BSc (Hons) Computer Science

University of Portsmouth

Third Year

Distributed Systems and Security

M30225

Semester 2

Hugh Baldwin

hugh.baldwin@myport.ac.uk

Contents

Part I

Distributed Systems

Lecture - Introduction to DS&S

10:00 27/01/25 Amanda Peart

Module Info

Spit into two parts, Distributed Systems covered in the first 8 weeks and lectured by Amanda, Security covered in the last 4 weeks, lectured by Fahad.

- Distributed Systems covered by a 60% exam, Weds 26th March
- Security covered by a 40% exam

Lecture - Distributed Systems

10:00 27/01/25 Amanda Peart

What is a Distributed System?

A distributed system is the interactions between two or more computers, which are connected in some way to form a single computing system. The two main issues with distributed systems are how to connect the devices, and how to make them interact.

Definition

A distributed system is one which is made up of a set of independent computers which appear to the user as a single system. They are typically connected using a coordination system, which may use any of many networking and communications standards.

Networking Requirements

Most systems typically require fast and reliable networking to coordinate between systems. The often also require access to large amounts of data, which may needed to be accessed by any or all of the systems at the same time, so fast and reliable storage is also necessary.

Parts of a Distributed System

There are several levels to a distributed system, from what runs on each machine to the system as a whole. The low-level systems work on a per-machine level, and include processes, threads, concurrency, etc. Middleware acts to synchronise and coordinate the different parts of the system and ensure efficiency. The application itself sits on top of the Middleware and manages high-availability and fault tolerance.

Design Issues

Naming

It's important that every part of a distributed system has a useful and descriptive name which have a global meaning. This may be supported by a name interpretation or translation system to allow programs to access named resources.

Access

It's important to limit who and what can access the system, both in terms of security and accessibility. They should support as many systems as possible

Communication

The performance and reliability of the communication system is very important to both the availability and overall performance of the system as a whole.

Software

Data abstraction is very important as it allows many different systems to interact with the same data on a high-level without needing to convert data constantly between different formats. Part of this is well designed and documented APIs that allow access to the information and systems.

Resource Management

Optimisation of resources is very important to make sure that there is capacity and availability where it's needed. This includes load balancing and load shifting.

Consistency

The data must be consistent across the system, and must appear the same for all users.

Lecture - Communication Models in Distributed Systems

10:00 03/02/25 Amanda Peart

Communication Models

The communication model of a distributed system defines how each process or node in the system interacts with the others. The models influence the overall design of the system, as well as the efficiency and reliability of the complete system.

Message Passing

The 'nodes' communicate by sending and receiving messages over the network. There are several variants of this model, which may use synchronous or asynchronous message exchange, but all of them require data to be serialised and deserialised before it is sent over the network. Because messages are sent discretely, it can easily suffer from delays and failures in the network.

Examples of message passing include

- Sockets (Both TCP and UDP)
- Remote Procedure Call (RPC)
- Message Queue Systems (RabbitMQ, Kafka)
- REST APIs

Shared Memory Model

Nodes communicate by reading and writing data to a shared memory space. This model is more commonly used in multi-threaded distributed systems running on a single physical machine, but can be used in a fully distributed system when some form of Distributed Shared Memory (DSM) is implemented. Because the memory is directly shared between nodes, there is no need to exchange messages, and as such the overhead is much lower. It also provides a global abstraction of all of the memory used by a system. There is however the overhead of some sort of synchronisation mechanism, such as locks or semaphores to ensure data integrity.

Examples of a shared memory model include

- Distributed Shared Memory (DSM)
- Distributed File Systems
  - Google File System (Used in Google Drive and other Google services)
  - Hadoop Distributed File System

Remote Procedure Call (RPC)

Nodes are able to invoke functions on remote systems as if it were a local function call, with the result being returned as normal to the caller. All network communication details are abstracted away. Can be either blocking (synchronous) or non-blocking (asynchronous). Also requires serialisation, as it typically uses message passing in the backend to actually initiate function calls and return results.

Examples of RPC include

- gRPC (Google Remote Procedure Call)
- Apache Thrift
- Java RMI (Remote Method Invocation)

Publish-Subscribe

Nodes communicate using event-based messaging, where publishers broadcast messages, not knowing who the subscribers are. It is a decoupled architecture, meaning that there is no direct connection between the sender and receiver(s). It is always asynchronous communication, as once the message is sent, the publisher does not receive any confirmations.

