Distributed Systems

2024

SKELETON

MODELING DS

- SOFTWARE ARCHITECTURE OF A DS
- ARCHITECTURAL STYLES (RUN-TIME ARCHITECTURES)
  - CLIENT-SERVER
  - SERVICE ORIENTED
  - REST
  - P2P
  - OBJECT ORIENTED
  - DATA CENTERED
  - EVENT BASED
  - MOBILE CODE
- INTERACTION MODEL
- FAILURE MODEL

COMMUNICATION

- PROTOCOLS AND MIDDLEWARE
- RPC
- RMI
- MESSAGE ORIENTED COMMUNICATION
  - SOCKETS
  - MPI
  - MESSAGE QUEUEING
  - PUBLISH-SUBSCRIBE
- STREAM ORIENTED COMMUNICATION

NAMING

- FLAT NAMING
  - SIMPLE SOLUTIONS
  - HOME BASED APPROACHES
  - DISTRIBUTED HASH TABLES (DHT)
  - HIERARCHICAL APPROACHES
- STRUCTURED NAMING
- ATTRIBUTE BASED NAMING
- REMOVING UNREFERENCED ENTITIES

SYNCHRONIZATION

- SYNCHRONIZING PHYSICAL CLOCKS
  - GPS
  - CRISTIAN'S ALGORITHM
  - BERKELEY'S ALGORITHM
  - NETWORK TIME PROTOCOL (NTP)
- LOGICAL TIME
  - SCALAR CLOCKS
  - VECTOR CLOCKS
- MUTUAL EXCLUSION
- LEADER ELECTION
- COLLECTING GLOBAL STATE — TERMINATION DETECTION
- DISTRIBUTED TRANSACTIONS
  - SERIALIZABILITY
  - PESSIMISTIC TIMESTAMP ORDERING
  - OPTIMISTIC TIMESTAMP ORDERING
  - DEADLOCKS

FAULT TOLERANCE

- CLIENT-SERVER COMMUNICATION
- PROTECTION AGAINST PROCESS FAILURES
- RELIABLE GROUP COMMUNICATION
- RECOVERY TECHNIQUES
  - UNCOORDINATED CHECKPOINTING
  - COORDINATED CHECKPOINTING
  - LOGGING

CONSENSUS IN DS - DISTRIBUTED AGREEMENT IN PRACTICE

- COMMIT PROTOCOLS
  - TWO PHASE COMMIT (2PC)
  - THREE PHASE COMMIT (3 PC)
- REPLICATED STATE MACHINES
  - PAXOS
  - RAFT
  - BYZANTINE FAULT TOLERANT REPLICATION & CONSENSUS — BLOCKCHAINS

REPLICATION AND CONSISTENCY

- CONSISTENCY MODELS
  - SINGLE LEADER PROTOCOLS
  - MULTI LEADER PROTOCOLS
  - LEADERLESS PROTOCOLS
- DATA-CENTRIC CONSISTENCY MODELS
  - SEQUENTIAL CONSISTENCY
  - LINEARIZABILITY
  - CAUSAL CONSISTENCY
  - FIFO CONSISTENCY
  - EVENTUAL CONSISTENCY
- CLIENT-CENTRIC CONSISTENCY MODELS
  - MONOTONIC READS
  - MONOTONIC WRITES
  - READ YOUR WRITES
  - WRITES FOLLOW READS
  - IMPLEMENTATION
- DESIGN STRATEGIES
- CASE STUDIES

DISTRIBUTED PLATFORMS FOR DATA ANALYTICS

- MAP REDUCE
- DATAFLOW PLATFORMS
  - SPARK
  - FLINK

PEER TO PEER

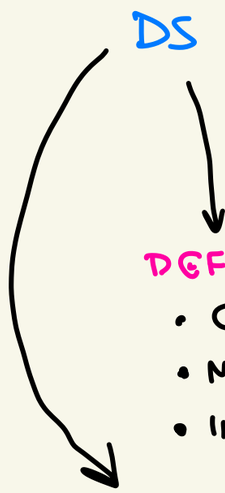
- NAPSTER
- QUERY FLOODING — GNUTELLA
- HIERARCHICAL TOPOLOGY — KAZAA
- COLLABORATIVE SYSTEMS — BITTORRENT
- SECURE STORAGE - FREENET
- STRUCTURED TOPOLOGY — DISTRIBUTED HASH TABLES (DHT) — CHORD

WHAT IS A DS?

"A collection of independent computers that appears to its users as a single coherent system"
-- A.S. Tanenbaum, M. van Steen

"One in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages"
-- G. Coulouris, J. Dollimore, T. Kindberg

"One on which I cannot get any work done because some machine I have never heard of has crashed"
-- L. Lamport



DS

DEFINING FEATURES:

- CONCURRENCY
- NO GLOBAL CLOCK (DIFFICULT TO SYNCH)
- INDIPENDENT AND PARTIAL FAILURES

CHALLENGES:

- HETEROGENEITY
- OPENNESS
- SECURITY
- SCALABILITY
- FAILURE HANDLING
- CONCURRENCY
- TRASPARENCY

The Eight Fallacies of Distributed Computing

Peter Deutsch

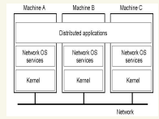
Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

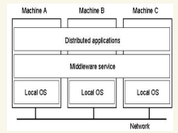
MODELING DS

SOFTWARE ARCHITECTURE OF A DS

- NETWORK OS BASED



- MIDDLEWARE BASED

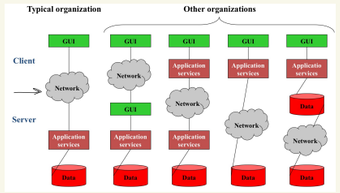


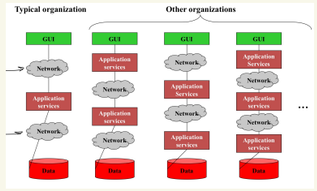
→ HIDES DISTRIBUTION USING API THAT PROVIDES COMMUNICATION AND COORDINATION

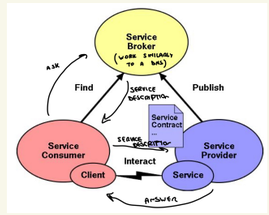
ARCHITECTURAL STYLES (RUN-TIME ARCHITECTURE)

↳ CLIENT - SERVER → MOST COMMON, COMMUNICATION IS MESSAGE BASED (RPC - REMOTE PROCEDURE CALL)

↳ TIERS: 2 TIERS 3 TIERS







SERVICE ORIENTED

↳ SERVICE CONSUMER (CLIENT) ASKS SERVICE BROKER FOR A SERVICE, BROKER CONTACTS A PROVIDER AND THEN CLIENTS BINDS TO PROVIDERS THROUGH A CONTRACT

↳ WEB SERVICES:
- SOFTWARE TO DO MACHINE-MACHINE WORK OVER A NETWORK
- INVOKED THROUGH SOAP, AN XML BASED PROTOCOL

REST (REPRESENTIONAL STATE TRANSFER)

↳ KEY GOALS: SCALABILITY, GENERALITY OF INTERFACES, INDIPENDENT DEPLOYMENT OF COMPONENTS, INTERMEDIARY COMPONENTS TO REDUCE LATENCY, MORE SECURITY

CONSTRAINS:
- CLIENT-SERVER AND STATELESS INTERACTIONS
  ↓
  MUST SUPPORT CODE ON DEMAND
- DATA MUST BE LABELED AS CACHABLE / NON-CACHABLE TO IMPROVE SCALABILITY
- COMPONENTS HAVE UNIFORM INTERFACE → CONSTRAINS:
  - COMPONENTS MUST HAVE ID
  - MANIPULATION THROUGH REPRESENTATION (DATA AND METADATA ABOUT DATA)
- SELF DESCRIPTIVE MESSAGES
- HYPERMEDIA TO MOVE BETWEEN STATES

P2P

↳ SHARING OF RESOURCES AND SERVICES THROUGH COMMUNICATION BETWEEN PEERS

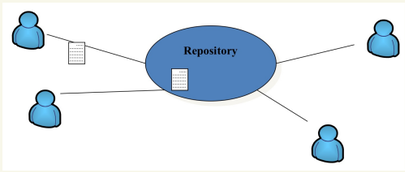
OBJECT ORIENTED

↳ COMPONENTS HAS DATA STRUCTURES PROVIDING API TO ACCES AND MODIFY IT

↳ PROS: • INFORMATION HIDING
• OBJECTS EASY TO REUSE
• REDUCED MANAGMENT COMPLEXITY

DATA CENTERED

↳ COMMUNICATION THROUGH COMMON REPOSITORY (USUALLY WITH RPC AND ALSO SYNCH)



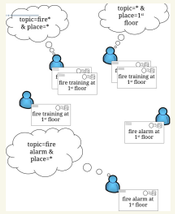
LINDA:

- Data is contained in ordered sequences of typed fields (**tuples**)
- Tuples are stored in a persistent, global shared space (**tuple space**)
- Standard operations:
  - out(t): writes the tuple t in the tuple space
  - rd(p): returns a copy of a tuple matching the **pattern** (or **template**) p, if it exists; blocks waiting for a matching tuple otherwise
    - If many matching tuples exist, one is chosen non-deterministically
  - in(p): like rd(p), but withdraws the matching tuple from the tuple space
  - Some implementations provide also an eval(a), which inserts the tuple generated by the execution of a process a
- Many variants:
  - Asynchronous, non-blocking primitives (probes): rdp(p) and inp(p)
    - Return immediately a null value if the matching tuple is not found
  - Bulk primitives: e.g., rdg(p)
  - …
- Some of the non-standard primitives have non-trivial distributed implementations
  - E.g., if atomicity is to be preserved, probes require a distributed transaction

- The tuple space model is not easily scaled on a wide-area network
  - How to store/replicate tuples efficiently
  - How to route queries efficiently
- The model is only proactive
  - Processes explicitly request a tuple query
    - reactive/asynchronous behavior must implemented with an extra process and a blocking operation
- As a consequence, commercial implementations:
  - Provide only client access to a server holding the tuple space
    - Instead of a fully distributed, decentralized implementation
  - Introduce reactive primitives
    - e.g., notify allows to register a listener, invoked when a matching tuple is written

EVENT BASED

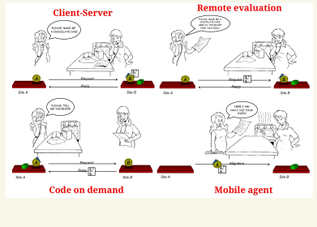
↳ SUBSCRIPTIONS AND PUBLICATION OF EVENTS BY THE COMPONENTS



COMMUNICATION IS:
- MESSAGE BASED
- ASYNCH
- ANONYMOUS
- MULTICAST
- IMPLICIT

MOBILE CODE

↳ RELOCATING COMPONENTS (CODE) AT RUN TIME



↳ STRONG MOBILITY: TRANSFER CODE AND EXECUTION
↳ WEAK MOBILITY: TRANSFER CODE

PROS: • GREAT FLEXIBILITY
CONS: • SECURITY

INTERACTION MODEL

BEHAVIOUR OF A DS DESCRIBED BY A DISTRIBUTED ALGORITHM

↳ SYNCHRONOUS DS: BOUNDS FOR PROCESS STEPS, MESSAGE TIME TRANSMISSION → ANY ASYNCH SOLUTION IS ALSO GOOD FOR SYNCH
↳ ASYNCHRONOUS DS: NO BOUNDS

PEPPERLAND EXAMPLE:

- The pepperland divisions are safe as long as they remain in their encampments
- If both charge at the same time they win, otherwise they loose
- Generals need to agree on:
  - Who will lead the charge
  - When the charge will take place
- We consider the case when messengers are able to walk from an hill to another without being captured by the enemies

- Even in asynchronous pepperland it is possible to agree on who will lead the charge
  - **How?**
- Charging together is a different issue
  - It is not possible in asynchronous pepperland
    - If the leader sends a messenger to the other general saying "charge!" the messenger may take three hours or just five minutes to reach the other general → (VARIABLE TIME TO ARRIVE)
    - Also differences on each division's clock do not allow strategies based on sending a message with the time to charge → DIFFERENT CLOCKS (TIME)
  - In synchronous pepperland it is possible to determine the maximum difference between charge times
    - Let min and max be the range of message transmission times
    - The leader sends a message "charge!", wait min minutes then charge
    - On receiving the "charge!" message the other general immediately charge
    - The second division may charge later then the first one but no more that (max-min) minutes
    - If we know that the charge will last longer then the victory is guaranteed

FAILURE MODEL

PROCESSES AND CHANNELS MAY FAIL :

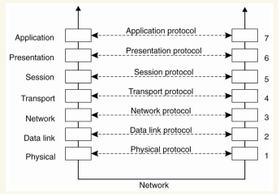
- OMISSION FAILURES : PROCESSES CRASH AND CHANNEL DON'T SEND/RECEIVE
- BYZANTINE FAILURES : PROCESSES SKIP/ADD STEPS AND CHANNELS CHANGE MESSAGES
  - SEND MORE/ NON-EXISTING MSG ARRIVE/ CORRUPTED
- TIMING FAILURES : TIME LIMITS VIOLATED (ONLY SYNCH SYSTEMS)

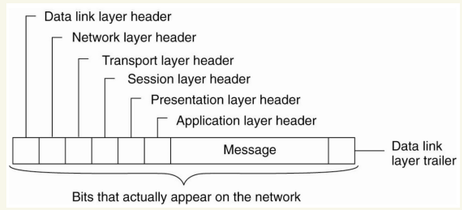
DETECTING FAILURES IS EZ IN SYNCH, IMPOSSIBLE IN ASYNCH

COMMUNICATION

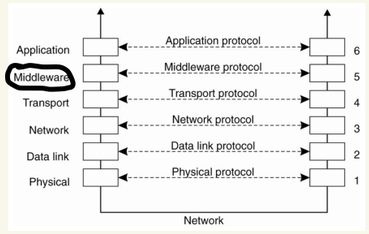
PROTOCOLS AND MIDDLEWARE

NETWORKS USE OSI MODEL AND MESSAGE ENCAPSULATION





IN DS MIDDLEWARE CAN BE SEEN AS A PROTOCOL LAYER

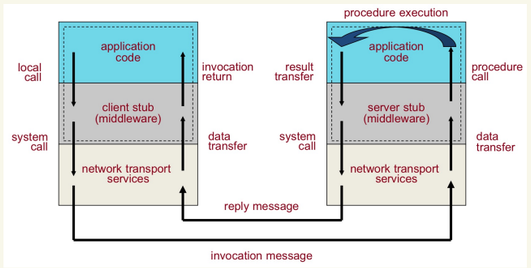


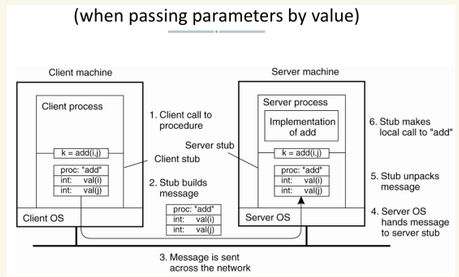
↓

OFFERS TRANSIENT/PERSISTENT AND SYNCH/ASYNCH COMMUNICATIONS

RPC (REMOTE PROCEDURE CALL)

PASSING PARAMETERS TO A PROCEDURE CAN BE DONE
- BY VALUE
- BY REFERENCE
- BY COPY/RESTORE





MIDDLEWARE THROUGH IDL (INTERFACE DEFINITION LANGUAGE) SUPPORTS

SERIALIZATION (FLATTENING STRUCTURED DATA) AND MARSHALLING (CONVERSION OF DATA REPRESENTATIONS) FOR PARAMETER PASSING

↑

MAPPINGS THROUGH VARIOUS LANGUAGES

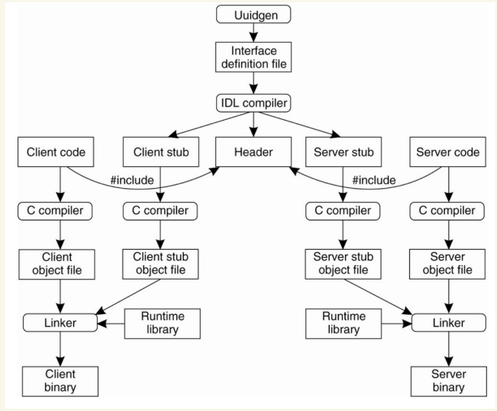
RPC IN PRACTICE

SUN MICROSYSTEM RPC IS THE STANDARD OVER THE INTERNET TO BIND CLIENTS TO SERVERS

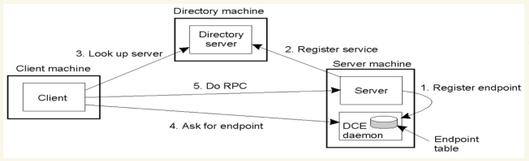
↓

SUN SOLUTION TO CLIENT-SERVER BINDING : PORTMAP (DAEMON PROCESS)

↳ SERVER TELLS A PORT TO PORTMAP WITH SERVICE IDENTIFIER
CLIENT CONTACTS A GIVEN PORTMAP AND REQUEST THE PORT FOR COMMUNICATION → PORTMAP DOESN'T TELLS US WHERE SERVER PROCESS IS



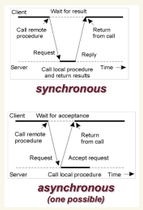
DCE'S SOLUTION:

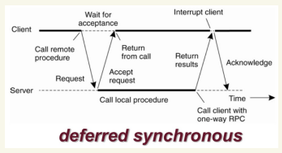


DCE DAEMON WORKS LIKE PORTMAP,
BUT CLIENTS ONLY NEED TO KNOW WHERE THE DIRECTORY SERVICE IS
HAS ANOTHER LOCAL DAEMON THAT FORKS THE PROCESS TO SERVE THE REQ OR REDIRECTS IT IF PROCESS IS ACTIVE

LIGHT WEIGHT RPC : COMMUNICATION EXPOLITS A SHARED MEMORY REGION → USES LESS THREADS/ 1 PARAMETER COPY INSTEAD OF 4

ASYNCH RPC :





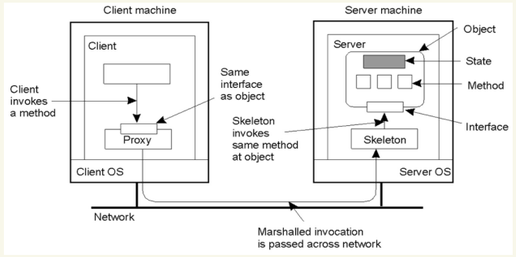
USUAL CALL BEHAVIOR PRESERVED
HAS MANY VARIANTS
POTENTIALLY WASTES CLIENT RESOURCES

SUN RPC HAS THE ABILITY TO PERFORM BATCHED RPC → RPCS THAT DO NOT REQUIRE RESULT ARE BUFFERED AND SENT ALL TOGETHER WHEN A NON-BATCHED CALL IS REQUESTED (ANOTHER FORM OF ASYNCH RPC)

RMI (REMOTE METHOD INVOCATION)

