Distributed Data Analytics
Distributed Systems

Thorsten Papenbrock
G-3.1.09, Campus III
Hasso Plattner Institut

Distributed Systems

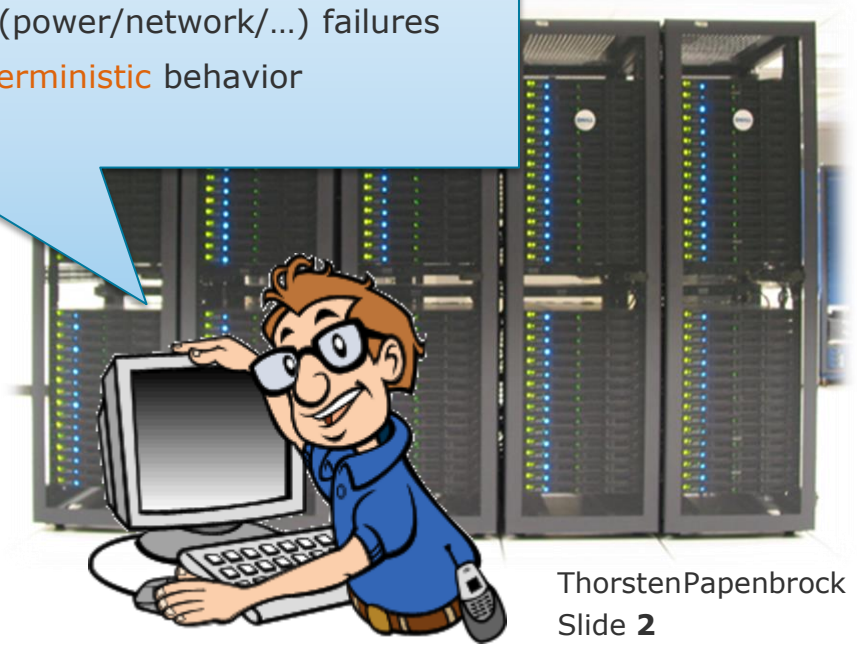
I am facing …

- software bugs
- power failures
- head crashes
- hardware aging
- …

Non-Distributed System Developer

I am facing everything he faces and …

- network faults
- clock deviation
- partial (power/network/…) failures
- nondeterministic behavior
- …



Distributed System Developer

Distributed Systems

"My system is predictable"

"I can debug easily"

"A well operating system should not have failures"

"I use parallelism whenever necessary"

Non-Distributed System Developer

"My system is predictably unpredictable"

"Debugging is hard"

"A well operating system properly deals with its failures"

"Parallelism is my bread and butter"

Distributed System Developer


Murphy's Law
Anything that can go wrong will go wrong
Everyone
Whenever
Anything
Nothing
Always
Another
Something
Everything
Life
Has
Believes
Is
Knows
Try
Left
Worth
Doing
Already
Worthwhile
Easy
Perfect
Equal
Better
Straightforward
Problems
Bigger
Creates
Doesn't
Cannot
Wrong
Fault
Expect
Expected
Worse
Unproven
False
Dreams
Goals
Plans
Tasks
Pleasing
Nobody
Likes
Screw It Up
Worst
Seems
Supposed
Too good to be true..
Things
Going
Well
Probably
Corner
Around
Problem
Last
Several
Smile
Fixing
Take
Longer
Than
Cost
More
Thought
Itself
From
Bad
Worse
Done
Been
Looks
As It
Seems
Ever
Created
Gets
Has
Have
Don't
Want
So better be prepared!
IQmatrix.com
© 2009 | IQ Matrix
Created by Adam Sicinski
IQX0067:

Reliability despite Unreliable Components

Building a reliable system from unreliable components

- With no special fault handling:
  - A distributed system is only as reliable as its weakest/strongest component
- With fault handling
  - A distributed system is (much) more reliable as its unreliable components

Fault handling examples

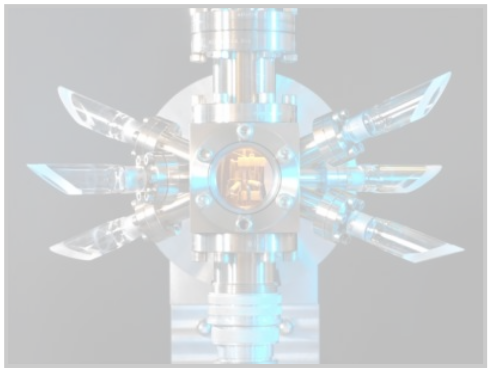
Some easily solvable faults

- Radio inference on wireless networks:
  - Error-correcting codes allow digital data to be transmitted accurately
- Unreliable Internet Protocol (IP):
  - Transmission Control Protocol (TCP) retransmits missing packages, eliminates duplicates, and reassembles packets in order

