port in order to interrogate this.

j) In the context of lab exercise 2, what would you do to launch a denial of service attack against the server?

(2 marks)

ans) Repeatedly send requests to the server without any delay in between requests. This will effectively overload the server by filling the server's request queue and bring the server to a halt.

Kommentar [13]: Does this need expanding?

Kommentar [14]: I don't think so, it looks really straight-forward

Kommentar [15]: You could mention it will not send another message until it receives an acknowledgement of failure, but I don't think it's necessary. ~ Andrew

Kommentar [16]: the acknowledgement of a failure is still a response, so the statement is also true in that case, I think.

Kommentar [17]: I think it's worth pointing out that one and ONLY one message is sent, as opposed to at-least-once semantics where it keeps trying until a response is received.

Kommentar [18]: 'Dont think you wait for a response. You're not hugely bothered if the message is received or not. You literally just send it the once and get on with what you were doing. no?'

Kommentar [19]: I got confused by the interface here and accidently added this onto the answer - thoughts?

Kommentar [20]: https://stackoverflow.com/questions/13330067/rpc-semantics-what-exactly-is-the-purpose

I believe you need to wait for a response/acknowledgement.

No idea where at most/least once are mentioned in the lectures though

Kommentar [21]: I see where you're coming from though, lecture 17 slide 27 says the MOM (message oriented middleware) sends and then forgets. They're asking for server/client here though so I assume it's more about RPC

Question 2.

a) Explain briefly what the role of a client stub and a server stub is in RPC.

(2 marks)

ans) Client and server stubs act as interfaces for RPCs. When the client generates a request, the client stub marshals it into a serial message which the server stub then unmarshalls to form a procedure call with the appropriate arguments for the server to call. Once the procedure is called on the server, the process is followed in reverse to return the result to the server. In other words, stubs are used to abstract the network's heterogeneity and allow client and server to interact via a common transparent interface.

b) Explain briefly what is meant by logical (Lamport) clocks and vector clocks. What property is captured by vector clocks that is not if Lamport clocks are used?

(3 marks)

ans) Logical clocks are monotonic (only ever increasing) counters which keep track of the 'happened-before' relationship of events within a distributed system. Every time a component is performing an action, it increases its logical clock by one and transmits it to every other component, which sets its own logical clock to either the maximum of its clock or the one received. (After this it must increment this as part of the 'receive message' action. This enables the system to keep track of the order in which events occurred.

Vector clocks operate in the same way, except that a copy of each component's clock is stored locally by every component in a vector (instead of a single system wide accumulator like clock), and clocks are updated separately for each process.

With logical clocks it can be inferred that $e \rightarrow e'$ implies $L(e) < L(e')$, but not the inverse. With vector clocks $L(e) < L(e')$ also implies that $e \rightarrow e'$

Kommentar [22]: Event e caused e'

Kommentar [23]: Timestamp associated with e is before timestamp associated with e'

c) Explain briefly what the four properties commonly denoted by the acronym ACID are when referring to transactions.

(4 marks)

ans) ACID:

- *Atomicity*
  - Transactions must be fully completed or not at all,
- *Consistency*
  - All machines must go from one valid state to another
  - A transaction should not violate systems integrity constraints. Maybe
- *Isolation*
  - Concurrent transactions result in a state that would have occurred if they were executed serially
- *Durability*
  - Transactions keep their state even in the event of something such as a power loss.

Kommentar [24]: To stay consistent you can't violate integrity constraints, e.g. erroneous entries in a database

d)

(i) Describe in detail how a centralised coordinating process can provide a mutual exclusive access service in a distributed system.

(3 marks)

ans) After a centralised coordinator is elected, it can restrict access to resources by operating a lock. The simplest way to do this is to have any process wishing to access the restricted resource request access from the coordinator. If the lock is free, the coordinator assigns it to the process until it no longer requires access, at which point the coordinator retrieves it. If the lock is taken (ie already assigned to a process), the process requesting access is placed in a queue. Once the lock is returned to the coordinator, whichever process is at the front of the queue is given it.

In a more complex system where multiple processes may access the same resource simultaneously (ie: multiple processes may read, but only one may write), tokens can be handed out by the coordinator to the requesting processes. These tokens work in the same way as a simple lock, with the exception that multiple tokens can be issued simultaneously.

