INTRODUCTION (Chapter 1)

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

Characteristics of a distributed system

Differences between the nodes and **communication** are (mostly) **hidden** from users
Interaction is **consistent** and **uniform** regardless of where/when the interaction takes place
Users should **not** notice **dead nodes** or **nodes being maintained**

Middleware

To **support heterogeneous** computers **middleware** is needed (page 3 top)
Logically above the **OS** layer (under **application** layer) (Figure 1-1 page 3)
Hides differences in **hardware** and **OS** from applications

Communication middleware

Remote Procedure Calls (*RPC*) (***Tight coupling***)

Send a request to another application by doing a local procedure call
The request is packaged and sent to the callee
The result is returned to the caller

Remote Method Invocation (*RMI*) (***Tight coupling***)

Same as RPC except it operates on objects instead of applications

Message Oriented Middleware (*MOM*) (***Loose coupling***)

Send messages to logical contact points
Other processes can indicate interests in specific messagetypes
They will recieve messages of that type
Publish / subscribe system

Goals for distributed systems

1. Should **make resources easily accessible**
2. Should reasonably **hide that resources are distributed** across a network
3. Should be **open**
4. Should be **scalable**

Distribution transparency (Goal number 2) (Figure 1-2 page 5)

Transparrency: *The distsys can represent itself to users and applications as a single computer*

1. **Access**	Hide differences in data representation and how a resource is accessed
2. **Location**	Hide where a resource is located

3. **Migration** Hide that a resource may move to another location
4. **Relocation** Hide that a recource may be moved to another location while in use
5. **Replication** Hide that a resource is replicated
6. **Concurrency** Hide that a resource may be shared by several competitive users
7. **Failure** Hide the failure and recovery of a resource

Openness (Goal number 3)

The distsys must describe the **syntax** and **semantics** of the services provided
The rules are **formalized in protocols**
Defined by **Interface Definition Language** (**IDL**) (*just like Java interfaces*)
Interfaces must be **complete** and **neutral**

complete everything necessary to make an implementation has been specified
neutral specifications do not prescribe what an implementation should look like

It should be easy to replace components (*code*)

Scalability (Goal number 4)

1. **In size** easy to add users and resources
2. **Geographically** users may lie far apart
3. **Administratively** easy to manage even if there are many independent administrators

1. Size problems

Centralized services, data and algorithms (*becomes bottlenecks*)
Sometimes not possible to decentralize (*eg. confidential information*)

Decentralized algorithm characteristics

1. No machine has complete information about the system state
2. Machines make decisions based only on local information
3. Failure of one machine does not ruin the algorithm
4. There is no implicit assumption that a global clock exists

2. Geographical problems

Many distsys are based on synchronus communication (*client blocks → too slow*)
The network is unreliable (*packages get lost*)
Connection is point-to-point (*LAN can use broadcasting*)

General solutions to scalability problems

Utilize asynchronous communication
Move data processing to the client
Split components up and distribute them across the system (*DNS*)
Replicate components across the system

Erroneous assumptions when building a distributed system

1. The network is reliable (*i.e. no packets are lost*)

2. The network is secure
3. The network is homogeneous (*i.e. computers have the same OS*)
4. The topology does not change (*i.e. noone disconnects or joins*)
5. Latency is zero
6. Bandwidth is infinite
7. Transport cost is zero
8. There is one administrator (*i.e. politics and security problems*)

Types of distributed systems

1. Distributed **computing** systems
2. Distributed **information** systems
3. Distributed **embedded/pervasive** systems

Distributed computing systems (Type 1)

1. **Cluster computing** Identical computers (*hardware, OS, etc.*) connected via LAN
2. **Grid computing** Heterogeneous computers typically connected via WAN (*Internet*)

Distributed information systems (Type 2)

1. **Transaction Processing Systems**
    - A transaction is either fully executed or not at all
    - **ACID** - **A**tomic, **C**onsistent, **I**solated & **D**urable
    - Subtransactions - a transaction may split up into subtransactions
        - Subtransactions **may roll back** (*not durable*) after commit if supertransaction fails
2. **Enterprise Application Integration**
    - Applications must be able to communicate
    - Popular communication middleware **RPC**, **RMI**, **MOM**

Distributed pervasive systems (Type 3)

Characteristics small, battery-powered, mobile, wireless connection
Examples Home systems, Electronic Health Care Systems and Sensor Networks

