Distributed Computing

Todd Davies

April 18, 2017

Introduction

Many of the most important and visible uses of computer technology rely on distributed computing. Understanding distributed computing requires an understanding of the problems and the challenges stemming from the coordinated operation of different hardware and software. The course focuses on a set of common techniques required to address the key challenges of distributed computing.

Aims

Many of the most important and visible uses of computer technology rely on distributed computing. This course unit aims to build on the course unit in the first year (COMP10052) which introduced students to the principles of distributed computing, and it focuses on techniques and

methods in sufficient breadth and depth to provide a foundation for the exploration of specific topics in more advanced course units. The course unit assumes that students have already a solid understanding of the main principles of computing within a single machine, have a rudimentary understanding of the issues related to machine communication and networking, and have been introduced to the area of distributed computing.

- Revision of the characteristics of distributed systems. Challenges. Architectural models.
- Remote Invocation and Distributed Objects
- Java RMI, CORBA, Web Services.
- Message-Oriented middleware
- Synchronous vs asynchronous messaging. Point-to-point messaging. Publish-subscribe.
- Concurrency, co-ordination and distributed transactions
- Ordering of events. Two-phase commit protocol. Consensus.
- Caching and Replication
- Security
- Service-Oriented Architectures, REST and Web Services

Contributing & Attribution

These notes are open-sourced on Github at `https://github.com/Todd-Davies/second-year-notes`. Please feel free to submit a pull request if you want to make any changes, or maybe open an issue if you find an error! Feedback is very welcome; you can email me at todd434@gmail.com.

These notes are based on the `COMP28112` lecture notes by Dr. Rizos Sakellariou.

Warning!

Since I write notes on the course before I start to look at past exam papers, I don't always produce notes that are in line with what is examined. Though these notes are still relevant to the course, I seem to have skimmed over topics that are heavily examined and written detailed content on stuff that never comes up. Sorry!

Contents

1 Distributed computing

A distributed system is a computing platform build with many computers that:

- Operate concurrently,
- Are physically distributed (and can fail independently)
- Are linked by a network
- Have independent clocks

Leslie Lamport once said that:

> You know you have a distributed system when the crash of a computer youve never heard of stops you from getting any work done. (Leslie Lamport)

The consequences of having a distributed system is that many problems can arise from a lack of synchronization and coordination between parts of the system. Problems include:

- Non-determinism
- Race conditions
- Deadlocks and synchronization
- No notion of a correct time (no global clock)
- No (visible) global state
- Parts of the system may fail independently

Despite these problems, we continue to (and increasingly commonly) build systems and software designed to run on

distributed hardware. This is for many reasons, including the fact that people are distributed and move around a lot, information needs to be shared, hardware can be shared to reduce costs or work in parallel etc.

Distributed systems have evolved from simple systems in the 70's and 80's. Early systems were for banks and airline booking systems, but the real proliferation of the technique arose with the internet in the early 90's.

There are eight so called *fallacies* of distributed computing:

1. **The network is reliable**
   The network could stop working at any time for a variety of reasons; hardware failure, malicious actors etc. In order to protect against this, we need to build clever software that can resend failed messages, reorder messages, verify the integrity of messages etc.

2. **Latency is zero**
   Latency is the time it takes from a message to get from one place to another. Even if the data is going at the speed of light, then a packet going from London to the east coast of the USA will take 30*milliseconds*. Developers should make as few calls to networked machines as possible, and transfer as much data as possible each time.

3. **Bandwidth is infinite**
   Bandwidth is how much data you can send in a certain amount of time, and is measured in bits per second. Bandwidth is growing as technology improves, but so do the data requirements of applications, meaning that it

is still an issue. Lost packets can reduce bandwidth, so increasing packet size can help. Compression can also be of use.

4. **The network is secure**
   Since networks are largely insecure, you need to think about application security all the time. Implementing access control etc is a good idea for networked applications.

5. **The network topology doesn't change**
   Since we don't control the network, servers could be added or removed, clients can change addresses etc and we won't know in advance. Distributed applications must be adaptive and work around these unexpected changes. The DNS system is a good example.

6. **There is one administrator**
   Different people are in charge of different networks, even different parts of networks. Diagnosing a problem may require the help (and cooperation) of multiple people and organisations.

7. **Transport cost is zero**
   Not only do networks cost money (buying bandwidth, servers etc), but they also cost in terms of computing resources. Serialising between data formats and protocols takes lots of CPU cycles.

8. **The network is homogeneous**
   Interoperability is required for heterogeneous systems to work together properly. Using standard technologies and data formats makes this easier (for example, returning data in JSON format from a REST API instead of a

binary blob).

