Theory

Table of Contents

1. Introduction

1.1 Define

```
Multiple computers
Common Task
Client sees a single service
```

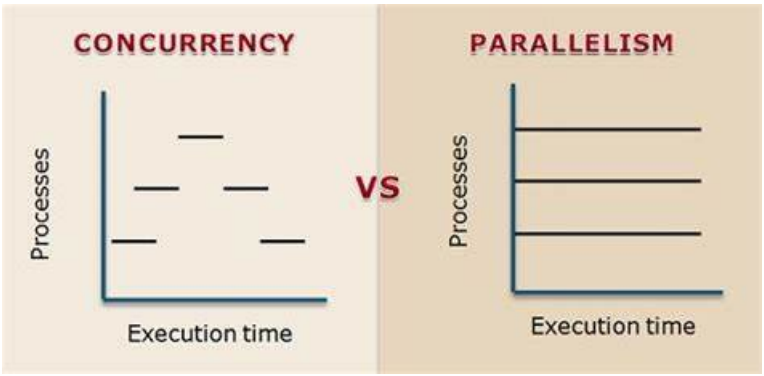
1.2 Why not DS?

```
Fault Tolerance is Hard ( a system as a whole continues to work, even when some parts are faulty)

- Non reliable communication
- Processes might crash
- Coordinated and uncoordinated indeterministic   failures
```

1.3 Why DS?

```
Parallel or Concurrent
Fault Tolerance
Physical Requirements
Isolation (Security)
Scalability
Resource Sharing
Price / Performance Ratio
Seamless Communication
Abstraction of Computation
```

2. Concurrency and Parallel Processing



	Single Core	Multi Core
Concurrent	☑	☑
Parallel	☒	☑

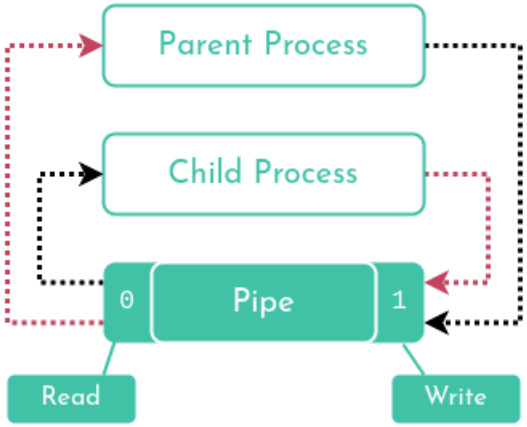
2.1 Thread, Process and Fork

```
- Threads (of the same process) run in a shared memory space
- Processes run in separate memory spaces.
- Each process is started with a single thread, often called the primary thread, but can create
additional threads from any of its threads.
- A thread is a subset of the process.
```

```
- A fork gives a copy of a process
- A fork has its own memory space (not shared)
```

2.2 Channels and Pipes

```
- Pipes are channels that connect processes for communication.
- They have a write end for sending bytes and a read end for receiving these bytes in FIFO
- Channels act like pipes between two processes or threads.
- One process puts data into the channel, and the other process retrieves it.
- Channels can be used for communication between concurrent threads within the same process
- They are simpler to use than pipes because they don’t involve file descriptors or system calls.
- Pipes are typically unidirectional. Data flows from the write end to the read end. To achieve full
duplex communication (both directions simultaneously), you’d need two pipes—one for each direction.
-  Channels can be bidirectional, allowing data to flow in both directions. Channels can handle
simultaneous communication in both directions within the same channel.
```



2.3 Locks

```
- A Lock can only give access to a single thread
- A mutually exclusive lock can give access to multiple threads
```

2.3.1 Example

```
Binary Lock
```

```c
// Binary Lock
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t lock;

void* critical_section(void* arg) {
    int thread_id = *((int*)arg);

    pthread_mutex_lock(&lock); // Acquire the binary lock
    printf("Thread %d: Entered critical section\n", thread_id);
    // Critical section code
    pthread_mutex_unlock(&lock); // Release the binary lock

    free(arg);
    return NULL;
```

```
}

int main() {
    pthread_t threads[5];
    int* thread_ids[5];

    pthread_mutex_init(&lock, NULL); // Initialize the binary lock

    // Create and start 5 threads
    for (int i = 0; i < 5; i++) {
        thread_ids[i] = malloc(sizeof(int));
        *thread_ids[i] = i;
        pthread_create(&threads[i], NULL, critical_section, thread_ids[i]);
    }

    // Wait for all threads to finish
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&lock); // Destroy the binary lock

    return 0;
}
```

Shared/Exclusive Lock

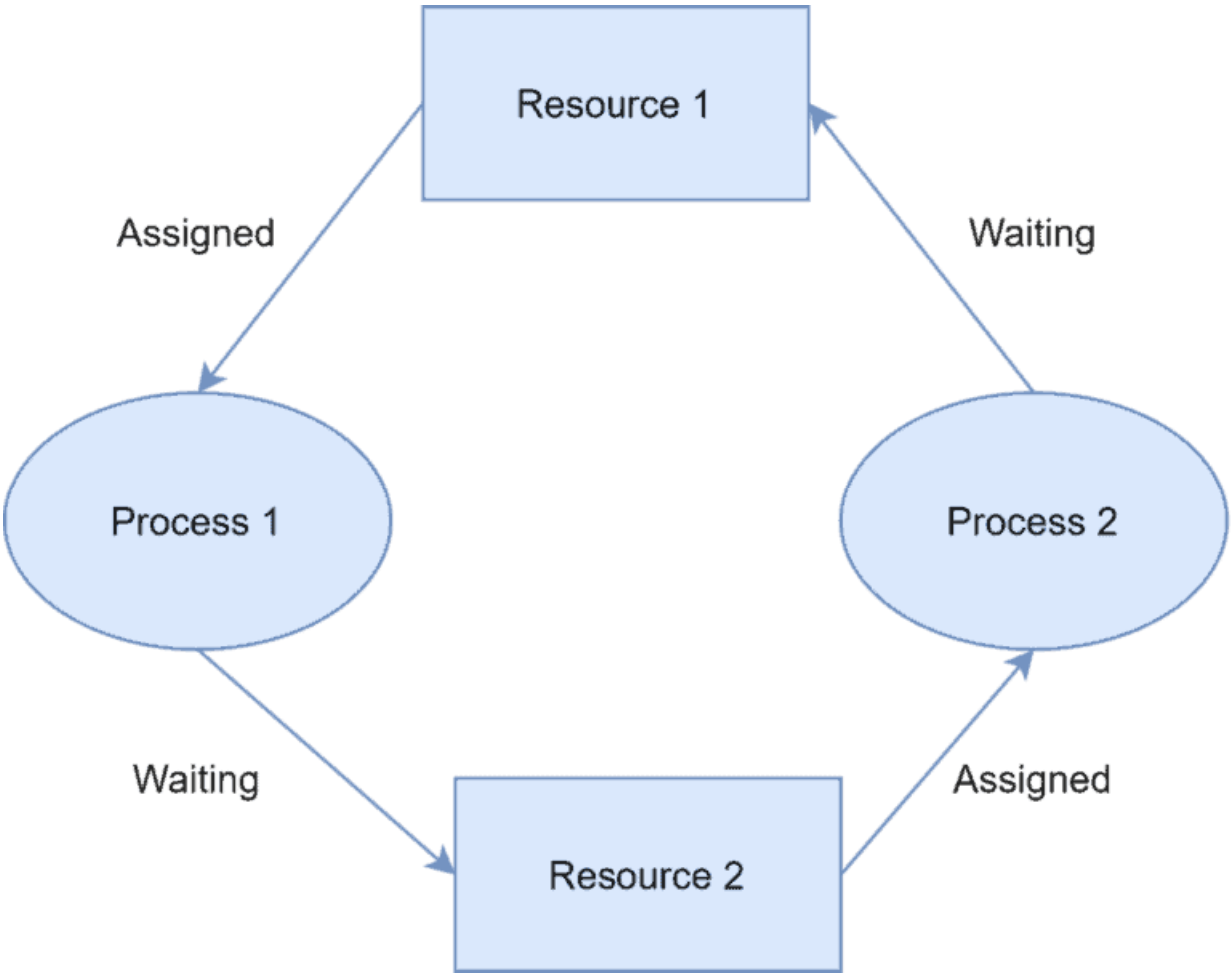
```
// Shared/Exclusive Lock
#include <stdio.h>
#include <pthread.h>

pthread_rwlock_t rwlock;

void* read_shared(void* arg) {
    pthread_rwlock_rdlock(&rwlock); // Acquire a shared read lock
    printf("Thread %ld: Reading shared resource\n", pthread_self());
    // Read from shared resource
    pthread_rwlock_unlock(&rwlock); // Release the shared read lock
    return NULL;
}

void* write_exclusive(void* arg) {
    pthread_rwlock_wrlock(&rwlock); // Acquire an exclusive write lock
    printf("Thread %ld: Writing to shared resource\n", pthread_self());
    // Write to shared resource
    pthread_rwlock_unlock(&rwlock); // Release the exclusive write lock
    return NULL;
}

int main() {
    pthread_t threads[5];

    pthread_rwlock_init(&rwlock, NULL); // Initialize the shared-exclusive lock

    // Create threads for shared read access
    pthread_create(&threads[0], NULL, read_shared, NULL);
    pthread_create(&threads[1], NULL, read_shared, NULL);
    pthread_create(&threads[2], NULL, read_shared, NULL);

    // Create threads for exclusive write access
    pthread_create(&threads[3], NULL, write_exclusive, NULL);
    pthread_create(&threads[4], NULL, write_exclusive, NULL);
```

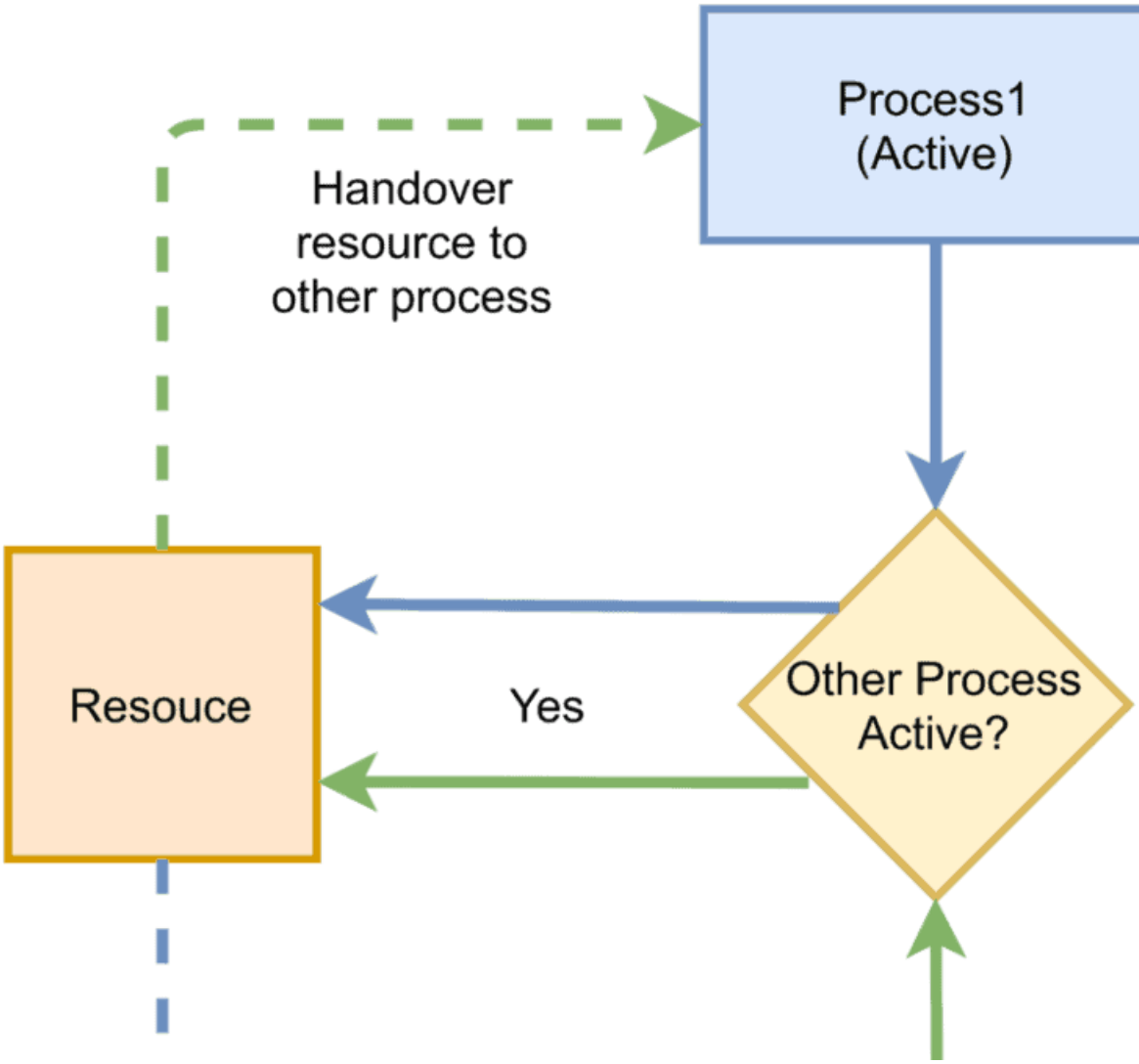
```
    // Wait for all threads to finish
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_rwlock_destroy(&rwlock); // Destroy the shared-exclusive lock

    return 0;
}
```