Examples of Publish-Subscribe include

- Real-time video and audio streaming
- Apache Kafka
- MQTT (Message Queueing Telemetry Transport)

Data Streaming

Nodes continuously send and receive data in real-time. This model is ideal for applications which require low latency data transmission. Typically used in real-time data analytics and monitoring. Often used in conjunction with a publish-subscribe system.

Peer-to-Peer (P2P)

All nodes act as both clients and servers, which are able to share data and resources without any centralised authority. With this decentralised architecture, it is highly scalable and reliable, but may run into consistency issues, especially if any rogue nodes connect.

Examples of P2P include

- Blockchains
- P2P file sharing such as Torrents and Usenet
- Various game networks

Tuple Space Model

Processes interact through a shared tuple space, where they can read, write and ‘take’ data tuples rather than directly exchanging messages. It is decoupled in both time and space, meaning that process need not be online all at once. Often used in distributed coordination and parallel computing.

Examples of the Tuple Space Model include

- JavaSpaces
- Apache ZooKeeper
- TIBCO Rendezvous

Comparison of Communication Models

Model	Type	Pros	Cons	Examples
Message Passing	Direct	Scalable, widely used	Requires message handling and serialisation	TCP/IP, RabbitMQ
Shared Memory	Indirect	Fast data access	Synchronisation overhead	Hadoop DFS
RPC	Direct	Abstracts network details	Can be blocking	gRPC, Java RMI
Publish-Subscribe	Indirect	Decoupled, scalable	Unpredictable latency	Kafka, MQTT
Data Streaming	Direct	Real-time, low latency	Complex implementation	Video & Audio streaming
P2P	Direct	Decentralised, fault tolerant	Data consistency issues	Blockchain, BitTorrent
Tuple Space	Indirect	Asynchronous coordination	Complexity in managing tuples	JavaSpaces

Lecture - Time & Coordination

09:00 10/02/25 Amanda Peart

Time Synchronisation

Time needs to be synchronised in distributed systems for many reasons, including

- To allow precise measurements of delays between distributed components
- To synchronise real-time applications such as audio and video streams
- To accurately timestamp events, transactions and security data
- To order events and achieve a consistent global state in the system

The main reason that synchronisation is important is to ensure accuracy and consistency when multiple users are accessing the same data, such as when booking flights, hotels, event tickets, etc.

Synchronisation Issues

Synchronisation is very important in distributed systems, as it is required when a large number of nodes are sharing resources, need to agree on the order of transactions or events, or when they need to recover from failures.

If clocks are not properly synchronised, it could lead to transactions resulting in the incorrect balance, tickets being double booked, and many other issues.

Clock Inaccuracy

No clock is perfectly accurate, as they constantly either lag behind or speed ahead of the true time. Each clock will also drift at a different rate, meaning that two synchronised clocks will always slowly drift apart if not corrected.

- Clock Skew is the instantaneous difference between the times recorded by two clocks
- Clock Drift is the difference in counting rate per unit of time as compared to some ‘ideal’ or perfect reference clock

Types of Synchronisation

There are two main types of clock synchronisation; external and internal. When synced externally, all clocks in a system are synchronised with a single external clock that acts as a reference point for ‘real time’. When synced internally, all clocks in a system are synchronised to match each other, even if they may not be in sync with ‘real time’.

Clock Synchronisation Methods

- Internal Synchronisation in a Synchronous system
- Cristian’s Method (External Synchronisation)
- The Berkeley Algorithm (Internal Synchronisation)
- The Network Time Protocol (NTP, External or Internal Synchronisation)

Synchronous Systems

A synchronous distributed system is one in which there is a known bound for: clock drift rate; maximum message transmission delay; time to execute each step of a process.