2 Parallelising processes

Many applications can be parallelised by doing homogeneous operations on different processors on different data. If this is the case, in ideal conditions, your speedup will be the same as the number of processors you're using as opposed to using just one processor.

Unfortunately for us, the speedup is not linear, since it takes time to split the data, coordinate the machines and collate the results. There is also a limit to how many processors will keep the speed improving or even keeping constant. If we have more processors than we can actually use, then the overhead of managing them will probably decrease performance, since they'll be doing nothing useful.

It is important to recognise that parallel computing is different to distributed computing. Although they have similar goals and are achieved using similar techniques, parallel computing is usually when you use multiple CPU's in the same computer, whereas distributed computing is using networked computers.

You can still parallelise an application over different systems using the network as a medium. Not all applications will benefit from this; the most suitable applications have CPU intensive sections that don't require much communication between nodes. If the proportion of your application that you can speed up is x (where $0 \leq x \leq 1$), then the maximum

speedup you can achieve is $\frac{1}{x}$.

The running time of a program executing on n CPU's, when it runs in t seconds on one cpu is:

> Note:
>
> Instead of n (for number), p (for parallelism) is used in the lecture notes.

$$\text{Running time} = \mathit{overhead} + t\left(1 - x + \frac{x}{n}\right)$$

Where *overhead* is the time it takes to setup, synchronise and communicate between CPU's. In practice, the *overhead* is a function that takes the number of processors as an argument (since it will usually increase as the number of processors increases).

2.1 Finding parallelisable portions of a program

Instructions are well suited to parallel execution if they are either independent of instructions around them (so the result of an instruction doesn't change the result of another), or the same instructions are executed on multiple data (such as mapping over an array).

Loops are a very good source of parallelism, since they are usually responsible for repetitive operations on large amounts of data.

3 Architectures of distributed systems

Broadly, there are two main architectures that can be used in the design of distributed systems; tightly coupled and loosely coupled architectures. Tightly coupled architectures look and (try to) behave as though they were a single computer, whereas loosely coupled architectures are often far more distributed and include client-server, peer-to-peer strategies.

3.1 Distributed Shared Memory

Distributed Shared Memory is a tightly coupled architecture that provides the programmer with an illusion of a single shared memory space. Since the programmer is not concerned with anything lower than the system calls to read and write to memory, and these system calls handle the distributed bit, the programmer can by abstracted from any concerns about message passing.

However, the machines are still connected by a network, and therefore there is a latency between the memory reads and writes and their completion. The middleware will try and minimise network traffic, but depending on the application, this could become a limiting factor.

There are also the issues of keeping track of the physical location of each virtual memory address, since they will be on multiple machines, and replicating the data when needed.

3.2 Loosely coupled architectures

Layered (figure ??)
In the layered style, messages must flow through a certain number of layers to reach the destination.

Object based (figure ??)
Here, objects can call other objects.

Event based (figure ??)
Often used as a publish-subscribe (pub-sub) architecture.

Shared dataspace (figure ??)
...

Middleware is used to abstract away the complexity of dealing with networks and their issues. Good middleware can effectively hide heterogeneity of the underlying platforms (e.g. Windows x86 computers talking to Linux ARM machines). The **end-to-end** argument reminds us of the limits of middleware; and states that the functions specific to each application should reside at the endpoints of the network, not the intermediary hosts. This means that though middleware can provide generic functions for you to use, you still have to apply these functions in your application in order to do something useful.

3.2.1 Client-Server model

In this model, many clients will invoke functions on a remote server. Servers (despite there being few of them) act as the slave, since they are passive and wait for requests to come.

Clients are masters, since they are active, and send requests to servers. Servers can be stateful or stateless.

Client server architectures, by nature, are asymmetrical. As a result of this, they often scale poorly, since the server load will increase proportional to the number of clients. Of course, you can use multiple servers to spread the load on any one server (horizontal scaling, done using proxies), and put caches between the client and the server to server common and easy requests.

Another way to reduce the load on servers is to use 'mobile code'. Popularised by Java, this is when code is downloaded to a client from the server, and the client executes the application locally. This reduces server load, and makes the app responsive on the user's computer.

The opposite side to this, is having thin clients, who rely on the server to do all of the heavy lifting for them (e.g. a very simple web browser like Lynx).

3.2.2 Peer to Peer (P2P) model

Here, all nodes in the network are the same, and they all talk to eachother. These networks have no central point of coordination (or failure), which provides resilience, yet at the same time, complexity.

Though P2P architectures scale well (since they are truly 'distributed'), it is hard to find, coordinate and use resources over the network because of a lack of any central organisation.

4 Remote Procedure Calls and Remote Method Invocation (RPC & RMI)

The basic idea of RPC and RMI is that a client will 'call' a process on a remote server to execute the code of a specific procedure, with a specific set of arguments.