For instance, a first process may require read access - it will be issued a read token. A second process may then also request read access - multiple processes scan read simultaneously, so it also gets a read token. A third process then requests write access - read and write are mutually exclusive, so the request is denied and the process is placed in a queue. The first process then terminates its operation and returns the read token it held - process two still holds a read token, so process three stays in the queue. Process two then returns its read token, at which points the coordinator holds all the read tokens and can issue a write one to the third process. No more tokens will be issued until that write token is returned as write operations require exclusive access.

In all cases, exclusive access can lead to deadlock, and so timeouts must be implemented to handle exceptional behaviour.

(ii) When the machine supporting such a process gets overloaded with other tasks it needs to find the least loaded machine in the network, and pass over the provision of the mutual exclusive access service to a process on that machine. Two algorithms are being considered for this. The first is to have the server ask each machine about its workload and then notify all the clients with the identity of the new server. The second is to use a ring-based election, initiated by the current server. Fully describe the latter, clearly stating any assumptions you make,

(4 marks)

ans)

Assumptions:

No failures, system is asynchronous.

All processes are initially marked as non-participating.

The process initiating the election marks itself as participating and sends it's workload to the next machine in the ring. When a process receives a workload it compares this with it's own workload. If its own workload is lower it marks itself participating and passes the next machine this value (and the machine identifier, sometimes these are the same). Otherwise it forwards the workload (and ID) it itself received to the next machine.

Once a process receives a message naming it as the coordinator it then sends a message around the ring announcing that it has been elected as the coordinator.

This has a worst case of 3n-1 messages in the case where the coordinator is the process directly before the one who begun the election in the ring.

ii contd) and compare it with the former with respect to the number of messages passed.

(4 marks)

Assumptions: Same as former.

The coordinator initially polls for loads, sends n-1 messages to other processes. Upon selecting the lowest load a further n-1 messages must be sent in order to notify of the nominated coordinator.

In the best case of the ring where the elector remains as the coordinator only 2(n-1) messages are required.

As such, the first algorithm is considerably more efficient with regards to message passing. However, it does rely on every process being aware of the identity of every other process in the system, whereas the ring algorithm only needs each process to know the identity of the next one in the ring, which may lead to scalability issues in larger systems.

Kommentar [25]: n-1-f: If any processes do not answer the first round of messages, they will be considered to have failed and not be contacted again.

Kommentar [26]: What if they are non-participating?

Question 3.

a) Describe clearly all the operations that take place during a Remote Procedure Call (RPC).

(4 marks)

Step-by-step procedure, probably more appropriate in paragraph format in the actual exam:

1. Client calls client stub procedure
2. Client stub marshalls message into serial form
3. Client OS sends to the server
4. Server OS hands message to the server stub
5. Server stub unmarshalls message
6. Server stub makes local call to the desired procedure(s)
7. Local method(s) execute and return results to server stub
8. Server stub build up the resulting message
9. Message sent back to client
10. Client OS hands message to client stub
11. Client stub unmarshalls return value from message and in turn returns result

b) Two computers are used to provide a replicated service. Each computer has a mean time between failures of 12 days; a failure takes on average 12 hours to fix. What is the availability of the replicated service?

(3 marks)

ans) The availability, A, of a replicated service over a period of time is given as (1 - Pa), where Pa = probability that all replicas have failed. The probability of failure, P, of a single node, uses the mean time between failures, F = 12 days, and the mean time to repair a failure, T = 12 hours:

P = T/(F+T)
P = 12/((12*24) + 12)
P = 0.04
Since we have two computers providing the service:
A = 1 - (0.04*0.04)
A = 0.998 (3 s.f.)

c) Consider a client-server application, which consists of 100 services provided by some server. Ten of these services must be executed strictly one after the other, not in parallel with any other services. The remaining 90 services may be executed concurrently and in any order. Assume that each service takes the same time to execute. What is the maximum speedup that can be obtained for the application if multiple identical servers are used to provide the required services?

(3 marks)

Ans) *[Ref: Amdahl's law I think. See Ans2 for wikipedia style approach]*: With p the fractional percentage of the application that can be parallelised, and n the number of threads, the theoretical speedup is:

$$S(n) = \frac{1}{(1-p) + (1/n)p}$$

Kommentar [27]: Basically the same thing, except uses B the percentage that cannot be parallelised instead of p the percentage that can be. Either way, B+p = 1 <=> 1-B=p <=> 1-p=B

As such, the maximum speedup attainable here assuming no overhead is:

$$S(100) = \frac{1}{(1-0.9)+(1/100)0.9} = \frac{1}{0.109} = 9.17\ldots$$

So the operation can be run roughly 9 times faster through parallelisation.