The general process for internally synchronising clocks in such a synchronous system is as follows;

- One process p_1 sends it's local time t to another process p_2 in a message m
- p_2 wants to set it's internal clock to $t + T_{trans}$, where T_{trans} is the time taken to transmit the message m, but T_{trans} is not known
- Since T_{trans} is unknown, it can't be set directly, and so is estimated as the known maximum transmission time minus the minimum estimated transmission time to get an uncertainty, $u = T_{max} - T_{min}$
- The clock of p_2 is then set to $t + \frac{(T_{max} - T_{min})}{2}$, or $t + \frac{u}{2}$ such that the skew is at most $\frac{u}{2}$

Cristian's Method

The general process for synchronising clocks with Cristian's method is as follows;

- A time server S receives signals from an accurate UTC source, and sets it's clock accordingly
- The round-trip time is used to estimate and account for the message latency inherent in the system
- The server process S then supplies the time to other processes accordingly

There are some issues with this method, as it does not account for what happens when the server fails, nor if the server itself becomes desynchronised for any reason.

The Berkeley Algorithm

The general process for synchronising clocks using the Berkeley algorithm is as follows;

- A master server polls other processes to collect their clock values
- The master uses the round-trip times to estimate the true clock values
- The master then calculates an average time within the network
- Any required adjustments are then sent to the other processes

This allows internal synchronisation, while attempting to eliminate the effects of faulty clocks by taking an average, and if the master fails, a new master can be elected from the remaining processes to take over

The Network Time Protocol

NTP is a standardised protocol used across the internet. It uses reference clocks as a so-called 'baseline', which include: the clocks which make up UTC; atomic clocks run by various government agencies around the world; radio stations used broadcast by the National Institute of Standard Time (NIST). The time server then broadcasts a short reference pulse at the start of each UTC second, with a resolution of $\pm 10msec$.

There are typically multiple layers in an NTP system, where the primary servers are directly synced to UTC, and all servers from there are synced to the primaries. Each layer is known as a 'stratum', where the higher the stratum, the further away from the primary the server is, and the lower it's overall accuracy. There are 3 modes of synchronisation, being multicast messages, Procedure-calls (which uses Cristian's algorithm), and symmetric synchronisation between a pair of servers. All messages are sent using UDP.

Logical Time

Since it is not possible to perfectly synchronise the clocks in a distributed system, we typically abandon the idea of 'physical' or 'real' time, and instead use logical or virtual time. This allows consistent event

ordering within a system, without the need to strictly sync with any external time source.

Events & States

A refined process model can be used, in which

- A distributed system is characterised as a collection N of processes, where P_i and $i = 0, 1, 2, \ldots, N-1$
- Each process P_i hsa a state s_i including the values of all of it's variables
- Processes communicate only via messages over a network
- Each process can only follow one of three actions; send a message, receive a message or change it's own state
- An event is then the occurrence of any single action

The events can then be ordered such that

- Events of a single process P_i can be placed in a total ordering denoted by $\rightarrow_i$ (happened before) between events, where $e \rightarrow_i e'$ if and only if e occurs before e' in P_i
- A history of process P_i is then a series of events ordered as by $\rightarrow_i$, e.g. History $(P_i) = h_i =< e_i^0, e_i^1, e_i^2, e_i^3, \ldots >$

Not all events are related bt $\rightarrow$, as they may occur in two different processes, without any chain of messages that can relate them. This means that they are not related, and are said to be concurrent, which is written as $e_i || e_j$

Lamport's Logical Clocks

A logical clock is a monotonically increasing counter related to a process. Each process P_i has a logical clock L_i which can be used to apply logical timestamps to events. L_i is incremented by 1 before each event occurring on process P_i. When P_i sends a message, it piggybacks the current logical time, so the sent message is (m, t) where $t = L_i$. When P_j receives (m, t), it sets $L_i = \max(L_i, t)$ and applies L_i before timestamping the receive event.

$e \rightarrow e'$ implies that $L(e) < L(e')$, but $L(e) < L(e')$ does not imply that $e \rightarrow e'$, which means that there is no way to detect when events occurred concurrently.

Lecture - Distributed Shared Memory

09:00 17/02/25 Amanda Peart

Distributed Shared Memory (DSM) simplifies communication between nodes in distributed systems by providing a single memory space that all nodes have equal access to. It simplifies programming, as there is no need to perform explicit message passing between nodes or processes. It theoretically also reduces the overhead inherent in resource sharing, as it does not require the data to be duplicated on every node.