There are two ways that RPC calls are fulfilled; synchronous calls and asynchronous calls. In a synchronous call, the idea is that the RPC call behaves like a normal (local) procedure call, and the program running on the local machine will stop executing until the RPC call has finished. In an asynchronous call, the local program will carry on executing, until the remote method is finished, at which point the remote machine will interrupt the local machine telling it to process the results of the call.

> Note:
>
> Serialising parameters into transmittable and server-understandable values is also called *parameter marshalling*. Doing the opposite (when the server receives the result) is called unmarshalling.

Since the local and the remote clients are (or at least are designed to be) running in different machines, the program arguments sent in the RPC call cannot be references to addresses in memory; they must be serialised into values that can be transmitted.

Some kind of middleware wrapper will handle the actual

sending of messages over the network. It will often provide 'stub' methods to provide the service for both the client and the server to make the system as transparent as possible. This is shown in Figure **??**

This middleware will have to be specific to the architecture *it* is running on, to make sure incoming data is passed to the program in the right format (e.g. little-endian not big-endian). If we communicate using a language and architecture independent medium such as a *Interface Definition Language*, then we can mostly avoid these pitfalls.

4.1 Interface Definition Languages

There are many different IDL's for example COBRA (Common Object Request Broker Architecture) and DCE (Distributed Computing Environment).

A COBRA file might look like this:

```
// Person.idl
struct Person {
  string name;
  string place;
  long year;
};
interface PersonList {
  readonly attribute string listname;
  void addPerson(in Person p);
  void getPerson(in string name,
      out Person p);
  long number();
```

};

RMI is included in the standard Java API. To be able to send an `Object` as a parameter, it must implement `Serializable`, which is an interface that lets you turn the object into a byte stream and back again. Having `Serializable` objects is good because it allows you to create arbitrarily nested serializable objects.

A major difference between RMI and RPC is that RMI can reference remote objects. The stub based architecture is similar, except the stub will forward the message to a per-class `dispatcher`, which then sends the message to the receiver's `skeleton` that handles unmarshalling. The `skeleton` will reply with another message to the dispatcher when the request finishes containing the result or any exceptions.

Remote objects are made available to RMI clients by exposing the `rmiregistry` which is a key-object store on the server. Clients interrogate the registry using the String that corresponds to the object they want.

4.1.1 Parameter passing

It is important to note that programming languages can use two different methods of passing parameters between functions; call by reference and call by value. The former is where you pass the function a pointer to a memory location (where a thing (object, value etc) is stored), while in the latter you actually pass the object/value directly to the function. If you're using RPC, then you need to use call by value, since the pointer you would pass with call by reference

is meaningless to the remote machine.

This means you will have to *serialise* any parameters that you want to send along with an RPC call. To serialise and deserialise an object, you need to be able to convert it into text (it can be human readable, such as JSON, or just binary data in ascii format) and back again.

4.2 Connecting and binding to an RPC server

Obviously, it would be silly to hard code the details of the server into client applications. Instead we can use a directory server (at a known location) to tell us the IP address and port of the server we want. The server can use a process to allocate different ports to different connections.

4.2.1 Name resolution

Names can be *pure* or *non-pure*. Pure names contain no information about the item they are associated with, while non-pure names do. An address is a non-pure name, since it tells you exactly where to find a server.

When a name is *resolved*, it is translated into data about the item it describes. Names are bound to attributes such as addresses, and as such, can often be directly used in resolution. For example, in a URL, you have the format:

```
http://{domain name}:{port}/{path}
```

The domain name (such as example.com) can be resolved into an IP address using a DNS lookup. This can be used in conjunction with the port and path to accurately describe where a resource is on the internet.

There are three different, yet commonly used names for the web:

Uniform **R**esource **I**dentifiers
: These identify resources on the web, and start with a URI scheme such as `http:`, `ftp:`, `ssh:` etc.

Uniform **R**esource **L**ocators
: These are a subset of URI's that give a location for a resource.

Uniform **R**esource **N**ames
: An URI which is not a URL. All URN's begin with `urn:`.

Names can also be flat, or hierarchical. If the name is flat, then it can be resolved all at once, but if it is hierarchical, then you have to resolve each part of the name in a different context.

Obviously, DNS is the most well known name resolution mechanism; it is responsible for turning URLs into IP addresses. Replication and caching are used to make the system able to handle the load. Since the data is very large, it is partitioned by the domain.

Name resolution can be implemented recursively (Figure **??**), or iteratively (Figure **??**). In the former case, the whole name is sent to the root nameserver which resolves the top most part, and forwards it on to the next nameserver, which continues until the name has been fully resolved, at which

point, it is sent back to the client. An iterative method would have the client poll each different nameserver in turn for a resolution of part of the name.