Summary

Pros Easier to integrate applications on different computers into a single system
Good scaleablity with respect to size of the underlying network
Cons More complex software
Degradation of performance
Weaker security

ARCHITECTURES (Chapter 2)

Component A modular unit with **well-defined interfaces** so it is replaceable in its environment

Connector Mediates **communication**, **coordination** and **cooperation** among components

Architectural styles

1. **Layered architectures**
2. **Object-based architectures**
3. **Data-centered architectures**
4. **Event-based architectures**

Layered architecture (Type 1) (Figure 2-1a page 35)

Components at layer *n* can only call components at layer *n-1*
Requests flow downward, and results flow upward

Object-based architecture (Type 2) (Figure 2-1b page 35)

Each object corresponds to a component
Communication via Remote Procedure Calls (RPC)

Data-centered architecture (Type 3)

Processes communicate through a common (passive or active) repository
E.g. systems that rely on shared distributed file systems

Event-based architecture (Type 4) (Figure 2-2a page 36)

Send out events, only nodes listening for the event-type recieves it
Publish/subscribe systems
Decoupled in space (don't need to know who wants to recieve the event)

Shared data spaces (Type 3 + 4) (Figure 2-2b page 36)

Hybrid between **data-centered** and **event-based** architecture
Decoupled in time and space (*shared data space remembers publishes*)

System architectures

1. Centralized architectures
2. Decentralized architectures
3. Hybrids

Centralized architecture (Type 1)

Client/server model	Server	a process implementing a specific service
	Client	a process that requests a service by sending a request

Connectionless or connection oriented communication

Connectionless good performance, impossible to tell if packages gets lost
Connection oriented bad performance, more reliable, advantage on WAN

Three logical levels in client/server model (Figure 2-5 page 41)

1. The **user-interface** level (e.g. a browser displaying an HTML page)
2. The **processing** level (e.g. the application)
3. The **data** level (e.g. the database)

The physical organization of theese can vary

Two-tiered architecture	One client and one server (*placement of logical levels vary*)
Multitiered architecture	Multiple physical layers (*physical division of logical levels*)

Decentralized architecture (Type 2)

Distribution of logical levels in a client/server environment

Vertical distribution	Add tiers (*each machine serves a single logical level*) Different jobs, exact same dataset
Horizontal distribution	P2P (*each machine operates on its share of the complete dataset*) Same job, (possibly) different parts of the dataset

Peer-to-peer systems

All peers are equal
A peer both act as **server and client** at the same time (*servent*)

Overlay network	A network where processes represent nodes and links represent possible communication channels (i.e. TCP connections)

Two types of P2P architectures

1. **Structured Peer-to-Peer Architectures**
2. **Unstructured Peer-to-Peer Architectures**

Structured Peer-to-Peer Architectures (Type 1)

Overlay network constructed using deterministic procedure

Membership managemnt	How nodes organize into an overlay network (*join/leave*)
Distributed hash table (DHT)	Most-used procedure to create overlay network

Items are assigned a random key
Nodes in the system are also assigned a random key
The items are deterministically and uniquely mapped to one node
When looking up a data item, the network address of the node having it, is returned

Chord system (DHT system) *(Figure 2-7 page 45)*

Nodes are organized in a ring
Item with key *k* is mapped to the node with smallest identifier $id \geq k$ (*successer of k, succ(k)*)

Content Addressable Network (CAN) (DHT system) *(Figure 2-8 page 46)*

d-dimentional coordinate space
The space is partitioned among peers

Items maps to coordinates which maps to peers *(see figure 2-8 page 46)*

Unstructured Peer-to-Peer Architectures (Type 2)

Overlay network constructed using random algorithms (*overlay netowrk is a* ***random graph***)
Data is assigned to random nodes (*finding data by flooding the network*)

Random graph	Graph generated by random algorithm
Partial view	The neighbours known by the node constructed by pushing/pulling from neighbours

Unstructured membership management *(Figure 2-9 page 48)*

Push mode	P1 sends part of its partial view to P2. P2 updates view
Pull mode	P1 sends a request to P2. P2 sends part of partial view. P1 updates view

The two modes are **complementary**, so they are often **used in conjunction**

Unstructured topology management *(Figure 2-10 page 49)*
Use a combination of structured and unstructured overlays
Random peers pushes randomly selected nodes to structured peers
Structured peers chooses its partial view by using a **ranking function** (*e.g. closest peers*)
Result is a network based on a specific topology (*according to the ranking function*)