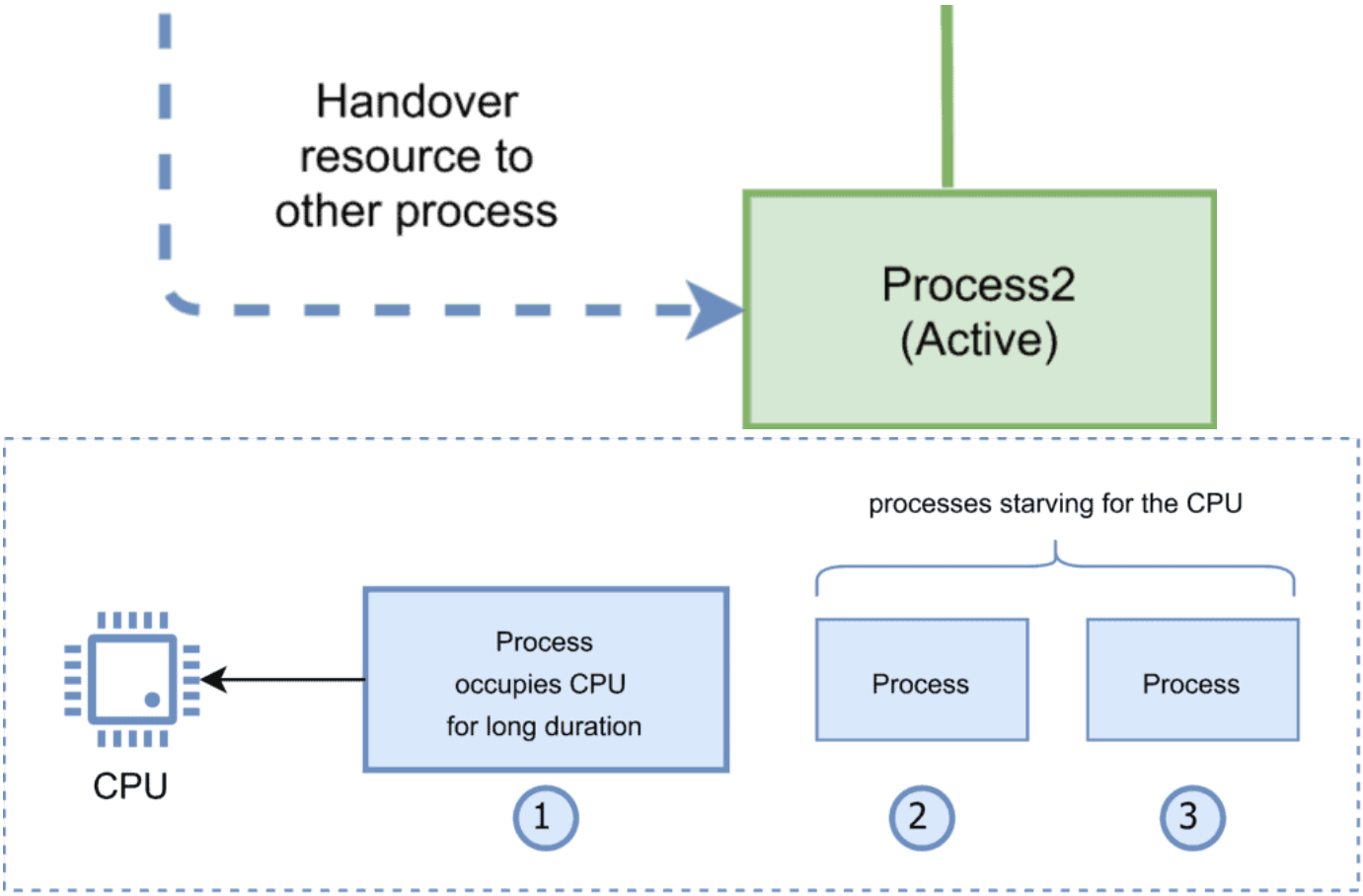
2.4 Deadlocks, Livelocks and Starvation

A deadlock is a state in which each member of a group of actions, is waiting for some other member to release a lock.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.


Resource 1
Assigned
Waiting
Process 1
Process 2
Waiting
Assigned
Resource 2


Process1
(Active)
Handover
resource to
other process
Resouce
Yes
Other Process
Active?



2.5 Race Conditions

```
A race condition occurs when two or more threads can access shared data and attempt to change it
simultaneously.
Can prevent using locks to ensure only one thread accesses the shared data at a time.
e.g.: Lower value than Expected (MapReduce)
```

2.6 Blind Writes

```
Blind writes occur when threads write to shared memory without proper synchronization.
Threads may overwrite each other’s changes, leading to incorrect results.
The final value of shared variable depends on which thread executes last.
Can prevent using properly syncing all threads.
e.g.: Higher value than Expected (MapReduce)
```

3. Communication and RPC

3.1 Latency and Bandwidth

```
- Latency: Time until message arrives
- Bandwidth: Data volume per unit time
```

3.2 Remote Procedure Calls (RPC)

Layer	Name	Protocols
Layer 7	Application	SMTP, HTTP, FTP, POP3, SNMP

Layer	Name	Protocols
Layer 6	Presentation	MPEG, ASCH, SSL, TLS
Layer 5	Session	NetBIOS, SAP
Layer 4	Transport	TCP, UDP
Layer 3	Network	IPV5, IPV6, ICMP, IPSEC, ARP, MPLS.
Layer 2	Data Link	RAPA, PPP, Frame Relay, ATM, Fiber Cable, etc.
Layer 1	Physical	RS232, 100BaseTX, ISDN, 11.

A communication protocol that enables a program to execute a subroutine or procedure on a remote system over a network.

3.2.1 Example

In this example, we have a Calculator type that represents a calculator instance. It has a single method Add that takes two integers and returns their sum.

In the main function, we create a new instance of Calculator, register it with the RPC server using rpc.Register(calculator), and start the RPC server on port 8000.

```go
// server.go
package main

import (
    "fmt"
    "net"
    "net/rpc"
)

type Calculator int

func (c *Calculator) Add(x, y int) (int, error) {
    return int(*c) + x + y, nil
}

func main() {
    calculator := new(Calculator)
    rpc.Register(calculator)

    listener, err := net.Listen("tcp", ":8000")
    if err != nil {
        fmt.Println("Failed to listen:", err)
        return
    }
    defer listener.Close()

    fmt.Println("Listening on port 8000...")
    rpc.Accept(listener)
}
```

In the client code, we create a new client connection to the RPC server using rpc.Dial("tcp", "localhost:8000").

We then call the Add method on the remote server using client.Call("Calculator.Add", []int{3, 5},

```
&result). The first argument is the name of the remote method, the second argument is the input arguments
(an array of integers in this case), and the third argument is a pointer to a variable where the result
will be stored.
```

```go
package main

import (
    "fmt"
    "net/rpc"
)

func main() {
    client, err := rpc.Dial("tcp", "localhost:8000")
    if err != nil {
        fmt.Println("Failed to connect:", err)
        return
    }
    defer client.Close()

    var result int
    err = client.Call("Calculator.Add", []int{3, 5}, &result)
    if err != nil {
        fmt.Println("Failed to call Add:", err)
    } else {
        fmt.Printf("3 + 5 = %d\n", result)
    }
}
```

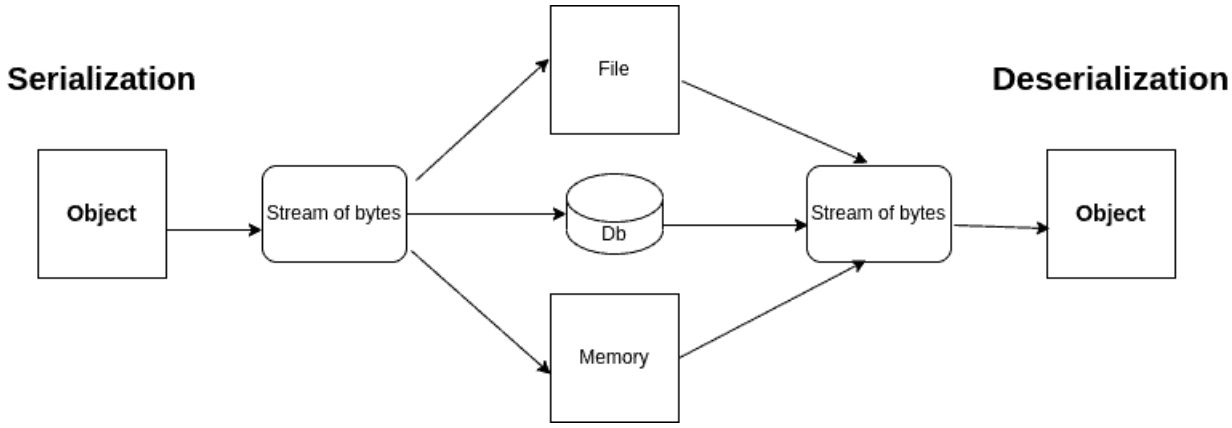
```
Output
```

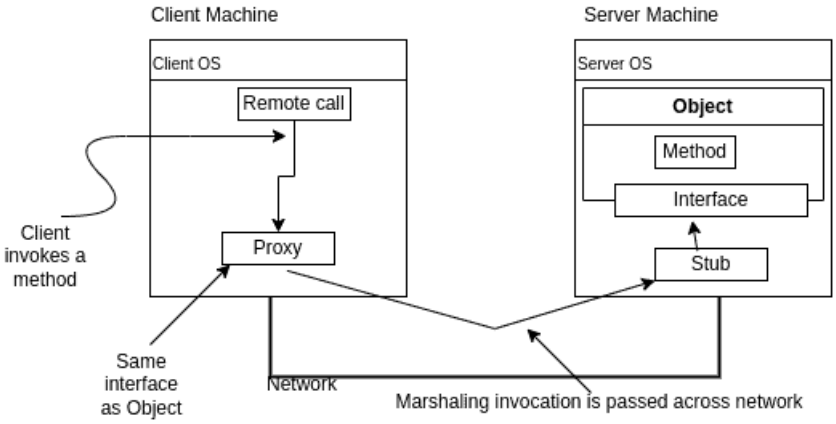
```
(Server terminal)
Listening on port 8000...

(Client terminal)
3 + 5 = 8
```

3.2.2 Invoking RPC

Protocol	Description
TCP (Transmission Control Protocol)	reliable, connection-oriented, guarantees the delivery of data packets in the correct order, provides error-checking mechanisms.
HTTP (Hypertext Transfer Protocol)	widely used protocol, useful when the client and server are separated by firewalls or proxy servers, often used in web services and RESTful APIs.
gRPC (Google Remote Procedure Call)	high-performance, uses HTTP/2, multiplexing, header compression, bidirectional streaming. efficient, scalable, and language-agnostic.
WebSocket	persistent, bidirectional communication channel between a client and a server over a single TCP connection, allowing real-time data exchange without the overhead of traditional HTTP requests.