- SAME IDEA AS RPC BUT BASE ON OOP → REMOTE OBJECT REFERENCES CAN BE PASSED AROUND
  - ↳ IDLS FOR DISTRIBUTED OBJECTS ARE RICHER THAN RPC

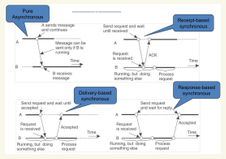
EXAMPLES: JAVA RMI / OMG CORBA

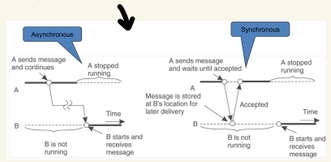


MESSAGE ORIENTED COMMUNICATION

ONE WAY MESSAGES/EVENTS, ASYNCH, OFTEN PERSISTENT, MORE DECOUPLING BETWEEN COMPONENTS

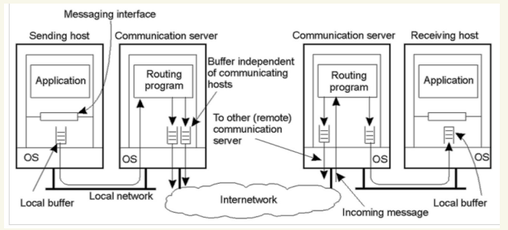
TYPES OF COMMUNICATION → TRANSIENT, PERSISTENT





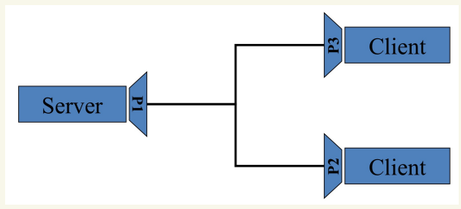
USING TCP (UNICAST) AND UDP

MESSAGE PASSING MIDDLEWARE → MESSAGE QUEUING, PUBLISH/SUBSCRIBE

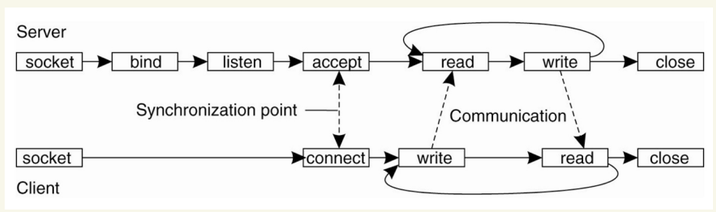


SOCKETS: ABSTRACTION FOR COMMUNICATION BETWEEN PROCESSES

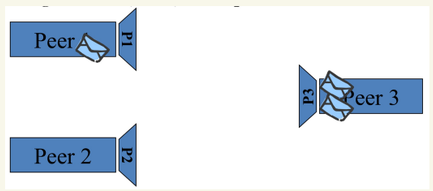
IDENTIFIED BY:
- SERVER IP
- INCOMING PORT
- CLIENT IP
- OUTGOING PORT



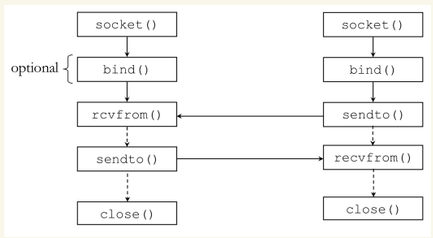
SOCKETS IN C →



- DATAGRAM SOCKETS:
  - CLIENT & SERVER USE THE SAME APPROACH, CREATE A SOCKET ON A PORT FOR INPUT/OUTPUT.
  - THERE IS NO CONNECTION AND THE SAME SOCKET MAY BE USED TO SEND/RECEIVE DATAGRAM FROM/TO MULTIPLE HOSTS



DATAGRAM SOCKETS IN C →



- MULTICAST SOCKETS → SOCKET API SIMILAR TO DATAGRAM COMMUNICATION ONE

↳ IP MULTICAST: PROTOCOL TO DELIVER UDP DATAGRAMS TO MULTIPLE COMPONENTS

MPI (MESSAGE PASSING INTERFACE) → NEED FOR HIGHER LEVEL SERVICES FOR ASYNCH AND TRANSIENT COMMUNICATION, MPI IS THE ANSWER

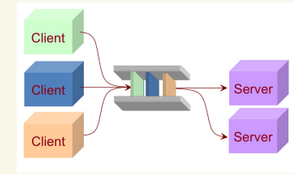
↳ ENABLES COMMUNICATION IN A GROUP OF PROCESSES, WHERE EACH PROCESS HAS IT'S OWN ID
DOESN'T SUPPORT FAULT TOLERANCE → CRASHES ARE FATAL

MPI PRIMITIVES:

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

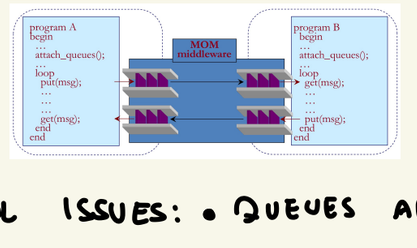
MESSAGE QUEUEING → POINT TO POINT, PERSISTENT, ASYNCH

↓
COMPONENTS HAVE INPUT QUEUE AND OUTPUT QUEUE



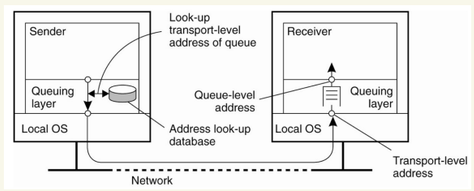
PRIMITIVES:

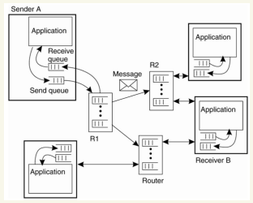
Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

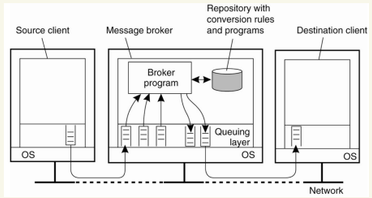


→ CLIENT SENDS REQUESTS TO SERVER QUEUE → DOESN'T NEED TO REMAIN CONNECTED
SERVER PROCESSES THE REQUESTS ASYNCHRONOUSLY AND RETURN RESULTS TO CLIENTS' QUEUES } EZ LOAD BALANCING

ARCHITECTURAL ISSUES:
- QUEUES ARE IDENTIFIED BY SIMBOLIC NAMES SO → NEED FOR LOOKUP SERVICE TO GET THE NETWORK ADDRESSES
- QUEUES ARE MANIPULATED BY QUEUE MANAGERS
- OVERLAY NETWORKS
- MESSAGE BROKERS DO MESSAGE CONVERTION







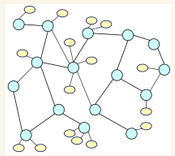
PUBLISH - SUBSCRIBE → ASYNCH, MULTIPOINT

↳ COMPONENTS CAN PUBLISH EVENTS OR SUBSCRIBE TO OTHER COMPONENTS TO GET NOTIFIED ON PUBLICATIONS

SUBSCRIPTIONS COLLECTED BY AN EVENT DISPATCHER → ALSO ROUTES NOTIFICATIONS BASED ON SUBSCRIPTIONS

EVENT DISPATCHER ARCHITECTURE:
MAY BE CENTRALIZED OR DISTRIBUTED

SET OF MESSAGE BROKERS IN AN OVERLAY NETWORK



EVERY BROKER STORES SUBSCRIPTIONS COMING ONLY FROM DIRECTLY CONNECTED CLIENTS → MESSAGES ARE FORWARDED FROM BROKER TO BROKER AND DELIVERED ONLY TO SUBSCRIBED CLIENTS

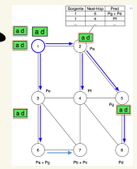
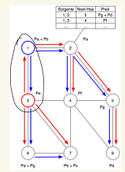
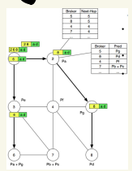
SUBSCRIPTION ARE:
- SUBJECT/TOPIC BASED
- CONTENT BASED

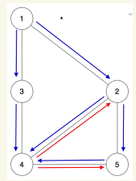
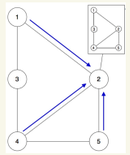
} CAN BE COMBINED

ACYCLIC GRAPH: SUBSCRIPTIONS NEVER SENT TWICE OVER THE SAME LINK AND MESSAGES FOLLOW ROUTES LAID BY SUBSCRIPTIONS

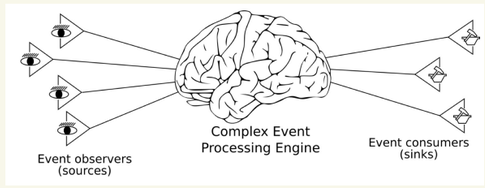
HIERARCHICAL FORWARDING: MESSAGES AND SUBSCRIPTIONS FORWARDED BY BROKERS TOWARDS THE ROOT OF THE TREE

CYCLIC TOPOLOGIES : • DHT (DISTRIBUTED HASH TABLE) → NODES ORGANIZED IN A STRUCTURED OVERLAY
↳ EFFICIENT ROUTING TOWARD NODES HAVING ID ≥ THAN ANY GIVEN ID

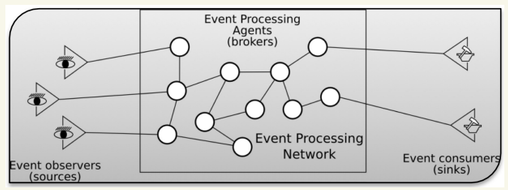
• CONTENT BASED ROUTING — FORWARDING — PSF (PER SOURCE FORWARDING) → 
iPSF (IMPROVED PER SOURCE FORWARDING) → 
PRF (PER RECEIVER FORWARDING) → 

ROUTING — DISTANCE VECTOR (DV) → 
LINK STATE (LS) → 

COMPLEX EVENT PROCESSING: HELPS TO DETECT PATTERNS OR IMPORTANT EVENTS FROM A SET OF SMALLER/BASIC EVENTS.
↓
ALLOW TO SET UP RULES TO DESCRIBE HOW THESE SMALLER EVENTS COMBINE TO CREATE A BIGGER ONE



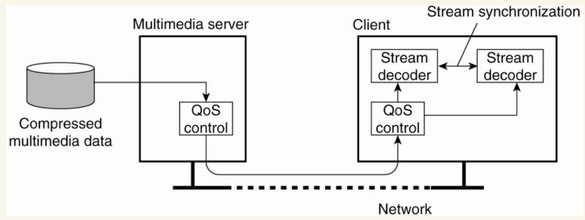
→



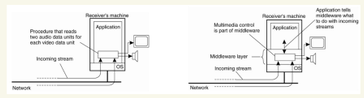
STREAM ORIENTED COMUNICATION

A STREAM IS A SEQUENCE OF DATA UNITS, TRASMISSION CAN BE
- ASYNC : DATA TRANSMITTED WITHOUT TIMING CONSTRAINS
- SYNC : MAX END-TO-END DELAY FOR EACH DATA PACK
- ISOSYNC: MAX AND MIN END-TO-END DELAY

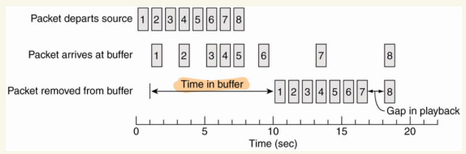
QoS (QUALITY OF SERVICE) IS DETERMINED BY:
- REQUIRED BIT RATE
- MAX DELAY FOR SETUP
- MAX END-TO-END DELAY
- MAX VARIANCE IN DELAY (JITTER)

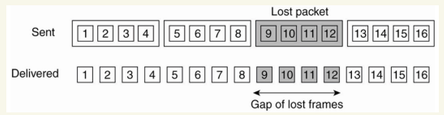


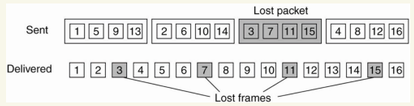
SYNCH 2 STREAM IS NOT AN EZ TASK, COULD BE DONE ON THE SENDER OR ON THE RECEIVER SIDE



QoS ENFORCING STRATEGIES:

• BUFFERING → 

• FORWARD ERROR CORRECTION → 

• INTERLEAVING DATA → 

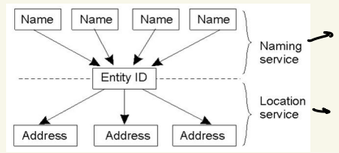
↳ USEFUL WITH BUFFERING

NAMING

SPECIAL CASE OF A NAME

NAMES REFERS TO ENTITIES, ENTITIES ARE ACCESSED THROUGH AN ACCESS POINT, AN ACCESS POINT IS CHARACTERIZED BY AN ADDRESS

IDENTIFIERS: UNIQUE AND IMMUTABLE NAME FOR AN ENTITY



MAPPING NAME TO ENTITY

LOCATING THE ENTITY

(NAME RESOLUTION)

NAME RESOLUTION: GETTING THE ADDRESS FROM THE NAME

- FLAT NAMING
- STRUCTURED NAMING
- ATTRIBUTE-BASED NAMING

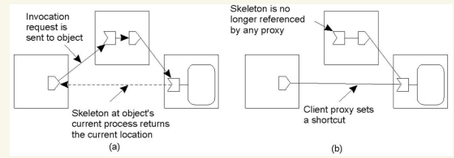
FLAT NAMING

NAMES ARE SIMPLE STRINGS WITHOUT STRUCTURES

SIMPLE SOLUTIONS

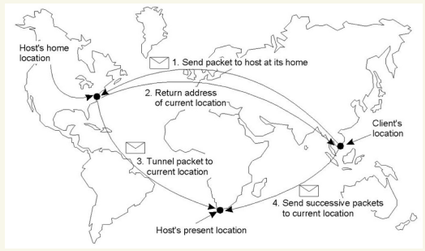
↳ FOR SMALL ENVIROMENTS (LAN)

- BROADCAST → SIMILAR TO ARP
- MULTICAST
- FORWARDING POINTERS → WHEN A THING MOVES CREATES A POINTER WHERE IT WAS POINTING AT WHERE IT IS NOW → CAN CREATE LONG CHAINS



HOME BASED APPROACHES → USED IN CELLPHONE NETWORKS

↳ HOME NODE KNOWS THE LOCATION OF THE MOBILE UNIT →



→ BIG LATENCY

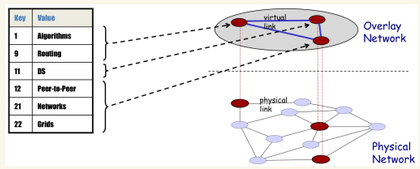
→ POOR SCALABILITY

→ HOME NODE SINGLE POINT OF FAILURE

DISTRIBUTED HASH TABLES (DHT)

↳ DATA STRUCTURE: PUT (ID, ITEM)
ITEM = GET (ID) → NAME RESOLUTION: ID ↔ NAME/IDENTIFIER, ITEM ↔ IDENTIFIER/ADDRESS

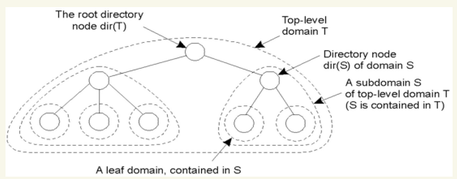
IMPLEMENTATION → STRUCTURED OVERLAY NETWORK (RING, TREE, ..)



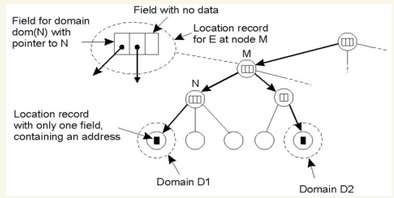
→ EXAMPLE: CHORD

HIERARCHICAL APPROACES

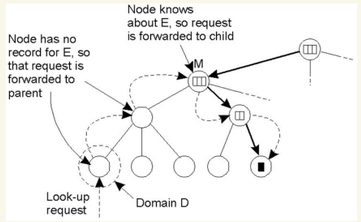
↳ NETWORK DIVIDED INTO DOMAINS, ROOT SPANS WHOLE NETWORK WHILE LEAVES ARE LAN/PHONE CELLS, EACH DOMAIN HAS A FATHER DIRECTORY NODE



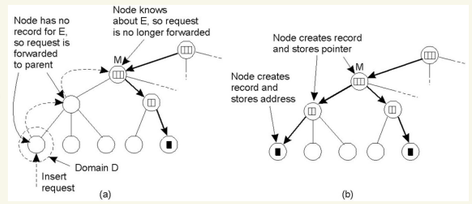
ARCHITECTURE →



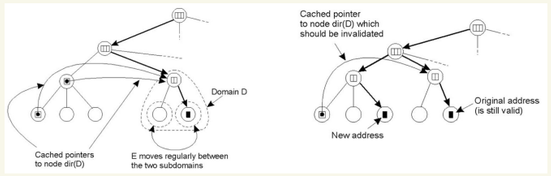
LOOKUP →



UPDATES →



CACHING AND SCALABILITY :



STRUCTURED NAMING

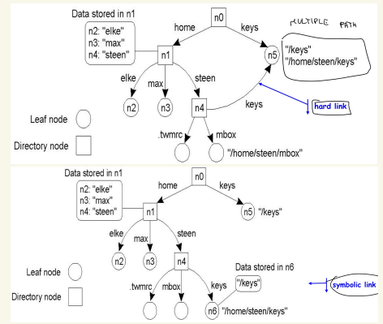
→ NAMES ORGANIZED IN A NAME SPACE → ORGANIZED GRAPH WITH LEAVES NODES AND DIRECTORY NODES

↓ REPRESENTS A NAMED ENTITY

↓

RESOURCES ARE REFERRED THROUGH PATH NAMES (ABSOLUTE OR RELATIVE)

HAS HARD LINKING ⇒

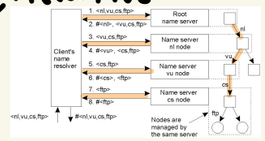


NAME SPACES IN DS ARE DISTRIBUTED AMONG NAME SERVERS, ORGANIZED HIERARCHICALLY

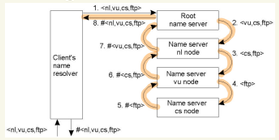
→ PARTITIONED INTO LAYERS → GLOBAL LEVEL (ROOT)
→ ADMINISTRATIONAL LEVEL (LEAVES)
→ MANAGERIAL LEVEL

NAME RESOLUTION :

→ ITERATIVE



→ RECURSIVE

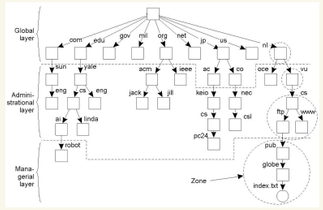


} USUALLY A MIX IS USED

↓

IN DNS CACHING AND REPLICATION IS WIDELY USED

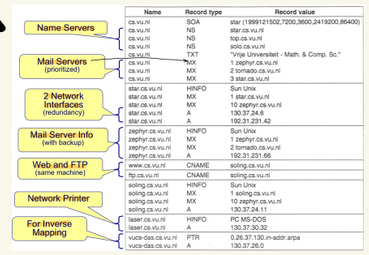
DNS →



↓

EACH NODE MAY REPRESENT SEVERAL ENTITIES THROUGH RESOURCE RECORDS →

EXAMPLE OF RECORDS



COMPARISON →

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

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

Name	Record type	Record value
cs.vu.nl	SOA	star (1999121502,7200,3600,2419200,86400)
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	"Vrije Universiteit - Math. & Comp. Sc."
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	HINFO	Sun Unix
star.cs.vu.nl	MX	1 star.cs.vu.nl
star.cs.vu.nl	MX	10 zephyr.cs.vu.nl
star.cs.vu.nl	A	130.37.24.6
star.cs.vu.nl	A	192.31.231.42
zephyr.cs.vu.nl	HINFO	Sun Unix
zephyr.cs.vu.nl	MX	1 zephyr.cs.vu.nl
zephyr.cs.vu.nl	MX	2 tornado.cs.vu.nl
zephyr.cs.vu.nl	A	192.31.231.66
www.cs.vu.nl	CNAME	soling.cs.vu.nl
ftp.cs.vu.nl	CNAME	soling.cs.vu.nl
soling.cs.vu.nl	HINFO	Sun Unix
soling.cs.vu.nl	MX	1 soling.cs.vu.nl
soling.cs.vu.nl	MX	10 zephyr.cs.vu.nl
soling.cs.vu.nl	A	130.37.24.11
laser.cs.vu.nl	HINFO	PC MS-DOS
laser.cs.vu.nl	A	130.37.30.32
vucs-das.cs.vu.nl	PTR	0.26.37.130.in-addr.arpa
vucs-das.cs.vu.nl	A	130.37.26.0

ATTRIBUTE BASED NAMING → USUALLY IMPLEMENTED USING DBMS

REFERRING TO ENTITIES THROUGH A SET OF ATTRIBUTES, NOT WITH THEIR NAME.