Superpeers

Keeps an index of data items and **acts as a broker** (*tell peers where the data is*)
Each peer is connected to a superpeer
Superpeers are connected in a seperate peer-to-peer network
How to select a superpeer? **Leader-election problem** *(Chapter 6)*

Hybrid architechture (Type 2 + 3)

Edge-Server systems	The ISPs are on the edge of the internet
Torrent systems	First connect to tracker (server), then join P2P network

Summary

Software architecture	Logical organization of the software (*interaction, structure etc.*)
System architecture	Physical organization (*where the logical components are placed*)
Architectural style	Organizing interaction between software components
Centralized architecture	Client server model
Decentralized architecture	Peer-to-peer model
Peer-to-peer	Processes organized into an overlay network All peers play an equal role (*except superpeers*)

Structured overlays Uses a deterministic algorithm to locate data items (*Chord system*)
Unstructured overlays Each peer has a partial view which which it updates (*push/pull*)

COMMUNICATION (Chapter 4)

All communication is based on low-level message passing
We need better abstractions **Remote Procedure Call (RPC)**
Message Oriented Middleware (MOM)

ISO OSI model Defines set of protocols to deal with communication issues (*7 layers*)
Protocol Communication rules
Connection oriented Handshake before data exchange (*telephone*) (*TCP*)
Connectionless Just send the data (*sending mail*) (*UDP*)

OSI model

Physical layer	Hardware stuff, send 1's and 0's (*how many volts for 0 and 1*)
Data link layer	Error correction and detection
Network layer	Concerned with routing (*IP*)
Transport layer	Provide a (reliable) connection. Higher level primitives (*TCP & UDP*)
Session layer	Synchronization and transfer checkpoints
Presentation layer	Gives meanings to the bit-streams
Application layer	Everything that does not fit into other levels

An adapted model where session and presentation is replaced by middleware layer is more appropriate for distributed systems

Middleware layer Provides application independent protocols

Communication types

Persistent The message is stored by middleware.
Sender and reciever do not need to be running at the same time
Transient Reciever must be running when the message is sent

Synchronous Sender blocks at least until the messages has been recieved
Asynchronous Sender does not wait, but gets interrupted when reply comes

Remote Procedure Call

Goal Achieve transparency (*calls are made the same way as normal procedure calls*)
Method Page 129

Problems

Big endian vs. **little endian**
Call by reference Replace it with call by copy/restore (*e.g. copy array onto stack*)

It is nescesarry to negotiate upon the **message format** and the **data representation**

Trancient message-oriented communication

Trancient communication uses **sockets**

Socket A communication endpoint to which an application can read and write data
An abstraction over the actual communication end point

Figure 4-15 demonstrates use of sockets

Message Passing Interface (MPI)

Sockets were deemed too inefficient (*only send and recieve primitives*)
Standardised primitives for **highly efficient** message passing
Built for **trancient communication**

Persistant message-oriented communication

Message-queuing systems = **Message-oriented middleware** (*MOM*)
Provides storage for messages until the reciever becomes available
Extensive support for persistant asynchronous communication
Loosely coupled in time

Source queue
Destination queue
Queue manager
Relays
Keep the topology of the queueing network static
Routers
Relays can be used to multicast and to log messages etc.
Message brokers (*converting messages from one format to another*) (*Figure 4-21 page 150*)

Multicast communication

Application level multicasting

Nodes organize into overlay network (*tree or grid*)
Grid is more robust
does not take routers into account, and may therefore be less efficient than network level routing

Constructing an overlay network

Quality of the overlay network is measured by
Link stress How often a package crosses the same link (*measured for each link*)

Stretch Latency ratio between two nodes **in the overlay** and **in the optimal path**
Relative Delay Penalty (RDP) Another word for **stretch**
Tree cost Minimizing the aggregated link costs (*bottom page 168*)

Switch trees Creates an overlay tree close to a minimum spanning tree
Allow nodes to switch parents (*decided by some criteria*) (*prevent loops*)
When a node has a dead parent, it just connects to the root as its new parent

Epidemic protocols (fast and scalable)

Gossip-Based Data Dissemination

Three methods
1. **Push only** Best when few nodes are infected
2. **Pull only** Best when many nodes are infected
3. **Push and pull** Best choice

Spreads information in time **O(*log(N)*)**

Infected node Holds information it is willing to spread
Susceptible node Has not yet recieved the information
Removed node Not willing or able to spread information

Death certificates Deletion of data (*page 173*)
Size of network One node sets a variable to 1. All others sets it to 0. Then calculate 1/n
Directional gossiping Nodes that are connected to few other nodes are contacted with high probability. Such nodes form a bridge to other remote parts of the network.