Ans2)

With B the fractional percentage of the application that cannot be parallelised, and n the number of threads, the theoretical speedup is:

$$S(n) = \frac{1}{B+(1/n)(1-B)}$$

As such, the maximum speedup attainable here assuming no overhead is:

$$S(100) = \frac{1}{0.1+(1/100)(1-0.9)} = \frac{1}{0.109} = 9.17\ldots$$

So the operation can be run roughly 9 times faster through parallelisation.

Ans)

For an infinite number of CPUs Amadahl's law reduces to 1/1-p

The proportion of the application which may be parallelised is 0.9f

1/1-0.9 = 10x speedup

d) Consider a simple server that carries out client requests without accessing other servers. Explain why it is generally not possible to set a limit on the time taken by such a server to respond to a client request. What would need to be done to make the server able to execute requests within a bounded time?

(4 marks)

Ans) Assuming the simple server is comprised of a single process, it can only treat one request at a time. If the server receives request faster than it can treat them, it must place them in a queue. Even if the execution time is bounded, the size of the queue cannot be without running the risk of turning away requests. As such, the overall response time for a request (queue time plus execution time) cannot be bounded.

To be able to create an upper bound for the response time, the server must either turn away requests when a fixed maximum are in the queue or (in a more advanced system) delegate those requests to a secondary server with the same queue size limit.

e) The following two processes access the shared variables x, y, z. Each process accesses a different replica of the store used to hold these variables. Before any process starts executing, the value of all three variables, x, y, z, is 0 in all the replicas.

Process A	Process B
x=1;	y=1;
if (y==0) z++;	if (x==0) z++;

(i) When both processes have completed executing the statements given, what are the possible values of z, if the replication uses the sequential consistency model? Justify your answer. **(3 marks)**

Kommentar [28]: This has been refuted, see comments etc

Kommentar [29]: New answer below

Kommentar [30]: This make sense? Basically the supermarket queue system - if more than n customers waiting at a till, open a new one.

Kommentar [31]: Hmm it does, but I'm unsure whether it's what they are looking for

Kommentar [32]: Might be best laid out as a table?

Kommentar [33]: With sequential, only outcomes are one or 0 right? (if either process runs entirely before the other then one, else if it's one operation each 0) I'm thinking table might be a bit overkill for that.

Kommentar [34]: Yeah I'm leaning towards that

One sequential order would be x=1; if(y==0) z++;[z is 1 at this point]; y=1;if(x==0) z++;[z is still 1 since x=1]. So z=1 is an option. Another sequential order, x=1; y=1; if(x==0) z++[z=0]; if(y==0) z++[z=0]. Therefore z=0. Any variance of these are acceptable and will produce either z=0 or z=1.

(ii) When both processes have completed executing the statements given, what are the possible values of z, if the replication uses the causal consistency model? Justify your answer. **(3 marks)**

Question 4. (see question for full details - link for lazy people)
a) Show via an execution sequence (an order in which statements are executed), how Tom can hold more than the maximum permitted number of reservations if two threads are concurrently processing messages from the in-queue and if the pseudo-code that processes a reservation request is not enclosed in an ACID transaction.

(4 marks)

ans) No ACID enclosure means two messages can be treated at the same time, without one having to be completed before the other. We assume here that all the requested slots are free.

Kommentar [35]: The table makes sense to me, but dunno if it will to anyone else. Would be clearer with a seq diagram but opening photoshop to draw one will take forever...

Step	Thread	Operation	What's going on
0	1	Dequeue Request	The thread retrieves a message from Tom.
	2	Pass	Thread is busy doing something (Chasing jerry probably)
1	1	If slot free && res_count<max_res	True, res_count == 0
	2	Pass	Thread is still busy doing something (Jerry's fast...)
2	1	Reserve slot, res_count +=1, return	1 slot booked, res count = 1, reply sent back to tom
	2	Pass	Thread is still busy doing something
3	1	Dequeue Request	The thread retrieves a message from Tom.
	2	Dequeue Request	The other thread also retrieves a message from Tom.
4	1	If slot free && res_count<max_res	True, res_count == 1
	2	If slot free && res_count<max_res	Also true, res_count == 1 (No one has written to it)
5	1	Reserve slot, res_count +=1, return	2 slots booked, res_count = 2, reply sent back to tom
	2	Reserve slot, res_count +=1, return	3 slots booked, res_count = 3 (we assume some sort of write lock on res_count, else res_count=??), reply sent back to tom, 3>max_res so tom has too many slots...