Distributed Systems

Unreliable Networks | Unreliable Clocks | Knowledge, Truth, Lies

A shark raiding an undersea cable



An atomic clock with minimum drift

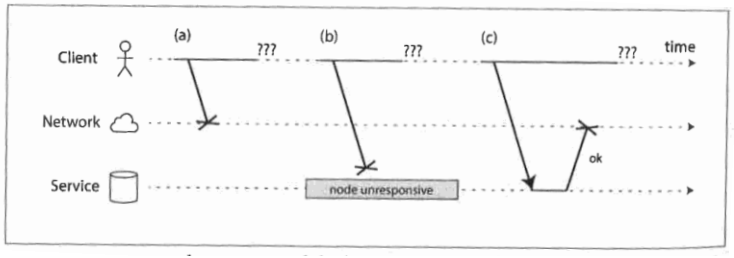
Students communi-cating their knowledge

Asynchronous Messaging Issues

Network

- Physical connection between autonomous, shared-nothing computing nodes
- Asynchronous messaging via packet binary sequences
- Nodes can send messages but no guarantees as to when/whether it arrives

Sender can't even tell if the packet was delivered …



Potential failures when sending a message

a) Request is lost on the network (e.g. cable unplugged)
b) Request is waiting in a queue and delivered later (e.g. recipient overloaded)
c) Remote node is unavailable (e.g. recipient crashed)
d) Response is delayed on the network (e.g. network overloaded)
e) Response is lost on the network (e.g. network switch misconfigured)

Detecting Faults

Using the operating system

- If a process on a node crashes, but the operating system (OS) still runs:
  - OS can close or refuse TCP connections to notify clients with an error
  - OS can trigger failover scripts to explicitly notify certain clients

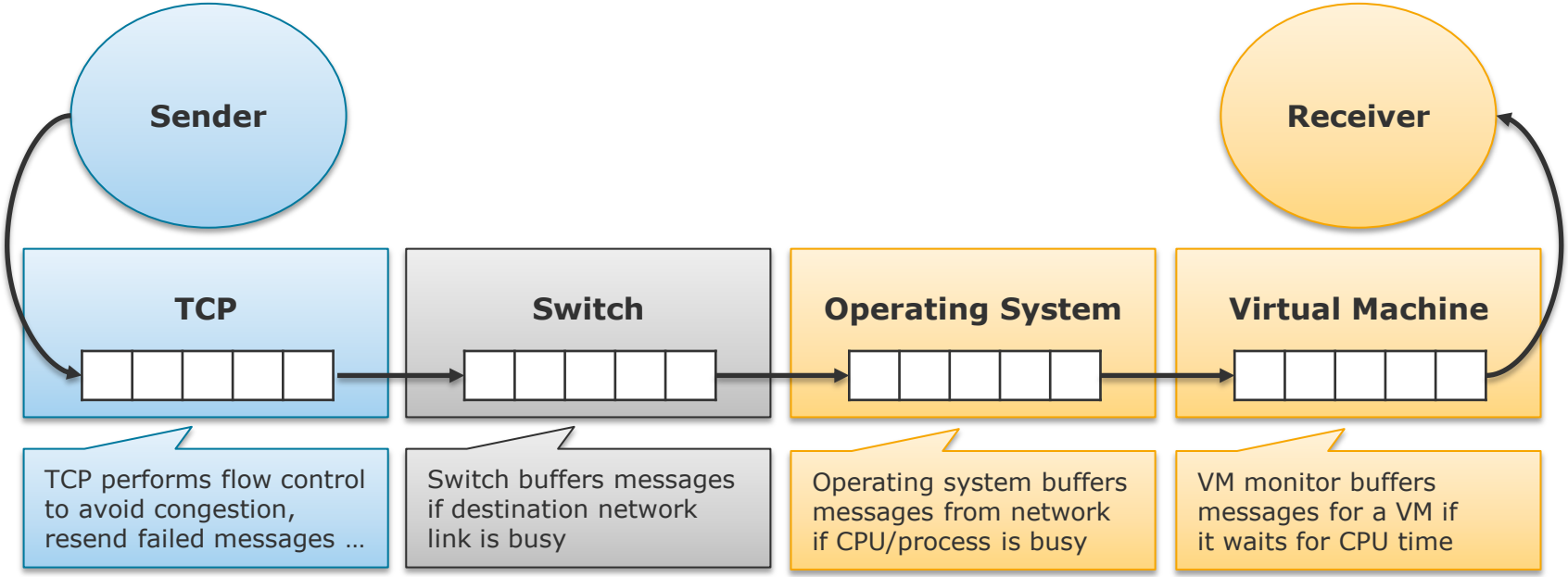
Using the network switch

- If the client has access to the network switch:
  - Switch can detect link failures on hardware level (e.g. detect if remote is powered on)