NAMING (Chapter 5)

1. **Flat naming**
2. **Structured naming**
3. **Attribute-based naming**

1. **Address**
2. **Identifier**
3. **Human-friendly name**

Access points and addresses

Each entity has one or more **access points**. The name of an access point si called an **address**
A telephone can be viewed as an **access point** of a person
The telephone number corresponds to the **address**

Identifiers

1. An identifier refers to at most one entity
2. Each eneity is referred to by at most one identifier
3. An identifier always refers to the same entity (*i.e. it is never reused*)

Location independent name Name independent of the entities address

Flat naming (Type 1)

Does not contain information about how to locate the access point of the associated eneity

Finding entities

Broadcasting (send to all on network) and multicasting (send to multicast group)
Works only on LAN
Spam all nodes (or the multicast group), and get reply from the one that holds the entity

Forwarding pointers
When an entity moves (from one address to another) it leaves a pointer to its new address

Problems
The chain can become very long. Long chains make it expensive to contact the entity
All intermediate "hubs" will have to maintain the link
High vulnerability to broken links
Solution
Let the last server stub connect directly to the home location (*Figure 5-2 page 185*)

Home-Based Approaches
Broadcast/multicast and forwarding pointer imposes scalability problems

Home Location Keeps track of the current location of the entity
Used as a fallback for services based on forwarding pointer based systems

Home agent (Figure 5-3 page 187)
Care-of-address (Figure 5-3 page 187)

Drawbacks
Uses a fixed home location (*which can be far away*)
The home location must always exist

Distributed hash tables (DHT)

Finger table Keeps references to other nodes for faster searching
Allow searching in time *O(log(N))*
(*Figure 5-4 page 189*)

Finger tables are kept up to date by deamons. Just query *succ(k)* for each entry in the table

Exploiting network proximity

Topology-based assignment of node identifiers

The identifier is based on i.e. the geographical placement of the node
Identifiers might not be spread equally and one node will hold all files

Proximity routing

Give each node multiple successors
Broken links give less problems

Proximity neighbor selection

A joining node gets information about the overlay from several nodes. Picks closest one

Hierarchical approaches (Chapter 5.2.4 page 191)

Stuctured naming (Type 2)

Name spaces Name spaces can be represented as a directed graph
Naming graphs Example graph over filesystem (*Figure 5-9 page 196*)
Most often a tree or directed asyclic graph
Name resolution Looking up a name

Logical partitioning of name spaces (page 203 and figure 5-14 page 205)

Global layer The root and its children (*stable - not often changed*)
Administrational layer Nodes managed within a single organization
Managerial layer Representing hosts in a local network (*Change regularly*)

DNS Zone Part of the name spaces that is implemented by a spearate server
Iterative name resolution Most often used (*caching only at user level*)
Figure 5-15 page 206
Recursive name resolution Con: Puts a higher performance demand on servers
Pro: Provides serverside caching
Pro: Cheaper with respect to communication
Figure 5-16 page 207

Flooding of the high level DNS servers can be avoided with a decentralized version

Attribute based naming (Type 3)

Describes entities by (*attribute, value*)-pairs
Each eneity has a set of associated attributes

Recource Description Framework Entities are described by triplets (*subject, perdicate, object*)

Hierarchical implementations (LDAP)

Combination of **structured** and **attribute-based** naming
LDAP Lightweight directory access protocol

Queries consists of a conjunction of pairs (*/C=DK/O=Aarhus Uni./CN=Main Server*)

Decentralized implementations

(*attribute, value*)-pairs need to be mapped so searching is efficient (*not exhaustive*)

Summary

Flat naming

1. **Broadcasting and multicasting** (works only on LAN)
2. **Forwarding pointers** (when an entity moves it leaves a pointer to its new location)
3. **Home base** (each time an entity moves it updates its home base)
4. **Structured peer-to-peer network** (*identifiers are hashed and assigned to peers (DHT)*)
5. **Hierarchical search tree** (*each node knows of all entities in its subtree*)