Kommentar [36]: Not an explicit variable, just a count in the database at a given time

b) After which statements in the pseudo-code for processing a reservation request can the server crash? Describe the state of the database, in-queue, and outqueue after the crash and recovery.

(4 marks)

Ans) A crash can occur at "Reserve slot for the student" if the database is unresponsive. In that case, any changes to the database must be reverted (it could have

written the change and not sent confirmation, causing the transaction to hang but the value to have changed), the request should be placed back in the in-queue and any replies placed in the out-queue must be removed.

c) When and why is the two-phase commit protocol used by the server? What happens when the middleware or the database fail during a two-phase commit?

(4 marks)

d) Suppose that two threads, running on different CPUs, are concurrently processing messages from the in-queue. Every incoming message requires processing time of 50 milliseconds and, on average, there are about 7200 messages per hour sent to the server. How long do you expect to have messages waiting in the in-queue before their processing starts? State any assumptions you make.

(4 marks)

ans) assumptions: (1) incoming message rate is constant
(2) incoming messages are always successfully received
(3) servers never fail in processing their messages

arrival rate = 7200/(60*60) = 2 messages/sec
processing time per CPU = 50/1000 = 20 messages/sec
It is expected that messages will have to wait 0 seconds in the in-queue before being processed.

Kommentar [37]: Just to clarify, this is waiting time in the in-queue and does not need to consider the processing time of a message

ans2)
QueueNumber = arrivalRate * timeToProcessRequest
QueueNumber = 2/s * 0.05
QueueNumber = 1

Kommentar [38]: Can this be clarified?

1 message takes 50ms to process, so expect to wait 0ms

Performance modelling - model a representation of the system, capturing its main characteristics to analyze the behaviour of the system.
They are used to answer “what if” questions regarding system capabilities without touching the production environment.
This is useful for example, to find how many servers would be needed to cope with a specific demand.

However, the model is not a reflection of reality, so this introduces some errors

Performance Modelling Approaches

- Analytical Approach - Derive formulae. Usually very complex.
- Simulation - e.g Monte-Carlo simulation (random numbers) & Discrete Event simulation

Analytical Modelling

Take for example, Amdahl’s law, which provides a basic formula/model to estimate the running time of a programme running on several CPUs

Another example is **Little’s Law**. Little’s Law allows us to produce a model of the average number of customers in a stable system, given the arrival rate and the average time taken to service a request.
Avg number of customers in = Arrival Rate * Service Time

Simulation Modelling

Monte-Carlo simulation can be used to calculate results based on repeated random sampling.

1. Generate Random Input
2. Simulate the effect of the input, and check the result.

For example, we can find the area of a circle inside a space of fixed size by scattering random points uniformly across the whole area, and find the percentage which lie within the circle, relative to the whole domain.

Discrete Event Simulation is accomplished by randomly generating discrete events, and then allowing the simulation to proceed as a chronologically ordered sequence of events, taking the generated events as input.
Discrete event simulation assumes some notion of time, as we simulate the “running” of the scenario.

A method of remote message passing.

RPC's are a high-level alternative to primitive "send" and "receive" message passing.
Provides **access transparency** - Using a local service has the same form as a remote one.

There are some (small) security problems, but also performance issues. Higher-level pieces of functionality tend to be "fat"

Concept of RPC - a **process** on the client calls a process on the server to execute the code of the procedure. Arguments are sent with the message, and the host process returns the result to the client.

Synchronous RPC
Host waits in idle to receive a message. Client sends request and then "waits" for the host to return the result. Client wastes time waiting

Asynchronous RPC
Client waits for acknowledgement after calling RP, allowing the client to continue working. The host sends an interrupt when the result is ready, sends the data, and the client then sends acknowledgement of receipt.

Asynchronous is "better" performance-wise as there is less time spent waiting by the client.

However, synchronous also tends to have useful applications, as sometimes the client has to wait for the result before proceeding with execution anyway.

Both have applications, for example
Synchronous - bank transfer
Asynchronous - download

RPC should promote transparent heterogeneity. Things should be expressed in a common way, which abstracts away from what an individual machine is doing.