Recursive resolution puts more burden on the name server, but also makes it easier to program clients, and also can improve caching (since the caches are centralised at the nameserver).

5 LDAP

The Lightweight Directory Access Protocol is a way of querying for information about services. LDAP works on top of the Internet Protocol, and provides a distributed directory information service.

6 Time and clocks

Being able to agree on a single clock is very advantageous in distributed systems, however, setting distributed systems to be set to the same time, and keeping them in time once they are is an issue.

6.1 UTC

Coordinated Universal Time is a time standard commonly used around the world. International Atomic Time is kept by around 200 atomic oscillators around the world, each

with a drift rate of around 1 in 10^13 seconds. Astronomical time is derived from stars and the sun, but diverges slightly because the earth's rotation is slowing. UTC is based off atomic time, but leap seconds are occasionally inserted so that it is in time with astronomical time. UTC is broadcast by satellites and radios.

GPS receivers are accurate to about one millisecond, whilst receivers from radios or telephone lines are accurate to a few milliseconds. Inexpensive crystal clocks have a drift of around 1 in 10^6 seconds, and this is usually what computers are accurate to.

6.2 Setting times

There are multiple ways of reaching a consensus about times between different machines:

Cristian's method:
: Here, a time server is polled by clients and returns the time on the server. The client times how long it took for the round trip time, divides it by two and adds it to the time that the server returned. This works well for short round trip times.

Berkeley Algorithm:
: The Berkeley algorithm has one master that polls other slaves. Each slave replies with its own time, and the master estimates their local times (using Cristian's method). The master then averages all of the slaves times and its own time, and eliminates any times with an excessive round trip time and any obvious outliers.

Once the master has determined the time, it then sends each clock a delta, which is how much to add or take off its own clock. If the master fails, then a distributed election algorithm elects a slave as the replacement.

Network Time Protocol (NTP):

NTP is designed for large scale internet (as opposed to the other two, which were designed for smaller scale networks). There is a network of servers, one is primary and the others are secondary. The primary server will have a UTC clock.

There are three methods of synchronisation; multicast mode, procedure call mode and symmetric mode.

Multicast mode:

Multicast mode is used on high speed LAN's and is not accurate. The time is broadcast to clients from the server at intervals, and the clients set their time equal to the received time plus a little bit for latency.

Procedure call mode:

This mode is very similar to Cristian's algorithm. The server basically just accepts requests and replies with the time. This is more accurate than multicast mode, and is used when accuracy is key, or multicast is not supported.

Symmetric mode:

This is the most accurate of the three; messages are exchanged and data is build up to improve the accuracy of the time as more messages are send and received. Each message contains time information about the previous message.

Each NTP server will interact with several peers to identify which ones are the most reliable. This can achieve accuracies of from $10 - 100$ms over the Internet.

6.3 Logical time

In a single processor, every event can be ordered in time using the local clock. However, if we are in a distributed system, then the synchronization between clocks may not be good enough to do this.

If any two events happen in a process, they occur in the order given by the process. If a message is sent from one process to another, then the act of sending and that of receiving is said to be an event. This defines a partial ordering of events, given by the *happens-before* relationship.

A logical clock is a monotonically (i.e. increases by one) increasing counter. Each process keeps its own logical clock and uses it to timestamp its own events. When a message is received, then the clock is updated to be the logical time of the received message, or the current logical time (whichever is highest), plus one.

A **Vector Clock** is similar to the **Lamport Clock** described above, but each process keeps track of the clock of each other process. It is in essence n Lamport clocks, one for each process. When a process receives a message, it *merges* its clock with the clock in the message, fining the max of each item.

Vector clocks capture causality which Lamport clocks do

not, however, vector clocks are more expensive in terms of bandwidth.

7 Elections, coordination and agreement

Implementing a semaphore is a common task for computer scientists.

> Note:
>
> See my COMP25111 notes for stuff on semaphores.

On a single machine, even on one with multiple processors, this is fairly easy to implement, but when memory isn't shared (like in a distributed system), then we can't just have a semaphore in memory; we need some system to manage access.

The simplest of these systems is a central server, where a client requests a token on entry of a critical section, and releases it on exit. Processes are held in a queue until there is a free slot for them to enter their critical section.

The trouble with this though, is that the server is a single point of failure, and if the critical sections are short or the server is slow etc, then the server could become a bottleneck.

7.1 Elections

If the server 'goes down', then no processes would be able to enter their critical sections. Obviously this is *slightly* detrimental to the running of the system, so ideally we would like a way of replacing the server if crashes.