Structured naming

Names are organized into a **name space**
A **name space** can be reperesented by a **naming graph**
A **naming graph** is organized by a **rooted asyclic directed graph**
Name resolution is done by traversing the **naming graph**

Attribute-based naming

Entities are described by **(*attribute, value*) pairs**
Name resolution requires an **exhaustive search** through all descriptors
    Can only be done when all descriptors are stored in a single database
Alternative is to **map pairs into DHT-based systems**
Gradually replace **name resolution** by **distributed search techniques** (*e.g. gossipping*)
Creation of **semantic overlay networks** (*for efficient lookups*)

SYNCHRONIZATION (Chapter 6)

Clock synchronization

Is it possible to synchronize all the clocks in a distributed system?

Clock offset	Difference between actual time and the computers time
Clock skew	How much slower/faster the machine clock is compared to actual time (*Figure 6-5 page 239*)

Real time systems depend on actual clock time (*makes use of external clock(s)*)
    How do we synchronize them with real world clocks?
    How do we synchronize the clocks with each other?

NIST sends out shortwave radio transmissions at every UTC second (*accurate to +/-10ms*)

Geostationary Environment Operational Satellite does the same (*more accurate*)

GPS system

Problems

1. It takes a while before data on a satellite's position reaches the receiver
2. The receiver's clock is generally not in synch with that of the satellite

Solution (*if all satellite clocks are accurate*) We need **4 satellites** to determine position

Synchronization

1. **A computer has a WWV reciever** (*keep all other computers synchronized to it*)
2. **No WWV reciever is present** (*keep all the machines together as well as possible*)

Clocks must be synched at least every *δ / 2ρ*

δ=maximum allowed clock difference

ρ=maximum drift rate (defined by manufacturer)

Network Time Protocol (NTP)

Estimate network delay (*Figure 6-6 page 240*)
Devide servers into strata (*server with reference clock (WWV) is stratum-1*)
Servers synchronize after other servers with lower strata
After synchronization the server will be Strata-(k+1) (*when synchronized with strata-k server*)
Accurate in the range of 1-50ms

The Berkeley algorithm

Usefull when no WWV reciever is present
The time server computes the average of all servers times and adjusts all clocks after that
Needs to be manually updated to conform with real time
Often real time is not nescesarry, just agreement on some consistent time is needed

Logical clocks

What usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order of which events occour.

1. **Lamport clocks**
2. **Vector clocks** (extension of Lamport clocks)

Lamport clocks

Happens-before a → b "a happens before b"

1. If a and b are events int the same process and a occurs before b, then a → b
2. If a is the event of sending a message and b is recieving that message, then a → b

Clocks are updated following 3 steps
1. Before executing an event, P_i executes $C_i \leftarrow C_i + 1$
2. When process P_i sends a message m to P_j it sets *m*'s timestamp *ts(m)* to C_i after step 1
3. Upon reciept of *m*, process P_j adjusts its time to $C_j \leftarrow max\{C_j, ts(m)\}$ and executes step 1

Used to implement totally ordered multicasting

Vector clocks

Lamport clocks does not guarantee that a happened before b even if a → b is true
Lamport clocks does not capture **causality**

Each process maintains a vector VC_i with two properties (*page 249*)

1. $VC_i[\ i\ ]$ is the local logical clock of P_i
2. $VC_i[\ j\ ]$ is P_i's knowledge of the local time at P_j

Three steps for maintaining the vector clock (*more or less like Lamport*) (*page 250*)
1. Before executing an event, P_i executes $VC_i[\ i\ ] \leftarrow VC_i[\ i\ ] + 1$
2. When process P_i sends a message *m* to P_j it sets *m*'s timestamp *ts(m)* to VC_i after step 1
3. Upon reciept of *m*, process P_j adjusts its vector to $VC_j[k] \leftarrow max\{VC_j[k], ts(m)[k]\}$ for each *k* and executes step 1

Causal ordering
Delay messages until theese conditions are met

1. $ts(m)[\ i\ ] = VC_i[\ i\ ] + 1$
2. $ts(m)[\ k\ ] \leq VC_j[k]$ *for all* $k \neq i$

Mutual exclusion

Token-based Only the process who has the token is allowed to access the recource
- **Token** A special message which is passed between the processes
- **Problem** When the token is lost (*process holding it crashes*)

Permission-based Require premission from the other processes

Centralized solution

Elect a process to be coordinator
Every time a process wants to use a resource it asks the coordinator for permission
Drawback Single point of failure (*the coordinator crashes*), coordinator becomes bottleneck