Argument wise, addresses are pointless. Value arguments are ok. Should be canonical (caller and callee agree on the format of messages)

Stubs
Instead of the code to provide a service, the client machine has a **stub procedure**

- Puts arguments in a message
- Sends message to server
- Waits for server reply
- Unpacks result
- Returns to application call

A client wants to execute the procedure “add” on another machine. It finds out that the procedure does not exist locally, so it calls stub. Stub takes parameters, and calls OS to organize transfer across network. OS on remote machine takes the message and hands it to the server stub. Stub unpacks the message, and makes local call to “add”. The process is then reversed with the result.

Parameter Marshalling

Packing the parameters into a message stub is known as parameter marshalling. Inversion = Unmarshalling.
Client/Server can use different ways to represent data (heterogeneity), so some interfacing is required.

Generating Stubs

The code for a stub can be generated once the procedure specification is known. Should be done in a language-independent way to accommodate heterogeneity.
Done by an Interface Definition Language (e.g CORBA).
The skeleton/syntax of the procedure is defined in an IDL file

Uuidgen

Generates a unique identifier, put in a header file, used to check the client and server stubs are compatible (sort of like a checksum?)

RMI In Java

Remote Method Invocation - Marshaling is simpler, Java <=> Java (homogeneity)
RMI means that both client and server use the same language.

RMI allows for references to remote objects. Interface lists the methods which can be invoked. RMI also has a stub (called the proxy), used to marshal arguments. Objects can be sent as arguments, but must be Serializable.

Proxy (stub) communicates with the dispatcher and skeleton.
A server has a dispatcher and skeleton for each remote object.
Dispatcher - Receives incoming message, and forwards to the correct method in the skeleton
Skeleton - Implements unmarshalling and invokes the corresponding method in the servant.
Also has the job of marshalling the response into a reply message to the proxy.

A typical implementation model of Java-RMI using stub and skeleton objects. Java 2 SDK, Standard Edition, v1.2 removed the need for a skeleton.

rmiregistry - How a server makes a remote object available to clients. A string is bound to the remote object, and the clients interrogate the registry using the string. The client must know the machine name and the port the registry is on.

Call-by-copy/restore can be used in RPCs to replace call by reference (pointers are only meaningful in the address space of the process using it)

- Client stub copies contents of a pointer to an array of characters to a message.
- Message sent to server, stub provides pointer to server to the array of characters. Modifications take place to the array of characters
- When the server finishes, the original message is set back to the client stub, which copies it to the client.

Why do we have replication in distributed systems?

One underlying reason is for **reliability**. As we saw in earlier lectures, redundancy (either time, information or physical) is an effective way of increasing the reliability of a system.

We can calculate the **availability** of a replicated service if we know the failure rates of all the components.
availability = 1 - P(All replicas failed)
where: P(all replicas failed) = the product of the failure rates of each replica. If we have n components with the same probability of failure, P, we get P^n

To calculate the probability of failure (p) of a single node, we use the mean time between failures (f), and the mean time to repair a failure (t):
p = t/(f+t)

Increasing performance

Another reason for replication is for **performance**
Placing copies of data in close proximity to the process that uses them decreases access time. Caching is a prime example of this, exploiting temporal and spatial locality to improve access speeds.

Kommentar [1]: and spatial?

Capacity Planning - Determining the necessary capacity to meet a certain level of demand.

Consistency

Aside from the monetary cost of replication, we must also consider problems with **consistency**. If we update one copy, we have to propagate the update through all other copies. The scale and difficulty of this propagation determines the price of the replication.

Furthermore, global synchronization takes a lot of time if the replicas are spread far apart, so we can see some problems if consistency rules are tight (A value could be updated, but before the update is propagated, the old value could be read elsewhere)

To solve this, we can simply loosen the consistency rules. This way, copies do not **have** to be the same everywhere, although the amount we can loosen constraints depends on the purpose of the system.

Consistency Models

A consistency model is a contract between processes and the data store. If the processes agree to obey certain rules, the data store promises to work correctly.

If consistency is **strict** then we need an absolute time ordering, so we can define which was the last write to a variable (So any read operations read the correct value).
This is easy in a local system, but with the absence of a global clock, this becomes hard in a distributed system.

Sequential Consistency Model - The result of any execution is the same as if operations of all processes were executed in some sequential order, and the operations of each program appear in the order specified by the program.

Causal Consistency - Potentially causally related writes must be seen by all processes in the same order. Concurrent (Not causally related) writes may be seen in a different order by different processes.