→ NAME SYSTEM QUERIED BY SEARCHING WITH SOME VALUES OF ATTRIBUTES

LDAP (LIGHTWEIGHT DIRECTORY ACCESS PROTOCOL) → COMBINATION OF STRUCTURED AND ATTRIBUTE BASED NAMING

→ HAS RECORDS (DIRECTORY ENTRIES) AND EACH IS A COLLECTION OF ⟨ATTRIBUTE, VALUE⟩ PAIRS

↓ → EACH RECORD HAS A NAME

↓ HAS TYPE AND CAN BE SINGLE OR MULTIPLE VALUED

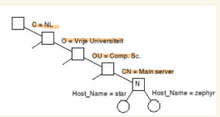
COLLECTION OF ALL RECORDS IS CALLED DIB (DIRECTORY INFORMATION BASE)

↓

DIT (DIRECTORY INFORMATION TREE)

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

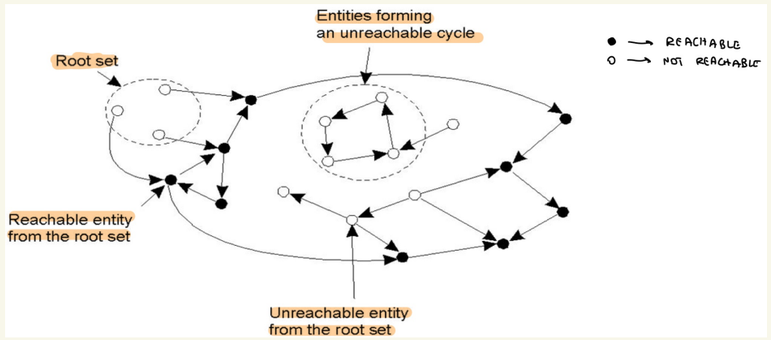


WHEN USING LARGE SCALE DIRECTORIES → DIB PARTITIONED AS DIT (AS DNS)

SERVER = DIRECTORY SERVICE AGENT DSA

CLIENT = DIRECTORY USER AGENT DUA

REMOVING UNREFERENCED ENTITIES

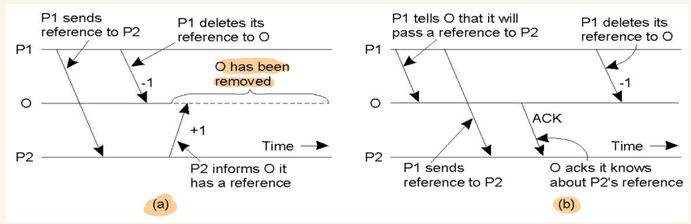


REFERENCE COUNTING

↓

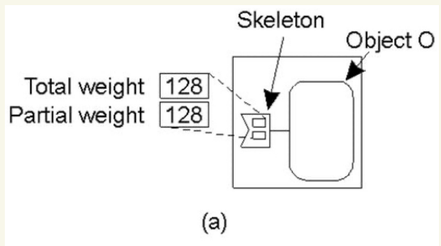
OBJECT KEEPS TRACK OF HOW MANY OTHER OBJECTS HAVE BEEN GIVEN REFERENCES

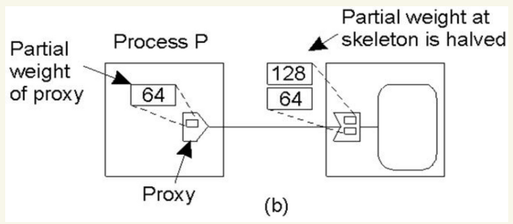
↳ RACE CONDITION WHEN PASSING REFERENCES THROUGH PROCESSES

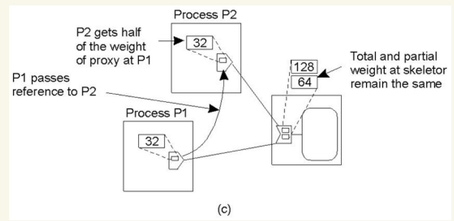


WEIGHTED REFERENCE COUNTING

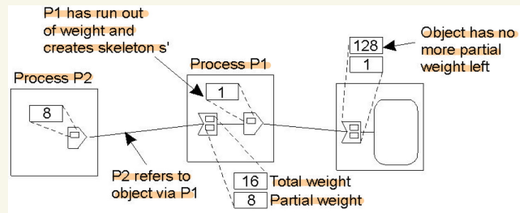
↳ WHEN TOTAL AND PARTIAL WEIGHTS BECOME EQUAL THE OBJECT CAN BE REMOVED







↳ USING INDIRECTION → REMOVES LIMIT ON NUMBER OF REFERENCES BUT INTRODUCES AN ADDITIONAL STEP TO ACCESS TARGET OBJECT



REFERENCE LISTING

↳ KEEP TRACK OF THE IDENTITIES OF THE PROXIES, INSTEAD OF THE NUMBER OF REFERENCES → • EZ TO MAINTAIN
• INSERTION/DELETION OF PROXY IS IDEMPOTENT
• STILL HAS RACE CONDITION

IDENTIFYING UNREACHABLE ENTITIES

↳ MARK AND SWEEP: ACCESS ENTITIES AND COLOR THEM STEP BY STEP, WHITE ENTITIES GARBAGE COLLECTED AT THE END

SYNCHRONIZATION

WE NEED TO ENSURE THAT ALL MACHINES SEE THE SAME GLOBAL TIME → SYNCHRONIZE ← TO ONE EXTERNAL CLOCK / AMONG THEMSELVES

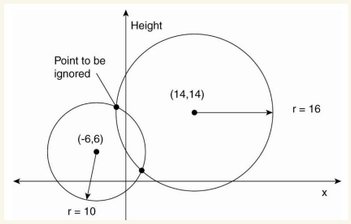
SYNCHRONIZING PHYSICAL CLOCKS

ρ : DRIFT RATE

δ : CLOCK SKEW

→ RESYNCH EVERY $\frac{\delta}{2\rho}$ SECONDS

GPS



→ TRIANGULATION

→ DISTANCE MEASURED WITH SIGNAL DELAY

→ SATELLITE AND RECEIVER MUST BE IN SYNC → THEY AREN'T SO WE MUST CONSIDER CLOCK SKEW

- Let:
  - Δ_r be the unknown deviation of the receiver's clock w.r.t. the atomic clocks installed on board of satellites
  - x_r, y_r, z_r be the unknown coordinates of the receiver
  - T_i be the timestamp of message sent by a satellite i
- Suppose that the messages sent by the i-th satellite is received at time T_i according to the receiver time, which corresponds to T_{now} in the real, actual time. Then:
  - $T_{now} = T_i - \Delta_r$
  - $\Delta_i = T_{i-} T_i$ is the measured delay of message
  - $c \times \Delta_i$ is the measured distance of satellite i
- So $c \times \Delta_i = c \times (T_{now} - T_i + \Delta_r) = c \times (T_{now} - T_i) + c \times \Delta_r$

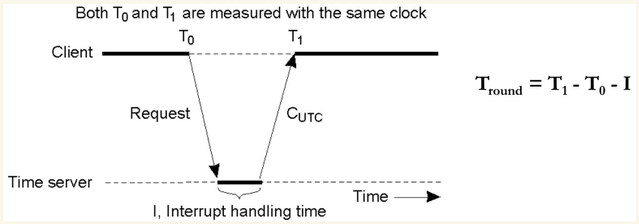
  where the first addendum must be equal to the real distance:

  $\sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$
- With four satellites we have four equations in four unknowns (including Δ_r)
  - We can solve them and determine both the node position and its clock skew

CRISTIAN'S ALGORITHMS

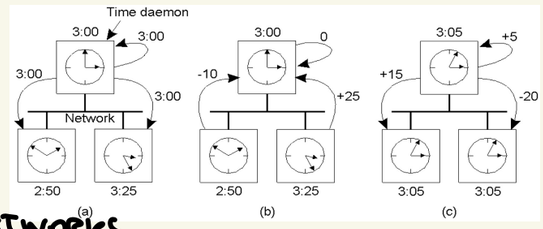
↳ EACH CLIENT PERIODICALLY REQUESTS TIME SERVER → PROBLEMS:

- TIME MIGHT RUN BACKWARDS OR CLIENT
- TAKES TIME TO DELIVER AND RECEIVE A MESSAGE → MEASURE RTT AND ADJUST



BERKELEY'S ALGORITHMS

↳ TIME SERVER COLLECTS TIME FROM ALL CLIENTS, AVERAGES IT AND SENDS IT BACK



- Servers exchange pairs of messages, each bearing timestamps of recent message events
  - The local time when the previous message between the pairs was sent and received, and the local time when the current message was transmitted
- If t and t' are the messages' transmission times, and o is the time offset of the clock at B relative to that at A, then:
  - $T_{i,2} = T_{i,1} + t + o$ and $T_i = T_{i,1} + t' - o$

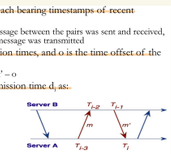
  This leads to calculate the total transmission time d_i as:
  - $d_i = t + t' = T_{i,2} - T_{i,1} + T_i - T_{i,1}$

  If we define o_i as:
  - $o_i = (T_{i,2} - T_{i,1} + T_{i,1} - T_i)/2$

  from the first two formulas we have:
  - $o = o_i + (t'-t)/2$

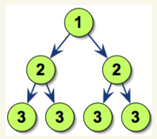
  and since $t, t' \geq 0$
  - $o_i - d_i/2 = o_i + (t'-t)/2 - t' \leq o \leq o_i + (t'-t)/2 + t = o_i + d_i/2$

  Thus o_i is an estimate of the offset and d_i is a measure of the accuracy of this estimate.



NETWORK TIME PROTOCOL (NTP) → DESIGNED FOR SYNCH OVER LARGE SCALE NETWORKS

↳ HIERARCHICAL SUBNET SYNCH (ORGANIZED IN STRATI)



→ SYNCHING IS:

- MULTICAST (LAN)
- PROCEDURE-CALL MODE (SIMILAR TO CRISTIAN'S)
- SYMMETRIC MODE (FOR HIGHER LEVELS)

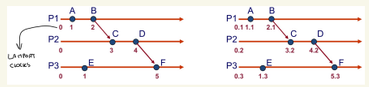
LOGICAL TIME → SOMETIMES IT IS SUFFICIENT TO AGREE ON A TIME, EVEN IF IS INACCURATE → ORDER AND CAUSALITY IMPORTANT!

SCALAR CLOCKS → $e \to e'$ MEANS THAT ← e HAPPENS BEFORE e' IN THE SAME PROCESS / e = SEND(MSG) AND e' = RECV(MSG) BETWEEN PROCESSES } CAPTURES CAUSAL ORDERING

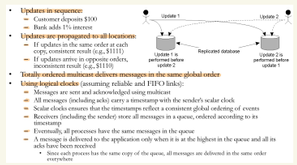
↓ TRANSITIVE

IF NEITHER $e \to e'$ NOR $e' \to e$, e AND e' ARE CONCURRENT $e \| e'$

EACH PROCESS P_i HAS A SCALAR CLOCK L_i —— $e \to e' \Rightarrow L(e) < L(e')$ → IF WE ATTACH P ID TO THE TIMESTAMP WE CAN ACHIEVE TOTAL ORDERING



↳ EXAMPLE (TOTALLY ORDERED MULTICAST)



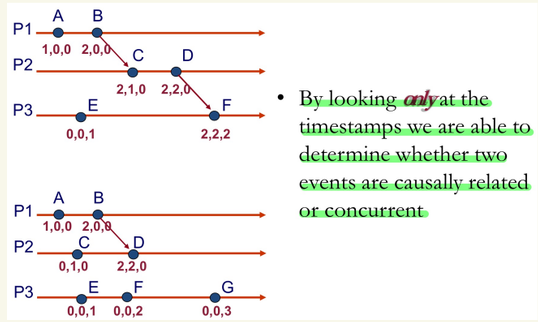
VECTOR CLOCKS → P_i HAS A VECTOR V_i OF N VALUES, WHERE (N = # OF PROCESSES)

- $V_i[i]$ IS THE N. OF EVENT HAPPENED AT P_i
- $V_i[j] = k$ MEANS THAT P_i KNOWS THAT k EVENTS HAVE OCCURRED AT P_j

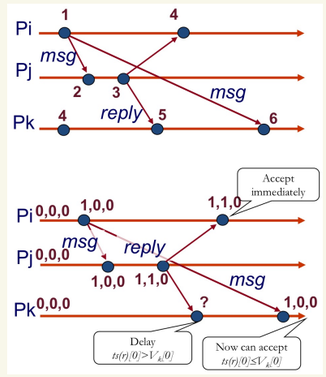
- $V_i[j] = 0 \ \forall i,j$ INITIALLY
- EVENT IN P_i INCREMENTS $V_i[i]$
- P_i ATTACHES A TIMESTAMP $t = V_i$ IN ALL MESSAGES IT SENDS
- WHEN P_i RECEIVES A MESSAGE WITH TIMESTAMP t IT SETS $V_i[j] = \max(V_i[j], t[j]) \ \forall j \neq i$ AND THEN INCREMENTS $V_i[i]$

- $V = V'$ IFF $V[j] = V'[j] \ \forall j$
- $V \leq V'$ IFF $V[j] \leq V'[j] \ \forall j$
- $V < V'$ IFF $V \leq V'$ AND $V \neq V'$
- $V \parallel V'$ IFF V ISN'T $< V'$ AND V' ISN'T $< V$

→
- $e \rightarrow e' \iff V(e) < V(e')$
- $e \parallel e' \iff V(e) \parallel V(e')$



- By looking ~~only~~ at the timestamps we are able to determine whether two events are causally related or concurrent

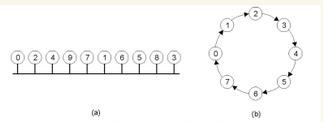


SLIGHT VARIATION TO IMPLEMENT CAUSAL DELIVERY IN A TOTALLY DISTRIBUTED WAY →

MUTUAL EXCLUSION → IN DS WE DON'T HAVE SHARED MEMORY

- SIMPLEST SOLUTION → HAVE A SERVER THAT COORDINATES ACCESSES TO EMULATE A CENTRALIZED SOLUTION
- USING SCALAR CLOCKS:
- TOKEN RING SOLUTION:

- To request access to a resource:
  - A process Pi multicasts a resource request message m, with timestamp Tm, to all processes (including itself)
  - Upon receipt of m, a process Pj:
    - If it does not hold the resource and it is not interested in holding the resource, Pj sends an acknowledgment to Pi.
    - If it holds the resource, Pj puts the requests into a local queue ordered according to Tm (process ids are used to break ties)
    - If it is also interested in holding the resource and has already sent out a requests, Pj compares the timestamp Tm with the timestamp of its own requests
      - If the Tm is the lowest one, Pj sends an acknowledgement to Pi, otherwise it put the request into the local queue above
- On releasing the resource, a process Pi acknowledges all the requests queued while using the resource
- A resource is granted to Pi when its request has been acknowledged by all the other processes



- Processes are logically arranged in a ring, regardless of their physical connectivity
  - At least for the purpose of mutual exclusion
- Access is granted by a token that is forwarded along a given direction on the ring
  - A process not interested in accessing the resource forwards the token
  - Resource access is achieved by retaining the token
  - Resource release is achieved by forwarding the token

→ COMPARISON

Algorithm	Messages per entry	Delay before entry (in message times)	Problems
Centralized	2	2	Coordinator crash
Distributed (Lamport)	$2(n-1)$	$2(n-1)$	Crash of any process
Token ring	1 to ∞	0 to $n-1$	Lost token, process crash

Request, grant

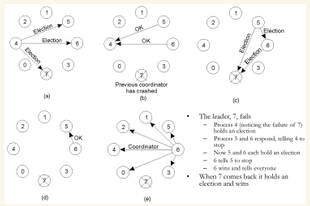
If nobody wants to enter the critical section, the token circulates indefinitely

LEADER ELECTION (A PROCESS IS THE COORDINATOR) → WE HAVE TO MAKE EVERYONE AGREE → ASSUMPTIONS
- DISTINGUISHABLE NODES
- PROCESSE KNOWS EACH OTHER IDs

- BULLY ELECTION ALGORITHM → OTHER ASSUMPTIONS: RELIABLE LINKS, POSSIBLE TO IDENTIFY CRASHES

Algorithm
- When any process P notices that the actual coordinator is no longer responding requests it initiates an election
- P sends an *ELECT* message, including its ID, to all other processes with higher IDs
- If no-one responds P wins and sends a *COORD* message to the processes with lower IDs
- If a process P' receives an *ELECT* message it responds (stopping the former candidate) and starts a new election (if it has not started one already)
- If a process that was previously down comes back up, it holds an election
  - If it happens to be the highest-numbered process currently running it wins the election and takes over the coordinator's job (hence the name of the algorithm)

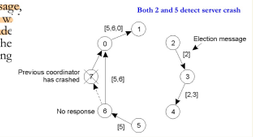
→

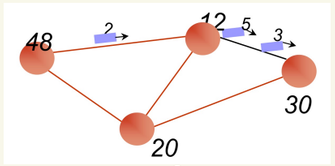


- The leader, 7, fails
  - Process 4 (noticing the failure of 7) holds an election
  - Process 5 and 6 respond, telling 4 to stop
  - Now 5 and 6 each hold an election
  - 6 tells 5 to stop
  - 6 wins and tells everyone
- When 7 comes back it holds an election and wins