3.2.3 Serialization and Marshalling





Serialization is persisting an object into a state independent of its execution environment. During serialization, the data is saved (in memory or physically) in a raw format, such as byte arrays or binary data. Deserialization is the reconstruction of the original object from the serialized data.

Marshaling is moving an object or method call into another execution part. It is more about the interoperability of objects between programs or threads. It can also involve serialization during its operation. Therefore, serialization is usually part of marshaling.

Serialization/Deserialization	Marshaling
Convert an object from and to a byte stream	Move objects from one thread or program to another Serialization can be used during this process
Apply to any context where serialization is required	Usually refers to remote procedure call or IPC
Store in memory or physically a copy of the original object	Pass by-value or by-reference a copy of the object
No code generation	Service implementation or template is generated

rpc example flow

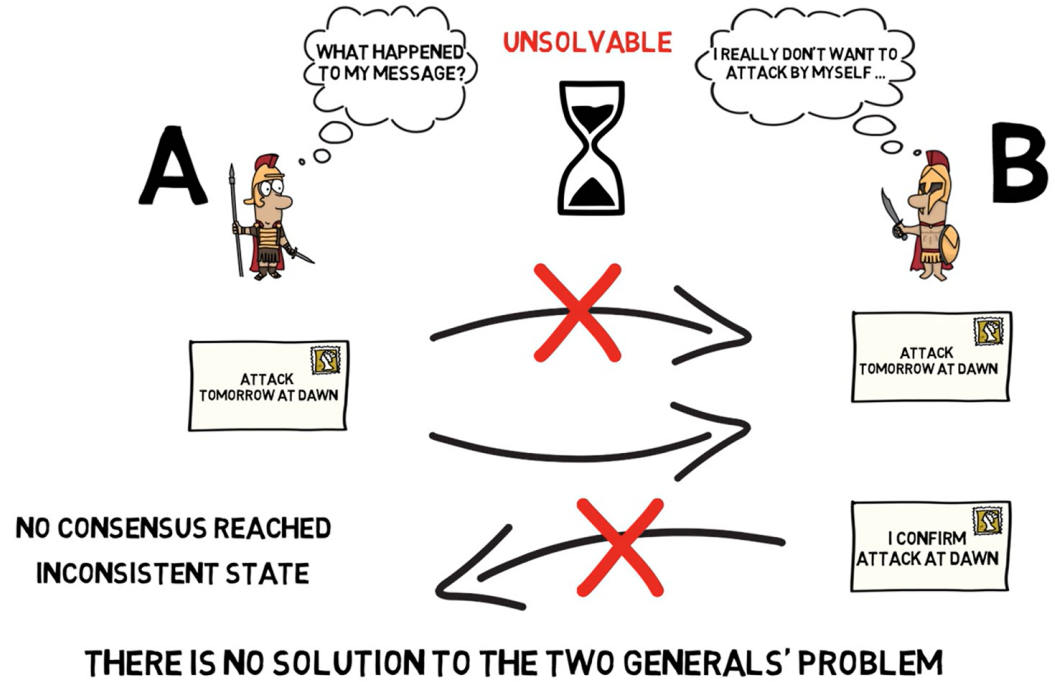
3.2.4 Protocol Buffer

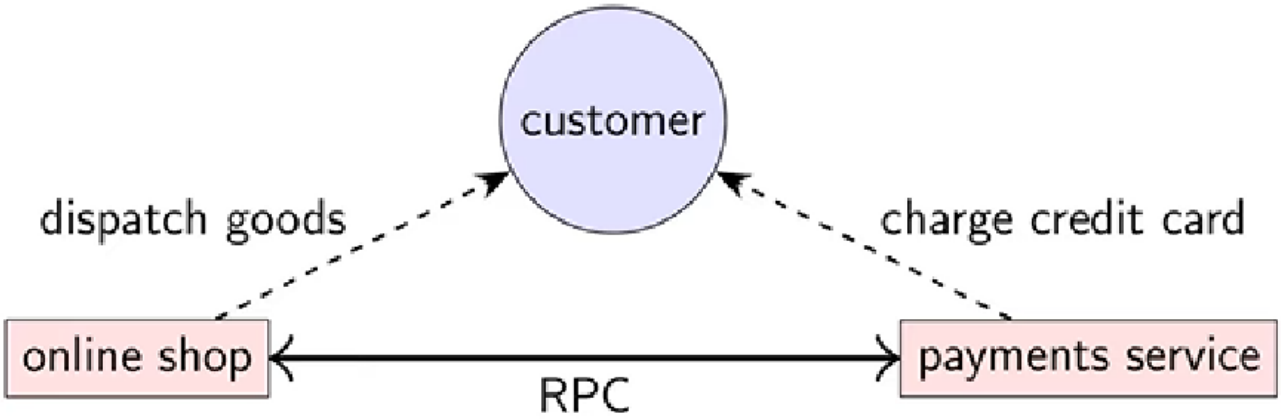
```
message PaymentRequest{
    message Card{
        required string cardNumber  =    1;
        optional int32 expiryMonth  =    2;
        optional int32 expiryYear   =    3;
        optional int32 CVC      =    4;
    }
}

message PaymentStatus{
    required bool success       =   1;
    optional string errorMessage        =   2;
}
service PaymentService{
```

```
    rpc ProcessPayment(PaymentRequest) return (PaymentStatus) {}
}
```

4. Models

4.1 Two Generals Problem

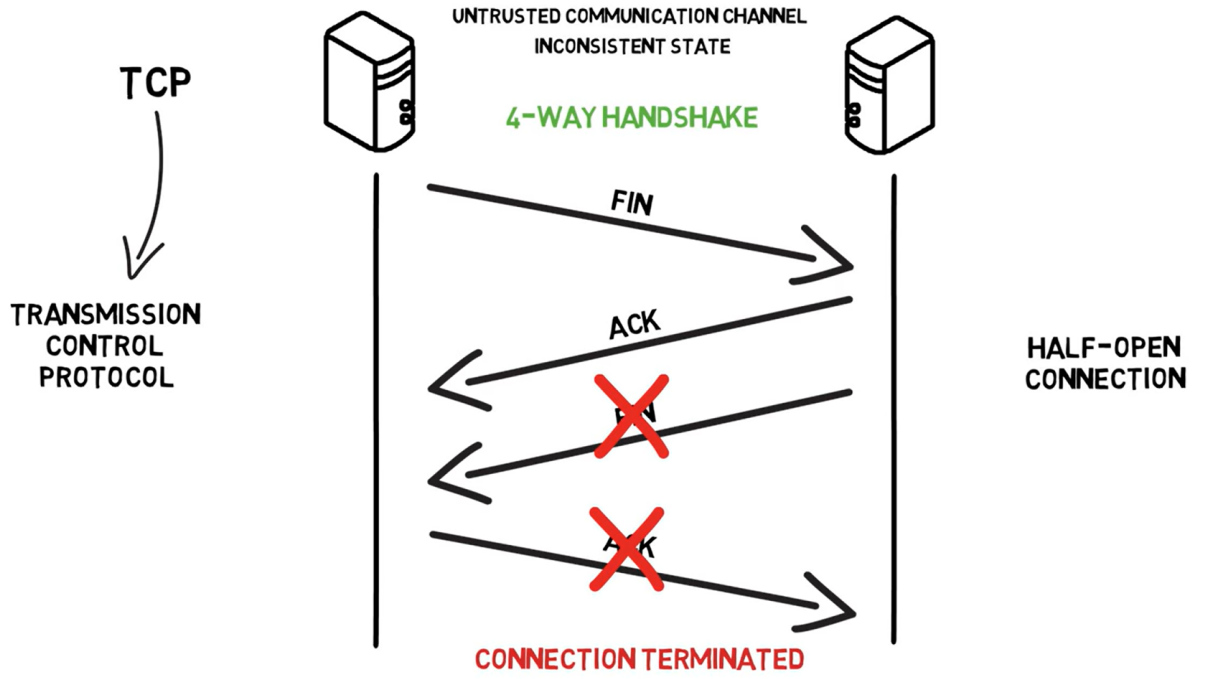


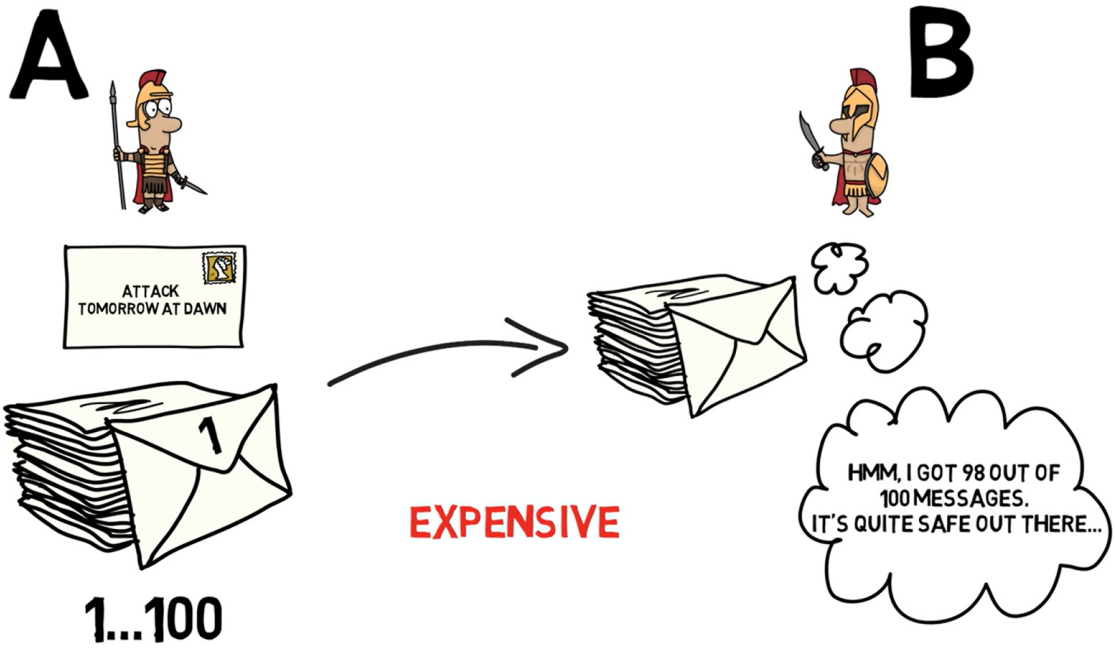


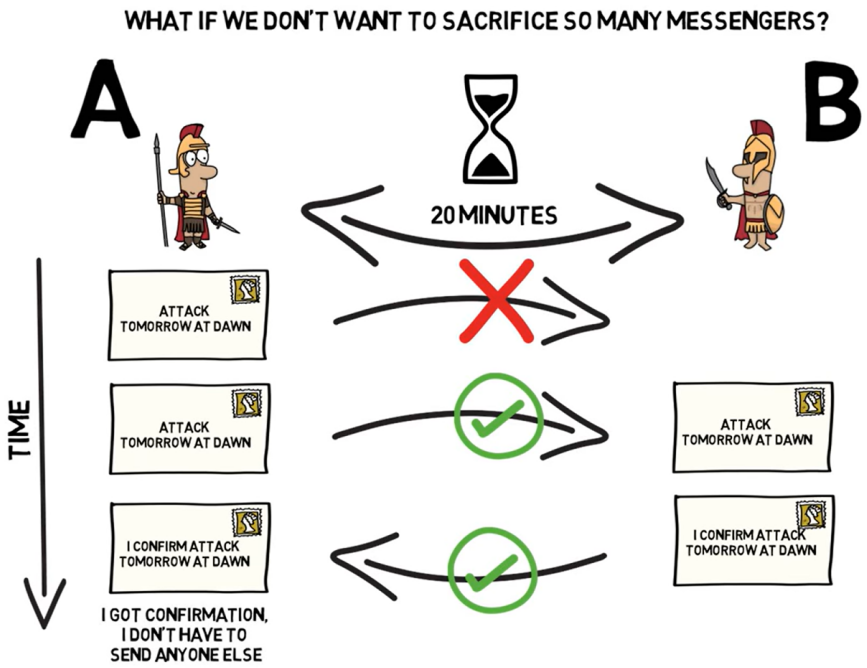
online shop	payments service	outcome
does not dispatch	does not charge	nothing happens
dispatches	does not charge	shop loses money
does not dispatch	charges	customer complaint
dispatches	charges	everyone happy