Using timeouts

- Log the sending time for each message
- Messages are declared lost if their recipient does not answer within a certain timeout
  - ➢ Most universal fault detection mechanism

Queues on the Network



- Many reasons for a packages being delayed (query congestion)
- Even if the receiver could guarantee a processing time for messages, the network cannot guarantee a transmission time for messages

Timeouts

Issues

- How to set the timeout?
  - Too long (conservative): program waits wastefully long before triggering fault handling
  - Too short (aggressive): more false message loss reports each triggering fault handling
- How to handle failures?
  - Resend message → Messages might get **handled multiple times**!
  - Reroute message → Messages might **worsen overload** if this caused the timeout!
  - Escalate as system error

Note that we cannot know:
- What caused the error?
- Has a message been worked on?

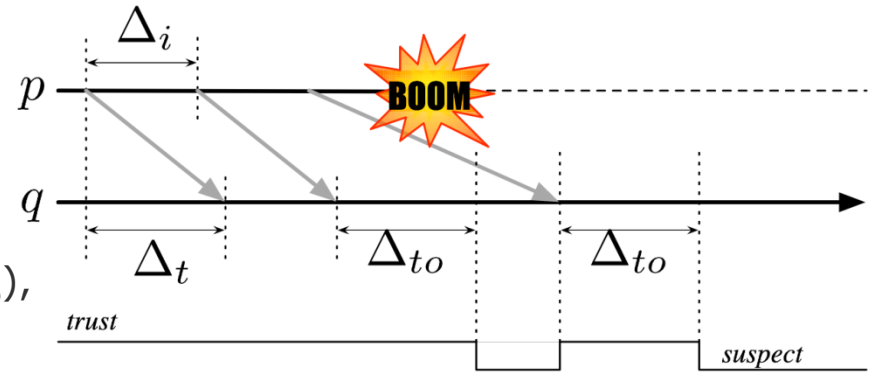
Analytical Systems

- Nodes with high CPU load due to analytical calculations
- Network with high traffic due to data-intensive nature
  - ➢ Overall high system load makes timeouts hard to predict

Defining Timeouts Experimentally

The traditional heartbeat method

- The monitored process p sends periodical heartbeat messages to the server process q
- Δ_i : the heartbeat send interval of p
- Δ_t : the initial wait time
- Δ_{to} : the timeout
- Upon receiving the first heartbeat (Δ_t), p measures the time to the next heartbeat (Δ_{to}), which is then set as the timeout
- Problems:
  - Static timeout: query congestion might naturally delay heartbeats on higher load
  - Initialization: if the second heartbeat is delayed, Δ_{to} is set too large
  - Binary trust: client is either trusted or suspected



Defining Timeouts Experimentally

The accrual failure detector method

- Accrual failure detector:
  - German: "anwachsende Fehler Erkenner"
  - Output a suspicion level for each node instead of binary trust or fixed timeout
- Suspicion level:
  - Measure describing the probability that node *p* has failed at time *t*
  - Defined as a continuous function for *p* over *t* : $susp\_level_p(t) \geq 0$
  - Properties
    - Asymptotic completeness: If p is faulty, $susp\_level_p(t) \rightarrow \infty$
    - Eventual monotony: If p is faulty, $susp\_level_p(t)$ *monotonically increases*
    - Upper bound: If p is correct, $susp\_level_p(t)$ *has an upper bound*
    - Reset: If p is correct, $susp\_level_p(t) = 0$ for some $t > t_0$
  - ➢ Used to adjust load balancing and timeout expectations