Sequential consistency is basically a stricter version of causal consistency. Causal consistency allows for the original order of a process's thread of operations to be muddled around, whereas sequential consistency enforces the original order to be maintained for each process.

Both Causal and Sequential Consistency ensure that operations occur in the order they are defined in the process. However causal consistency allows the propagation of values from local/shared stores to be out of order

Replica Location Management

Where do we place the replicas to minimize data transfer?
This is an optimization problem, but in practise tends to be a management issue.

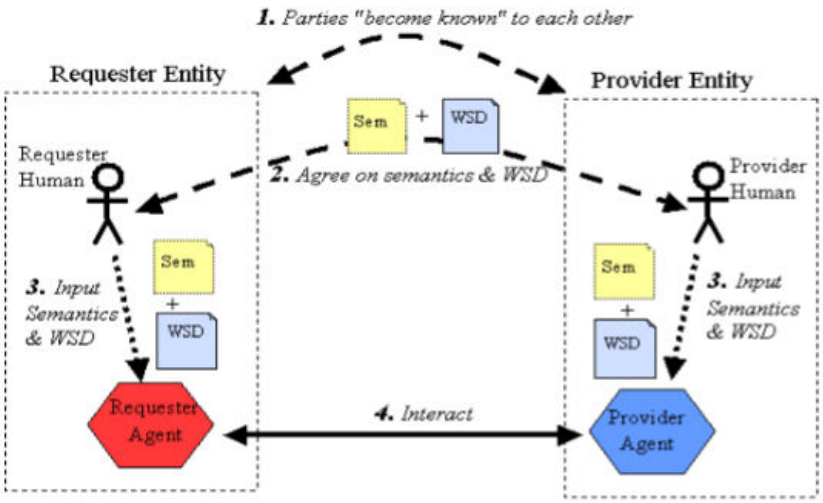
Service Oriented Architecture (SOA)

.Principles for building systems meet the following requirements

- Reliable, fault-tolerant, responsive, scalable and secure
- Interoperate across computing platforms and administrative domains

Service

A service is a collection of code and data that **stands alone**
The ONLY way in and out of the service is via **messages**.
Services are durable and survive crashes.



Semantic descriptions are needed to ensure that the requestor gets the "right" service.

Below are the 4 tenets of SOA

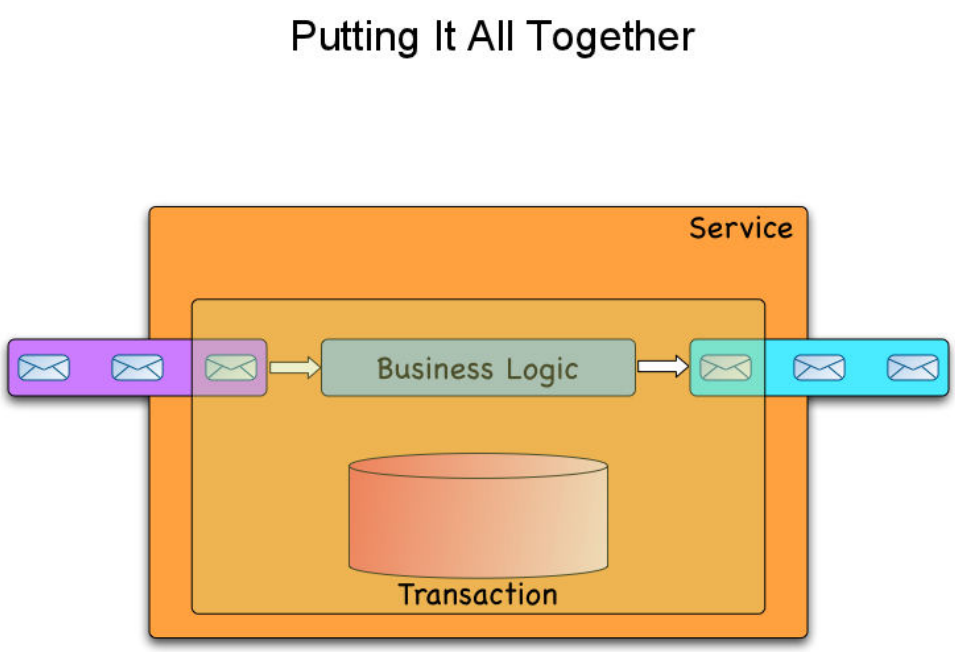
- Service boundaries are explicit - No ambiguity as to whether a piece of code lies within or outside a service.
- Services are autonomous - Services are developed and managed independently. A service can be re-written without impacting other services
- Services share schema and contract - **Schema defines the messages** a service can send and receive. **Contract defines permissible message sequences**. Services do not share implementations, and are independent of computing platform or languages.
- Service compatibility is based on policy - E.g. security policy specifies who can use the service

Service Schemas and Contracts

Service Schema - Defines the messages that are used for different pieces of functionality

Service Contract - Defines the causal relationship between messages. For example, certain messages must come before others in a message conversation.