↳ RING - BASED ALGORITHM →

- Assume a (physical or logical) ring topology among nodes
- When a process detects a leader failure, it sends an *ELECT* message containing its ID to the closest alive neighbor
- Upon receipt of the election message a process *P*:
  - If *P* is not in the message, add *P* and propagate to next alive neighbor
  - If *P* is in the list, change message type to *COORD*, and re-circulate
- On receiving a *COORD* message, a node considers the process w the highest ID as the new leade (and is also informed about the remaining members of the ring
- Multiple messages may circulate at the same time
  - Eventually converge to the same content





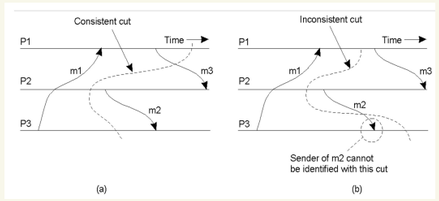
COLLECTING GLOBAL STATE → IN A DS GLOBAL STATE IS THE STATE OF ALL PROCESSES WITH THE MESSAGES IN TRANSIT OVER THE LINKS

↳ HAVING DIFFERENT CLOCKS WE MUST ACCEPT RECORDED STATES OF PROCESSES AT DIFFERENT TIMESTAMPS

↓

DISTRIBUTED SNAPSHOT REFLECTS A STATE IN WHICH THE DS MIGHT HAVE BEEN

↳ CUTS:



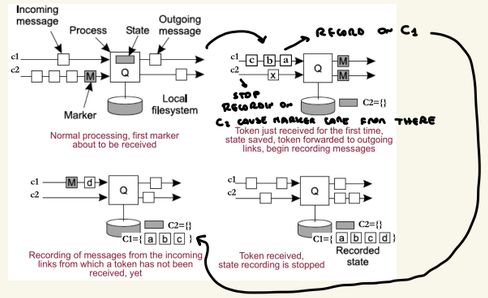
→ CUT = UNION OF HISTORIES OF ALL PROCESSES UP TO A CERTAIN EVENT → CONSISTENT IF ∀ EVENTS e IT INCLUDES, IT ALSO INCLUDES ALL THE EVENTS HAPPENED BEFORE e

↳ DISTRIBUTED SNAPSHOT ALGORITHM

↳ SELECTS A CONSISTENT CUT

↳ MESSAGES ARE ORDERED

- Assume FIFO, reliable links/nodes + strongly connected graph
- Any process *p* may initiate a snapshot by
  - Recording its internal state
  - Sending a token on all outgoing channels
    - This signals a snapshot is being run
  - Start recording a local snapshot
    - I.e., record messages arriving on every incoming channel
- Upon receiving a token, a process *q*
  - If not already recording local snapshot
    - Records its internal state
    - Sends a token on all outgoing channels
    - Start recording a local snapshot (see above)
  - In any case stop recording incoming message on the channel the token arrived along
- Recording messages
  - If a message arrives on a channel which is recording messages, record the arrival of the message, then process the message as normal
  - Otherwise, just process the message as normal
- Each process considers the snapshot ended when tokens have arrived on all its incoming channels
  - Afterwards, the collected data can be sent to a single collector of the global state



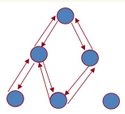
TERMINATION DETECTION → TERMINATION CONDITION: WHEN PROCESSING IS COMPLETE ∀ NODES AND THERE ARE NO MORE MESSAGES IN THE SYSTEM

↳ DIJKSTRA - SCHOLTEN TERMINATION DETECTION

- CREATE A TREE OF ACTIVE PROCESSES
- WHEN A NODE FINISHES AND IT'S A LEAF REMOVE IT
- WHEN THE ROOT FINISHES AND IT'S THE LAST REMAINING THE COMPUTATION HAS TERMINATED

↓

- Each node keeps track of nodes it sends a message to (those it may have woken up), its children
- If a node was already awake when the message arrived, then it is already part of the tree, and should not be added as a child of the sender
- When a node has no more children and it is idle, it tells its parent to remove it as a child



Processing
Idle

← MESSAGES ACTIVATE PROCESSES

COMPARISON:

- Use distributed snapshot
  - Overhead is one message per link
  - Plus cost to collect result
  - If system not terminated, need to run again!
- Use Dijkstra-Scholten
  - Overhead depends on the number of messages in the system
    - Acknowledgments sent when already part of network and when become idle
    - Can be added to network more than once, so this value is not fixed
  - Does not involve never-activated processes
  - Termination detected when last ack received

Simple solution (from Tanenbaum):

- Let call *predecessor* of a process *p*, the process *q* from which it got the first marker. The successors of *p* are all those processes *p* sent the marker
- When a process *p* finishes its part of the snapshot, it sends a DONE message back to its predecessor only if two conditions are met:
  - All of *p*'s successors have returned a DONE message
  - *P* has not received any message between the point it recorded its state and the point it had received the marker along each of its incoming channels
- In any other case *p* sends a CONTINUE
- If the initiator receives all DONE the computation is over; otherwise, another snapshot is necessary

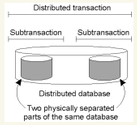
↳ USING DISTRIBUTED SNAPSHOTS

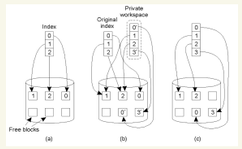
DISTRIBUTED TRANSACTIONS → TRANSACTIONS ARE SEQUENCE OF OPERATIONS, WE MUST PROTECT A SHARED RESOURCE FROM CONCURRENT ACCESSES

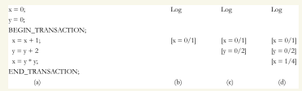
Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

→ TRANSACTIONS ALSO HAVE ACID PROPERTIES (ATOMIC, CONSISTENT, ISOLATED, DURABLE)

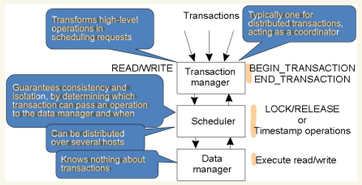
↓ TYPES
- FLAT
- NESTED → 

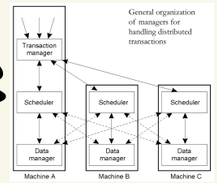
- DISTRIBUTED → 

ACHIEVING ATOMICITY APPROACHES
- PRIVATE WORKSPACE → 

COPY WHAT THE TRANSACTION MODIFIES INTO A SEPARETE MEMORY SPACE, IF ABORTED THE NEW MEMORY SPACE GETS DELETED, OTHERWISE IT'S COPIED INSIDE PARENT'S MEMORY
- WRITE AHEAD LOG → 

IF ABORTED THEN ORIGINAL STATE GETS RESTORED BASED ON LOGS.

CONTROLLING CONCURRENCY → 

DISTRIBUTED SCHEDULER → 

BEGIN_TRANSACTION
x = 0;
x = x + 1;
END_TRANSACTION
	BEGIN_TRANSACTION
	x = 0;
	x = x + 2;
	END_TRANSACTION
		BEGIN_TRANSACTION
		x = 0;
		x = x + 3;
		END_TRANSACTION

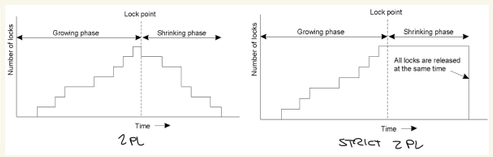
Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3	Illegal

SERIALIZABILITY → INTERLEAVING BETWEEN OPERATIONS IN TRANSACTIONS MUST BE CORRECT

↓
- PROPER SCHEDULING
- MUTUAL EXCLUSION
- PESSIMISTIC/OPTIMISTIC CONCURRENCY CONTROL

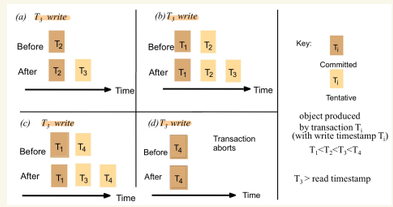
2PL (2 PHASE LOCKING) → PROCESS ASKS SCHEDULER FOR A LOCK

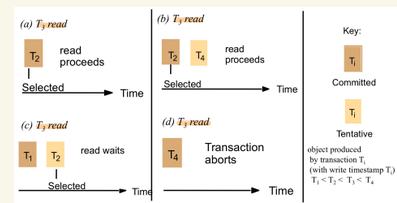
Two-phase locking
- The scheduler tests whether the requested operation conflicts with another that has already received the lock: if so, the operation is delayed
- Once a lock for a transaction T has been released, T can no longer acquire it
- *Strict 2PL*, i.e., releasing the locks all at the same time, prevents cascaded aborts by requiring the shrink phase to take place only after transaction termination
- Proven that 2PL leads to serializability… but it may deadlock
- Widely used



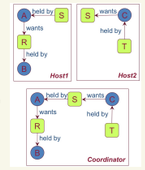
PESSIMISTIC TIMESTAMP ORDERING → EACH TRANSACTION IS TIMESTAMPED

↓
- WRITE ARE TIMESTAMPED AND TENTATIVE, UNTIL A COMMIT
- EACH DATA ITEM ALSO HAS A LAST READ TIMESTAMP
- SCHEDULER:
  - WHEN RECEIVE $W(T,x)$ AT TIME t_s → IF $t_s > t_{READ}(x)$ AND $t_s > t_{WRITE}(x)$ DOES A TENTATIVE WRITE WITH TIMESTAMP t_s ELSE ABORTS T → WRITE
  - WHEN RECEIVES $R(T,x)$ AT TIME t_s → IF $t_s > t_{SVR}(x)$ AND THE PREVIOUS WRITE ON X IS COMMITTED READ X AND SET $t_{SRD}(x) = MAX(t_s, t_{SRD}(x))$ ELSE ABORT T → READ



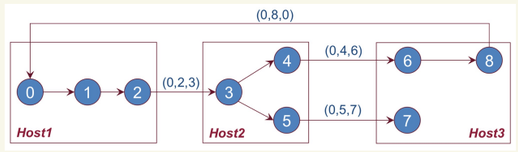


OPTIMISTIC TIMESTAMP ORDERING → LET TRANSACTIONS DO WHAT THEY WANT AND FIX CONFLICTS LATER



DEADLOCKS

- CENTRALIZED DEADLOCK DETECTION: EACH MACHINE HAS A RESOURCE GRAPH FOR HIS RESOURCES AND REPORTS TO A COORDINATOR
- DISTRIBUTED DEADLOCK DETECTION: NO COORDINATOR, PROCESSES CAN REQUEST MULTIPLE RESOURCES SIMULTANEOUSLY, WHEN A PROCESS GETS BLOCKED IT SENDS A PROBE MESSAGE TO THE PROCESS HOLDING RESOURCES IT WANTS TO ACQUIRE



↓

(INITIATOR, SENDER, RECEIVER) → CYCLE WHEN INITIATOR = RECEIVER → DEADLOCK!

↓

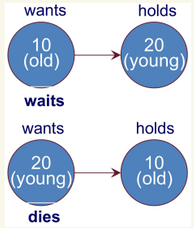
WHEN DEADLOCK → INITIATOR SUICIDE
OR
PROCESS WITH HIGHER ID KILLED

- DISTRIBUTED PREVENTION (USING TIMESTAMPS)

  WITH PREEMPTION

  WAIT-DIE ALGORITHM

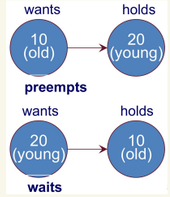
  - When a process A is about to block for a resource that another process B is using, allow A to wait only if A has a lower timestamp (it is older) than B; otherwise kill the process A
  - Following a chain of waiting processes, the timestamps will always increase (no cycles)



OR REQ:
→ OLDER WAITS
YOUNGER DIES

  WOUND-WAIT ALGORITHM

  - Preempting the young process aborts its transaction, and it may immediately try to reacquire the resource, but will wait
  - In wait-die, young process may die many times before the old one releases the resource: here, this is not the case



→

OLDER STEALS ACCES
YOUNGER WAITS

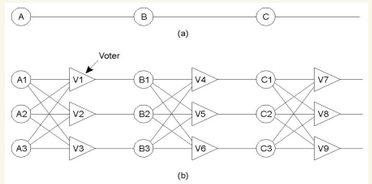
FAULT TOLERANCE → ACHIEVE GOOD AVAILABILITY, RELIABILITY, SAFETY AND MAINTAINABILITY

↳ SYSTEM FAILS WHEN IT CAN'T PROVIDE HIS SERVICES → FAILURE RESULT OF AN ERROR → ERROR CAUSED BY FAULTS
- TRANSIENT
- INTERMITTENT
- PERMANENT

↓

BEST TECNIQUE = REDUNDANCY!

↳
- OMISSION
- BYZANTINE
- TIMING



Replicate each sub-module: they won't be all rotten

CLIENT - SERVER COMMUNICATION → TCP PROVIDES RELIABLE COMMUNICATION BUT CRASH FAILURES ARE NOT EASILY MASKED

↳ WITH RPC WE HAVE SOME PROBLEMS:

- SERVER CAN'T BE LOCATED → CLIENT HANDLES IT AS AN EXCPTION
- LOST REQUESTS → CLIENT CAN SEND IT AGAIN, IF TOO MANY REQUESTS ARE LOST IT CAN ASSUME THE SERVER IS DOWN
- SERVER CRASHES → CLIENT CAN'T TELL WHEN THE SERVER CRASHED, IT DOESN'T KNOW ALSO IF THE SERVER COMPUTATION STARTED OR NOT
- LOST REPLIES → CAN'T TELL IF REPLY IS LOST OR REQUEST DID NOT ARRIVE
- CLIENT CRASHES → COMPUTATION STARTED BY A DEAD CLIENT IS AN ORPHAN THAT NEEDS TO BE KILLED
  - EXTERMINATION
  - REINCARNATION
  - GENTLE REINCARNATION
  - EXPIRATION

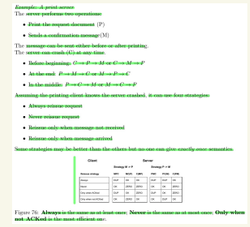


Figure 76: **Always** is the same as at least once. **Never** is the same as at most once. **Only when not ACKed** is the most efficient one.

PROTECTION AGAINST PROCESS FAILURES

↳ USE REDUNDANCY → TASK OF A PROCESS TAKEN BY A GROUP OF PROCESSES

↓

HOW BIG THE GROUP NEEDS TO BE?
- SILENT FAILS → $k+1$ PROCESSES ALLOW THE SYSTEM TO BE K-FAULT-TOLERANT
- BYZANTINE FAILURES → $2k+1$ PROCESSES REQUIRED TO BE K-FAULT-TOLERANT

↓

HOW CAN THE PROCESSES IN THE GROUP AGREE?

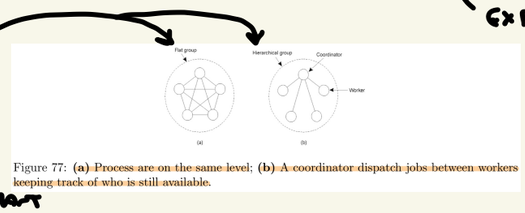


Figure 77: **(a)** Process are on the same level; **(b)** A coordinator dispatch jobs between workers keeping track of who is still available.

CONSENSUS PROBLEM → CONDITIONS:
- EACH PROCESS STARTS WITH SOME INITAL VALUE
- ALL NON-FAULTY PROCESSES HAVE TO REACH A DECISION
- FOLLOWING PROPERTIES MUST HOLD: AGREEMENT, VALIDITY, TERMINATION

↳ IN ASYNCH SYSTEMS PROBLEM IS NOT SOLVABLE

↓

PEPPERED WITH RELIABLE COMMUNICATIONS BUT FAILURES IN PROCESSES → ASSUMING SYNCHRONOUS SYSTEM

↳ CRASH FAILURES

↳ CONSENSUS IS POSSIBLE, WE NEED THAT PROCESSES TAKE AT LEAST F+1 ROUNDS (SUFFICIENT THAT A SINGLE PROCESS ISN'T FAULTY)

BOUND ON NUMBER OF FAILURES

↓

FLOODSET ALGORITHM

Let v_0 be a pre-specified default value. Each process maintains a variable W (*subset of V*) initialized with its start value.
The following steps are repeated for $f+1$ rounds
1. Each process sends W to all other processes
2. It adds the received sets to W

The decision is made after $f+1$ rounds
- If $|W|=1$: decide on W's element
- If $|W|>1$: decide on v_0 (or use the same function on all the processes to dedice, e.g., $max(W)$)

Remember: each process may stop in the middle of the send operation

In conclusion, supposing that we want to be 5-fault-tolerant we will need $5+1=6$ steps to agree. Then after the 6^{th} step, processes have to agree on a value. Since all of the on that step will have the same sets, they need to agree on the value using the same function.

Example
Suppose to have 4 not reliable processes, each starting with its own value:
$W_1=\{0\}$, $W_2=\{1\}$, $W_3=\{0\}$, $W_4=\{0\}$.

At each step, each process i send its W_i to all other processes. When a process i receives a message from process j, it adds the W_j to its own W_i.
In this case, after the first step, the processes' state will look like this:
$W_1=\{0,1\}$, $W_2=\{1,0\}$, $W_3=\{0,1\}$, $W_4=\{0,1\}$. Finally they can reach an agreement using the same function to ensure that each process selects the same value. For instance, if they use $max(W)$, the value on which they will agree will be 1.

In case all processes started with $W=\{0\}$, then they all agree on 0, because everybody in the end will have $W=\{0\}$.

The complexity of the FloodSet algorithm can be reduced by using an optimized version:
- Each process needs to know the entire set W only if its cardinality is greater than 1
- The improved algorithm broadcasts W at the first round
- In addition each process broadcasts W again when it first learns of a new value

PROVING CORRECTNESS

Lemma:
1. If no process fails during a particular round r: $1 \leq r \leq f+1$, then $W_i(r)=W_j(r)$ for all i and j that are active after r rounds.
2. Suppose that $W_i(r)=W_j(r)$ for all r and j that are active after r rounds. Then for any round r_1: $r \leq r_1 \leq f+1$, the same holds, that is, $W_i(r_1)=W_j(r_1)$ for all i and j that are active after r_1 rounds.
3. If processes i and j are both active after $f+1$ rounds, then $W_i=W_j$ at the end of round $f+1$.

In other words, if there is a round where no crashes happen, then in the end of that round we have the same W for every process and from that round on, the value of W does not change anymore.

↳ BYZANTINE FAILURES → PROCESSES DOESN'T ONLY FAIL BUT CAN ALSO HAVE WEIRD BEHAVIORS (SEND MESSAGES, CHANGE STATE, ...)