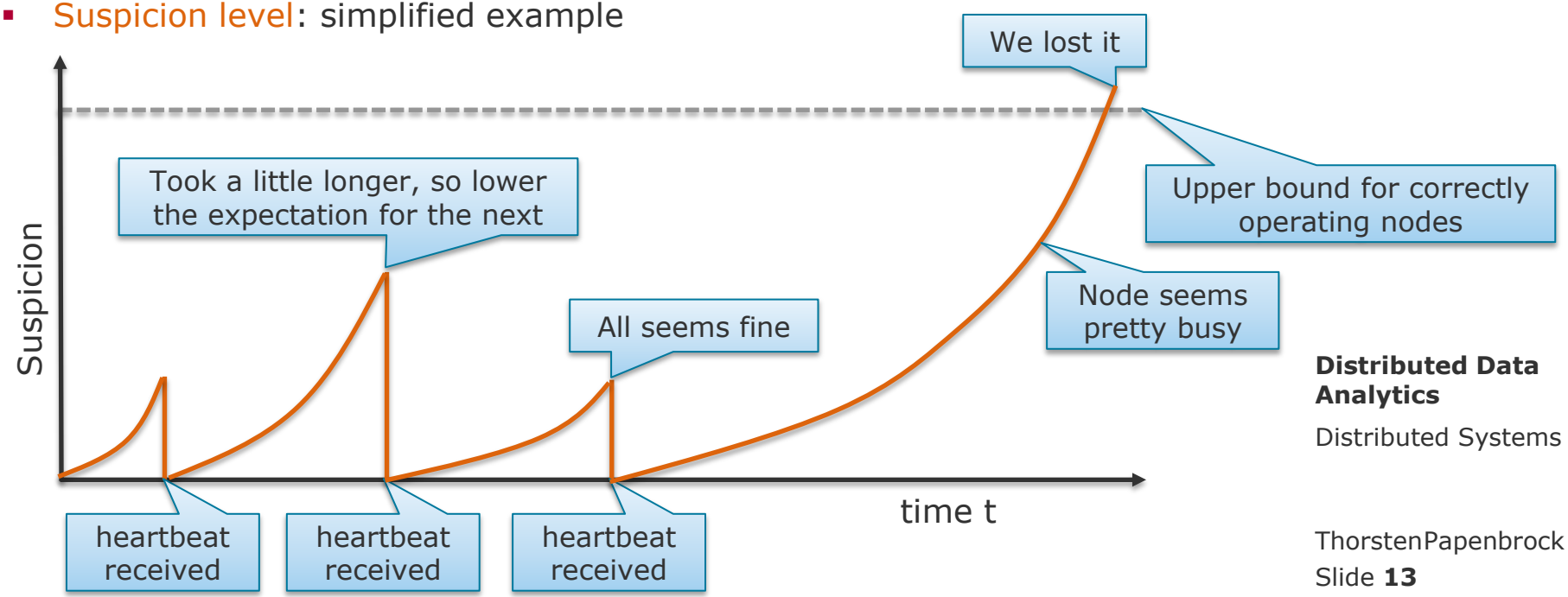
Trust is interpreted from the development of suspicion

i.e., whenever a heartbeat arrives

Defining Timeouts Experimentally

The accrual failure detector method

- Suspicion level: simplified example



Defining Timeouts Experimentally

The accrual failure detector method

- Suspicion level interpretation:
  - Example interpretation algorithm:
    - Initialize two dynamic thresholds T_{high} and T_{low} to the same arbitrary values >0 and start trusting a node
    - S-transition:
      - Whenever $susp\_level_p(t)$ crosses T_{high} upwards, $T_{high} = T_{high} + 1$ and suspect p
    - T-transition:
      - Whenever $susp\_level_p(t)$ crosses T_{low} downwards, $T_{low} = T_{high}$ and trust p
    - ➢ The longer the algorithms monitors $susp\_level_p(t)$, the better T_{high} captures real node failures
  - Suspicion dynamically adjusts to the current latency and load
    - ➢ T_{high} becomes a fix threshold that is robust against load changes

Defining Timeouts

Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama,
"The φ Accrual Failure Detector",
Japan Advanced Institute of Science and Technology, School of Information Science,
Technical Report IS-RR-2004-010, May 2004