Think of a schema as the semantics and a contract as the syntax of a service.



Transactions inside a service

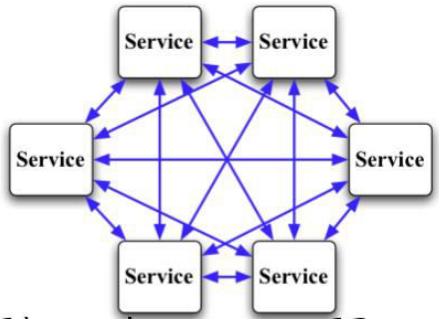
A service crash could occur at any point during fetching a message from the in-queue, processing the messaging, and sending the message to the out queue.

A two-phase commit is used between the queueing system and the database.

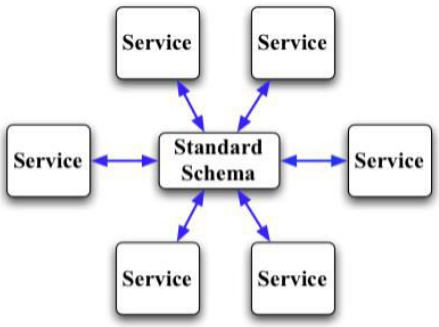
Remember that services share schemas, so when we get a new message, we convert the message from the shared schema to the service's internal schema (language/platform dependent). Upon sending an outgoing message, we convert back from our internal schemas to the language/platform independent shared schema.

Why do we need these shared schemas?

If we had 12 nodes, there would be N*(N-1) possible interfaces between any two nodes, so 132 possible ways to interpret the data.



However, if we use a centralized shared schema, we only need to be able to transform to and from each platform's internal schema, so we need 2N transformations.
So with 12 services, we would only need 24 transformations, which is a massive decrease from 132.



So using an internal schema significantly increases the potential for scalability, because as you can see, 2N is much more scalable than N*(N-1)
We're basically talking O(N) vs O(N^2)

Downsides
However, there are some complications with mapping a service's internal schema to another schema.

We can have a problem with **naming conflicts**
This is where two services express the same concept but with a different naming convention.

We can have a similar problem with the structure of a message, producing **structural mismatches**.
Data representation is another avenue for potential mismatch (endian-ness, ranges etc)

How do we work around these downsides?
If we can find what some information **means**, we have a chance of being able to produce different syntax with the same meaning.
This is known as a **semantic description** of the data

Ontologies are tools for capturing the “meaning” of data.

However, agreement on a common standard is hard, as each service will want to reduce the costs of writing code to transform their own data to and from the standard

Message Oriented Middleware (MOM)

MOM is basically the plumbing to ship messages between services.
The message passing should be asynchronous and non-blocking, meaning the user should not have to wait for a response.

In Point-to-point messaging, the sender specifies a recipient, and the message is delivered to that one recipient.

MOM supports three types of P2P delivery modes:

- At most once
- At least once
- Exactly once

At Most Once (Best effort)
MOM sends the message, and then forgets about it.
Useful when we do not need to guarantee that the message arrives e.g. in non-critical applications.
MOM does not write to any stable storage, so messages can be lost permanently due to failures.

At Least once
Guarantees that a message will eventually reach the destination, but there may be duplicates, so the recipient must be able to cope with this (recognize and ignore duplicates).
This method survives network and service failures.

Exactly Once
The recipient is guaranteed to eventually receive exactly one copy of the message. Useful in critical applications (e.g bank transactions)

Inorder Delivery - The messages are received in the same order as they were sent.

Publish Subscribe Message Delivery
MOM also supports publish/subscribe message passing.
Senders publish a message on a **topic** rather than a destination, and recipients subscribe to topics.
Topics are organized in a hierarchy, so if a recipient has subscribed to a topic, they receive all messages for that topic, and all it’s sub-topics.