Decentralized solution

Each resource is replicated *n* times, and every replica has its own coordinator
To access a resource a process must get more than *n/2* votes
Coordinator crash is taken care of (*page 254*) (*more stable than centralized version*)

Distributed algorithm

Multicast a message containing the resource name, the process number and a timestamp
The system is built on **totally ordered multicasting**
When it has recieved OK messages from all peers, it can access the resource
After access it sends OK message to pending requests

Three different scenarios for the recipent of the message (*page 256 top*)
1. The reciever is not using, and does not want to use the resource - sends OK reply
2. The reciever uses the resource - queues the request
3. The reciever wants to use the resource
    3.1. If the timestamp is higher than the one in its own request - queues the request
    3.2. If the timestamp is lower than the one in its own request - sends OK reply

Drawbacks Slower, more complicated, more expensive and less robust than centralized

A Token Ring Algorithm

On a LAN the processes is ordered into a logical ring
A token is passed around the ring
When a process has the token it can use the resource (*to which the token applies*)
Drawbacks If the token is lost. It is hard to tell if it is lost or being used (*dead vs slow peer*)

Comparison (Figure 6-17 page 259)

Election algorithms

Assume all processes has a unique number
Assume every process knows the process number of all other processes

The Bully Algorithm (LAN) (Figure 6-20 page 265)

When a process notices the **coordinator does not respond** it initiates an election as follows:
1. P sends an ELECTION message to all processes with higher numbers
2. If no one responds, P winds the election and becomes coordinator
3. If one of the higer-ups answers, it takes over. P’s job is done

When an ELECTION message is recieved the reciever starts a new election
When a process wins an election it multicasts to all other processes

A Ring Algorithm (LAN) (Figure 6-21 page 266)

Nodes are ordered into a ring
When a process
When a process notices the **coordinator does not respond** it initiates an election
    Builds a ELECTION message and sends it around the ring
    Each process aggregates its number to the ring

When the it comes back to the sender it is converted to COORDINATOR message
Every node updates their coordinator
When it comes back it is deleted

Wireless coordinator election (Ad hoc networks) (Figure 6-22 page 268)

A node can start an election by sending an ELECTION message
When a node receives a message for the first time, it designates the sender as its parent and multicast the ELECTION message to its neighbours (*which creates a tree*)
If it already has a parent it acknowledges the message
When all acknowledgments are recieved it reports the best node back to its parent
When the initiator of the election is finished, the root multicasts who is the new leader

Selecting superpeers

Requirements for superpeers
1. Normal nodes should have low-latency access to superpeers
2. Superpeers should be evenly distributed across the overlay network
3. A give fraction of the number of peers should be superpeers
4. Each superpeer should not need to serve more than a fixed number of normal nodes

Solution 1
Reserve *k* bits to identify superpeers (*i.e. k=3, then **111**00101 is a superpeer*)
Route lookup for entity *p* to ***p AND 11100000*** (*which will contact the superpeer responsible*)
A node can check if it is superpeer by looking up <u>*id AND 11100000*</u> and see if it gets the lookup

Solution 2
Distribute *n* tokens in the network
Use gossipping to distribute tokens evely
When a token position is stable, it is a superpeer

CONSISTENCY AND REPLICATION (Chapter 7)

Replication	Enhance reliability and improve performance
Consistency model	A contract between processes and data store

Continous consistency

Conit	A unit on which to seperately measure consistency
Tentative update	A local update not yet committed

Continuous consistency ranges (defining inconsistencies)
1. Deviation in **numerical values** between replicas (*numerical deviation or write count*)
2. Deviation in **staleness** between replicas (*time since last update*)
3. Deviation with respect to the **ordering of update operations** (*different ordering of writes*)

Consistent ordering of operations

Example How to order the commits of tentative updates

Sequential consistency

All processes must see the **same interleaving** of write operations (*Figure 7-5 page 283*)
All outputs of *Figure 7-7 page 284* are valid in terms of sequential consistency (*more exists*)
A program that *only works for some* of the outputs violates the contract and is incorrect

Causal consistency

Weaker than sequential consistency (*Example in figure 7-8 and 7-9 page 285*)

Writes that are potentially causally related must be seen by all processes in the sam order. Concurrent writes may be seen in a different order on different machines.