The φ accrual failure detector

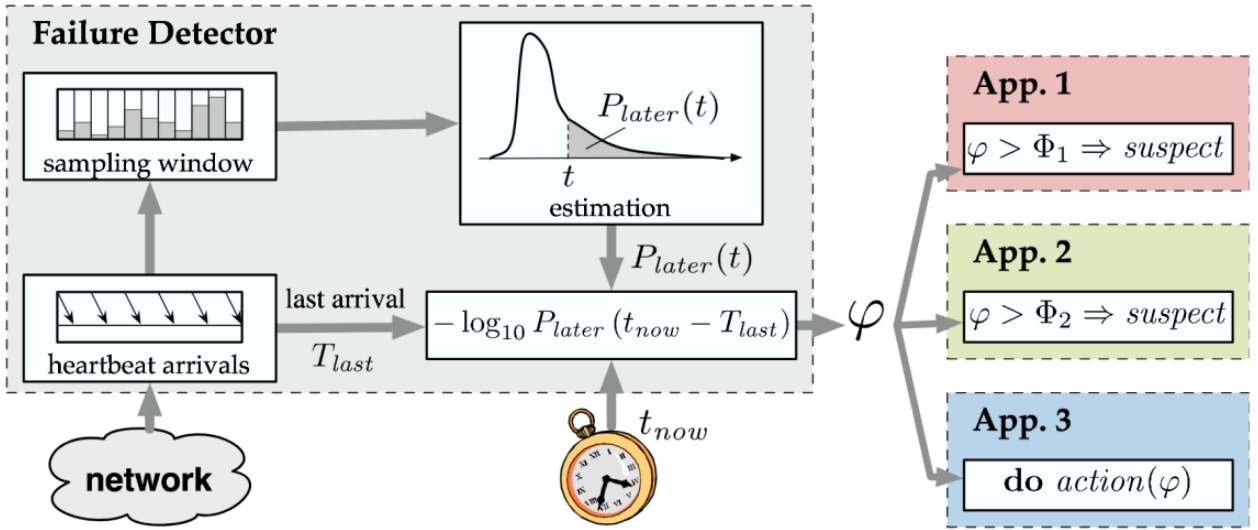
- A concrete implementation of the accrual failure detection method
- Implemented in Akka, Spark, Cassandra, …
- φ (Phi):
  - Suspicion level: $\varphi_p(t) = susp\_level_p(t)$
  - Comparable: if $\varphi_p(t) > \varphi_q(t)$, p is more likely to fail at time t than q, i.e., p differs more clearly from its usual timing than q
    - ➢ Useful for fault detection and load balancing
- General idea:
  - Continually measure response times (jitter) and availability of nodes via heartbeat
  - Calculate $\varphi_p(t)$ based on p's heartbeat history

Defining Timeouts

Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama,
"The φ Accrual Failure Detector",
Japan Advanced Institute of Science and Technology, School of Information Science,
Technical Report IS-RR-2004-010, May 2004

The φ accrual failure detector

- Variables
  - T_{last}: Arrival time of most recent heartbeat
  - t_{now}: Current time
  - P_{later}: Probability that a heartbeat will arrive more than t time units after the previous one
- Heartbeat arrivals
  - Heartbeats arrive with a sequence number to restore their send order
- Sampling window
  - Stores the arrival times in a fixed sized window (last x heartbeats per node)
  - Pre-calculates the arrival intervals, sum, and sum of squares of all samples