Publish Subscribe sees the same delivery modes as P2P

- At Most Once
- At Least Once
- Exactly Once
- In-Order Delivery

Interoperability

Services need to be able to exchange messages, and should be able to choose the MOM of their choice.
However, different MOMs from different vendors do not interoperate. This implies that each service needs to use the same MOM, but this violates our “service autonomy” tenet.

There are emerging technologies to work around this.

REST - Representational State Transfer

Kommentar [1]: Where did the body of this section come from?

The principle behind the Web.
The key abstraction is “resources”, i.e **anything** that can be named.
Resources can be represented in many ways, i.e physical, HTML, XML etc
Resources can contain links to other resources

REST provides the image of a network of interlinked resources.

There are two many problems in synchronization

- The difficulty in setting two spatially separated clocks to the same time
- Once the clocks are synchronized we must cope with clock drift, to maintain synchronization

UTC (Coordinated Universal Time)

International Atomic Time is the time derived by atomic clocks, extremely precise.
Astronomical Time derived from the stars/sun.

UTC is based on atomic time, but has alterations to keep it in step with astronomical time.
UTC is broadcast by radio and satellite.

Computer Time

In reality, few computers in a network have access to GPS/Radio receivers, and so need a xdifferent way of setting them time.

In a coarse grain synchronization, it may be safe to assume the latency between two points is equal in both directions
However, in a finer, more precise synchronization this is unlikely to be true

Cristian's Clock Synchronization

Using a "time server", clients set up their own clocks by measuring the round-trip time to process their request, and adding half of that to the time specified in the reply.
Make request to server, server replies with it's current time. Measure RTT, and set current time to the time in the response + half the RTT.
This method **assumes identical latency in both directions.**

The Berkeley Algorithm

1 processor "master", polls other "slaves"
Slaves reply with their times, and the master estimates the local times using RTT as in Cristian's clock method.
The master averages all these local times (including it's own), eliminating any with abnormal RTTs.
The master sends back the relative delta (+/-) for each client.
If the master fails, an election algorithm exists to elect a new master.

Both of the above algorithms assume equal latency in both directions, and hence probably aren't a good idea for large scale networks with more variable latency. As such, they were designed mainly for intranets.

Network Time Protocol

Designed to be able to handle the larger scale internet.
Has a network of servers:

- Primary - Calibrated with UTC time
- Secondary - Synchronized with primary

NTP provides three methods of synchronization

- Multicast mode
- Procedure Call mode
- Symmetric mode

Multicast - For use in high speed LANs. Simply send the time to all servers on the LAN at once, and each sets their clock to the given time, assuming a small delay. This method is not highly accurate
Procedure-call - Basically Cristian's algorithm. Server accepts requests and replies with time.
Symmetric mode - Highest accuracy. Messages are exchanged and data is built up to improve the accuracy of the synchronization. Each message contains timing information about the previous message

Logical Time (Lamport Time)

In a single processor, every event can be uniquely ordered in time using the local clock. We want to be able to do this in a distributed system, where we cannot use physical time.

We must obey the following principles

- If two events happen in the same process, they occur in the order given by that process.
- If a message is sent between two processes, the sending occurs before the receiving.
- These define a partial ordering of events given by the "happens-before" relationship

Logical Clocks

A logical clock is a monotonically increasing software counter
Each process keeps it's own clock, and uses it to timestamp events.
L++ before each event.
Each message sent contains current L (as t)
Each message received sets L = max(L, t) + 1
Will keep **shaky-synchronizing** the two clocks

Using Lamport logical time establishes a partial ordering between events, but it is also possible to establish a **total ordering,** so there is a order between any pair of events.
This is achieved by using an ordering of process identifiers, to resolve cases where logical clocks are the same in different processes.

Vector Clocks

A vector clock in a system with n processes is an array of n integers.
Each process keeps it's own clock.

Messages between processes contain the vector clock of the sender as a timestamp.
Each clock starts with all integers as 0.

so process 1 increments v[1]
When a process receives a timestamp t, it resets each element of it's own clock to be
V[j] = max(V[j], t[j]) for j= 1….n
This is called "merging" of the clocks

Comparing elements of vector clocks allow us to know which events took place before other events, and to see if the two clocks are in sync
v1 = v2 if v1[j] = v2[j] for all j
same with greater than/less than

If v(e1) < v(e2), then event e1 happened before e2

Advantages and Disadvantages of Vector Clocks
We don't end up with unnecessary orderings between unrelated events.
The extra overhead of using a vector clock is the extra data sent in the timestamp