DSM is an abstraction which allows multiple computers which do not share any physical memory to access the same data as if it were a single unified address space. It is usually entirely transparent to the system running on it, as there is typically no need for the program to directly interact with the system implementing the abstraction.

Key Features

- **Transparency**– The user and the system do not need to know that the memory is distributed
- **Scalability**– Nodes should be able to be added and removed without affecting performance or reliability
- **Consistency Control**– It is essential to ensure that the memory is exactly the same on all nodes at the same time, to ensure that nodes are not using different or stale values
- **Synchronisation**– Using locks, semaphores or other synchronisation techniques to prevent conflicts between nodes
- **Performance**– Store data locally to ensure quick access

Centralised vs Distributed DSM

- Centralised
  - Shared memory managed by a single server
  - Data is accessed only through the single server
  - Data is stored in a single place
  - This makes it easier to manage and ensure consistency, but has a single point of failure and a large bottleneck on the performance of the single server
- Distributed
  - Memory is stored across multiple nodes but appears as one
  - Each node manages part of the memory space
  - This reduces the reliance on a single machine both in terms of reliability, and performance, but is significantly harder to manage and ensure consistency between nodes

DSM Models

- Page-Based
  - Transfers entire memory pages

  - Can be efficient for large datasets
  - Is susceptible to page faults causing large overhead
- Object-Based
  - Transfers single object instances
  - Reduced data transfers for each object
  - Harder to enforce consistency
- Variable-Based
  - Transfers single variable values
  - Ideal for sharing small amounts of data infrequently
  - Very bad in terms of scalability
- Library-Based
  - Run-time RPC/IPC
  - Very scalable if implemented correctly
  - Code may fail at runtime, causing unrecoverable errors

Consistency Models

Consistency Model	Guarantees	Pros	Cons
Strict	Always have the latest data	Strongest consistency	High overheads
Sequential	All process see events in the same order (in logical time)	Correct execution order	Expensive synchronisation
Casual	Related writes appear in order	Balances order and efficiency	Complex dependency tracking
Eventual	Will synchronise over time	Highly scalable	Temporary data inconsistency
Weak	Synchronisation occurs only at set points	High performance	Data frequently out of date

Strict Consistency

All processes see memory updates immediately, which gives the strongest consistency and ensures values are always correct, but comes with the high overheads inherent in real-time synchronisation.

Sequential Consistency

All processes see operations occur in the same order in logical time, which is easier to implement as it does not rely on physical time, and guarantees a correct order of execution. It does come with the downsides of not relying on synchronisation, and not being ideal for performance sensitive applications.

Casual Consistency

Related writes are always seen in order, but independent writes can appear in any order. If event e_A occurs before event e_B, and they are related, all processes must see e_A before e_B. This is good as it reduces the overhead from synchronising all writes, including ones that don't need to be, but does still require tracking relationships between related events, without guaranteeing a single order of operations.

Eventual Consistency

If no updates occur, all processes will eventually converge on a single correct state. This is the best for highly scalable distributed systems, as it improves performance by greatly reducing the overhead of synchronisation, but will lead to temporary inconsistencies in the data.

Weak Consistency

No consistency is guaranteed for individual writes, as it is only enforced at specific synchronisation points, such as when a process explicitly requests the latest data. This reduces synchronisation overhead even further, and works well when immediate accuracy is not required. This does require the overhead of handling out-of-date data, and needs a way to predict when data may not be correct.

Synchronisation in DSM

To improve performance, DSM usually caches data locally on each node. This makes it more efficient to access the data, but results in problems keeping the cached data consistent with other nodes. It also causes issues if too large of a page size is used, as the entire page may need to be evicted from cache and replaced.

Main Approaches

- Hardware-based (DASH & KSR)–
  - Some multi-processor systems rely on specialised hardware devices designed to load and store instructions with addresses automatically converted to those stored in DSM, and to communicate with remote memory as necessary
- Page-based (IVY, MUNIN & The Cloud)–
  - DSM is implemented as a space in virtual memory, which occupies the same address range in the address space of every process which needs access
- Library-based (Orca, Linda & OpenMP)–
  - Memory is not shared on the memory address level, and is instead handled by run-time communications between processes using library calls
  - The compiler inserts library calls in the place of memory calls when accessing items stored in DSM
  - The libraries access local data and communicate as needed to maintain consistency

DSM Approaches

There are several approaches that can be used when synchronising memory space in a DSM system. Some systems may synchronise only part of the address space, such as variables or data structures that may need to be accessed by more than one process at a time. Another method is to share the entire memory space, which can be useful if each process may need access to any data randomly. Alternatively, only encapsulated data types such as objects and data structures are shared, which may include methods in the case of remote process calls.