Desired: online shop dispatches *if and only if* payment made

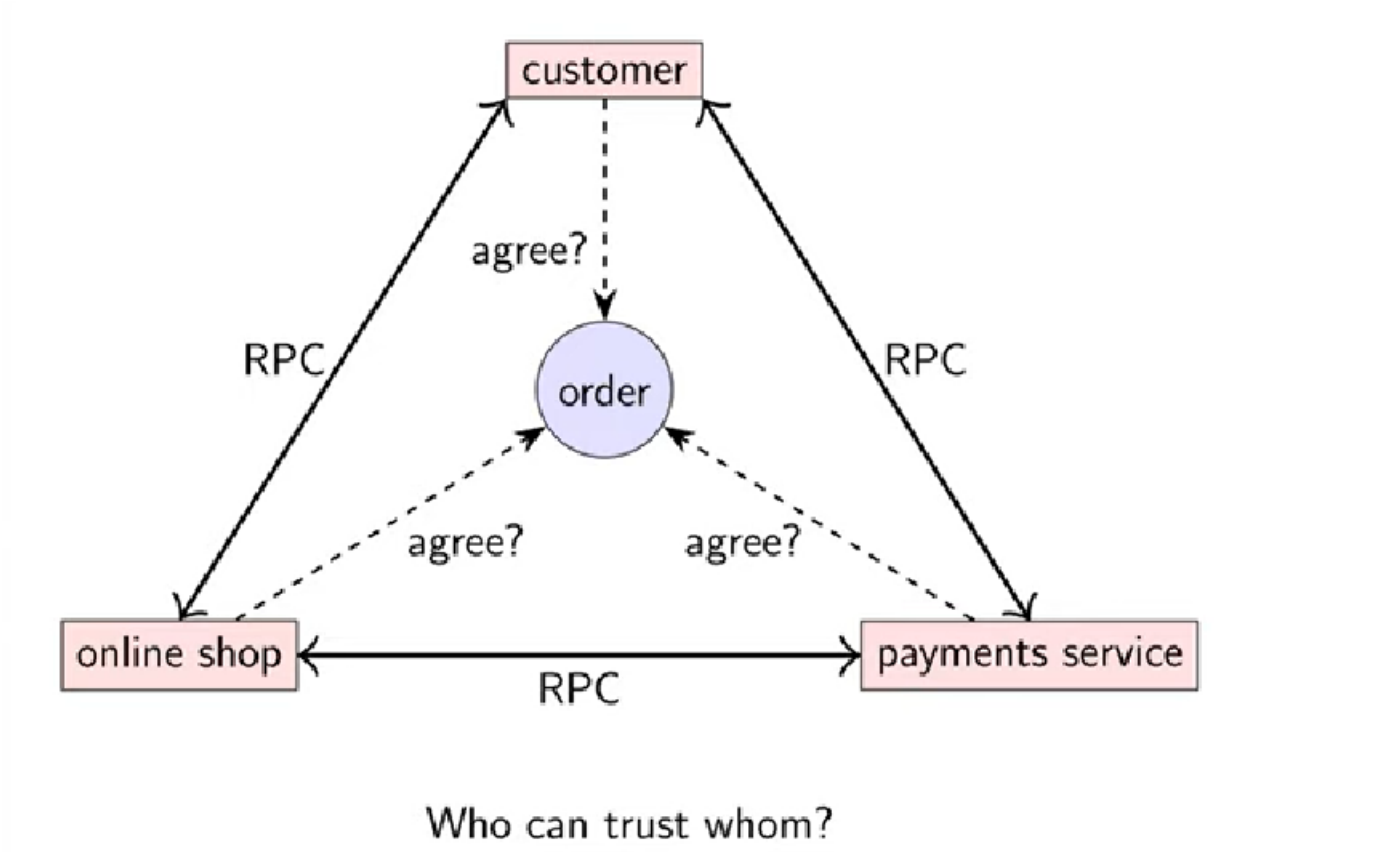
TCP is reliable, but doesn't solve two generals problem.


TCP
TRANSMISSION CONTROL PROTOCOL
UNTRUSTED COMMUNICATION CHANNEL
INCONSISTENT STATE
4-WAY HANDSHAKE
FIN
ACK
HALF-OPEN CONNECTION
CONNECTION TERMINATED


A
B
ATTACK TOMORROW AT DAWN
1
1...100
EXPENSIVE
HMM, I GOT 98 OUT OF 100 MESSAGES. IT'S QUITE SAFE OUT THERE...



4.2 Byzantine Generals Problem




```
- Up to f generals might behave maliciously
- Honest generals don’t know who the malicious ones are
- The malicious generals may collude
- Nevertheless, honest generals must agree on plan

need 3f+1 generals in total to tolerate f malicious generals (<1/3 may be malicious)
```

4.3 Systems Models

```
- Network behavior ( messages may be loss)
- Node behavior (crashes / faults )
- Time behavior  (latency)
```

4.4 Network Behaviour

```
- Reliable (perfect) links: Message is received if and only id it is sent
- Fair-loss links: Message may or be lost, duplicated, or reordered. if kept retrying message eventually
gets through
- Arbitrary links: A malicious adversity may interfere with messages (eavesdrop, modify, drop, spoof,
reply)

- Network partitioning : some links dropping/ delaying all messages for extended period of time
```

4.5 Node Behaviour

```
- Crash-Stop: A node is faulty if it crashes (at any time). After crashing, it stops execution (forever)
- Crash-Recovery: A node may crash at any moment, losing its in-memory (volatite)   state. It may resume
executing sometime later
- Byzantine (fail-arbitrary): A node is faulty if it deviates from the algorithm, Faulty nodes may do
anything, including crashing or malicious
```

4.6 Time Behaviour

```
- Synchronous: Message latency no greater than a known upper bound. Nodes execute algorithm at a known
speed
- Partially synchronous: System is asynchronous for some finite ( but, unknown) period of   time,
Synchronous otherwise
- Asynchronous: Messages can be delayed arbitrarily. Nodes can pause execution arbitrarily. No timing
guaranties
```

4.7 Violations of Synchrony

```
Networks
  - predictive latency
  - Message loss requiring retry
  - Congestion/ contention causing queueing
  - Network/route reconfiguration

Nodes
  - predictable speed
  - Operating system scheduling issues (priority inversion)
  - Stop-the-world garbed collection pause
  - Page fault (trashing)
```

4.7.1 Congestion

```
- Congestion occurs when there is too much traffic on the network, and the available resources (like
bandwidth or buffers) are not enough to handle it.
- It causes delays, packet loss, and reduced throughput (data transfer rate).
```

```
- Congestion can happen due to high traffic volumes, sudden traffic bursts, inefficient routing, or
network failures.
```

4.7.2 Contention

```
- Contention happens when multiple devices or processes try to access or use the same shared resource at
the same time.
- It leads to increased latency (delay) because entities have to wait for the resource to become
available.
- Contention can also reduce overall throughput and cause unfairness, where some entities get more access
than others.
```

4.7.3 Stop the World Garbage Collection

```
In programming, garbage collection is the process of automatically reclaiming memory occupied by objects
or data structures that are no longer in use by the program. This helps prevent memory leaks and
simplifies memory management for developers.

The "Stop-the-world" part of the phrase refers to the fact that, in some garbage collection
implementations, the entire application or program execution is temporarily suspended or paused while the
garbage collection process is happening. This means that all running threads or processes are stopped,
and no code is executing during this period.
```

4.7.4 Page Fault and Thrashing

```
A page fault is an exception or interrupt that occurs when a program tries to access a memory page that
is not currently in the computer's physical memory (RAM). When this happens, the operating system needs
to bring the required page from disk into memory before the program can continue executing.

Thrashing is a situation that occurs when a computer spends most of its time handling page faults,
instead of executing productive instructions. This happens when the system's physical memory is too small
to hold the working set of active memory pages required by the running programs.
```

4.7.5 Priority Inversion

```
A scheduling problem that causes a high-priority task to be blocked or delayed by a lower-priority task.

In operating systems, priority inversion can happen due to various reasons, such as:

- Sharing of resources: If a low-priority task holds a resource (like a lock or a semaphore) that a high-
priority task needs, the high-priority task may get blocked until the low-priority task releases the
resource.
- Interrupts: If a low-priority task is interrupted by a higher-priority task, but then the higher-
priority task gets blocked (e.g., waiting for I/O), the low-priority task may continue executing,
delaying the higher-priority task.
- Scheduling algorithms: Some scheduling algorithms can cause priority inversion due to their design or
implementation.
```

4.8 Availability

```
Availability – Uptime -> fraction of time that a service is functioning correctly
```

```
Two nines -> 99%    (down 3.7 days /year)
Three nines -> 99.9%    (down 8.8 hours /year)
Four nines -> 99.99%    (down 53 minutes /year)
Five nines -> 99.999%   (down 5.3 minutes /year)
```

4.9 Faliure Detection

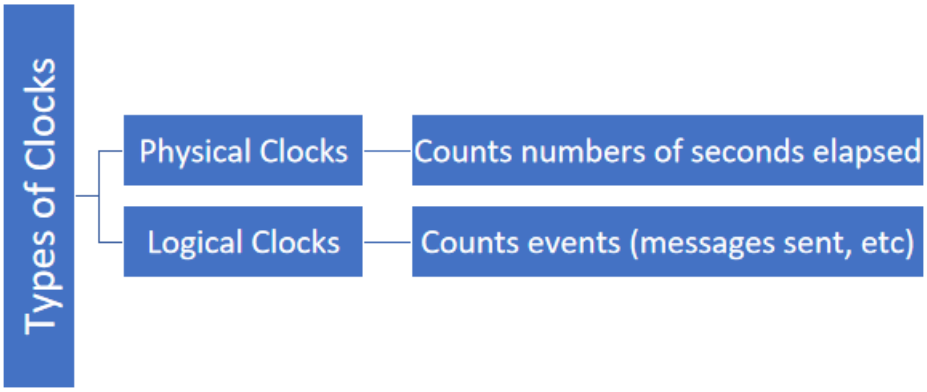
```
For crash stop/ crash recovery : send message, wait response, label node as crashed if no reply within
some timeout

Cannot tell the difference between crashed node, temporarily unresponsive node, lost messages, and
delayed messages.

Perfect timeout-based failure detectors exists only in a synchronous crash-stop system with reliable
links.

Eventually perfect failure detector
- May temporarily label a node as crashed even though it is correct
- May temporarily label a node as correct, even though it has crashed
- But, eventually, label a node as crashed if and only if it has crashed
```

5. Time

```
For software systems:

    Time is represented as numerical values, like timestamps or durations.
    Software uses time for scheduling tasks, tracking events, and measuring performance.
    Time is typically obtained from hardware clocks or external time sources.

For Operating Systems:

    Operating systems use time for scheduling processes and managing resources.
    They have a system clock and timers to keep track of time.
    Time is important for fair resource allocation and maintaining system stability.

For Distributed Systems:

    In distributed systems, there is no single global notion of time across different machines.
    They use techniques like logical clocks or time synchronization protocols to establish a consistent
view of time.
    Consistent time is crucial for coordinating operations and ensuring data consistency across different
nodes.
```

5.1 Time for Distributed Systems

```
- Scheduler  (also in operating systems) : For Scheduling, Timeouts, Failure Detectors, Retry time ,
etc..
- Performance measurement statistics, profiling : Time a process had been running, CPU usage, etc
- Log files & databases : Records when an event occurs
- Date with timelimited validity : cache entries   DNS / TLS / etc
```



5.2 Physical Clocks

Physical clocks are hardware devices that measure the passage of time. These clocks use oscillators, such as quartz crystals or atomic oscillators, to keep track of time. However, physical clocks are subject to imperfections and can experience variations in their timekeeping due to factors like temperature, aging, and manufacturing tolerances.