Potentially causally relatedP1 writes x, P2 reads x and writes y
y might be causally related to x ($y \leftarrow x + 1$) but not nessesarily ($y \leftarrow 2$)
Concurrent writes Writes that are not potentially causally related

Implementation requires keeping track of which processes have seen which writes.
Done by keeping a dependency graph of which operations depend on which
Can be done with **vector clocks**

Grouping operations

Read and *write* are too primitive operations so grouping operations atomically is done
Synchronization variable An lock that must be acquired before data access is permitted

Three criteria for entry consistency
1. When P1 does an acquire, the operation must not complete until all data guarded by the lock (*synchronization variable*) are brought up to date
2. Before updating the process must enter a critical section in exclusive mode
3. If a process wants to enter a critical region in *nonexclusive mode* it must fetch the most recent copies of the guarded shared data from the owner of the synchronization variable

Consistency model Describes what can be expected on a set of data when concurrent updates take place
Coherence model Concerned with only a **single data item** instead of a set of items

Client-centric consistency models

1. **Monotonic reads**
2. **Monotonic writes**
3. **Read your writes**

4. **Writes follow reads**

Eventual consistency

Systems with no (*or very rare*) write-write conflicts
Updates are propagated to other servers in a lazy fashion (*no need to rush*)
All replicas will become consistent if no updates happens for a long period

Problems occour when clients access different replicas over a short period of time
Solved by introducing **client-centric consistency**

Monotonic Reads

If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

Monotonic Writes

A write operation by a process on a data item x is completed before any successive operation on x by the same process.

Read Your Writes

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

Writes follow Reads

A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same fr a more recent value of x than was read.

Replica Management

Placing replica servers	Physical placement of replica servers
Placing content	Placement of specific data items

Replica server placement (chapter 7.4.1 page 296)

Split the network into cells and place replica servers in the *k* most crowded cells

Content replicaiton and placement

Server initiated replicas
A file F is replicated if a number of requests (***replication threshold***) is exceeded
If requests to F drops below some **deletion threshold** it is deleted (*unless it is the only replica*)
Client initiated replicas (aka. client caches)
Cache hit When the file requested is already cached

More cache hits can be generated by letting clients share a cache

Content distribution

1. **Propagate only a notification of an update** (*invalidation protocols*)
    Use little network bandwidth
    Best when read-to-write ratio is small
2. **Transfer data from one copy to another**
    Uses a lot of network bandwidth, but all replicas are up to date when queried
    Best when read-to-write ratio is high
3. **Propagate the update operation to other copies** (*aka.* ***active replication***)
    Needs more processing power

Pull (*client-based protocol*) **vs. push** (*server-based protocol*) **protocols**

Push	Often used between permanent and server-initiated replicas Maintains high level of consistency Best when read-to-write ratio is high
Pull	Efficient when read-to-write ratio is low Check table Figure 7-19 page 304
Hybrid	Use **leases** (*when a lease is valid the server pushes updates to the client*)

Three types of leases

Age-based	Data that has not been updated for long is not likely to be update soon Long lease given
Renewal-frequency-based	Long lease to clients how often needs to refresh their cache (*gives long leases to popular content*)
State-space overhead	When a server gets overloaded it shortens leases to reduce state

Consistency protocols

Continous consistency

1. **Bounding Numerical Deviation** (*push updates when deviation becomes too big*)
2. **Bounding Staleness Deviations** (*keep a realtime vector clock and pull updates*)
3. **Bounding Ordering Deviations** (*Specify a maximum amount of tentative writes*)
    Primary-based or **quorum-based** protocols are used for ordering of tentative writes

Primary-based protocols

In the case of **sequential consistency**, primary-based protocols prevail
Each data item has a **primary** that is responsible for coordinating write operations on it

Remote-write protocols	*Figure 7-20 page 309*
Local-write protocols	*Figure 7-21 page 310*

The primary is moved to the clients server so **successive writes will be fast**

Replicated-write protocols

Write operations can be carried out at multiple replicas instead of only one

Quorum-based protocols

To write, a client must contact <u>over</u> half the servers and get an OK. Then update all
To read, a client contacts <u>more than</u> half the servers and reads from the one where the file has the highest version number

Quorums (more general than above)
Define a **read quorum** *Qr* and a **write quorum** *Qw* satisfying (*Figure 7-22 page 313*)