If we have n clients/processes, and we want to select one to be a server, then it makes sense to select one that has the lowest load. We could give each server an identifier, which is a pair of $\frac{1}{load+1}$ and i, where i is the number (or unique ID) of the client.

We also need to make sure that all the clients can learn about and agree on the result of the election, and let any client initiate an election at any time (so we could have multiple elections concurrently occurring).

7.1.1 Ring-based election algorithm

If we arranged the processes in a logical ring, then each process knows which is next in the order (this assumes there are no failures in the system).

1. Initially, all processes are non-participants in the election.
2. The initiating process changes itself to be a participant and sends its identifier (remember, that's $\frac{1}{load+1}$) to its neighbour).
3. Each receiver will forward its own identifier if it is higher than the one it received. If it is already a participant (i.e.

its forwarded an identifier previously), then the message is discarded. If the received identifier is lower than its own identifier, the node forwards its own identifier.

4. If the identifier of the receiving process is the same as the received identifier, then that process has been elected. It now sends an elected message around informing the other nodes of its ascension to serverhood.

5. When each node receives an election result, it reset to non-participating and passes the result on.

The largest number of messages sent is $3n - 1$ which isn't too bad. However, failure tolerance is limited (you could re-build the ring if one node crashed, but that's a lot of effort) and its hard to add new clients to the ring.

7.1.2 Bully algorithm

The bully algorithm is designed to be more resilient, since it takes into account that processes may crash (although messages are still assumed to be reliable). Each process knows about each other process and they can all communicate with each other.

Timeouts are used to detect the failure of processes. If a reply has not been received in a time T, then the replying process is assumed to have crashed. If the initiator of an election doesn't learn who has won by $T^1 + T^2$, then it assumes some process crashed and it starts another election.

Three types of messages are used to communicate:

Election messages are sent to initiate an election

Answer messages are sent in response to election messages

Coordinator messages are sent to announce election winners

This is how the method works:

1. The initiating client will send a message to all the clients with a higher identifier than itself. The initiator now waits for the result.
2. Any client that receives an election message eventually will send an answer back. It should then start its own election, by sending a message to clients that have a higher identifier than it.
3. Only when a process gets no replies (within the timeout), does it consider itself elected, at which point, it sends a coordinator message to all lower processes.

This process if $O(n^2)$ in the worst case, which is where the smallest process initiates. Timeouts must be realistic, otherwise the election can be slow.

7.2 Transactions

In an ideal distributed system, we must be able to tolerate faults. That is to say that if a partial failure occurs, then we must be able to continue operating in an acceptable way. Fault tolerance gives rise to dependability, which is defined as:

> Dependability is defined as the trustworthiness of a

computing system which allows reliance to be justifiably placed on the service it delivers.
(IFIP 10.4 Working Group on Dependable Computing and

Dependability has four facets:

Availability:
The probability that the system will be operating correctly at any given moment.

Reliability:
The length of time the system can run without failure.

Safety:
If a failure occurs, what will the consequences be? How will they affect the system?

Maintainability:
How easily can a failed system be repaired?

Likewise, there are five types of service failures:

Crash:
The server has crashed, and no longer responds to anything we try to do.

Omission failures:
The server fails to do *something*. This could be responding to incoming requests, receiving incoming requests, sending messages etc

Response failures:
This is where the server responds, but its response is incorrect.

Timing failures:

The server responds too slowly (or within a certain time window).

Arbitrary (byzantine) failures:
Output may be produced that never should have been produced, which may seem correct, but isn't actually. These arbitrary responses can happen at arbitrary times.

7.2.1 The Two Generals' Problem

The Two Generals' Problem is a thought experiment that illustrates the design challenges of coordinating action through an unreliable medium. Here is a description of the problem:

> Two armies, each led by a general, are preparing to attack a fortified city. The armies are encamped near the city, each on its own hill. A valley separates the two hills, and the only way for the two generals to communicate is by sending messengers through the valley. Unfortunately, the valley is occupied by the city's defenders and there's a chance that any given messenger sent through the valley will be captured.
>
> While the two generals have agreed that they will attack, they haven't agreed upon a time for attack. It is required that the two generals have their armies attack the city at the same time in order to succeed, else the lone attacker army will die trying. They must thus communicate with each other to decide on a time to attack and to agree to attack at that time, and each general must know that the other general

knows that they have agreed to the attack plan. Because acknowledgement of message receipt can be lost as easily as the original message, a potentially infinite series of messages are required to come to consensus.