5.3 Atomic Clocks

Atomic clocks and GPS clocks are highly precise physical clocks. Atomic clocks use the vibrations of atoms to measure time with incredible accuracy, while GPS clocks synchronize with the time signals transmitted by the Global Positioning System (GPS) satellites. These clocks are used as reference standards for time measurement.

5.4 Skew and Drift

Skew refers to the difference in time between two clocks at a given moment.

Drift is the rate at which the clocks diverge from each other over time.

These issues arise due to various factors, such as network latencies, clock imperfections, and environmental conditions.

5.5 Logical Clocks

Logical clocks are a software-based approach to maintaining a consistent notion of time in distributed systems. They use logical timestamps, rather than physical time, to order events in a system. Logical clocks help maintain causality and ordering of events, even when physical clocks are not perfectly synchronized.

5.6 Leap Seconds

Leap seconds are periodic adjustments made to UTC to keep it synchronized with the Earth's rotation. These adjustments are necessary because atomic clocks are more stable than the Earth's rotation. Leap seconds are either added or removed from UTC to compensate for the difference between TAI and UT1.

5.7 Time Sync, NTP and PTP

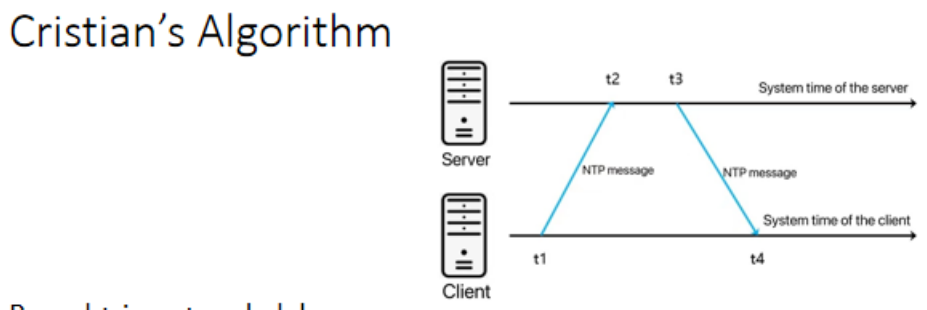
NTP and PTP are protocols used for synchronizing clocks in distributed systems. NTP is widely used for internet time synchronization, while PTP is designed for highly precise time synchronization in local area networks, often used in industrial and scientific applications.

5.7.1 Client Sever Sync

In distributed systems, it is essential to synchronize the time between servers and clients. This synchronization ensures that events are properly ordered and that timestamps are consistent across the system. Time synchronization protocols like NTP are commonly used for this purpose.

5.8 Cristian's Algorithm

Involves a client requesting the current time from a server and adjusting its clock based on the server's response, taking into account the network delay.

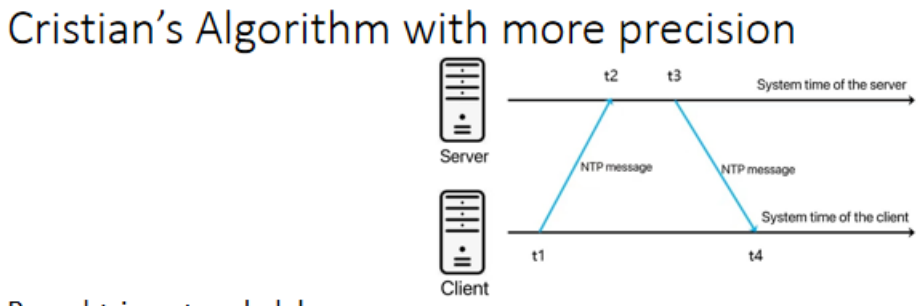


Round-trip network delay :

$$\partial = (t_4 - t_1) - (t_3 - t_2)$$

estimated server time when client receives response : $t_3 + \frac{\partial}{2}$

estimated clock skew $\theta = t_3 + \frac{\partial}{2} - t_4 = \frac{t_2 - t_1 + t_3 - t_4}{2}$



Round-trip network delay :

$$\partial = (t_4 - t_1) - (t_3 - t_2)$$

estimated server time when client receives response : $t_3 + \frac{\partial}{2}$

estimated clock skew $\theta = t_3 + \frac{\partial}{2} - t_4 = \frac{t_2 - t_1 + t_3 - t_4}{2}$

5.9 Berkeley Algorithm

Iterative clock synchronization algorithm that aims to improve the accuracy of clock synchronization over time. It involves periodically exchanging time information between multiple clocks and adjusting them

```
based on the received data.
```

Berkeley Algorithm

- Master Is chosen
- Master Uses Cristian's Algorithm to find clock drift with each slave
- Master Computes the mean value drift
- Master sends an update to each slave regarding the adjustment

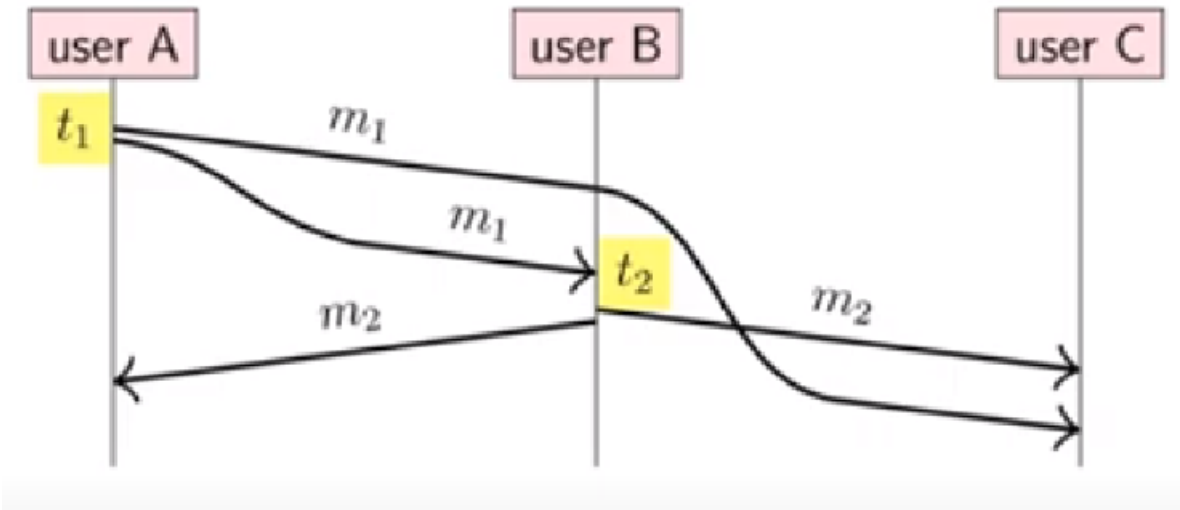
This will ensure that clocks of most slaves are relatively synchronized with each other

Algorithm also aims to minimize the amount by which each slave needs to adjust its clock

5.10 Time-of-Day and Monotonic Clocks

```
Time of Day clocks (e.g., system clocks) represent the current date and time, but they can be adjusted,
which can cause discontinuities.

Monotonic clocks, on the other hand, are clocks that always increase monotonically, even across system
reboots or time adjustments.

Monotonic clocks are useful for measuring elapsed time and ordering events, while Time of Day clocks are
used for absolute time representation.
```

6. Ordering



6.1 Event

```
An event in a distributed system is any significant occurrence or action that can be distinctly
identified. Events include sending or receiving messages, computation steps, and state changes in a
process.
```

6.2 Message

A message is a unit of communication sent from one process to another in a distributed system. Messages are used to share information, synchronize actions, and coordinate tasks among distributed processes.

6.3 Sent Order

Sent order refers to the sequence in which messages are sent from a process. Maintaining the correct sent order ensures that messages are dispatched in the intended sequence, which is critical for the system's coherence and reliability.

6.4 Recieved Order

Received order refers to the sequence in which messages are received by a process. Ensuring the correct received order is crucial for the consistency of the system's state and the proper execution of operations based on the messages received.

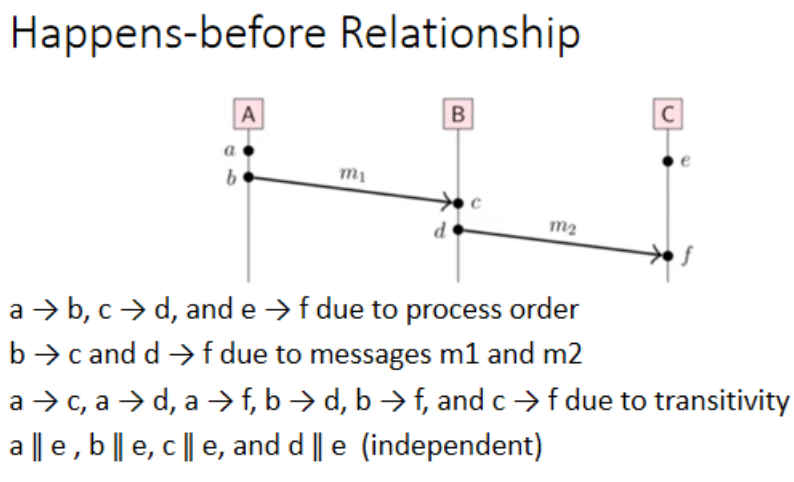
6.5 Issues in Ordering

Network Delays: Variability in network delays can cause messages to arrive out of order.
Clock Skew: Differences in the clocks of distributed nodes can lead to inconsistencies in the perceived order of events.
Concurrency: Simultaneous actions by different processes can make it difficult to establish a single global order.
Fault Tolerance: Node failures and message losses can disrupt the intended order of events.

6.6 Happens-Before Relationship

If event A occurs before event B in the same process, then A → B.
If event A is the sending of a message and event B is the receipt of that message, then A → B.
If A → B and B → C, then A → C (transitivity).

It is possible that neither a → b nor b → a. In that case, a and b are concurrent (a ‖ b)



6.7 Causality

```
Causality in distributed systems refers to the relationship between events where one event is understood
to have caused another. If event A causally affects event B, then A must happen before B. This ensures
that the system's behavior is consistent with the cause-and-effect relationships of events.

When a → b, then a might have caused b
When a ‖ b, known a cannot have caused b
```

Let $\prec$ be a strict total order on events

if (a $\rightarrow$ b) ➔ (a $\prec$ b) ; then $\prec$ is a **causal order**

$\prec$ is "consistent with causality"

There is a causal relationship

6.8 Lamport Clock

```
Each process maintains a counter that is incremented for each event. When a process sends a message, it
includes the counter value. Upon receiving a message, a process updates its counter to be greater than
both its current value and the received counter value
```

- Each node maintains a counter (t)
  counter is incremented on every local events (e)
- Let L(e) be value of t after increment
- Every message sent over network is appended with current t
- Receiver adjusts its internal t to the received t (iff its greater) and then increment the event count

Properties : **If a → b ➔ L(a) < L(b)**
But, L(a) < L(b) does not imply a → b
Possible -> L(a) = L(b) for a ≠ b