↓

WE NOW HAVE N GENERALS WHO HAVE TO REACH A COORDINATED DECISION, THEY ANNOUNCE THEIR TROOP STRENGHT AND DECIDE BASED ON THE TOTAL NUMBERS OF TROOPS, BUT SOME GENERALS ARE TRAITORS

↳
- AGREEMENT: NO TWO LOYAL GENERALS LEARN DIFFERENT TROOP STRENGHT VALUES
- VALIDITY: IF ALL LOYAL GENERALS ANNOUNCES V, THEN ALL LOYAL GENERALS LEARN THAT HIS TROOP STRENGHT IS V
- TERMINATION: ALL LOYAL GENERALS EVENTUALLY LEARN TROOP STRENGHT OF ALL OTHER LOYAL GENERALS.

↓

SOLUTION STEPS:
- SEND TROOP STRENGTH TO OTHERS
- FORM A VECTOR WITH RECEIVED VALUES
- SEND VECTOR TO OTHERS
- COMPUTE VECTOR USING MAJORITY (CONFIRM MOST USED VALUE) FOR EACH VECTOR POSITION

⟹

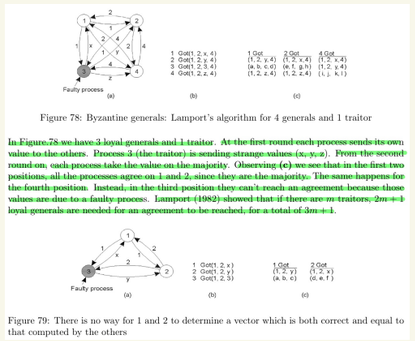


Figure 78: Byzantine generals: Lamport's algorithm for 4 generals and 1 traitor

~~In Figure 78 we have 3 loyal generals and 1 traitor. At the first round each process sends its own value to the others. Process 3 (the traitor) is sending strange values (x, y, z). From the second round on, each process take the value on the majority. Observing (c) we see that in the first two positions, all the processes agree on 1 and 2, since they are the majority. The same happens for the fourth position. Instead, in the third position they can't reach an agreement because those values are due to a faulty process. Lamport (1982) showed that if there are m traitors, $2m+1$ loyal generals are needed for an agreement to be reached, for a total of $3m+1$.~~

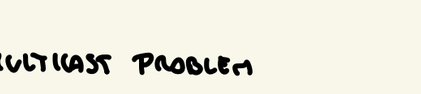


Figure 79: There is no way for 1 and 2 to determine a vector which is both correct and equal to that computed by the others

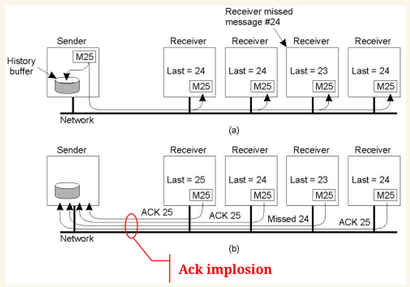
RELIABLE GROUP COMMUNICATION → FIXED GROUPS AND NON-FAULTY PROCESSES

↳ 2 APPROACHES
- ACKS: SENDER MULTICAST OVER UDP AND WAIT FOR ACK, IF NOT RECEIVED RESEND THE MESSAGE → ACK IMPLOSION
- NACKS: SENDER NEEDS TO CACHE MESSAGES, NACK SENT ONLY IF A PROCESS MISSES A PACKET.

- WITH NON-FAULTY PROCESSES

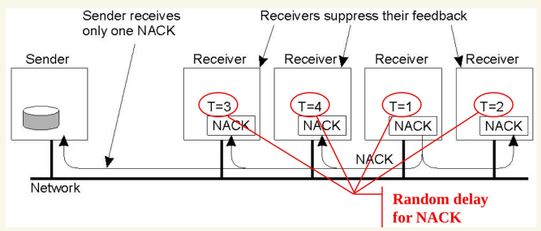
↓

BASIC APPROACH



↘

SCALABLE RELIABLE MULTICAST:
- NON-HIERARCHICAL FEEDBACK CONTROL

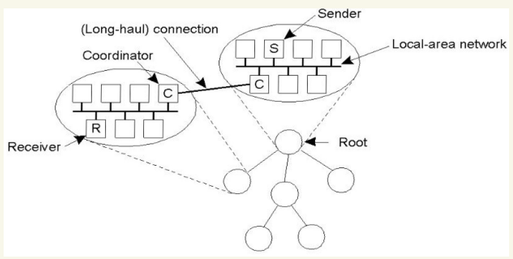


- HIERARCHICAL FEEDBACK CONTROL

↑

RECEIVERS ORGANIZED IN GROUPS HEADED BY A COORDINATOR, GROUPS ORGANIZED IN TREES

↓
- CAN ADOPT ANY STRATEGY
- CAN REQUEST RETRANSMISSION TO ITS PARENT COORDINATOR
- REMOVE MESSAGES FROM BUFFER IF ACK RECEIVED



- WITH FAULTY PROCESSES → THEY CAN FAIL OR JOIN/LEAVE GROUPS DURING COMMUNICATION

WE NEED THE MESSAGE ORDERED AND DELIVERED TO EVERYONE OR TO NO-ONE → ATOMIC MULTICAST PROBLEM

↳ CLOSE SYNCHRONY CAN'T BE ACHIEVED

↓

VIRTUAL SYNCHRONY

↓

MODEL

- CRASHED PROCESSES GET PRUNED FROM THE GROUP AND NEED TO REJOIN
- MESSAGES FROM A CORRECT PROCESS ARE PROCESSED BY ALL CORRECT PROCESSES
- MESSAGES FROM A FAILING PROCESS ARE PROCESSED EITHER BY ALL CORRECT PROCESSES OR BY NONE
- ONLY RELEVANT MESSAGES ARE RECEIVED IN A "SPECIFIC" ORDER

↳ USEFUL TO DISTINGUISH BETWEEN RECEIVING AND DELIVERING A MESSAGE

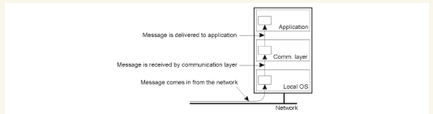


Figure 83: Messages received by the communication layer are buffered there and later delivered when some condition holds.

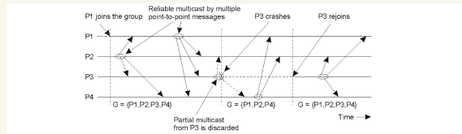


Figure 84: The message sent by P3 "while" crashing are discarded.

GROUP VIEW: SET OF PROCESSES TO WHICH A MESSAGE SHOULD BE DELIVERED AS SEEN BY THE SENDER ⇒ ORDERING REQ: GROUP VIEW CHANGES DELIVERED IN A CONSISTENT ORDER WITH RESPECT TO OTHER MULTICAST AND EACH OTHER ⇒ RELIABLE MULTICAST (VIRTUALLY SYNCHRONOUS)

↓

VIEW CHANGES OCCURS WHEN A PROCESS JOINS/LEAVES THE GROUP (POSSIBLE CRASH)

↳ ALL MULTICAST MESSAGES MUST TAKE PLACE BETWEEN VIEW CHANGES (CUTS) →

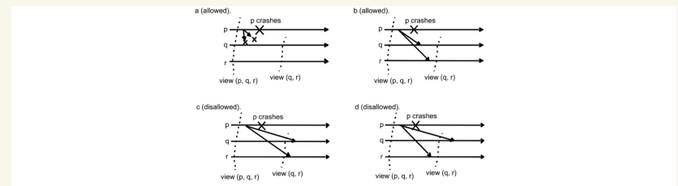


Figure 85: (a) p crashes, the message it sent is not delivered to anyone; (b) p crashes, but q and r received the message from p before knowing that p crashed (before the cut); (c) p crashes, q and r received the message after becoming aware of the fact that p crashed, not acceptable; (d) p crashes, r receive the message before knowing that p crashed. The same does not hold for q.

⇓

MESSAGE ORDERING WITH VIRTUAL SYNCHRONY PROPERTY

↳ MULTICASTS → UNORDERED →

FIFO →

CAUSALLY-ORDERED

ATOMIC →

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

MESSAGES FROM THE SAME SENDER MUST BE RECEIVED IN THE SAME ORDER BY EVERYONE

Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

IMPLEMENTATION OF VIRTUAL SYNCHRONY:

↳

- We assume that processes are notified of view changes by some (possibly distributed) component.
- When a process receives a view change message, it stops sending new messages until the new view is installed. In the meanwhile it multicasts all the pending unstable messages to the non-faulty members of the old view, and marks them as stable and multicasts a flush message.
- Eventually all the non-faulty members of the old view will receive the view change and do the same (duplicates are discarded).
- Each process installs the new view as soon as it has received a flush message from each other process in the new view. Now it can restart sending new messages.

→

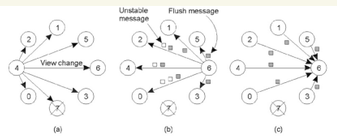


Figure 87: Graphic schema of the previously described protocol.

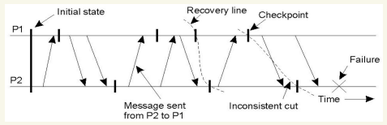
87, we have that:

- Process 4 notices that process 7 has crashed, and sends a view change
- Process 6 sends out all its unstable messages, followed by a flush message
- Process 6 installs the new view when it has received a flush message from everyone else

RECOVERY TECHNIQUES → RESUME PROCESS TO HIS PREVIOUS CORRECT STATE

- FORWARD RECOVERY → NEW CORRECT STATE TO RESUME COMPUTATION
- BACKWARD RECOVERY → RECOVERY LAST SAVED CORRECT STATE

• UNCOORDINATED CHECKPOINTING → PERIODICALLY SAVE STATE ON STORAGE



→ NEED TO FIND MOST RECENT CONSISTENT CUT

INITIAL STATE ∩

DOMINO EFFECT!

↳ IF EVERY PROCESS INDEPENDENTLY SAVES HIS OWN STATE AND THEN WE HAVE A FAILURE → SEARCH FOR A CONSISTENT CUT

↳ ALSO TRIVIAL TO IMPLEMENT →
- INTERVALS BETWEEN CHECKPOINTS ARE TAGGED
- EACH MESSAGE LABELED WITH A REFERENCE TO THE INTERVAL HE'S IN
- RECEIVER CAN STORE INFORMATION ABOUT INTERVALS AND MESSAGES

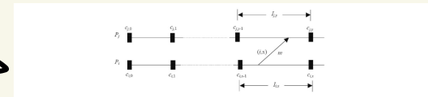
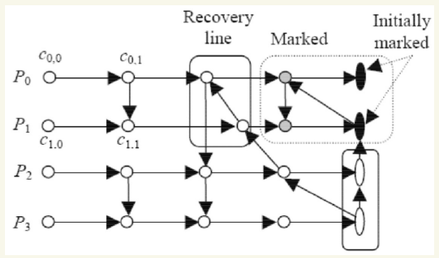


Figure 91: Each process knows if its own interval depends on other processes interval

WHEN FAIL

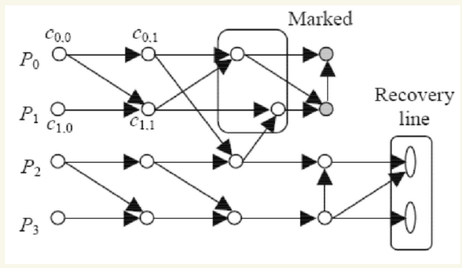
↳ PROCESS IN RECOVERY ASKS TO EVERYONE DEPENDENCY INFO → ALL PROCESSE SEND INFO TO A GLOBAL COORDINATOR THAT BUILDS A DEPENDENCY GRAPH → RECOVERING PROCESS COMPUTES THE RECOVERY LINE AND SENDS A ROLLBACK REQUEST → RECOVERY LINE CAN BE COMPUTED USING
- ROLLBACK DEPENDENCY GRAPH
- CHECKPOINT GRAPH

- ROLLBACK DEPENDENCY GRAPH



IF THERE'S A DEPENDENCY BETWEEN INTERVALS → TRANSFORM IT INTO CHECKPOINT DEPENDENCY
THEN FROM THE CRASH FOLLOW THE ARROWS AND MARK STATES REACHABLE FROM THEM
THEN ROLLBACK TO THE LAST UNMARKED STATES FOR EACH STATE

- CHECKPOINT DEPENDENCY GRAPH



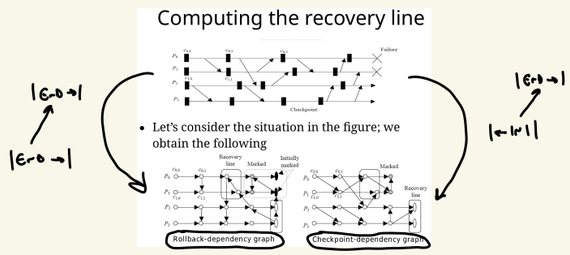
REMOVE FAILURE STATES AND START CONSIDERING ALL THE LAST STATES FOR EACH PROCESS

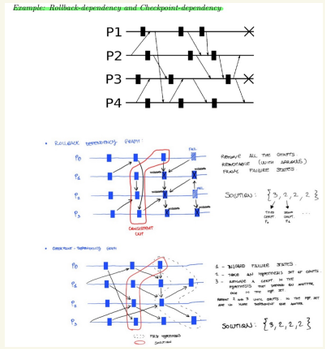
↑ HP GROUP

REMOVE A STATE FROM THE HP GROUP THAT DEPEND ON ANOTHER STATE IN THE HP GROUP UNTIL THERE ARE NO MORE DEPENDENCIES INSIDE THE HP GROUP.

FINAL HP GROUP IS THE RECOVERY LINE

EXAMPLE FOR BOTH :





- COORDINATED CHECKPOINTING
  - COORDINATOR SENDS CHECKPOINT REQ.
  - RECEIVERS TAKE CHECKPOINT AND QUEUE OUTGOING MESSAGES, THEN ACK THE COORDINATOR
  - COORDINATOR SENDS CHECKPOINT - DONE
  - IMPROVED USING - INCREMENTAL SNAPSHOT → ONLY CHECKPOINT-REQ PROCESSES THAT DEPEND ON THE RECOVERY OF THE COORD.
    - GLOBAL SNAPSHOT → PROCESSES NOT BLOCKED WHILE CHECKPOINT IS BEING TAKEN

- LOGGING → CAN BE USED WITH CHECKPOINTING
  - START FROM CHECKPOINT AND REPLAY ISTRUCTION STORED IN A LOG → WORKS IF SYSTEM IS PIECEWISE DETERMINISTIC
    - EXECUTION PROCEDES DETERMINISTICALY BETWEEN THE RECEIPT OF TWO MESSAGES
  - MESSAGES HEADERS CONTAINS INFO TO REPLAY THE MESSAGES
    - STABLE IF CAN BE LOST
    - UNSTABLE OTHERWISE
      - DEP(m) → PROCESSES THAT DEPEND ON THE MESSAGE m
      - COPY(m) → PROCESSES THAT HAVE A COPY OF m, BUT NOT IN STABLE STORAGE YET

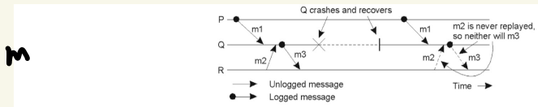


Figure 93: We may arrive in a state in which $M1$ is sent again because logged, while $M2$ is not re-sent because it was not logged, and then $M3$ cannot be sent again because maybe it was causally related to $M2$.

ORPHAN PROCESS: Q IS ORPHAN IF $\exists$ m SUCH THAT Q IS IN DEP(m) AND ALL PROCESSES IN COPY(m) CRASHED
↓
THERE IS NO WAY TO REPLAY m THEN

- PESSIMISTIC LOGGING: ENSURE THAT ALL UNSTABLE MESSAGES ARE DELIVERED AT LEAST AT ONE PROCESS
- OPTIMISTIC LOGGING: MESSAGES LOGGED ASYNCHRONOUSLY ASSUMING THAT THEY'LL BE LOGGED BEFORE A FAULT

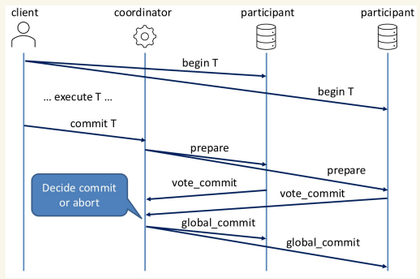
CONSENSUS IN DS - DISTRIBUTED AGREEMENT IN PRACTICE

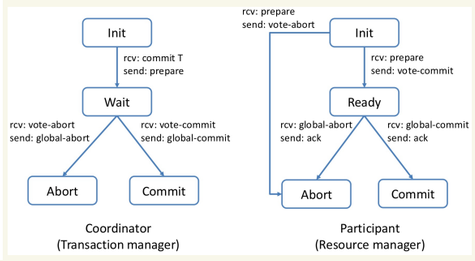
Consensus	Atomic commit
One or more nodes propose a value	Every node votes to commit or abort
Nodes agree on one of the proposed values	Commit if and only if all nodes vote to commit, abort otherwise
Tolerates failures, as long as a majority of nodes is available	Any crash leads to an abort

COMMIT PROTOCOLS

↳ DISTRIBUTED ATOMIC COMMIT → ALL NODES EITHER COMMIT OR ABORT → IF ONE CRASH ALL ABORTS

- TWO PHASE COMMIT (2PC) → ASSUMING ALL NODES NEED TO REACH AN AGREEMENT





POSSIBLE FAILURES → PARTICIPANT → COORDINATOR ASSUMES ABORT

→ COORDINATOR → PARTICIPANT IN → INIT STATE → CAN DECIDE TO ABORT

→ READY STATE → CAN'T DECIDE BY ITSELF

COORDINATOR:

```
write start_2PC to local log;
multicast prepare to all participants;
while not all votes have been collected {
        wait for any incoming vote;
        if timeout {
                write global-abort to local log;
                multicast global-abort to all participants;
                exit;
        }
        record vote;
}
if all participants sent vote-commit {
        write global-commit to local log;
        multicast global-commit to all participants;
} else {
        write global-abort to local log;
        multicast global-abort to all participants;
}
```

Log is assumed to be on durable storage. It survives crashes.