(`http://wikipedia.com/wiki/Two_Generals_Problem`)

7.2.2 Redundancy

> Note:
>
> See my COMP28512 notes for information on Forward Error Encoding

We can mask failures by building levels of redundancy into the system; say we need three servers to process peak load on a website, then maybe we should run with four in case one goes down, this would be an example of *physical redundancy*. We could also just try actions again if they fail the first time, which is called *time redundancy*, or we could utilise *information redundancy* by sending more information than we need (for example, using Forward Error Encoding) to be able to correct errors in transmissions.

The main concern though, is that the failure of one part of the system shouldn't leave us in an inconsistent state.

7.2.3 ACID

If we can make operations atomic, then if they do not succeed, then we can roll back the system to whatever state it was in prior to the operation taking place. An example of a good use of atomic operations is when multiple threads are updating a variable (since variables can be corrupted if they are updated concurrently by multiple threads).

To ensure that different concurrent threads don't interfere, we could ensure that they are only sequentially executing on the same resources. This is slow though, and not scalable.

However, we also need to make updates *durable*, so that they 'stick around' after they have been made (i.e. they are *persistent*), and that application constraints are not violated (that's called maintaining *integrity*).

This brings us to ACID, which describes the four properties we need to implement in order to have an effective and safe system:

- Atomicity
- Consistency
- Isolation
- Durability

7.2.4 Implementing transactions

Transactions let us have the ACID properties at a relatively low cost. A transaction defines a set of operations that

either commit, or abort. If it commits, then the effects are persistent. This means that we can recover from all sorts of failures, make sure the system is highly available and manage high levels of concurrency.

Transactions are implemented using locks. We want to make it look like things are happening in a serial manner, but we also want to serve clients as fast as possible. To do this we have two phase locking:

Acquire/Expanding phase:
: Here, we acquire the locks to gain permission to read or write. We need a read lock before reading, and a write lock before writing. Read locks only conflict with write locks, but write locks conflict with read and write locks.

Release/Shrinking phase:
: We release the locks when the transaction terminates (whether it has committed or aborted).

We could try and avoid locks by using **optimistic concurrency control**. This is when multiple transactions are allowed to happen at the same time, and then the results are checked a the commit phase, where they are aborted only if there is a collision.

We could also use **timestamp ordering** to detect and resolve collisions between transactions. This involves maintaining a time at which objects were last accessed, and comparing that to the time of the timestamp. If the comparison fails, then the transaction is immediately aborted.

7.2.5 Transaction recovery

There are two methods of recovering from an aborted transaction:

Backward recovery:
Here, we roll back from the current state of the system to a previously known and valid state (a checkpoint). This is usually the state of the system at the start of the transaction.

Forward recovery:
Here, we try and correct the current state by bringing the system into a new valid state from the one we're in now. We must know in advance what errors have and may occur for this to work.

7.2.6 Distributed transactions

The all or nothing principle states that all the databases must commit the transaction, or they all abort; everybody must agree on the single outcome. There are many ways of trying to achieve this:

One-phase atomic commit:
A client tells a central coordinator to commit or abort a transaction, and the coordinator will forward this on to all of the other clients.

Since this is not an interactive protocol, clients cannot tell the coordinator if they are unable to commit.

Two-phase commit:

The two phase commit is a more interactive protocol. The coordinator first asks the clients if they can commit a transaction. If they can, then they reply with a yes. If all clients can, then a global commit message is sent, and if not, then a global abort message is sent.

If the coordinator fails, then the protocol must handle choosing another, or let all the individual nodes talk to all the other nodes. Transactions should be short (so that there is minimum wait time for processes), and there is a danger of distributed deadlocks).

Distributed deadlock can be avoided by each client forward its '*wait for graph*' to the server, and the server can analyse it to find deadlocks. This should be carefully implemented though; you don't want to stop random deadlocks when they are actually normally functioning tasks.

8 Byzantine Fault Tolerance

If there are two generals on a either side of a valley that want to attack a city in the valley, but can only win the battle if they both attack at the same time, then they need to coordinate their attack so that they do. This is hard, since the valley is such that the only way they can communicate is by sending runners through the valley, where they might get killed. If the generals do manage to coordinate an attack over this 'channel' then, great; they'll take the city. However, if the last message in the sequence of messages was not sent, then the receiving general may decide not to attack (though

the sending general wouldn't know any different) and the attacking armies would lose.

We could send lots of messages and hope they get through, thereby reducing the chance that no messages will get through and disaster will be the result. We can never totally eliminate the risk though.

The worst kind of problem is when one general is incompetent or malicious and won't co-operate; then we're really done for! Dealing with these problems is related to the field of byzantine fault tolerance.

Suppose we have four generals now. If there is one malicious general, then we could use majority voting about attacking and retreating to ignore the effects of his meddling. However, if there are three generals, then we cannot:

In general, it is possible to counteract the effects of a malicious general if $n \leq 3f$, where f is the number of malicious generals, and n is the total number of generals.