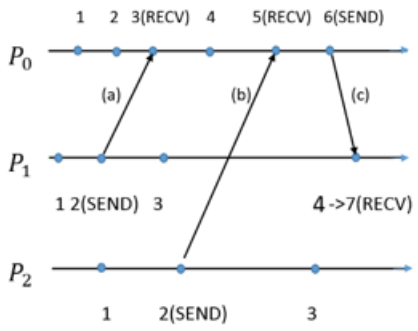
Lamport Clock Algorithm

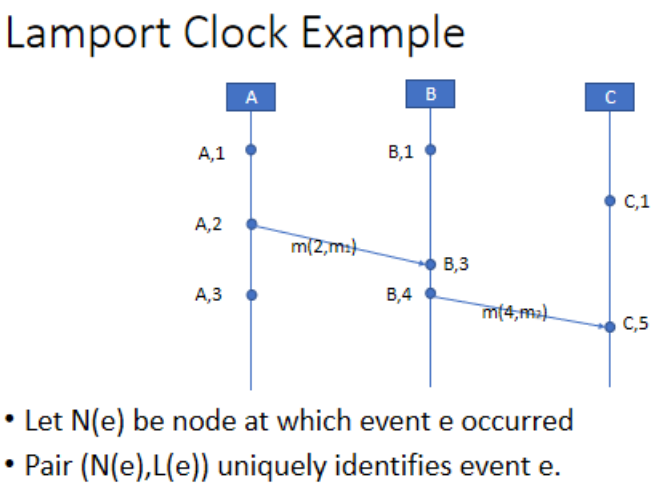
on initialization do
  t:=0 (each node maintains local variable t)
end initilization

on any event occurring at the local node do
  t := t+1
end any event

on request to send message m do
  t := t+1; send (t,m) via network link
end request to send message

on receiving (t',m) via network link do
  t := max(t',t) +1
  deliver m to the application
end receiving

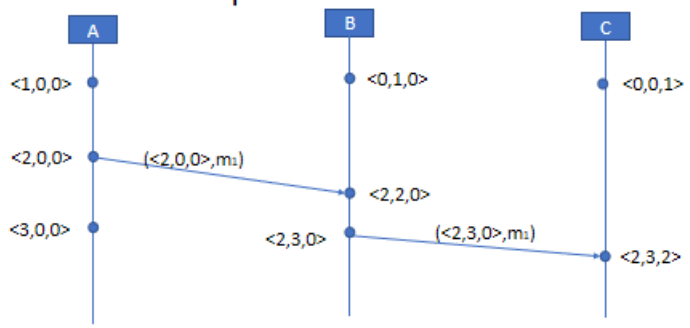




6.9 Vector Clock

```
A vector clock is an advancement over Lamport clocks, providing a way to capture causality between
events. Each process maintains a vector of counters, with one counter for each process in the system.
When an event occurs, a process increments its own counter in the vector. When sending a message, it
includes its vector. Upon receiving a message, a process updates its vector by taking the element-wise
maximum of its own vector and the received vector. Vector clocks allow for the determination of
concurrent, causally related, or causally unrelated events.
```

Vector Clock Example



- Vector timestamp of an event e represents a set of events;

e and its causal dependencies : {e} ∪ {a | a → e }

Vector clock ordering

Define following order on vector timestamps

(in a system with n nodes)

T = T' iff T[i] = T'[i] for all i ∈ {1,.....,n}

T ≤ T' iff T[i] ≤ T'[i] for all i ∈ {1,.....,n}

T < T' iff T < T'[i] && T ≠ T'[i]

T ‖ T' iff T ! ≤ T'[i] && T'[i] ! ≤ T (incomparable!)

V(a) ≤ V(b) iff ({a} ∪ {e | e → a }) ⊆ ({b} ∪ {e | e → b })

(V(a) < V(b)) ↔ (a → b)

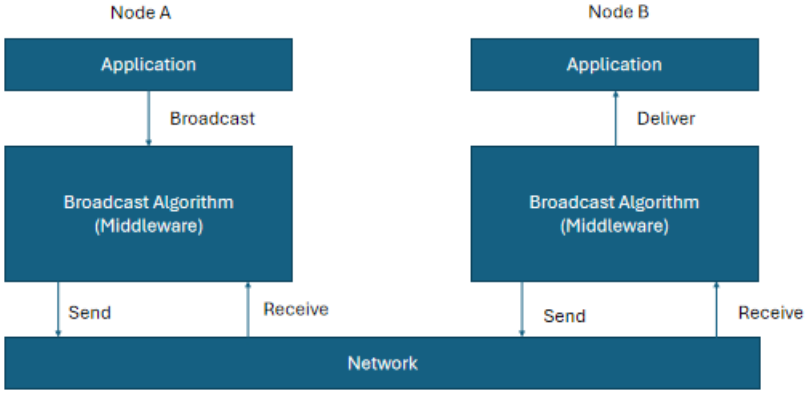
(V(a) = V(b)) ↔ (a = b)

(V(a) ‖ V(b)) ↔ (a ‖ b)

7. Broadcast Protocols

```
Why Broadcast Protocols in Distributed Systems?

- Data Replication: Ensuring all nodes have the same data.
- Coordination: Synchronizing actions among nodes.
- Fault Tolerance: Ensuring reliability and availability.
```

Sending and Receiving



Network sends point-to-point messages. After BC algorithm receives a message – it may buffer / queue before delivering to the application

7.1 Unicast

```
Unicast is a communication method where data is sent from one sender to one specific receiver. Each
message is individually addressed to a specific recipient.

Example: When you send an email to a single recipient, it is an example of unicast communication.
```

7.2 Broadcast

```
Broadcast is a method where a message is sent from one sender to all nodes in the network. Every node
receives the same message simultaneously.

Example: Sending a message to all devices on a local network segment using network broadcast.
```

7.3 Multicast

```
Multicast involves sending a message from one sender to a specific group of nodes. Only the nodes that
are part of the multicast group receive the message.

Example: Streaming a live video to a group of subscribers using multicast IP addresses.
```

7.4 Point-to-Point Communication (Non-IP Multicast)

```
Point-to-point communication involves direct communication between two nodes without relying on IP
multicast.

Example: A direct TCP connection between two servers for data exchange.
```

7.5 Best Effort vs. Reliable Communication

```
Best Effort communication does not guarantee message delivery. Messages may be lost, duplicated, or
received out of order.

Reliable Communication ensures that messages are delivered, typically in the correct order, and without
duplication.
```

7.6 Issues in Message Delivery

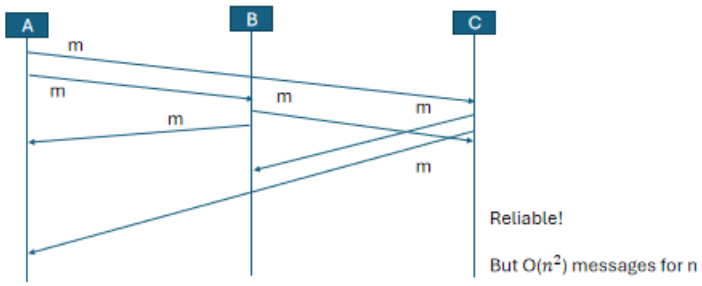
```
Message Loss: In best-effort, messages may get lost. Solutions include acknowledgments and
retransmissions.
Duplication: In reliable communication, unique message identifiers can help prevent duplicates.
Ordering: Out-of-order messages can be managed using sequence numbers or logical clocks.
```

7.7 Asynchronous, Partially Synchronous Timing Models

```
Asynchronous: No assumptions are made about the time taken for messages to be delivered or actions to be
completed.
Partially Synchronous: Some bounds on message delivery times and execution speeds are known, which helps
in designing more efficient protocols.
```

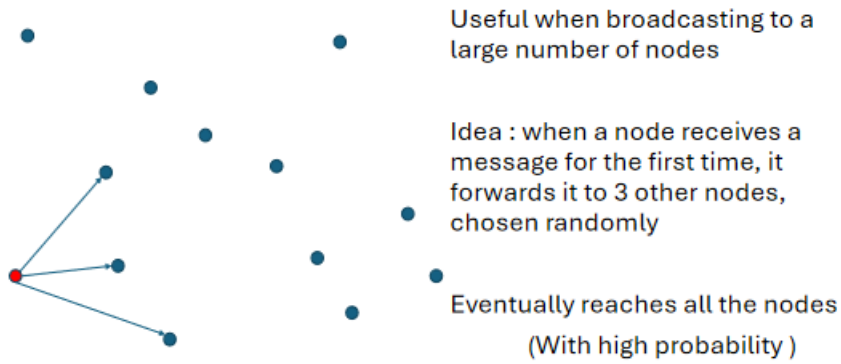
7.8 Eager Reliable Broadcast

When a node receives a message, it immediately sends it to all other nodes, ensuring fast and reliable delivery.



7.9 Gossip/Epidemic Protocol

A node tells a random subset of other nodes about a new update, which then tell other nodes, eventually reaching all nodes. This approach is robust and scalable.



7.10 Reliable Broadcast Paradigm

All nodes receive the same set of messages.
Messages are delivered without duplication.
The order of messages is preserved as needed (FIFO, Causal, Total Order).

Reliable Broadcast Paradigm

FIFO Broadcast

If m_1 and m_2 are broadcasted by the same node;
broadcast(m_1) → broadcast (m_2); then m_1 must be delivered before m_2

Causal Broadcast

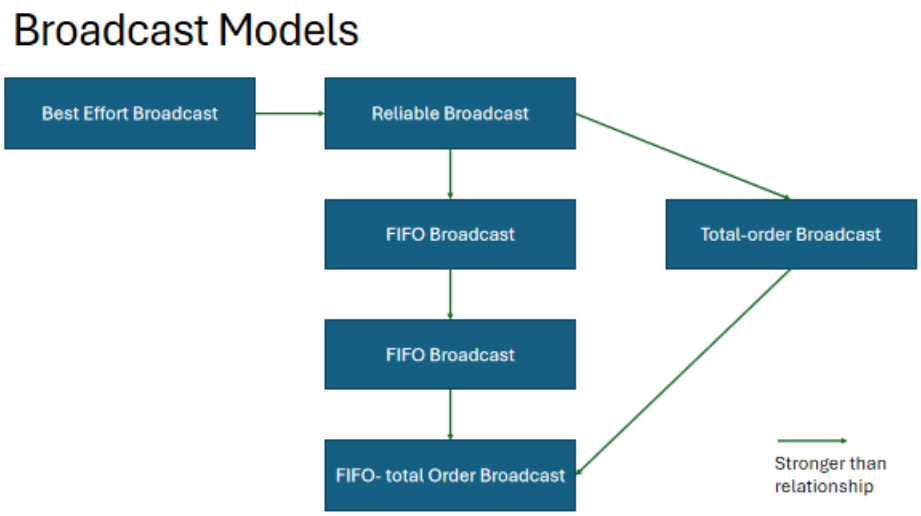
broadcast(m_1) → broadcast (m_2); then m_1 must be delivered before m_2

Total Order Broadcast

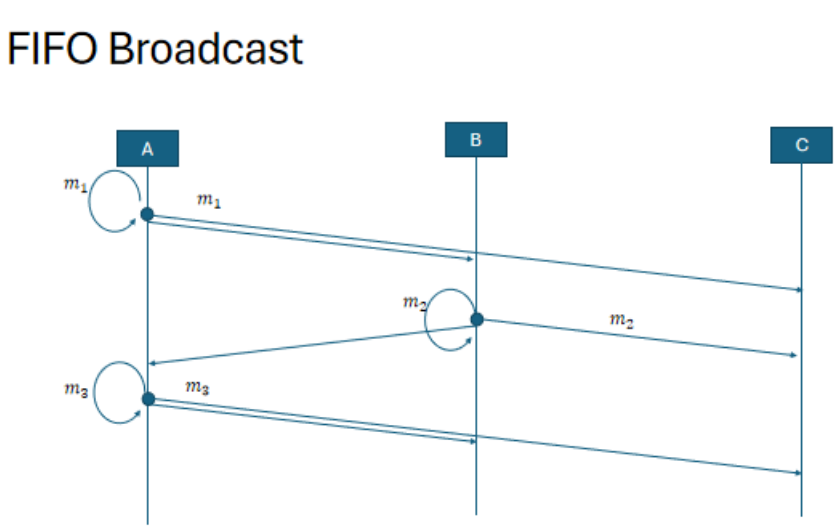
If m_1 is delivered before m_2 on one node; then m_1 must be delivered before m_2 on all nodes