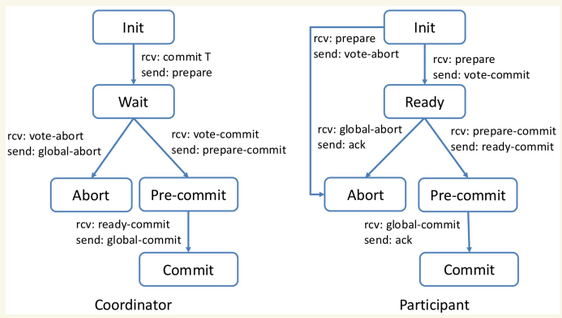
PARTICIPANT:

```
write init to local log;
wait for prepare from coordinator;
if timeout {
        write vote-abort to local log;
        exit;
}
if participant votes vote-commit {
        write vote-commit to local log;
        send vote-commit to coordinator;
        wait for global-decision from coordinator;
        if timeout {
                multicast decision-request to other participants;
                wait until global-decision is received; /* remain blocked */
        }
        write global-decision to local log;
} else {
        write vote-abort to local log;
        send vote-abort to coordinator;
}
```

⇒ 2PC IS SAFE BUT IT MAY BLOCK

- THREE PHASE COMMIT (3PC)

↳ WE HAVE ANOTHER PHASE → SPLIT COMMIT/ABORT PHASE INTO TWO PHASES



POSSIBLE FAILURES → PARTICIPANT → COORD BLOCKED IN → WAIT STATE : ASSUME ABORT

→ PRE-COMMIT STATE : COMMIT AND COMMIT THE FAILED ONE WHEN RECOVERS

→ COORDINATOR

→ GUARANTEES SAFETY, NEVER LEADS TO INCORRECT STATE

↳ PARTICIPANT BLOCKED IN → INIT STATE → ABORT

→ READY → CONTACT OTHER PARTICIPANTS AND FINDS :

- ABORT → ABORT
- COMMIT → COMMIT
- INIT → ABORT
- PRE-COMMIT → COMMIT
- # PRE-COMMIT + # READY FROM MAJORITY → COMMIT
- NO-ONE PRE-COMMIT + # READY FROM MAJORITY → ABORT

→ IN SYNCH → LIVENESS

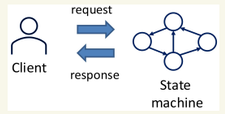
↳ IN ASYNCH → MAY NOT TERMINATE

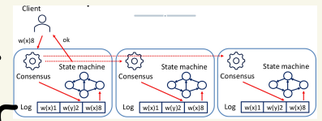
FLP THEOREM: CAN'T HAVE LIVENESS AND SAFETY TOGETHER WITH NETWORK PARTITIONS AND ASYNCH SYSTEM

CAP THEOREM: ANY DS THAT HAS SOME SHARE REPLICATED DATA CAN'T HAVE ALL THREE OF THESE:

- CONSISTENCY
- HIGH AVAILABILITY
- TOLERANCE TO PARTITIONS

REPLICATED STATE MACHINES → GENERAL PURPOSE CONSENSUS ALGORITHM ALLOWS MANY MACHINES TO WORK AS A COHERENT GROUP, OPERATING ON IDENTICAL COPIES OF THE SAME STATE → STATE MACHINE





↳ REPLICATED LOG ENSURES THAT MACHINES EXECUTE THE SAME COMMAND IN THE SAME ORDER

↳ GUARANTEES: • SAFETY
• LIVENESS/AVAILABILITY

POSSIBLE FAILURES: • UNRELIABLE, ASYNCH COMMUNICATION
• PROCESSES MAY FAIL

PAXOS → AGREEMENT ON SINGLE DECISIONS ONLY, NOT A SEQUENCE

↳ DIFFICULT TO USE AND UNDERSTAND

RAFT → GOAL: UNDERSTANDABILITY → RAFT EQUIVALENT TO MULTI-PAXOS

↳ APPROACH: PROBLEM DECOMPOSITION → LEADER ELECTION
→ LOG REPLICATION MANAGED BY LEADER
→ SAFETY → KEEP LOG CONSISTENT

⇓

NODES CAN BE IN 3 STATES: LEADER, FOLLOWER AND CANDIDATE

↳ LEADER → ALL COMMANDS PASS THROUGH THEM AND HE'S RESPONSIBLE FOR COMMITTING AND PROPAGATING
FOLLOWER → IF IT DOESN'T HEAR LEADER FOR A WHILE IT BECOMES A CANDIDATE
CANDIDATE → RUNS ELECTIONS, IF IT WINS IT BECOMES LEADER

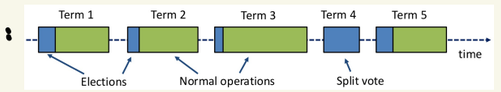
↳ NEEDS THE LOG UP TO DATE TO WIN

↓

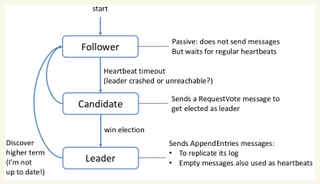
OPERATIONS:

IF FOLLOWER DOESN'T HEAR FROM THE LEADER (TIMEOUT) THEY START AN ELECTION

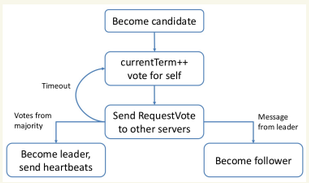
→ WORKFLOW:



→ SERVER STATE AND MESSAGES

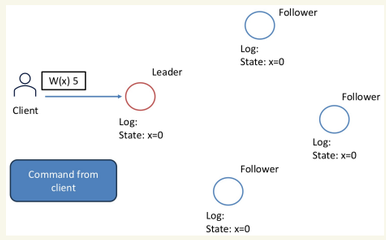


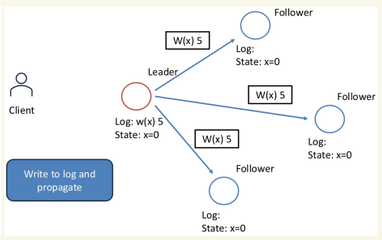
→ LEADER ELECTION

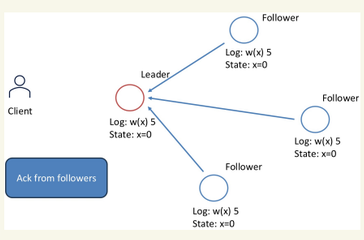


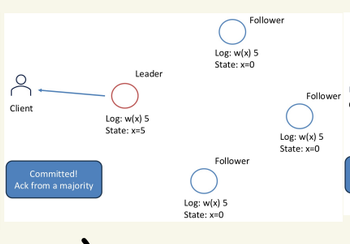
→ SAFETY

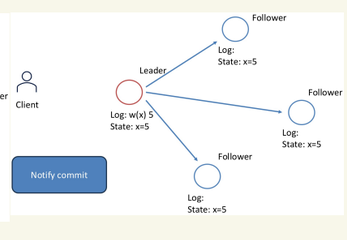
→ LIVENESS



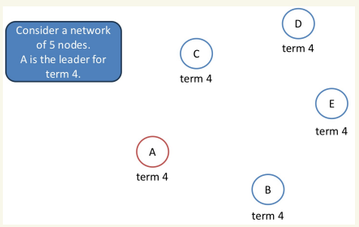


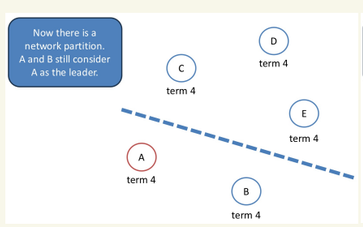


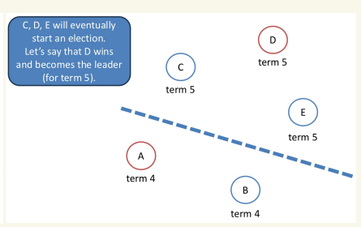


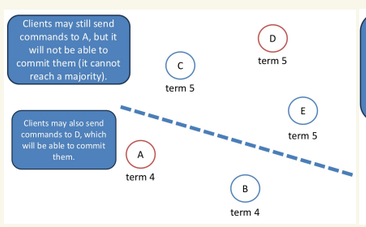


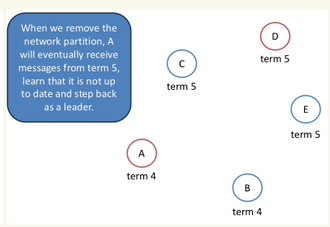
↓ WHAT HAPPENS WHEN THE GROUP GETS SPLIT (NETWORK PARTITION)





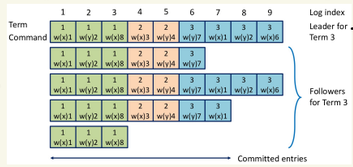






THIS CREATES → COULD BE CAUSE BY CRASHES ALSO

LOG INCONSISTENCIES



→ LEADER ASSUMES HIS LOG IS CORRECT
↓
NORMAL OPERATIONS WILL FIX INCONSISTENCIES

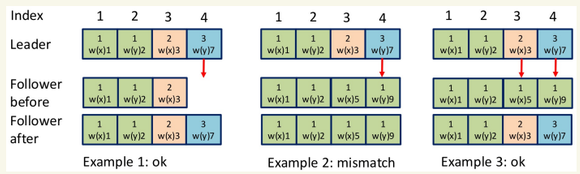
↓

RAFT GUARANTEES THE LOG MATCHING PROPERTY

CONSISTENCY CHECK

MESSAGES APPENDED HAS <INDEX, TERM> OF THE ENTRY PRECEDING THE NEW ONE(S)

FOLLOWERS MUST CONTAIN MATCHING ENTRIES → OTHERWISE REJECTS REQUEST AND LEADER RETRIES WITH LOWER LOG INDEX



RAFT GUARANTEES LINEARIZABLE SEMANTICS → ALL OPERATIONS BEHAVE AS IF THEY WERE EXECUTED ONCE AND ONLY ONCE ON A SINGLE COPY

SOME DBMS USE REPLICATED STATE MACHINES AND COMMIT PROTOCOLS

BYZANTINE FAULT TOLERANT REPLICATION / CONSENSUS → BLOCKCHAINS

BYZANTINE ENVIRONMENT

↳ CRYPTOCURRENCIES CAN BE SEEN AS REPLICATED STATE MACHINES → STORED IN A REPLICATED LOG

USER MAY TRY TO DOUBLE SPEND CREATING INCONSISTENCIES IN LOG

↓ CURRENT BALANCE

BLOCKCHAIN IS THE REPLICATED LOG THAT RECORDS TRANSACTIONS → CONTAINS ALL TRANSACTIONS DONE, EACH BLOCK OF THE CHAIN HAS MULTIPLE TRANSACTIONS

↓

ADDING A BLOCK TO THE CHAIN REQUIRES SOLVING A MATHEMATICAL PROBLEM → SOLUTION DIFFICULT TO FIND BUT EASY TO VERIFY → REWARD: BITCOINS

- Proof of work computed by special nodes (miners) that collect new (pending) transactions into a block
  - Incentive: they earn Bitcoins if successful
- When a miner finds the proof, it broadcasts the new block
  - This defines the *next* block of valid transactions
- The other miners receive it and try to create the next block in the chain
  - Global agreement on the order of blocks!

- What if two miners find a proof concurrently?
  - The proof is very complex to compute
  - Very unlikely that two computers will find a solution at the same time
  - Very difficult for someone to *force* their desired order of transactions, since it would require a lot of compute power
- If two concurrent versions are created, the one that grows faster (includes more blocks) survive
  - If same length → deterministic choice
- Still, there may always exist a longer chain I'm not aware of
  - No one can be 100% sure of a given sequence

DOUBLE SPENDING →

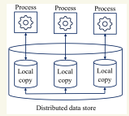
- Someone with enough computational power and 1 Bitcoin can create two concurrent chains
  - Chain 1: the Bitcoin is used to pay A and the chain is propagated to A
  - Chain 2: the Bitcoin is used to pay B and the chain is propagated to B
  - A and B can never be 100% sure that a conflicting chain does not exist!

REPLICATION AND CONSISTENCY

REPLICATE TO ACHIEVE FAULT TOLERANCE, INCREASED AVAILABILITY AND PERFORMANCE, REDUCE LATENCY

↳ CHALLENGES: CONSISTENCY ACROSS REPLICAS → COST FOR CONSISTENCY CAN BE VERY HIGH
GOAL: PROVIDE CONSISTENCY WITH LITTLE OVERHEAD

CONSISTENCY MODELS → FOCUS ON DISTRIBUTED DATA STORE



↳ CONTRACT BETWEEN PROCESSES AND DATA STORE
- STRICT GUARANTEES → EZ TO DEVELOP BUT COSTLY
- WEAK GUARANTEES → DIFFICULT TO DEVELOP LESS COSTLY

- GUARANTEES ON CONTENT → SET A MAXIMUM DIFFERENCE BETWEEN VERSIONS IN REPLICAS
- GUARANTEES ON STALENESS → SET A MAXIMUM PROPAGATION TIME BETWEEN REPLICAS
- GUARANTEES ON THE ORDER OF UPDATES → CONSTRAIN POSSIBLE BEHAVIORS IN CASE OF CONFLICTS
  - DATA-CENTRIC
  - CLIENT-CENTRIC

SINGLE LEADER PROTOCOLS → SINGLE ORGANIZATION/DATA CENTER

CLIENTS ONLY CONTACT THE LEADER, HE WRITES IN LOCAL THEN CONTACT THE FOLLOWERS TO MAKE THEM WRITE TOO

CLIENTS WANTS TO READ → CAN CONTACT LEADER OR REPLICA BASED ON THE SYSTEM USED

- SYNCH → WRITE COMPLETES AFTER ALL THE FOLLOWERS REPLIED → SYNCH REPLICATION IS SAFER
- ASYNCH → WRITE FINISHES WHEN LEADER IS DONE, FOLLOWERS UPDATED IN ASYNCH
- SEMI-SYNCH → WRITE COMPLETES WHEN LEADER RECEIVE AT LEAST K REPLIES

NO WRITE-WRITE CONFLICT POSSIBLE

READ-WRITE CONFLICTS STILL POSSIBLE

MULTI LEADER PROTOCOLS → ADOPTED IN GEO-REPLICATED APPLICATIONS → IN PRACTICE RARE CONFLICTS AND EASY TO SOLVE

WRITES ARE CARRIED OUT AT DIFFERENT REPLICAS CONCURRENTLY → POSSIBLE TO HAVE WRITE-WRITE CONFLICTS (DUE TO CONCURRENCY)

NATIVELY SUPPORTED IN SOME DBs

LEADERLESS PROTOCOLS

CLIENT CONTACTS MULTIPLE REPLICAS TO PERFORM A READ/WRITE

USE QUORUM-BASED PROTOCOLS TO AVOID CONFLICTS

HIGHLY AVAILABLE PROTOCOLS → DOES NOT REQUIRE SYNCH/BLOCKING COMMUNICATION → IF NODE FAILS CLIENT CAN GET REPLY FROM REPLICA

↳ RELATED TO CAP THEOREM → WITH NETWORK FAILURES (P) → WE CAN HAVE AVAILABILITY (A) OR CONSISTENCY (C) BUT NOT BOTH

DATA-CENTRIC CONSISTENCY MODELS → W(x)a / R(x)a MEANS WE ARE WRITING/READING a FROM x

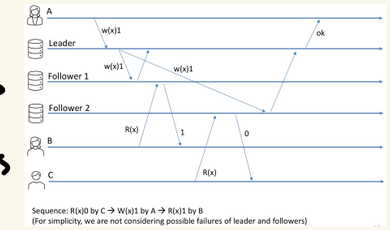
↳ IN DS WRITES AREN'T INSTANTANEOUSLY VISIBLE AND GLOBAL ORDER IS NOT MAINTAINED

- **SEQUENTIAL CONSISTENCY** → ALL PROCESSES SEE THE SAME INTERLEAVING AND OPERATIONS WITHIN A PROCESS MAY NOT BE RE-ORDERED

P1: W(x)a	P1: W(x)a
P2: W(x)b	P2: W(x)b
P3: R(x)b R(x)a	P3: R(x)b R(x)a
P4: R(x)b R(x)a	P4: R(x)a R(x)b
Consistent	NOT Consistent

→ ALL REPLICAS NEED TO AGREE ON A GIVEN ORDER OF OPERATIONS

↳ USE SINGLE LEADER REPLICATION WITH SYNCH →



↳ WORKS ASSUMING: • LINKS ARE FIFO: RECEIVE MESSAGES IN THE ORDER THEY WERE SENT

↓ LEADERLESS IMPLEMENTATION

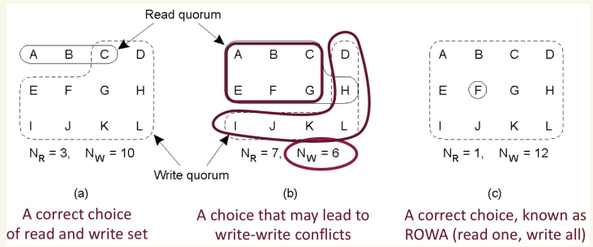
QUORUM BASED: • TO W/R CLIENTS CONTACT MULTIPLE REPLICAS

• READS REQUIRE QUORUM TO ENSURE THAT WE'RE READING THE LATEST VALUE

• STICKY CLIENTS: CLIENTS ALWAYS READ FROM THE SAME REPLICA

• $R + W > N.\text{TOT OP}$ → AVOID R/W CONFLICTS

$W > \frac{N.\text{TOT OP}}{2}$ → AVOID W/W CONFLICTS



TO ENSURE CONSISTENCY CLIENTS FIX THE REPLICAS THAT ARE NOT UP TO DATE (THEY READ MANY IN PARALLEL) → CALLED READ-REPAIR

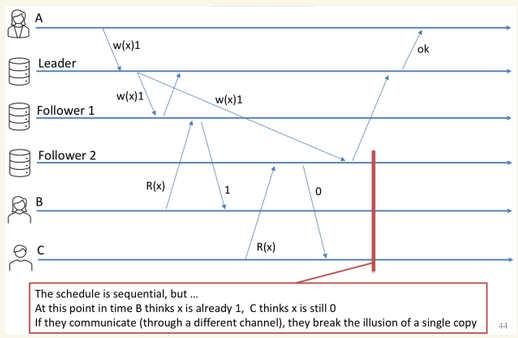
ALSO NODES EXCHANGE DATA IN BACKGROUND TO REMAIN UP-TO-DATE (ANTI-ENTROPY)

SEQUENTIAL CONSISTENCY LIMITS AVAILABILITY → HIGH LATENCY

↳ CLIENTS BLOCKED WITH NETWORK PARTITIONS

- **LINEARIZABILITY** → STRONGEST POSSIBLE CONSISTENCY GUARANTEE → WHEN A CLIENT WRITES, ALL OTHER CLIENTS NEED TO SEE THE NEW VAL

↳ INCLUDES NOTION OF TIME (SEQUENTIAL CONSISTENCY DOESN'T) → OPERATIONS BEHAVE AS IF THEY TOOK PLACE AT A SINGLE POINT OF TIME



- GLOBAL ORDER IS MAINTAINED
- ALL WRITES BECOME VISIBLE AT SOME ISTANT IN TIME
- OPERATIONS HA A DURATION

⇒ IF OPERATIONS ON A VARIABLE ARE LINEARIZABLE ∀ VARIABLES

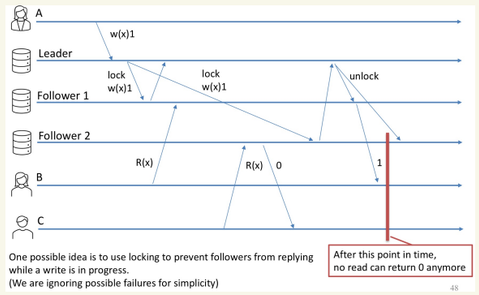
↓

GLOBAL SCHEDULE IS LINEARIZABLE

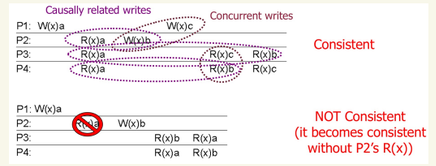
P1: W(x)a	P1: W(x)a
P2: R(x)a	P2: R(x)NIL R(x)a
Consistent	NOT Consistent (sequential but not linearizable)