1. *size(Qr) + size(Qw) > N* *N = number of servers (prevents read-write conflicts)*
2. *size(Qw) > N/2* *N = number of servers (prevents write-write conflicts)*

Implementing Client-Centric Consistency

Each client keeps track of two sets of writes
Read set Writes relevant for the read operations performed by the client
Write set (identifiers of) writes performed by the client

General solution (*naive*)
Update the server using either the clients read or write set (*whatever is appropriate*)
Execute the request and return result
Update the clients read and/or write set
Problem The read and write sets can get very big (*require a lot of bandwidth*)

Improved solution
Replace the read and write set with a vector clock
If the clients vector clock is greater than the one on the server, the server updates

FAULT TOLERANCE (Chapter 8)

Dependable systems requirements

1. **Availability** (*how likely it is that the system is up and running at any give instant*)
2. **Reliability** (*how long without interruption the system runs*)
3. **Safety** (*if a system breaks down, nothing catastrophic happens*)
4. **Maintainability** (*how easy a failed system can be repaired*)

Fault tolerance A system continues to operate normally even when faults are present

Transient fault The fault only occours once
Intermittent fault The fault comes and goes randomly or periodically (*loose contact*)

Permanent fault Burn-out chips, software bugs, disk head crashes etc.

Different types of failures in figure 8-1 page 324

Three kinds of redundancy

1. **Information redundancy** (*e.g. an error correction code is added to messages*)
2. **Time redundancy** (*the messages is sent again after a time interval*)
3. **Physical redundancy** (*e.g. more replica servers are set up*)

Process resilience

Organize processes into groups (*when a message arrives all processes in a group recieves it*)
Point If a process fails, another process from the group will execute the request

Group structure (Figure 8-3 page 329)

Flat group No single point of failure, more complex decision making
Replicated-write protocols are used as the form of replication
Hierarchical group Single point of failure (*coordinator*), simple decision making
A primary-backup protocol is used for replication (*coordinator is primary*)

Agreement in faulty systems

Not always possible (*Figure 8-4 page 333*)
Byzantine fault The faulty process just returns random stuff (*sick process*)

Byzantine agreement problem
A **k fault tolerant** system should have at least *3k + 1* processes to reach agreement

Failure detection

Ping other nodes to see if they are alive and wait for timeout (*crude*)
Use gossipping to tell other nodes that you are alive
When a possibly dead node is detected, ask neighbours if they think the same

Reliable client server communication

Point-to-point communicaion

TCP masks omission failures by resending and acknowledging packages
TCP does not mask connection failures though

RPC semantics in the presense of failures

Five types of failures can happen

1. The cilent is unable to locate the server
2. The request message from the client ot the server is lost
3. The server crashes after recieving a request
4. The reply message from the server to the client is lost

5. The client crashes after sending a request
   Orphan extermination, reincarnation, gentle reincarnation, expiration (*page 342*)

Reliable group communication

Solution 1 Create reliable point-to-point channels to all clients
Solution 2 Send multicast and let all acknowledge or say they miss a package
On large networks **feedback implosion** can occour (*figure 8-9 page 344*)

Reducing feedback

Feedback supression (support relatively small multicast groups)

When a client in a group is missing a message it sends a retransmission request to the group
After that it sends a request for retransmission to the original sender
Other group members that have not recieved the package does not need to contact the sender
Only one retransmission request needs to be sent to the sender

Hierarchical feedback control (support very large multicast groups)

All groups are organized into a tree (*the group containting the sender is the root*)
Each group has a coordinator (*who multicasts messages within the group*)
The sender multicasts a message to all coordinators
If a message is not recieved, the parent group is requested to send it
Acknowledgments on reciept can also be used **without making feedback implosion**

Atomic multicast

All clients or **no clients** recieve the message (*also the messages is totally ordered*)
Virtual synchrony Figure 8-13 page 350
Figure 8-16 page 353

Distributed commit

One-phase A coordinator tells when participants should commit
Two-phase See 4 steps page 355 (*blocking commit protocol (it may have to wait for coordinator to recover from a crash)*)
Three-phase Same as 2PC but a PRECOMMIT state is introduced which makes the protocol nonblocking

Recovery

Backward recovery Recover to latest checkpoint (*go back in time*)
Example: retransmission of packages
Forward recovery Bring process to new state (*possible errors has to be known in advance*)
Example: **erasure correction** (*page 364 top*)

Combination of **checkpointing** and **message logging** (*sender or reciever*) is widely used