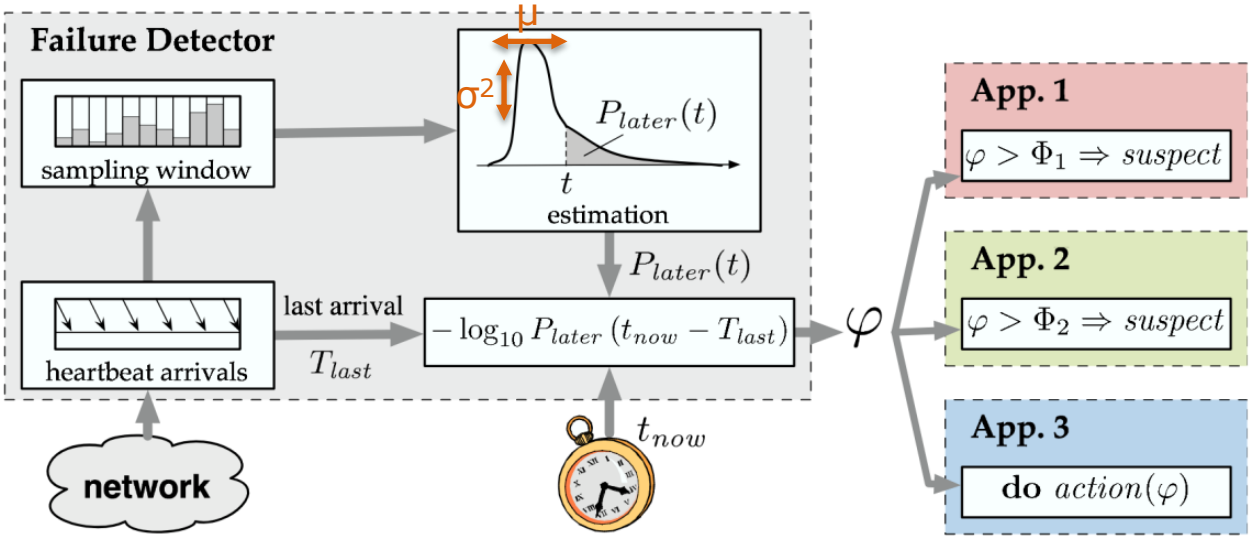


Defining Timeouts

Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama,
"The φ Accrual Failure Detector",
Japan Advanced Institute of Science and Technology, School of Information Science,
Technical Report IS-RR-2004-010, May 2004

The φ accrual failure detector

- Variables
  - T_{last}: Arrival time of most recent heartbeat
  - t_{now}: Current time
  - P_{later}: Probability that a heartbeat will arrive more than t time units after the previous one
- Estimation
  1. Calculate the mean μ and the variance σ^2 for the samples
  2. Calculate $P_{later}(t)$:



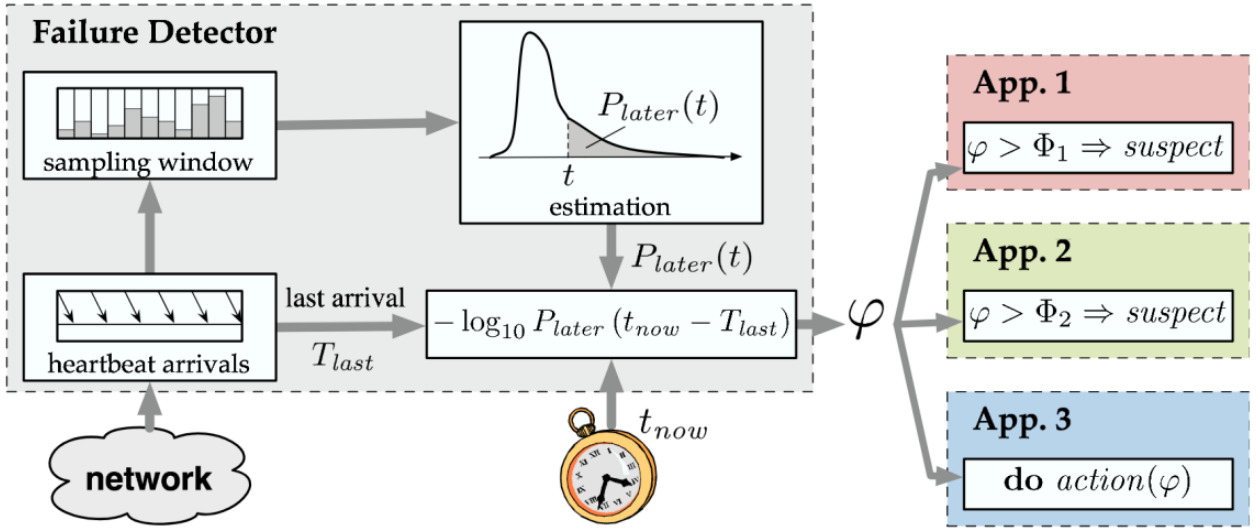
$$P_{later}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_{t}^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

Defining Timeouts

Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama,
"The φ Accrual Failure Detector",
Japan Advanced Institute of Science and Technology, School of Information Science,
Technical Report IS-RR-2004-010, May 2004

The φ accrual failure detector

- Variables
  - T_{last}: Arrival time of most recent heartbeat
  - t_{now}: Current time
  - P_{later}: Probability that a heartbeat will arrive more than t time units after the previous one
- φ calculation
  3. Calculate φ using P_{later} and the time since *p*'s last heartbeat:



$$\varphi(t_{now}) \stackrel{\text{def}}{=} -\log_{10}(P_{later}(t_{now} - T_{last}))$$

P_{later} gets increasingly smaller; $-\log_{10}$ turns small in very large values

Interpretation by application:

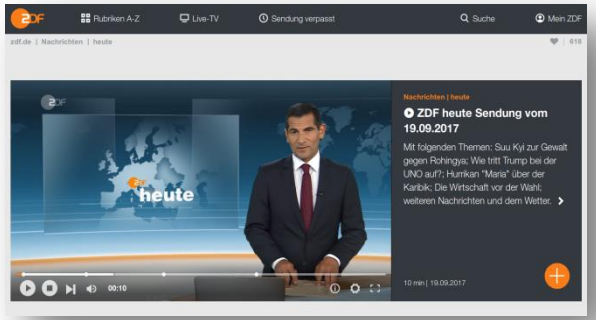
E.g. failure detection with T_{high} and T_{low} where $T_{high} = \Phi$