Ring-Based Multi-Processors

In a system with ring-based multi-processing, there is no centralised global memory. The memory space of each processor is divided into private and shared data spaces, which are made up of 32-byte blocks. A token-ring is used to share blocks between processors, and each block of data has a ‘home’ processor, usually the one it originates from.

To read a word from shared memory, the address is passed to the controller, which checks the block table to see if the block is present. If it is, the request is already complete and the data can be read. If not, the controller waits for it's turn on the ring, where it then puts a request packet out. As the packet is passed around, each controller checks if it has the block. If it has the block, it provides the block of memory and clears the request's exclusive bit. When the token returns to the requesting controller, it always has the requested block.

To write a word to shared memory, if the block is present and is the only copy (the exclusive bit is set), it is written locally. If the block is present but not the only copy, an invalidation packet is sent around the ring to invalidate all other copes. When the packet returns, the exclusive bit is set and the data is written locally. If the block is not present, a packet is sent that combines a read request with an invalidation request. The first processor which has the block copies it onto the packet and discards it's own copy, any other processors just discard their copies.

NUMA Multi-Processors

Because it is very difficult and expensive to perform hardware caching in multi-processor machines, there is an alternative method of accessing the memory of other processors. Non-Uniform Memory Access (NUMA) is a method for accessing memory of a processor other than the one requesting it. Access to remote memory is typically much slower than local memory, usually in the ratio of 10:1. This means that a processor can directly access the data of another processor, including executing code directly, but the performance will typically be much worse than if it was running from local memory.

NUMA creates a single virtual address space across all processors, with no attempt to reduce the performance deficit via caching. However, it is possible to replicate or migrate a page of memory from remote to local to improve performance of frequent access. Replicated pages are read-only to ensure integrity, but migrated pages can also be written to. There is usually a page scanning daemon running, which watches how the pages are being used to make sure they are not replicated or migrated too often. If a page is migrated too frequently, it may be frozen in place such that it stops wasting bandwidth moving it back and forth, or if a page is read frequently from another processor, it may be invalidated so it is automatically moved.

Page-Based DSM

The idea of page-based DSM is to emulate the systems used in a multi-processor system using the MMU (Memory Management Unit) and OS-level interrupts to handle page migration. It is possible to replicate read-only data or read-write data with consistency support.

This introduces a new problem known as false sharing, in which pages are constantly replicated back and forth due to unrelated writes that happen to fall into the same page of memory. For example, if process 1 is accessing A and process 2 is accessing B, which both happen to be in the same page, each time process 1 writes to A, the page is replicated to process 2, and any time process 2 writes to B, the page is replicated to process 1, even though process 1 never accesses B and process 2 never accesses A. The larger the pages used, the more frequently false sharing will occur.

If pages are not replicated, or are only replicated as read-only there are no issues. If there is only one writeable copy of any page at a time, there is no chance of inconsistent data. When there are replicated read-write pages, there is a high likelihood of data inconsistencies.

Lecture - Web Services and Naming Systems

10:00 22/02/25 Amanda Peart

Web Service Architecture

- Provider
  - Creates and offers web services
  - Describes their services in a standardised format to be published in a registry
- Registry
  - Contains data about services offers by different providers
  - Usually includes technical details, as well as company info like addresses and contact details
- Consumer
  - Retrieves information about services from the registry
  - Uses information from the description to directly connect to the provider and actually use the service

Web Service Definition Language (WDSL)

WDSL is a standard method of describing the services offered by a web service. It tells the client what services it offers, how the client should connect to the service, and what the expected results should be. It specifies the protocols and endpoints used, as well as the location of the service.

Typically formatted as XML, especially when used with SOAP APIs, and tells the client exactly what the format of SOAP messages should be, both the request the client must make, and the response it should receive.