9 Performance

Performance is sometimes hard to achieve with distributed systems, especially since it is often chosen as the trade off for something else. For example, XML may be sent instead of raw binary data because it is human readable, but XML is also very expensive compared to binary formats.

It would be bad if we created a system, only to realise it couldn't hope to achieve a performance anywhere near to

close enough to what we need. As a result, performance modelling is often used to estimate how a system will react to load by capturing the most relevant characteristics of the system and simulating them. Different circumstances can be modelled, and pitfalls observed in advance to building the system.

Since the model is obviously not reality, this can introduce some errors into our predictions, but in general, it points you in the right direction in terms of knowing how a system will perform. Performance tuning can be done in a model too, for example, what is the best load balancing strategy and how does it scale for different numbers of servers?

There are two main approaches to modelling:

Analytical solutions:

This is where you derive formulas that describe the behaviour of a system. This is apparently more advanced maths than we're expected to know, but the general gist can be found by googling for "Queuing Theory".

An example of this could be Amdahl's Law. This basic model is used to estimate the running time of a program on p CPU's. The premise is that if 95% of the program can be parallelised, then the maximum speed up can only be 20×, since the last five percent cannot be parallelised and must run one one core.

Another simple example is the time it takes for a http response:

$$time = RTT + t_{request} + t_{process} + t_{reply}$$

Where RTT is the round trip time, the $t_{request}$ and $t_{response}$ are the size of the http message divided by the bandwidth, and $t_{process}$ is how long it takes to process the request.

Queuing Theory is based on applied probability theory, and studies the behaviour of systems with queues of 'work' waiting to be done. If we are interested for example, in the average queue length, we can work that out using **Little's Law**. This states:

Average number of customers = arrival time×service time

Simulations:

Monte-Carlo simulations or Discrete-Event simulations can be used to model the behaviour of a system.

Monte-Carlo methods generate random inputs and simulate what follows from those inputs, while Discrete Event simulation proceeds as a chronological sequence of events (though the events are still randomly generated). The main differentiator is that Discrete Event simulation has the notion of time.

10 Redundancy

Redundancy is a technique to increase availability. If a server crashes, then we can have a replica of that server that we can use instead, and henceforth failures can be tolerated by the use of redundant components. For example, in the

internet, there is usually more than one route between two machines, so if one route fails, then the other might succeed. The trade off with replicating services, is that there is a cost associated with the maintenance of the replicas (ensuring they're always available etc).

An example of redundancy in practice is the primary flight computer in Boeing 777 aircraft. The probability that all the computers will fail, is the probability of the first one failing, multiplied by the probability of the second, multiplied by the probability of the third. If one fails every 500 flights (which would be extremely poor), then the probability of them all failing is $\frac{1}{500}^3 = 8 \times 10^{-}9$.

If we replicate services, then we don't just get redundancy as a benefit, we also get performance! Caches are a good example of this; by replicating the data and putting it 'close' to the user of the data, then the time to access the data decreases.

10.1 Problems with replication

If you're taking `COMP25212` then you're probably quite familiar with the joys of consistency between caches. However, for those of you that aren't, let it be known, that this can be a sticky problem sometimes.

If a copy of data is modified, then the copy is obviously different from all the other copies. If we want to ensure that the data is consistent over all of the copies, then we need to copy the data from the updated copy to all the other copies. This could increase our bandwidth and CPU time

requirements significantly, the cure may be worse than the disease!

If it takes time to copy the updated cache value to all other caches, then we may need to stop processing on other nodes while the update takes place in case the other nodes use the old value and end up using the wrong data. This has the potential to be very slow, especially if we're working over the internet.

The solution is to loosen the consistency constraints. This is easier in some settings than in others, for example, web browsers can cache web pages harmlessly, and if the user wants to see the most up to date content, they an manually refresh the cache. However, a CPU couldn't do that with its L1 cache, since multiple cores may end up working with completely distinct data.

We can use *consistency models* to agree a contract between processes and caches. If the processes agree to obeying certain rules, then the cache will agree to respond correctly. For example, if we are reading a cached value, then we would normally want the most up to date version of that value. However, the consistency model might dictate that the most up to date value might not be returned, but it will specify what value would be returned.

Strict consistency is often required in some systems, which makes sense on local machines (e.g. between processor cores), since communication time is low, but on distributed systems, this could be a massive bottleneck. Consequently, we need a solution that has an absolute global time to dictate which events happened after which other events.