WITH SINGLE LEADER →

- LEADER ORDERS WRITE ACCORDING TO THEIR TIMESTAMPS
- REPLICAS ARE UPDATED SYNCHRONOUSLY AND ATOMICALLY
- DIFFICULT TO GUARANTEE WITH FAILURES →



- **CAUSAL CONSISTENCY** → POTENTIALLY CAUSALLY RELATED WRITES MUST BE SEEN IN THE SAME ORDER ∀ PROCESSES
  CONCURRENT WRITES MAY BE SEEN IN ANY ORDER AT DIFFERENT NODES



↓

USES LAMPORT'S NOTION OF HAPPENED-BEFORE → CAUSALITY BETWEEN READS AND WRITES

↓

CAUSAL ORDER DEFINED AS :
- W BY P IS CAUSALLY ORDERED AFTER EVERY PREVIOUS OPERATION IN P → EVEN WITH DIFFERENT VARIABLES
- R BY P ON X IS CAUSALLY ORDERED AFTER A PREVIOUS W BY P ON X
- CAUSAL ORDER IS TRANSITIVE (· → · → ·)
- OPERATIONS THAT ARE NOT CAUSALLY ORDERED ARE CONSIDERED CONCURRENT

USING MULTI-LEADER IMPLEMENTATION ! - WRITES TIMESTAMPED WITH VECTOR CLOCKS (THEY STATE WHAT PROCESS KNEW WHEN IT WROTE)
- UPDATE IS APPLIED TO A REPLICA ONLY WHEN ALL W THAT ARE POSSIBLE CAUSES OF THE UPDATE HAVE BEEN RECEIVED AND APPLIED

↓

HIGHLY AVAILABLE
ASSUMING STICKY CLIENTS

- **FIFO CONSISTENCY** → W DONE BY P ARE SEEN BY ALL OTHER PROCESSES IN THE ORDER IN WHICH THEY WERE ISSUED

P1: W(x)a						
P2:	R(x)a	W(x)b	W(x)c			
P3:				R(x)b	R(x)a	R(x)c
P4:				R(x)a	R(x)b	R(x)c

Consistent

P1: W(x)a						
P2:	R(x)a	W(x)b	W(x)c			
P3:				R(x)b	R(x)a	R(x)c
P4:				R(x)c	R(x)b	R(x)a

Not Consistent

↓

EZ TO IMPLEMENT EVEN WITH MULTI-LEADER

UPDATES FROM P CARRY A SEQUENCE NUMBER → REPLICA UPDATES FROM P ONLY IF IT HAS RECEIVED ALL THE PREVIOUS WITH LOWER SEQUENCE NUMBER

↓

STILL REQUIRES ALL W TO BE VISIBLE TO ALL PROCESSES

NOT ALL WRITES NEEDS TO BE SEEN BY ALL PROCESSES

- **SUMMARY**

Consistency	Description
Linearizable	All processes must see all shared accesses in the same order. Operations behave as if they took place at some point in (wall-clock) time.
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time.
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order.

1. Linearizable Consistency
What it means: All processes (computers, servers, etc.) see every read and write operation happening in the same order and at a specific moment in time.
Key idea: It's as if every operation happened at an exact point on a global clock.
Example: Imagine a shared document. If one person types "Hello" and another types "World," everyone will see "Hello World" in that exact order, no matter what. Even if two people are far apart, they'll see the exact same sequence of changes.

2. Sequential Consistency
What it means: All processes see every read and write operation in the same order, but the order doesn't need to match actual time.
Key idea: Operations appear in a single, consistent order across the system, but it doesn't have to line up with real time.
Example: For that same shared document, you might see "Hello" before "World" even if "World" was actually typed first. Everyone will still see the final document as "Hello World," just in the same, agreed-upon order.

3. Causal Consistency
What it means: Only operations that are related (causally linked) must be seen in the same order by all processes. Independent operations don't need to follow a specific order.
Key idea: Actions that logically depend on each other are seen in order; others may not be.
Example: If you like a post and then comment on it, all users will see your like before your comment. But if another user comments at the same time, those two comments may appear in any order.

4. FIFO (First-In, First-Out) Consistency
What it means: Each process sees the operations from any other process in the order they were sent. However, writes from different processes might be seen in different orders.
Key idea: Each person's changes are seen in the order they made them, but the order across people isn't guaranteed.
Example: If you send "Hello" and then "World," everyone will see those two in that order from you. But if someone else also sends "Goodbye," users might see "Hello Goodbye World" or "Goodbye Hello World" depending on the timing.

Quick Summary
Linearizable: Consistent across time. All operations are seen at the same "real" time by everyone.
Sequential: Consistent order, but doesn't reflect real time. Everyone agrees on an order.
Causal: Dependent operations are in the same order, unrelated ones don't matter.
FIFO: Each person's actions are in their order, but the overall order between people can vary.

- **EVENTUAL CONSISTENCY** → WE HAVE NO SIMULTANEOUS UPDATES AND MOSTLY READS (LIKE DNS)

↓ UPDATES ARE GUARANTEED TO EVENTUALLY PROPAGATE TO ALL REPLICAS → EZ TO IMPLEMENT
FEW CONFLICTS IN PRACTICE
DEDICATED DATA-TYPES (CONFLICT FREE REPLICATED DATA-TYPES)

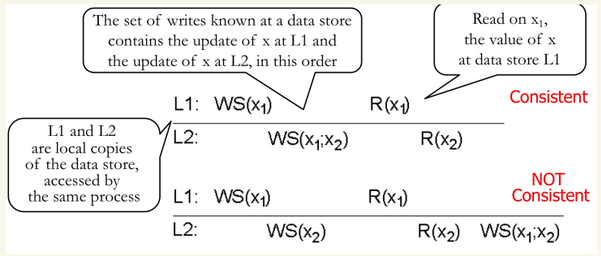
TRADEOFF BETWEEN COMPLEXITY AND PERFORMANCE

CRDTs → GUARANTEE CONVERGENCE EVEN IF UPDATES RECEIVED IN DIFFERENT ORDER

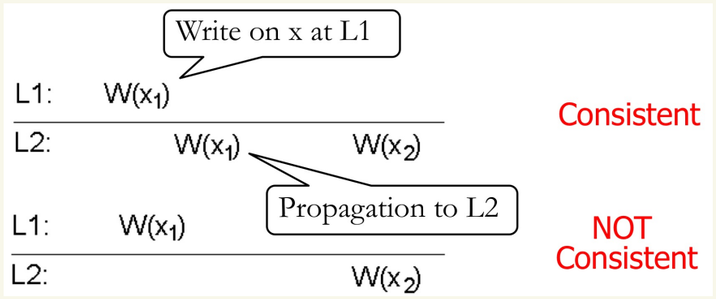
CLIENT-CENTRIC CONSISTENCY MODELS → CLIENT CHANGES REPLICA IT CONNECTS TO DYNAMICALLY

PROVIDE GUARANTEES ABOUT ACCESSES TO DATA STORE FROM A SINGLE CLIENT PERSPECTIVE

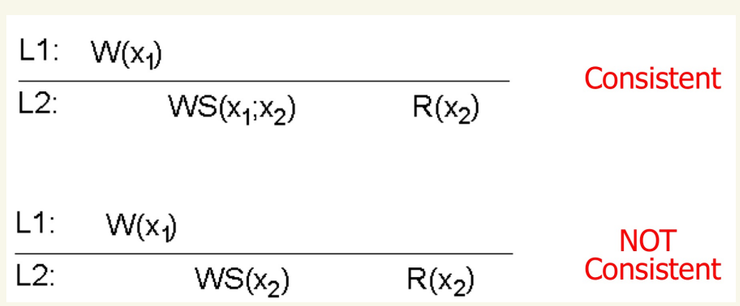
MONOTONIC READS → IF P READS X, ANY SUCCESSIVE R(x) BY P WILL ALWAYS THE SAME VALUE OR A MORE RECENT ONE



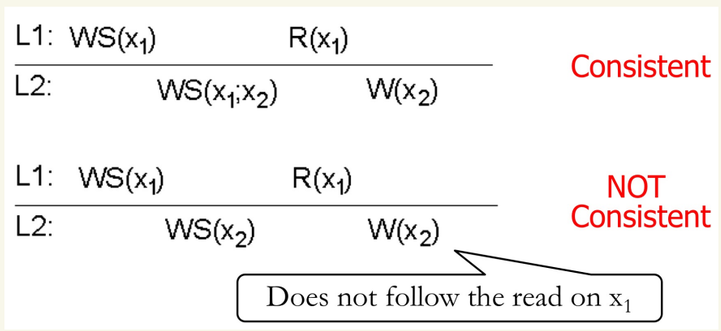
MONOTONIC WRITES → W(x) IN P IS COMPLETED BEFORE ANY OTHER SUCCESSIVE W(x) IN P



READ YOUR WRITES → EFFECT OF A W(x) OF P WILL ALWAYS BE SEEN BY A SUCCESSIVE R(x) BY P



WRITES FOLLOW READS → W(x) BY P FOLLOWING A PREVIOUS R(x) ON P IS GUARANTEED TO TAKE PLACE ON THE SAME OR MORE RECENT VALUE OF X THAT WAS READ



CLIENT-CENTRIC CONSISTENCY IMPLEMENTATION

- EACH R/W GETS A IDENTIFIER → CAN BE ENCODED BY VECTOR CLOCKS
- 2 SETS DEFINE FOR EACH CLIENT
  - READ-SET → RELEVANT WRITE IDENTIFIERS FOR THE READ OPERATIONS PERFORMED BY CLIENT
  - WRITE-SET → WRITE IDENTIFIERS FOR W PERFORMED BY CLIENT
- MONOTONIC READS → BEFORE READING CLIENT CHECKS THAT ALL W IN THE READ-SET HAVE BEEN DONE
- MONOTONIC WRITES → AS ABOVE BUT WITH THE WRITE-SET
- READ-YOUR-WRITES → AS MONOTONIC WRITES
- WRITE-FOLLOWS-READS → SERVER STATE BROUGHT UP TO DATE BY READ-SET AND THEN W ADDED TO WRITE-SET

DESIGN STRATEGIES → HOW TO REPLACE REPLICAS? WHAT TO PROPAGATE? HOW TO PROPAGATE UPATES BETWEEN THEM?

- REPLICA REPLACEMENT
  - PERMANENT REPLICAS
  - SERVER-INITIATED REPLICAS
  - CLIENT-INITIATED REPLICAS
- UPDATE PROPAGATION
  - PROPAGATE ONLY NOTIFICATION
  - TRANSFER DATA TO ALL COPIES
  - PROPAGATE INFO TO ENABLE THE UPDATE TO OCCUR IN OTHER COPIES (ACTIVE REPLICATION)
- HOW TO PROPAGATE →

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Comparison assuming one server and multiple clients, each with its own cache

- PROPAGATION STRATEGIES
  - LEADER-BASED PROTOCOLS — PROPAGATION CAN BE SYNCH, ASYNCH OR SEMI-SYNCH
  - LEADERLESS PROTOCOLS
    - READ REPAIR
    - ANTI-ENTROPY

CASE STUDIES

SEE HOW ATOMICITY, CONCURRENCY CONTROL AND REPLICATION ARE CONSIDERED IN MODERN DISTRIBUTED DATABASES

- Broadly speaking, modern systems can be classified based on the decision they take with respect to the CAP theorem
- As network partitions may always occur, systems may choose to offer either
  - Strong guarantees (C) through blocking communication
    - Higher latency, potentially not available in the presence of failures
  - High availability (A) through non-blocking communication
    - Lower latency, lower guarantees
- Various trade-offs have become popular over time

NoSQL (2000s)
- Key-value stores, wide-columns, document stores
  - Operations to access/modify individual items
- Isolation: no need, no multi-item transactions
- Atomicity: no need, no multi-item transactions
- Replication: asynchronous and/or multi-leader

Strong guarantees

High availability

NewSQL (2010s)
- Relational
- Isolation: serializable (locking and timestamp)
- Atomicity: (blocking) commit protocols
- Replication: sequential consistency / linearizable
- New techniques to limit costs
  - Case studies in the next slides

Case study: Spanner

- Designed for very large data bases
  - Many partitions, each partition is replicated
- Standard techniques
  - Single leader replication with Paxos for fault-tolerant agreement on followers and leader
  - 2PC for atomic commit
  - Timestamp protocols for concurrency control
- Novelty: TrueTime
  - Very precise clocks (atomic clocks + GPS)
  - Offer an API that returns an uncertainty range
  - The "real" time is certainly within the range
- Read-write transactions use TrueTime to decide when to commit
  - The transaction coordinator asks a transaction timestamp to TrueTime
  - It waits to release all locks and commit only when the uncertainty range is certainly passed …
  - … the commit timestamp is certainly passed for every node
  - Transactions are ordered based on time: linearizable!
- Read-only transactions also acquire a timestamp through TrueTime
  - They need not lock, but simply read the latest value at that time
  - Significant optimization when read-only operations are frequent

Case study: Calvin

- Designed for the same settings as Spanner
- Adopts a sequencing layer to order all incoming requests (read and write)
  - Replicated for durability
  - Essentially, a replicated log implemented using Paxos
- Operations are required to be deterministic …
- … and they are executed everywhere in the same order
- Guarantees: linearizability provided by the sequencing layer
- Advantage
  - Agreement (order of execution) achieved before acquiring locks: lower lock contention
  - No need for 2PC: as transactions are deterministic, they either succeed or fail in all replicas
    - One message from partitions that may lead to an abort is sufficient

Case study: VoltDB

- Developers specify how to partition database tables and transactions
  - E.g., hotel and flights tables both partitioned by city
- Single-partition transactions may execute sequentially on that partition without coordinating with other partitions
  - Standard protocols for other transactions

BIG DATA / DISTRIBUTED PLATFORM FOR DATA ANALYTICS

↳ DATA SCIENCE USES SCIENTIFIC METHODS, PROCESSES, ALGORITHMS AND SYSTEMS TO EXTRACT KNOWLEDGE AND INSIGHTS FROM DATA

↳ MADE POSSIBLE BY BIG DATA ↳ COLLECT THE DATA AND DECIDE WHAT TO DO WITH IT → CONSEQUENCES → BIG VOLUME, NEED VELOCITY, VARIETY, VERACITY

MAP REDUCE → WE HAVE MAP REPRESENTED AS A GRAPH AND WE WANT TO COMPUTE THE SHORTEST PATH FOR EACH POINT TO OTHER POINTS OF INTEREST

→ WE CAN DISCARD TOO FAR AWAY NODES

→ NEED TO CONSIDER THE SCALE OF THE PROBLEM AND THE COMPUTING INFRASTRUCTURE WE HAVE (ASSUMING BIG SCALE AND "NORMAL" HARDWARE)

→ WE COULD SPLIT THE DATASET INTO BLOCKS

→ WE CAN REPARTITION THE DATA BY NODE AND COMPUTE SHORTEST PATH FOR ALL NODES

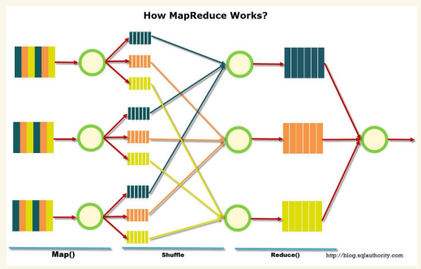
↓

SOLUTION (MAPREDUCE)

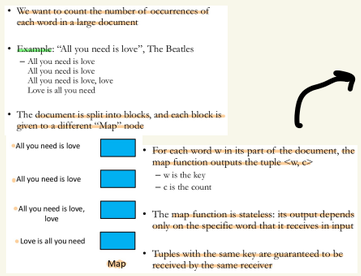
↳ COMPUTATION SPLIT INTO 2 PHASES

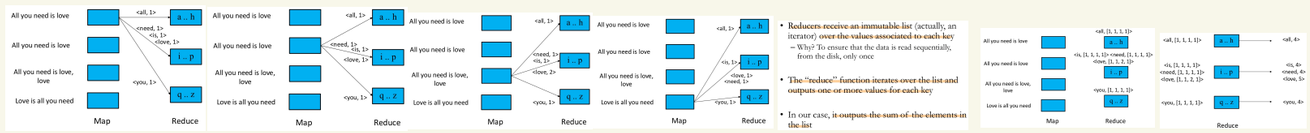
- MAP → PROCESSES INDIVIDUAL ELEMENTS GIVING ONE OR MORE <KEY, VALUE> PAIRS
- REDUCE → PROCESSES ALL THE VALUES WITH THE SAME KEY AND OUTPUT VALUE

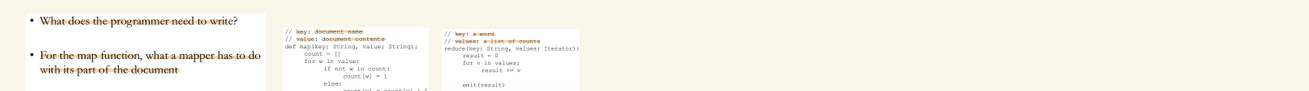
} DEVS ONLY SPECIFY THIS TWO FUNCTIONS



WORD COUNT EXAMPLE







- Reducers receive an immutable list (actually, an iterator) over the values associated to each key
  - Why? To ensure that the data is read sequentially, from the disk, only once
- The "reduce" function iterates over the list and outputs one or more values for each key
- In our case, it outputs the sum of the elements in the list

- What does the programmer need to write?
- For the map function, what a mapper has to do with its part of the document
- For the reduce function, how to process the list of values associated to a given key

```
// key: document-name
// value: document-contents
def map(key: String, value: String):
    count = {}
    for w in value:
        if not w in count:
            count[w] = 1
        else:
            count[w] = count[w] + 1

    for w in count:
        emit(w, count[w])
```

```
// key: a-word
// value: a-list-of-counts
reduce(key: String, values: Iterator):
    result = 0
    for v in values:
        result += v

    emit(result)
```

↓

FOR MAPREDUCE THE PLATFORM PERFORMS

- SCHEDULING: HAS A MASTER AND WORKERS, MASTER ASSIGNS MAP/REDUCE TASKS TO FREE WORKERS
- DATA DISTRIBUTION. WANTS TO LIMIT THE USAGE OF NETWORK BANDWIDTH → MASTER SCHEDULES MAP TASKS BASED ON THE LOCATION OF THESE REPLICAS
- FAULT TOLERANCE: → MASTER FAILURE → RECOVER FROM CHECKPOINT
  → WORKER FAILURE → MASTER KNOWS RE-EXECUTE THE TASK THAT IT WAS DOING