Simple Object Access Protocol

An API standard which defines how web services talk to each other, and how clients should invoke methods on the server. Typically uses XML as the serialized format which is exchanged between the client and server. Each message consists of an envelope that defines what is contained within a message and how to process it, and a set of standards for how the message is formatted. Most SOAP services use HTTP but it is not the only protocol that can be used.

Naming Systems

Types of Names

- **Human Readable Names**– Names used by humans, typically abstracted, e.g. URLs, domain names, hostnames, etc.
- **Identifiers**– Unique identifiers used within systems, e.g. IP addresses, MAC addresses, etc.
- **Addresses**– Internal addresses used by the application, e.g. Memory addresses, storage addresses, etc.

Naming Schemes

- **Flat Naming**– No structure, typically used in peer-to-peer networking
- **Hierarchical Naming**– Organised into a hierarchy like a tree, used in DNS, Active Directory, etc.
- **Attribute Based Naming**– Names based on the attributes of resources

Name Resolution

How names are mapped to an address or identifier. Can be centralized to a single server, or can be distributed hierarchically, as in DNS. Can also use caching to improve speed and efficiency, but can introduce consistency issues.

Challenges

The main challenges in naming schemes are scalability, consistency and security. This is because a naming system needs to be able to handle a large number of names, always be correct, and needs to prevent unauthorized access to name resolution and spoofing naming. There's also the issue of making a naming system that actually makes sense, since it may be based heavily on business-specific schemes, which may not be useful to anyone outside of the organization.

Quality of Service (QoS)

- Accessibility– Capability of a service to answer requests
- Availability– When and how the service is available, how long it takes to recover when failed
- Scalability– Coping with both high and low numbers of user requests, and how efficiently it is able to adapt
- Interoperability– Ability to function in different environments (languages, platforms, libraries)
- Security– To be discussed in part II

Lecture - Mobile and Ubiquitous Computing

10:00 03/03/25 Amanda Peart

Ubiquitous Computing

Ubiquitous computing is the concept of integrating computing and computers into everyday objects and environments, making technology available anywhere and any time, often without requiring explicit user interaction.

For example, a building system which uses RFID cards or infrared tags to identify and track users as they move around the building, adjusting rooms to suit their preferences, or automatically disabling lights when everyone leaves a room.

Sensors and Actuators

Devices in a ubiquitous computing system typically fall into two categories; sensors and actuators. A sensor is a device which measures physical parameters, typically of the environment around it, this could be light level, temperature, humidity, etc. An actuator is a device which is controllable via the software system, this could be things like air conditioning units, lights, motors, locks, etc.

Association and Discovery Services

Volatile components of a system need to interoperate, and access to the network is essential for devices to be able to communicate with one another and fulfil their purposes. The challenge with this is the vast number of devices which need to be associated, and that they're typically embedded devices with little-to-no controls.

A discovery services is one which allows users, devices and other software to discover where services are located, and how they should be accessed.

Discovery Services

There are several ways to discover services on a network, one of the simplest is to have a specialised server providing these services, which allow other services to register and de-register from it's directory, and allows clients and services to loop up other services available on the network. This works well in a network that already has dedicated servers, but in peer-to-peer networks, another solution is needed.

In a peer-to-peer network, each device maintains it's own directory of services, and these services are advertised to new devices when they join the network. This does mean that there is no central directory, and that it would be possible for devices to spoof services maliciously.

Discovery Challenges

There are two different models for advertising services, being the push and pull models. In the push model, the discovery service periodically broadcasts what services are available on the network. This means that clients always have the most up-to-date information, but network bandwidth is wasted, especially when there are no changes to the available services. In the pull model, clients request the service directory from the server. This is more efficient in terms of bandwidth, but means that clients may not have up-to-date information and more requests may be needed to gather all the information they need.

Lecture - Fault Tolerance

10:00 17/03/25 Amanda Peart

Fault tolerance is the ability of a system to continue operating in spite of any failures. It is very important as failures are more or less inevitable, given the sheer complexity of many distributed systems.