Ignoring Timeouts

TCP vs. UDP

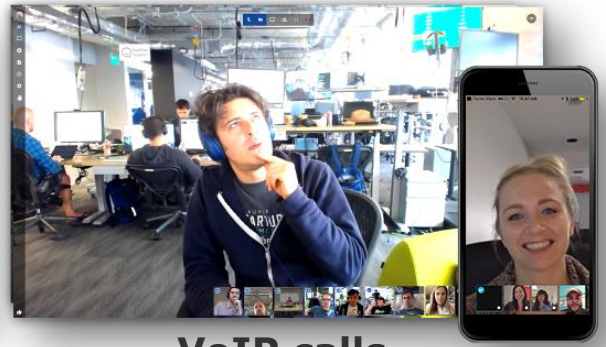
- User Datagram Protocol (UDP) does not use timeouts
  - No guarantee of delivery, ordering, or de-duplication
  - ➢ Preferable if outdated messages are worthless:

sensor processing



video streaming

gaming



VoIP calls

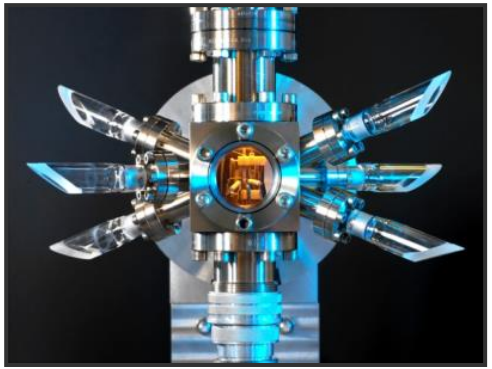
- ➢ Problematic for most analytical use cases!

Distributed Systems

Unreliable Networks

A shark raiding an undersea cable

Unreliable Clocks



An atomic clock with minimum drift

Knowledge, Truth, Lies

Students communi-cating their knowledge

Clocks vs. Networks

Unreliable clocks

Often cause
silent, creeping failures and data loss

Unreliable networks

Usually cause
noticeable crashes and failures

Both need to be considered in application logic!

About Clocks

Computer clocks

- Actual hardware devices: quarz crystal oscillator
- Not perfectly accurate and not in sync with other clocks

Clock usage in distributed systems

1. Measure duration e.g.:
   - Has this request timed out yet?
   - What's the 99th percentile response time of this service?
   - How long did the user spend on this page?
2. Measure points in time e.g.:
   - When was this heartbeat send?
   - When does this cache entry expire?
   - What's the timestamp of this error message?

About Clocks

Kinds of clocks

a) Time-of-day clock:

- Returns the current time according to some calendar (e.g. millis since 01.01.1970 UTC)
- Example: `clock_gettime(CLOCK_REALTIME)` (Linux)
  `System.currentTimeMillis()` (Java)
- Can be changed completely (e.g., synchronized via NTP)
- Used to measure points in time

b) Monotonic clock:

- A constantly forward moving clock with no reference point (specific values are meaningless)
- Example: `clock_gettime(CLOCK_MONOTONIC)` (Linux)
  `System.nanoTime()` (Java)
- Can be speeded up or slowed down (e.g., by 0.05% via NTP)
- Used to measure durations (time intervals)

Unreliability

Clock drift

- Natural deviation of clock speeds due to …
  - machine temperature
  - gravitation
  - aging and abrasion
- Unavoidable even if clocks get synchronized frequently

Illusion of synchronized clocks

- Clock drift: 17 sec drift for clocks synchronized once a day (Google)
- Back-shifts: clocks being forced to sync to past times
- Network delay: no synchronization can work around network delay
- Leap seconds: necessary time adjustment due to earth rotation
- Virtualization: VMs use virtualized clocks that pause if VM has no CPU time

Use Libraries for Time-Calculations!