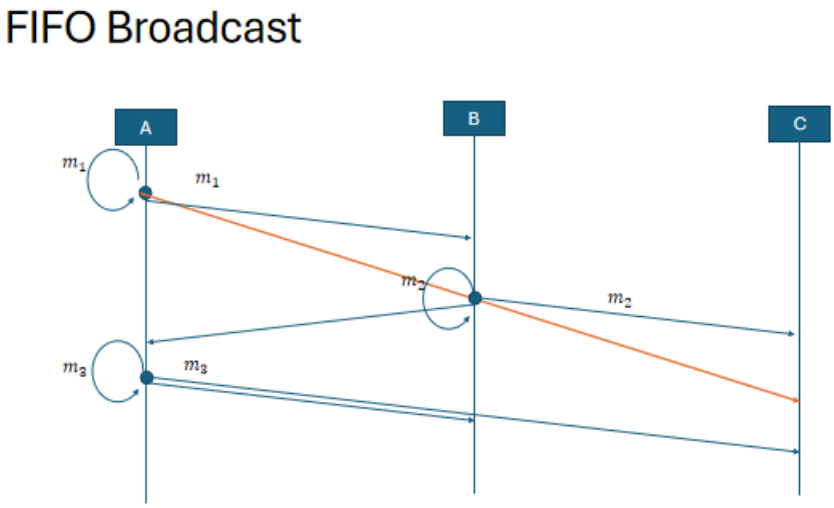
FIFO-Total Order Broadcast

Combination of FIFO broadcast and total order broadcast

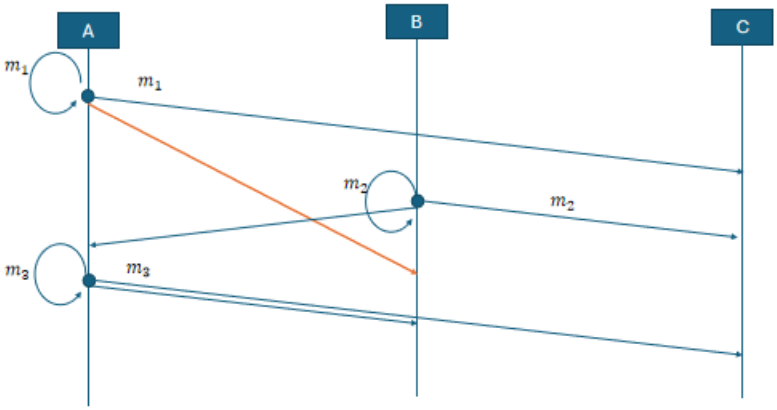

Broadcast Models
Best Effort Broadcast
Reliable Broadcast
FIFO Broadcast
Total-order Broadcast
FIFO Broadcast
FIFO- total Order Broadcast
Stronger than relationship

7.10.1 FIFO


FIFO Broadcast
A
B
C
m_1
m_1
m_2
m_2
m_3
m_3


FIFO Broadcast
A
B
C
m_1
m_1
m_2
m_2
m_3
m_3

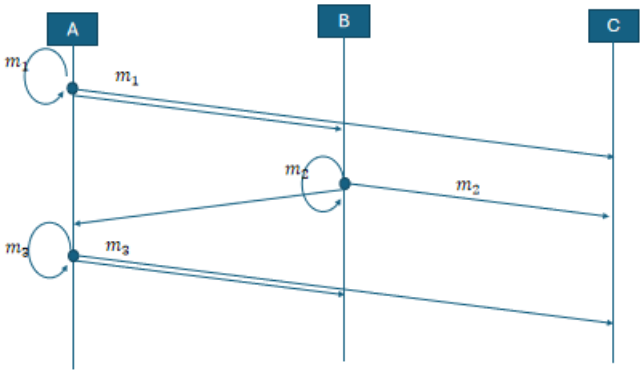
FIFO Broadcast



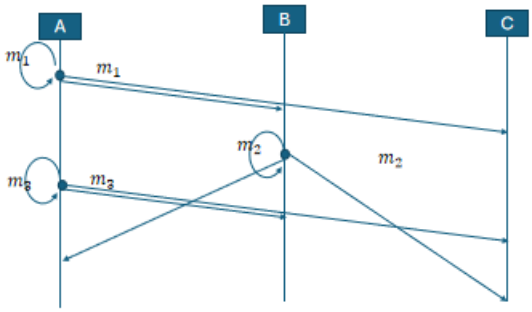
FIFO Broadbast

- **Valid Orders**

(m_2, m_1, m_3)
(m_1, m_2, m_3)
(m_1, m_3, m_2)



7.10.2 Causal



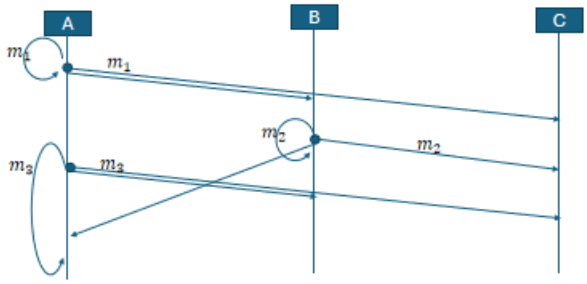
- Causally related messages must be delivered in causal order

broadcast(m_1) → broadcast (m_2) && broadcast(m_1) → broadcast (m_3)

➔ Valid orders (m_1, m_2, m_3)
(m_1, m_3, m_2)

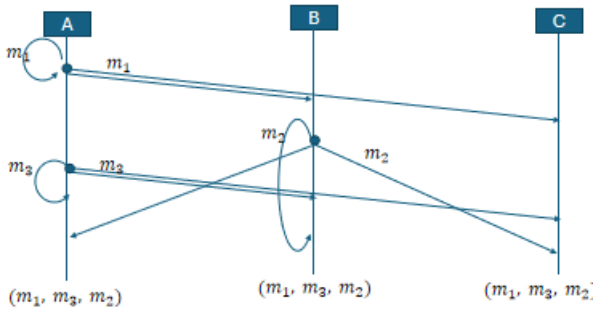
7.10.3 Total Order

All nodes must deliver
In the same order



INCLUDES node's deliveries to itself
This case : (m_1, m_2, m_3)

All nodes must deliver
In the same order



INCLUDES node's deliveries to itself
This case : (m_1, m_3, m_2)

Single Leader

- One leader is designated (sequencer)
- To broadcast messages, send it to the leader;
  leader broadcasts it via FIFO broadcast

Problem : Leader crashes ➔ no more messages are delivered
Solution : changing the leader / safety is difficult

Single Leader

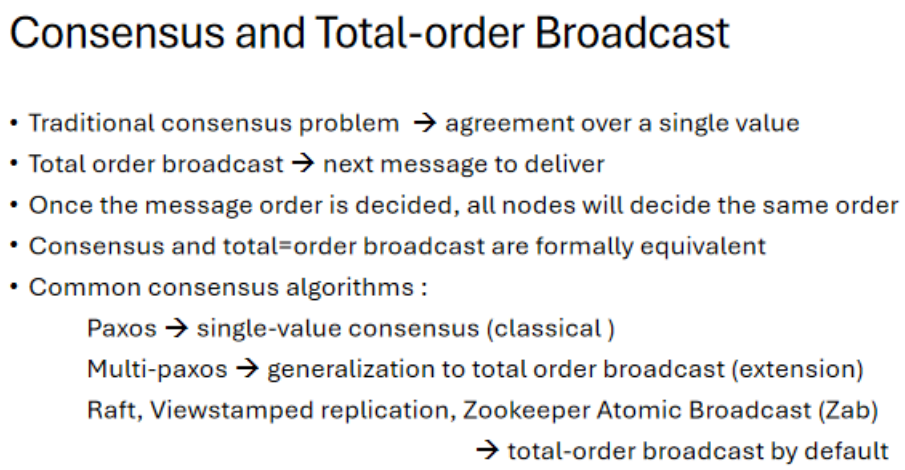
- One leader is designated (sequencer)
- To broadcast messages, send it to the leader;
  leader broadcasts it via FIFO broadcast

Problem : Leader crashes ➔ no more messages are delivered
Solution : changing the leader / safety is difficult

```
Neither of Single Leader approach or Lamport Clock approach are fault tolerant
```

7.10.4 FIFO Total Order

```
Sending all messages via a single leader
Concensus is needed to confirm the ordering of messages
```



7.11 Implementing Fault Tolerance in Broadcast Protocols

```
FIFO:

    Fault Tolerance: Use acknowledgment and retransmission for message loss.
    Example: If a message is lost, the sender retransmits until it is acknowledged.

Causal:

    Fault Tolerance: Use vector clocks and ensure retransmission upon failure detection.
    Example: Nodes track dependencies with vector clocks and retransmit if messages are missing.

Total Order:

    Fault Tolerance: Use consensus algorithms (like Paxos or Raft) to agree on message order.
    Example: Nodes use a consensus protocol to agree on a sequence number before delivering messages.

FIFO Total Order:

    Fault Tolerance: Combine approaches from FIFO and total order, such as leader-based sequencing with
acknowledgments.
    Example: A leader assigns sequence numbers and ensures all nodes acknowledge receipt before
confirming delivery.
```

8. Replication

```
Creating copies of data or services across multiple nodes.
Why it is Needed?

    - Improve fault tolerance.
    - Enhance data availability.
    - Increase system performance.

*** Can implement total order broadcast by sending all messages via a single LEADER
    Problem : What if leader crashes ? Become unavailable ?
```

8.1 Probability of Faults in Replication

```
More replicas reduce the likelihood of total data loss.
Higher fault tolerance with more replicas.
```

If any replica has the probability of p of being faulty or unavailable at any time and faults are independent (not true, but approximation)

Probability of all n replicas being faulty p^n

Probability of ≥ 1 (atleast one)

out of n replicas being faulty : 1- $(1-p)^n$

8.2 Availability and Faultiness

```
Increased replicas improve availability.
Fault tolerance depends on the number of replicas and replication strategy.
```

Replicas (n)	P(≥ 1 faulty)	P(≥ $\frac{n+1}{2}$ faulty)	P(all n faulty)
1	0.01	0.01	0.01
3	0.03	$3 * 10^{-4}$	10^{-6}
5	0.049	$1 * 10^{-5}$	10^{-10}
100	0.63	$6 * 10^{-74}$	10^{-200}

8.3 Retry and Deduplication

```
Retry: Resending failed operations.
Deduplication: Ensuring duplicate operations are not processed multiple times.
```

8.4 Idempotence

A function f is idempotent if $f(x) = f(f(x))$

Increment → not idempotent $f(likeCount) = likeCount + 1$

Idempotent : $f(likeSet) = likeSet \cup \{UserID\}$

Idempotent requests can be retired without deduplication

```
class Post {
    constructor() {
        this.likeSet = new Set();
    }

    like(userId) {
        this.likeSet.add(userId);
    }

    getLikeCount() {
        return this.likeSet.size;
    }
}

// Example Usage
const post = new Post();

// User likes the post
```

```
post.like('user');
console.log(post.getLikeCount());
```

8.5 Retry Semantics

```
At Most One: Operation may fail or succeed once, no duplicates.
At Least One: Operation guaranteed to succeed at least once, may have duplicates.
Exactly One: Operation succeeds exactly once, no duplicates
```

8.5.1 At Most Once

```
Operation may fail or succeed once, no duplicates.

Use Case: Payment Processing

Example: When a payment is initiated, it's crucial to ensure that the transaction either completes
successfully or fails without any retries to avoid double charging the customer.

Why: Duplicate operations could lead to multiple charges, which would be unacceptable.
```