Types of Failure

- Crashing
  - A node or the software running on it completely stops responding, for example if power fails to the node, or database software crashing in the middle of transactions.
- Omission
  - Messages are lost or dropped in transmission, for example network congestion causing transactions to take much longer than expected, or network devices failing and packets being lost.
- Timing
  - Part of the system responds too late or too early, for example a banking system not processing transactions in realtime, or transactions not being processed in chronological order leading to inconsistent results.
- Byzantine
  - A node sending incorrect data due to code errors or malicious attacks, for example a node in a blockchain network intentionally sending false transactions, or the software producing incorrect results due to a logic error.
- Network Partition
  - Some nodes become disconnected from the system as a whole, for example two datacenters losing communication, causing inconsistencies in the stored data.

Redundancy and Replication Techniques

- Data Replication
  - Primary-Backup model, where one server is the main server which handles writes, but there is one or more backup nodes which are ready to take over requests if the primary node fails
- Quorum-Based Redundancy
  - The majority of nodes must agree before an update or transaction can be completed
- Process Redundancy
  - Active Replication– All nodes run in parallel
  - Passive Replication– One node acts as a primary, all others act as passive backups
- Voting Mechanisms
  - Used in datastores like databases to ensure all nodes agree before committing a transaction

Recovery Mechanisms

- Checkpointing
  - Save the state of the entire system periodically, so only data modified after that point is lost in the event of a failure
- Rollback
  - Restoring the state of the system to a checkpoint after a failure
- Message Logging
  - Store messages before they are processed so they can be re-processed in the event of a crash or other failure
- Self-healing
  - Using algorithms or machine learning to monitor the state of the system and predict if system is likely to fail imminently, and either resolve automatically or page administrators

Consensus Algorithms

Consensus algorithms are needed when distributed nodes are required to agree on a single value, even with failures.

- Paxos Algorithm
  - An algorithm designed to ensure consensus in unreliable networks where nodes crash or messages are lost
  - The algorithm works in 3 fazes; the prepare phase, accept phase and learn phase
  - During the prepare phase, a proposer sends a proposal with a unique numbered to all acceptors, if the acceptors haven't already accepted a proposal with a higher number, they respond with the last accepted value
  - During the accept phase, the proposer selects the most recently accepted value and sends an accept request, which the acceptors acknowledge if no proposal with a higher number exists
  - If a majority of acceptors accept a proposal, it becomes the agreed value and is then sent to learners, who apply the accepted value.
  - It is typically used where durability is most important for large datasets, such as a file or database
- Raft Algorithm
  - In a Raft network, there is one Leader and multiple Follower servers. There may also be Candidate servers depending on the state of the network
  - The elected leader node is the only one which can interact with the client. All other nodes become followers and sync their value to that of the leader
  - The follower nodes sync their value to the leader at regular intervals. If the leader server stops working, one of the followers can contest an election and become the leader
  - At the time of contesting to become leader, followers become candidates and request votes from other nodes. The node with the most votes then becomes the leader. When the network first starts up, all nodes are candidates
  - When the client sends a request to the leader, it appends the request to it's log and sends the update to the followers. The followers then copy the entry and send an acknowledgement to the leader, which applies the entry to the system only when a majority of followers have ac-

knowledged the change

- Byzantine Fault Tolerance
  - BFT is typically used in networks where nodes may act maliciously or fail with invalid data
  - It uses cryptography, redundancy and a quorum system to ensure integrity in spite of faulty nodes
  - It will only operate reliably when the assumption that at most $\frac{(n-1)}{3}$ nodes are faulty
  - This also means that at least $3f+1$ nodes are required to tolerate f failures, and at least $2f+1$ 'honest' nodes must reach a consensus to outvote any malicious nodes
  - To achieve fault tolerance, 3 steps are typically followed: nodes exchange messages to verify information from other nodes; consensus is required before a decision is finalised; verification ensures that faulty or malicious nodes cannot influence the system
  - There are 3 main types of BFT algorithms
    * Practical BFT– Uses a leader-based approach to reach consensus among nodes
    * Federated BFT– Used in decentralised networks where each participant may wish to select specific nodes to trust
    * Proof of Work– Used in several blockchains

(To be filled in after seminar)

Fault Detection Mechanisms

- Heartbeats
  - Nodes periodically send a 'heartbeat', effectively just a message saying they're still online and working as expected
- Timeouts
  - If responses are taking longer than expected or aren't being received at all, the node is marked as failed
- Gossip Protocols
  - Nodes spread information about failures throughout the system using peer-to-peer communication

Part II

Security