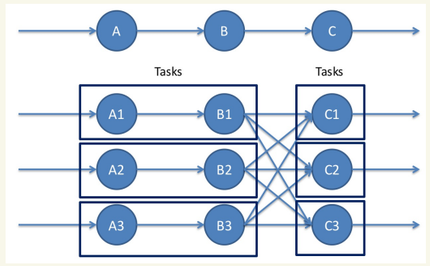
⇓

STRAGGLERS → TASKS THAT TAKE A LONG TIME

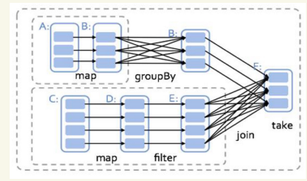
MAPREDUCE

- STRENGTHS → EZ FOR DEVS, GOOD FOR LARGE SCALE, VERY GENERAL
- LIMITATIONS → HIGH OVERHEAD, VERY FIXED PARADIGM

DATA FLOW PLATFORMS → DIFFERENT FROM MAPREDUCE → FROM 2 STEPS TO ARBITRARY ACYCLIC GRAPH OF OPERATIONS
→ FROM BATCH TO STREAM PROCESSING
→ FROM DISK TO MAIN-MEMORY OR HYBRID APPROACHES



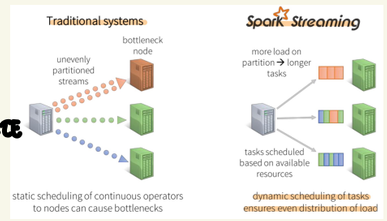
→ WE SEE 2 NEW SYSTEMS
- APACHE SPARK → SCHEDULING TASKS AND BATCH PROCESSING
- APACHE FLINK → PIPELINE TASKS AND CONTINUOUS PROCESSING



- APACHE SPARK → SIMILAR TO MAPREDUCE BUT WITH ARBITRARY NUMBER OF STAGES
  INTERMEDIATE RESULTS CAN BE CACHED IN MAIN MEMORY IF THEY'RE GOING TO BE REUSED
  SCHEDULING ENSURES COMPUTATION IS CLOSE TO THE DATA
  SUPPORT FOR STREAMING DATA THROUGH MICRO-BATCH APPROACH → INPUT STREAM SPLIT INTO SMALL BATCHES OF DATA

- APACHE FLINK → JOB ISN'T SPLIT INTO SCHEDULED STAGES → INSTEAD USE A PIPELINE ARCHITECTURE → SERIES OF OPERATORS WORK CONCURRENTLY
  ALL OPERATORS INSTANCIATED AS SOON AS THE JOB IS SUBMITTED
  IDEAL FOR STREAM PROCESSING → ALSO STREAM OF THE BATCH PROCESSES THROUGH THE OPERATORS

COMPARISON
- LATENCY → FLINK PROVIDES LOWER LATENCY THANKS TO STREAMING
- THROUGHPUT → SPARK BETTER BECAUSE A SCHEDULING APPROACH OFFERS MORE OPPORTUNITIES TO OPTIMIZE THROUGHPUT
- LOAD BALANCING → SPARK BETTER BECAUSE OF SCHEDULING → DYNAMIC SCHEDULING DECISIONS CAN CONSIDER DATA DISTRIBUTION (NOT POSSIBLE WITH STREAMING)



DYNAMICALLY ADAPT RESOURCE USAGE TO LOAD THAT CAN CHANGE OVER TIME

- ELASTICITY → SPARK BETTER BECAUSE SCHEDULING DECISIONS TAKE PLACE DYNAMICALLY AT RUNTIME
  ELASTICITY ISN'T POSSIBLE IN PIPELINED APPROACHES

- FAULT-TOLERANCE → SPARK RELIES ON RE-SCHEDULING LOST TASKS
  FLINK RELIES ON CHECKPOINTING AND RESUMING COMPUTATION FROM CHECKPOINT

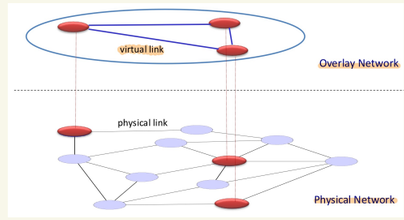
PEER - TO - PEER → TAKE ADVANTAGE OF RESOURCES AT THE EDGES OF THE NETWORK

↳ PROMOTES SHARING OF RESOURCES AND SERVICES THROUGH DIRECT EXCHANGE BETWEEN PEERS

↓

CHARACTERISTICS :
- ALL NODES ARE POTENTIAL USERS/PROVIDERS OF A SERVICE
- NO CENTRAL ADMINISTRATION → NODES ARE INDEPENDENT
- NODES CAN COME AND GO DYNAMICALLY FROM NETWORK
- CAN HAVE A INTERNET-WIDE SCALE

⇒



FUNDAMENTAL ISSUE IN P2P IS RETRIEVING RESOURCES DUE TO THE GEOGRAPHICAL DISTRIBUTION OF PEERS

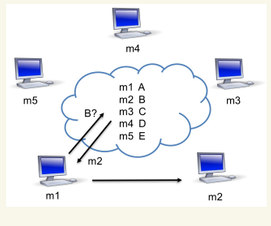
↳ 2 RETRIEVAL OPERATIONS CAN BE DONE
- SEARCH FOR SOMETHING → LIKE LOCATE ALL DOCUMENTS
- LOOKUP FOR A SPECIFIC ITEM

↓

CAN RETRIEVE DATA ITSELF OR A POINTER TO IT

NAPSTER → SHARE STORE AND BANDWIDTH OF INDIVIDUAL USERS

CENTRALIZED SEARCH
- JOIN → CLIENTS CONTACT CENTRAL SERVER
- PUBLISH → SUBMIT LIST OF FILES TO CENTRAL SERVER
- SEARCH → QUERY SERVER TO FIND WHO HAS THE WANTED FILE
- FETCH → GET THE FILE FROM THE PEER DIRECTLY



→
- PROS: SIMPLE AND SEARCH OF IS O(1)
- CONS: SERVER MAINTAINS O(N) STATES
  SERVER DOES SEARCH
  SERVER SINGLE POINT OF FAILURE

QUERY FLOODING - GNUTELLA

↳ NO CENTRAL AUTHORITY
USES FLOODING AS A SEARCH ALGORITHM → EACH QUERY FORWARDED TO ALL NEIGHBORS → PROPAGATION LIMITED BY HOPS TO LIVE

↓

CHARACTERISTICS
- JOIN : CLIENT CONTACTS FEW OTHER NODES (NEIGHBORS) → FIRST CONTACTS A WELL-KNOWN "ARCHOR" NODE THAT PINGS OTHER NODES THAT PONG BACK, THEN DIRECT CONNECTION IS ESTABLISHED
- PUBLISH : NO NEED
- SEARCH : ASK NEIGHBOR WHO ASK NEIGHBORS AND SO ON, WHEN FOUND REPLY TO SENDER
- FETCH : GET FILE DIRECTLY FROM PEER

↓

PROS: FULLY DECENTRALIZED, SEARCH COST DISTRIBUTED

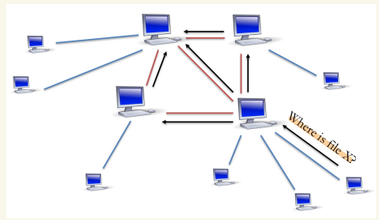
CONS: FLOOD OF REQUESTS, SEARCH SCOPE IS O(N) AND SEARCH TIME IS O(2D), NODES LEAVE OFTEN, NETWORK UNSTABLE

HIERARCHICAL TOPOLOGY - KAZAA

↳ WE HAVE NODES AND SUPERNODES → QUERY FLOODING THROUGH SUPERNODES AND NORMAL NODES LOOKUP TO SUPERNODES TO SEARCH

↓

CHARACTERISTICS
- JOIN : CLIENTS CONTACT A SUPERNODE → MAY BECOME SUPERNODE IN THE FUTURE
- PUBLISH : SEND LIST OF FILES TO SUPERNODE
- SEARCH : SEND QUERY TO SUPERNODE , SUPERNODES FLOOD QUERY TO OTHER SUPERNODES
- FETCH : GET THE FILE DIRECTLY FROM PEER → CAN FETCH SIMULTANEOUSLY FROM MULTIPLE PEERS



↓

PROS: CONSIDER NODES HETEROGENEITY AND NETWORK LOCALITY

CONS: NO REAL GUARANTEES ON SEARCH SCOPE OR SEARCH TIME

COLLABORATIVE SYSTEMS - BITTORRENT → NO FREE RIDERS (CLIENTS THAT JUST DOWNLOAD FROM NETWORK)

↳ ALLOWS MULTIPLE PEOPLE TO DOWNLOAD THE SAME FILE WITHOUT SLOWING DOWN EVERYONE ELSE'S DOWNLOAD

↓

DOWNLOADERS SWAP PORTIONS OF A FILE WITH ONE ANOTHER INSTEAD OF ALL DOWNLOADING FROM A SINGLE SERVER

↓

EVERY CLIENT TRYING TO UPLOAD TO OTHER CLIENTS GETS THE FASTEST DOWNLOAD

↓

CHARACTERISTICS
- JOIN → CONTACT A "TRACKER" SERVER AND GET A LIST OF PEERS
- PUBLISH → RUN A TRACKER SERVER
- SEARCH → OUT-OF-BAND (USE GOOGLE FOR EXAMPLE TO SEARCH A TRACKER SERVER FOR A SPECIFIC FILE)
- FETCH → DOWNLOAD CHUNKS OF THE FILE FROM PEERS WHILE UPLOADING YOUR CHUNK

BITTORRENT TERMINOLOGY:

- Torrent: a meta-data file describing the file(s) to be shared
  - Names of the file(s)
  - Size(s)
  - Checksum of all blocks (file is split in fixed-size blocks)
  - Address of the tracker
  - Address of peers
- Seed: a peer that has the complete file and still offers it for upload
- Leech: a peer that has incomplete download
- Swarm: all seeders/leeches together make a swarm
- Tracker: a server that keeps track of seeds and peers in the swarm and gathers statistics
  - When a new peer enters the network, it queries the tracker to obtain a list of peers

HOW DOES CONTENT GETS DISTRIBUTED?

- Breaks the file down into smaller fragments (usually 256KB in size)
  - The .torrent holds the SHA1 hash of each fragment to verify data integrity
- Peers contact the tracker to have a list of the peers
- Peers download missing fragments from each other and upload to those who don't have it
- The fragments are not downloaded in sequential order and need to be assembled by the receiving machine
  - When a client needs to choose which segment to request first, it usually adopts a "rarest-first" approach, by identifying the fragment held by the fewest of its peers
  - This tends to keep the number of sources for each segment as high as possible, spreading load
- Clients start uploading what they already have (small fragments) before the whole download is finished
- Once a peer finishes downloading the whole file, it should keep the upload running and become an additional seed in the network
- Everyone can eventually get the complete file as long as there is "one distributed copy" of the file in the network, even if there are no seeds

→ CHOKING: TEMPORAL REFUSAL TO UPLOAD

↓

PROS: WORKS WELL
INCENTIVE PEERS TO SHARE

CONS: PARETO EFFICIENCY IS A WEAK CONDITION
CENTRAL TRACKER SERVER NEEDED TO BOOTSTRAP SWARM

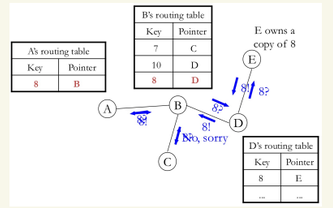
SECURE STORAGE – FREENET → ENSURE FREEDOM OF COMMUNICATION OVER INTERNET → ALLOWS ANYBODY TO PUBLISH AND READ INFO ANONIMOUSLY

↳ GOALS :
- ALLOW ONE-TO-MANY PUBLISHING OF INFO
- PROVIDE ANONIMITY
- RELY ON DECENTRALIZED NETWORK
- BE SCALABLE IN A BIG WAY
- ROBUST AGAINST FAILURE AND MALICIOUS ATTACK

→ CHARACTERISTICS
- JOIN : CLIENT CONTACTS FEW OTHER NODES AND GET UNIQUE ID
- PUBLISH : ROUTE FILE TOWARD NODES WHO STORE OTHER FILES WITH ID CLOSEST TO FILE ID
- SEARCH : QUERY FOR FILE ID USING STEEPEST-ASCENT HILL-CLIMBING SEARCH WITH BACKTRACKING
- FETCH : WHEN QUERY REACHES FILE ID SEARCHED IT RETURNS IT

ROUTING → NODES HAVE LOCAL INFO AND KNOW APPROXIMATELY WHAT THEIR NEIGHBOR STORE TOO
REQUEST FORWARDED TO "BEST GUESS" NEIGHBOR, IF INFO FOUND WITHIN THE HOPS TO LIVE IT IS PASSED BACK THROUGH THE NODE-CHAIN
INTERIDIATE NODES STORE THE INFO IN THEIR LRU (LEAST RECENTLY USED) CACHE

EXAMPLE →



PUBLISHING → SEARCH FOR THE DAT ID WE WANT TO INSERT, IF FOUND DON'T INSERT
EACH NODE ADDS ENTRY TO ROUTING TABLE ASSOCIATING THE KEY AND THE DATA SOURCE

ANONIMITY → MESSAGES ARE FORWARDED BACK AND FORTH, NODES CAN'T TELL WHERE A MESSAGE ORIGINATED

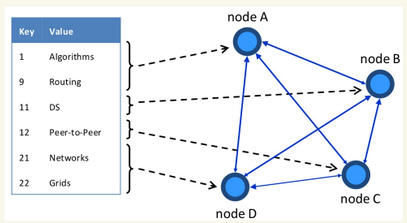
SECURITY AND CENSORSHIP RESISTANCE → FILE IDS SHOULDN'T COLLIDE → ROBUST HASHING SO THAT FILE IDS ARE RELATED TO THEIR CONTENT

↓

PROS : SMART ROUTING MAKE QUERIES FAST
SMALL SEARCH SCOPE
ANONYMITY MAY GIVE PLAUSIBLE DENIABILITY

CONS : NO GUARANTEES
ANONIMITY MAKES THINK HARDER

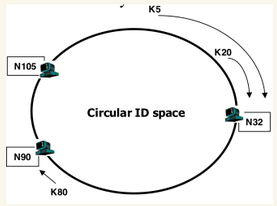
STRUCTURED TOPOLOGY – DISTRIBUTED HASH TABLES (DHTs)



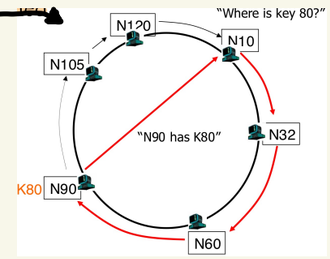
DISTRIBUTED HASH-TABLE DATA STRUCTURE :
- PUT (ID, ITEM)
- ITEM = GET (ID)

→ CHARACTERISTICS
- JOIN : CLIENTS CONTACT A BOOTSTRAP NODE AND INTEGRATE INTO THE DISTRIBUTED DATA STRUCTURE, GET A NODE ID
- PUBLISH : ROUTE PUBLICATION FOR FILE ID TOWARD A CLOSE NODE ID ALONG THE DATA STRUCTURE
- SEARCH : ROUTE A QUERY FOR FILE ID TOWARD A CLOSE NODE ID
- FETCH : PUBLICATION CONTAINS FILE → FETCH FROM WHERE QUERY STOPS
  PUBLICATION CONTAINS REFERENCE → FETCH FROM THE LOCATION INDICATED IN THE REFERENCE

CHORD → NODES AND KEYS ORGANIZED IN A LOGICAL RING
NODES HAVE UNIQUE m-bit ID (HASH OF IP USUALLY)
ITEMS HAVE UNIQUE m-bit KEY (HASH OF ITEM) → ITEM WITH KEY K MANAGED BY NODE WITH SMALLEST ID > K

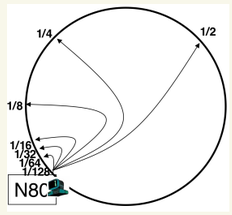


NODES HAVE A REFERENCE TO THEIR SUCCESSOR AND PREDECESSOR, SEARCH IS PERFORMED LINEARLY

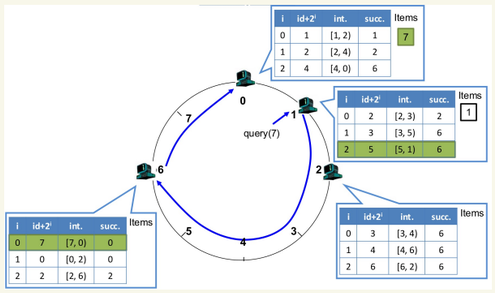


↓

EACH NODE MAINTAINS A FINGER TABLE WITH m ENTRIES → ENTRY i IN TABLE OF NODE h IS THE FIRST NODE WITH ID $\geq h+2^i$



ROUTING: WHEN A NODE RECEIVE A QUERY SEARCHING FOR AN ITEM K IT :
- CHECKS IF IT HAS IT LOCALLY
- IF NOT FORWARDS QUERY TO THE NODE IN TABLE THAT IS RESPONSIBLE FOR THE INTERVAL OF KEYS WHERE K IS



JOINING: WHEN NODE N JOINS :
- NEED TO INITIALIZE PREDECESSOR AND TABLE OF N → ASSUMING N ALREADY KNOWS N' INSIDE SYSTEM
  ↓
  USES N' TO INITIALIZE HIS TABLE
- UPDATE TABLES OF EXISTING NODES
  ↓
  NODE N WILL BECOME i-TH FINGER OF NODE P IFF
  - P PRECEDES N BY AT LEAST 2^{i-1} → NEED TO CHECK FOR EACH FINGER
  - i-TH FINGER OF P SUCCEEDS N
  ↓
  FIRST P THAT MEETS THOSE CONDITIONS IS THE IMMEDIATE PREDECESSOR OF NODE $N-2^{i-1}$

FINGER i = SUCCESSOR OF NODE $N+2^i$

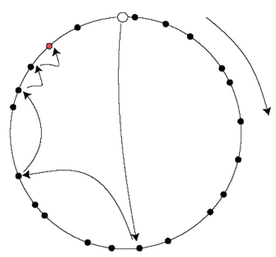
↳ TOT COST: $\log^2(N)$

STABILIZATION: TO STABILIZE ROUTING CHORD USES PERIODIC PROCEDURES TO UPDATE SUCCESSOR AND TABLES

FAILURE AND REPLICATION: EACH NODE HAS A LIST OF SUCCESSORS OF SIZE R, CONTAINING THE FIRST R SUCCESSORS

SUMMARY:

- ~~Routing table size?~~
  - Log_N fingers
  - With $N = 2^m$ nodes
- ~~Routing time?~~
  - Each hop expects to 1/2 the distance to the desired id => expect O(log N) hops
- ~~Joining time~~?
  - With high probability expect $O(\log^2 N)$ hops



→ PROS: GUARANTEED LOOKUP
$O(\log N)$ PER NODE STATE AND SEARCH SCOPE

CONS: UNUSED
MORE FRAGILE THAN UNSTRUCTURED NETWORKS
DOESN'T CONSIDER PHYSICAL TOPOLOGY