Sequential Consistency is when the result of any execution is the same as if operations of all the processes were executed in some sequential order and the operations of each process appear in the order specified by the program. This include reads and write. *Casual Consistency* is when the reads can appear out of order, but the potentially causal writes (i.e. writes that could affect each other) must still be in the same order. *Eventual consistency* gives a time limit for the updates to propagate to all the clients. This is a weak consistency guarantee.

Choosing the correct locations on the network for replica (cache) servers is important since we want the minimum amount of travel time for each packet to reach one. We can use a greedy heuristic to do this, you just find the total cost of accessing each site from the other sites and choose the one with the minimum total cost. The heuristic bit, is that when you want to replace another replica, for each node, calculate its new shortest distance with the new cache in place.

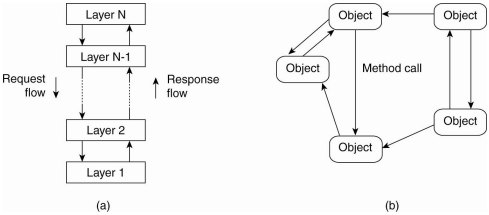

Figure 1: The layered (a) and object-based (b) architectural style

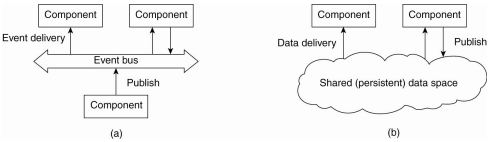

Figure 2: The event-based (a) and shared-dataspace(b) architectural style

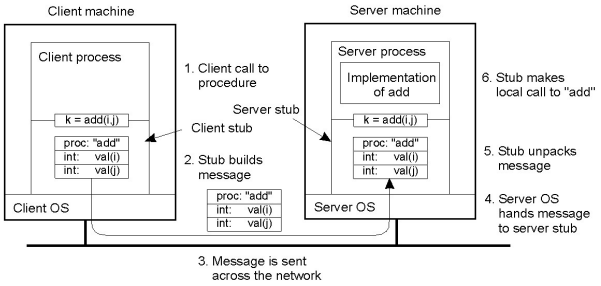

Figure 3: How middleware stub methods could be used to facilitate RPC.

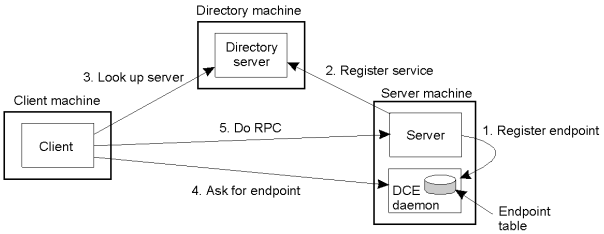

Figure 4: How a directory server can be used can be used to find the location and port of the remote server.

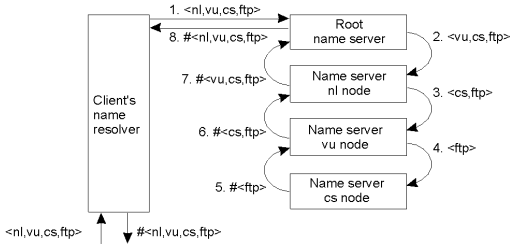

Figure 5: Recursive domain name resolution.

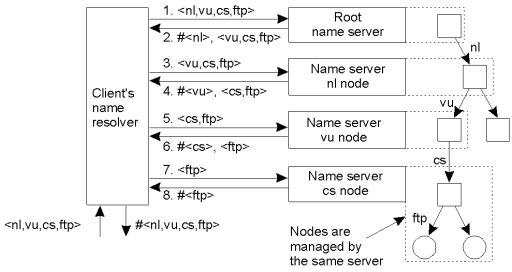

Figure 6: Iterative domain name resolution.

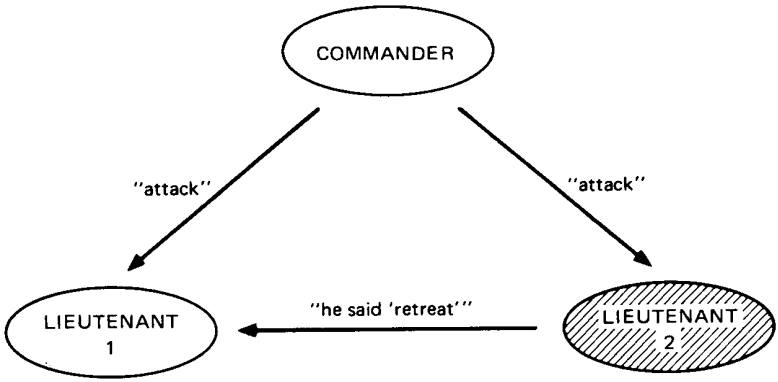

Fig. 1. Lieutenant 2 a traitor.

Figure 7: One malicious general out of three