8.5.2 At Least Once

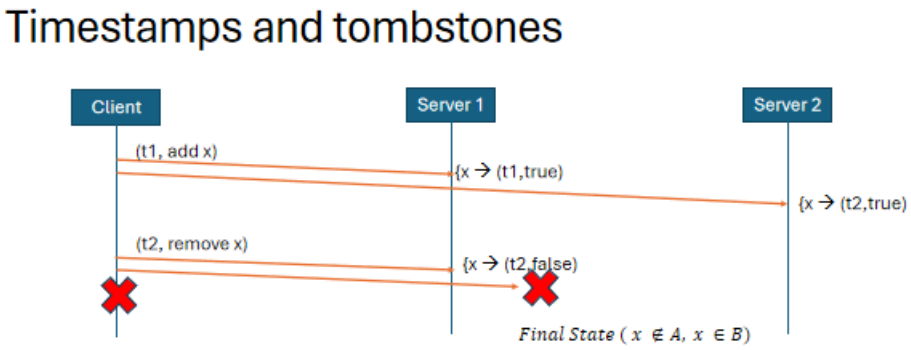
```
Operation guaranteed to succeed at least once, may have duplicates.

Use Case: Log Delivery

Example: In a distributed logging system, logs must be reliably delivered to a central server or data
store. If a log message is lost, it should be retransmitted until acknowledged.

Why: It’s more important that logs are eventually received, even if it means some logs are received
multiple times, which can be handled with deduplication on the receiving end.

Use Case: Event Notification

Example: Sending notifications to users about important events (e.g., system alerts, transaction
confirmations) should ensure delivery, even if the same notification is sent multiple times.

Why: Missing notifications can be critical, whereas receiving duplicates is less problematic.
```

8.5.3 Exactly Once

```
Operation succeeds exactly once, no duplicates. (Use idempotent operations)

Use Case: Distributed Database Writes

Example: Writing to a distributed database where each write operation must be applied exactly once to
maintain data consistency.

Why: Duplicate writes can corrupt the database state, and missed writes can lead to data loss.
Implementing an exactly-once guarantee ensures that each write is processed only once, preserving data
integrity.

Use Case: Message Queue Processing

Example: In a distributed message queue system, each message should be processed exactly once to ensure
```

```
accurate downstream processing.

Why: Processing a message multiple times can lead to duplicated actions (e.g., sending the same email
multiple times), and missing messages can lead to data loss or incomplete processing.
```

8.6 Timestamps and Tombstones

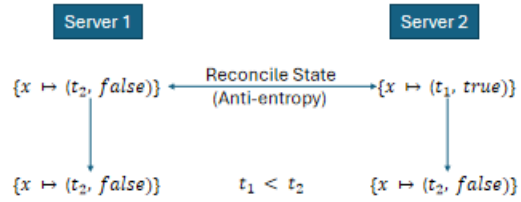


Remove x → doesn't actually remove x; it labels x with "false"
To indicate its invisible **(tombstone)**
Every record has **logical timestamp** of last write

8.8 Replica Reconciling

```
Process of ensuring all replicas have the same data.
Techniques: Conflict resolution, merging updates.
```

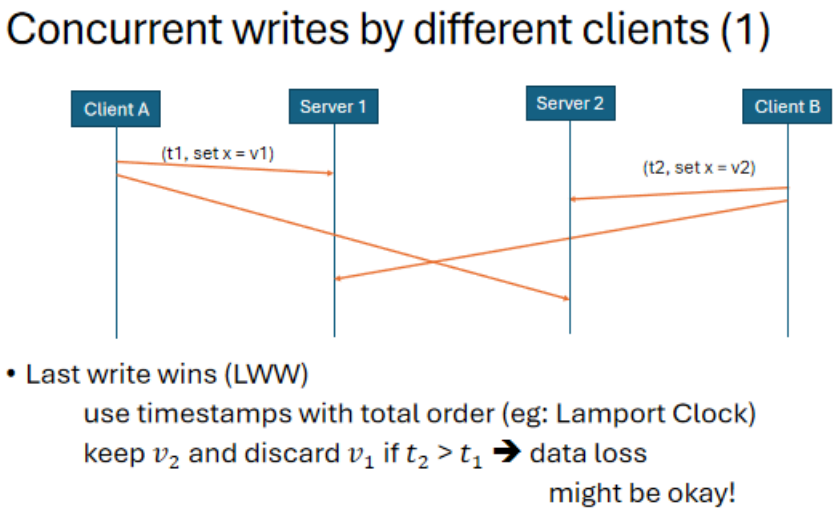
Replicas periodically communicate among themselves to check for any inconsistencies



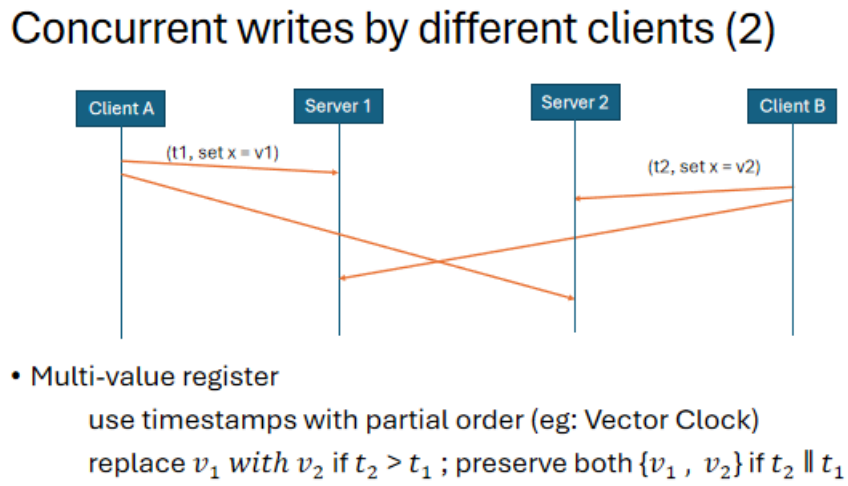
Propagate the record with the latest timestamp
Discard the records with earlier timestamps (for a given key)

8.9 Concurrent Writes by Different Clients

```
Last Write Wins: The latest write operation overrides previous ones.
```




```
Multi-Value Register: All concurrent writes are stored, resolved later.
```



8.10 Read After Write Consistency (RAW)

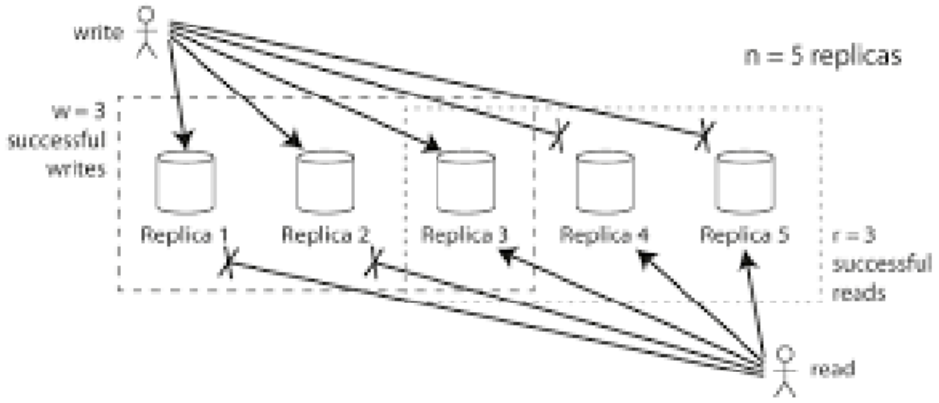
```
Ensures that a read operation immediately reflects a prior write operation. This means that immediately
after a write, the updated data should be visible to any read request.
It provides a guarantee that after you make a change to data, you can immediately read the latest version
of that data.

Single Master Systems:
- Write operations are directed to a single master node, and reads can be served from the same master or
a synchronized replica. Ensures the latest data is available immediately after a write.

Quorum-Based Systems:
- Writes and reads are acknowledged based on a majority quorum.
Ensures that after a write, a majority of replicas reflect the change, so subsequent reads can get the
updated data.

Synchronous Replication:
- Write operations are propagated to all replicas synchronously.
Ensures that all replicas have the updated data immediately after a write.
```

8.10.1 Strategies to Achieve RAW Consistency

```
Primary-Secondary Replication:
Use a primary node for writes and ensure secondary nodes are updated synchronously.

Strong Consistency Protocols:
Implement protocols like Paxos or Raft to ensure that a write is acknowledged only when it has been
committed by a majority of nodes.

Immediate Cache Invalidation:
Invalidate or update cache entries immediately after a write to ensure that subsequent reads fetch the
latest data from the database.
```

8.11 Quorum

```
Read Quorum: Minimum number of replicas to read from for consistency.
Write Quorum: Minimum number of replicas to write to for consistency.

Majority Quorum: A majority of nodes must agree on an operation for it to be considered successful.
r = w = (n + 1)/2

Read Repair: Inconsistent data detected during reads is corrected and synchronized with other replicas.
```



In a system with *n* replicas

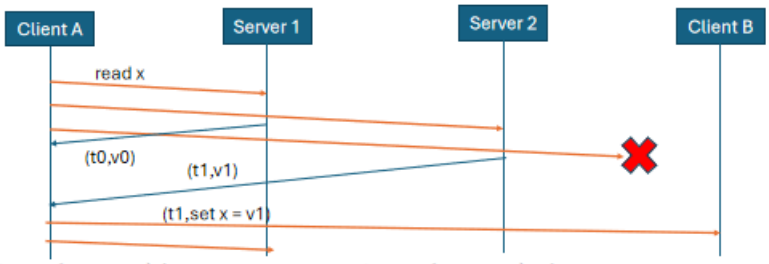
- If a write is acknowledged by *w* replicas (write quorum)
- And subsequently read from *r* replicas (read quorum)
- And $r + w > n$

  then the read will see the previous written value !!!

  (or a value that subsequently overwrote it)

Read quorum and write quorum share ≥ 1 replica

Read Repair



- Update (t_1, v_1) is more recent than (t_0, v_0) since $t_0 < t_1$
- Client can propagate (t_1, v_1) to other replicas

8.12 State Machine Replication

```
Ensuring all replicas follow the same sequence of operations.
Considerations: Deterministic state machines, consistent order of operations.
```

- State Machine Replication (SMR)
  - FIFO-total-order broadcast every update to all the replicas
  - Replica delivers update message : apply it to own state
  - Applying an update is deterministic
  - Replica is a state machine → starts at a fixed initial state
    goes through the same sequence of state transitions
    In the same order
    → All replicas endup in the same state

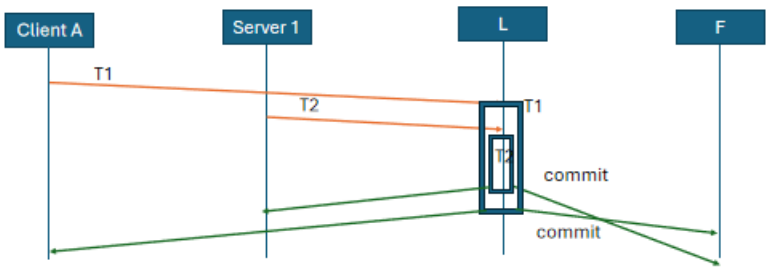
8.12.1 Limitations

Limitations

- Cannot update states immediately
- Have to wait for delivery through broadcast
- Need fault-tolerant total-order broadcast

Replication Using Causal (weaker) Broadcast

- If replica state updates commutative, replicas can process updates in different order, still endup in the same state
- Updates *f and* g are commutative if $f(g(f(x)) = g(f(x)))$



Summary of Broadcast

Broadcast Model	Assumptions about state update function
Total-order	Deterministic (SMR)
Causal	Deterministic, concurrent update commutative
Reliable	Deterministic, all updates commutative
Best-effort	Deterministic, commutative, idempotent, tolerates message loss

8.13 Leaders for Consensus

Consensus System Models

- Paxos, Raft etc
  Assumes partially synchronous, crash-recovery system model

Why not Asynchronous ? → FLP results (Fitcher, Lynch, Paterson)
There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model

Paxos, Raft, etc use clocks used for timeouts/ failure detector to ensure progress. Safety (Correctness) does not depend on timing

These are also consensus algorithms for a partially synchrounous Byzantine system models (Used in blockchains)

```
Single Leader: One node responsible for coordination.
Shadow Leader: Backup leader ready to take over.
Leader Crash: Mechanism to detect and handle leader failure.
Leader Elections: Process to choose a new leader.
Split Brain: Scenario where network partition leads to multiple leaders.
Network Partitioning: Handling communication breakdown between parts of the system.
```

9. Consistency in Replicas