Tom Scott

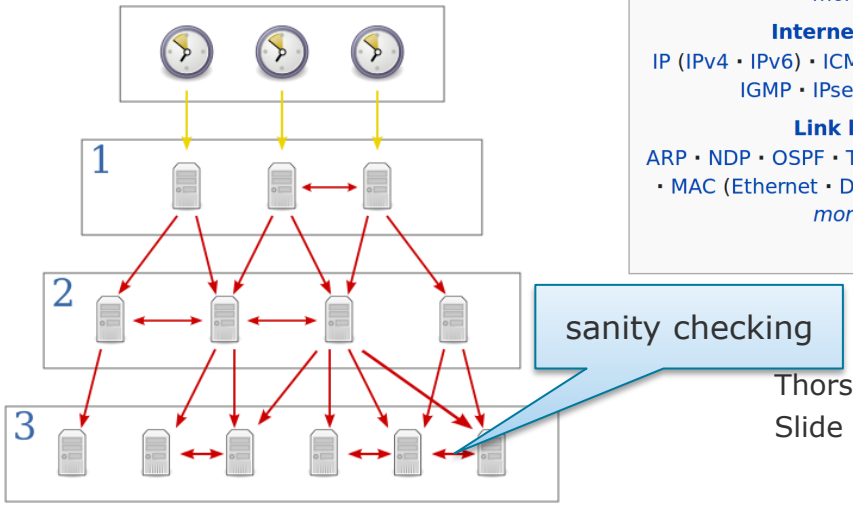
https://www.youtube.com/watch?v=-5wpm-gesOY

Synchronization

David L. Mills

Network Time Protocol (NTP)

- Most popular clock synchronization protocol for packet-switched, variable-latency data networks
- Assumption:
  - Some nodes (server) have very precise clocks (atomic, GPS, …)
- Protocol:
  - Nodes with less precise clocks synchronize their clocks with these reference clocks directly or indirectly
  - ➢ The closer a node is to the reference clocks, the more precise it can (potentially) sync its clock



Internet protocol suite

Application layer
BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · **NTP** · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer
TCP · UDP · DCCP · SCTP · RSVP · *more...*

Internet layer
IP (IPv4 · IPv6) · ICMP · ICMPv6 · ECN · IGMP · IPsec · *more...*

Link layer
ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · DSL · ISDN · FDDI) · *more...*

V · T · E

Synchronization

Network Time Protocol (NTP)

- Synchronization Algorithm:
  - Client nodes regularly poll server nodes and calculate:

1. time offset

$$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$$

Offset send

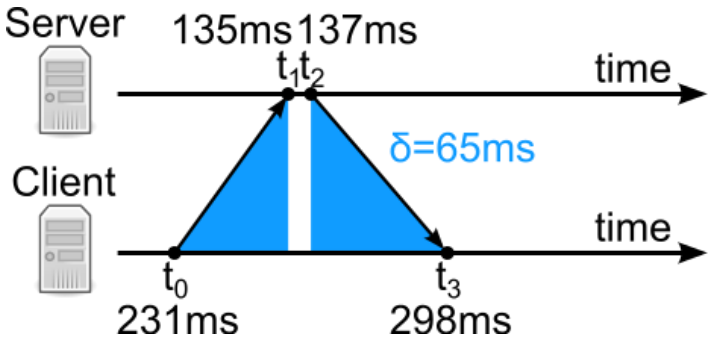
Offset receive

t_1 and t_3 include transmission time so it is added and deleted

Because we calculated the offset twice!

2. round-trip delay

$$\delta = (t_3 - t_0) - (t_2 - t_1)$$



t0, t1, t2, and t3 are timestamps attaches to the sync message

- θ and δ are passed through statistical analysis removing outliers
- Client then gradually adjusts its local clock using θ

Synchronization

Network Time Protocol (NTP)

- Most popular clock synchronization protocol for packet-switched, variable-latency data networks
- Computers synchronize their time with a group of servers
- Server get their time from more accurate time sources

Confidence in local time t

- Estimation about the deviation between local and system time
- A client's local time t can be expected to be t + uncertainty
- Uncertainty ≈ own expected clock drift since last NTP-sync + network round-trip time + server's uncertainty

➢ Systems that rely on synchronized clocks try to estimate uncertainty and incorporate it in their application logic

Internet protocol suite

Application layer
BGP · DHCP · DNS · FTP · HTTP · IMAP · LDAP · MGCP · NNTP · **NTP** · POP · ONC/RPC · RTP · RTSP · RIP · SIP · SMTP · SNMP · SSH · Telnet · TLS/SSL · XMPP · *more...*

Transport layer
TCP · UDP · DCCP · SCTP · RSVP · *more...*

Internet layer
IP (IPv4 · IPv6) · ICMP · ICMPv6 · ECN · IGMP · IPsec · *more...*

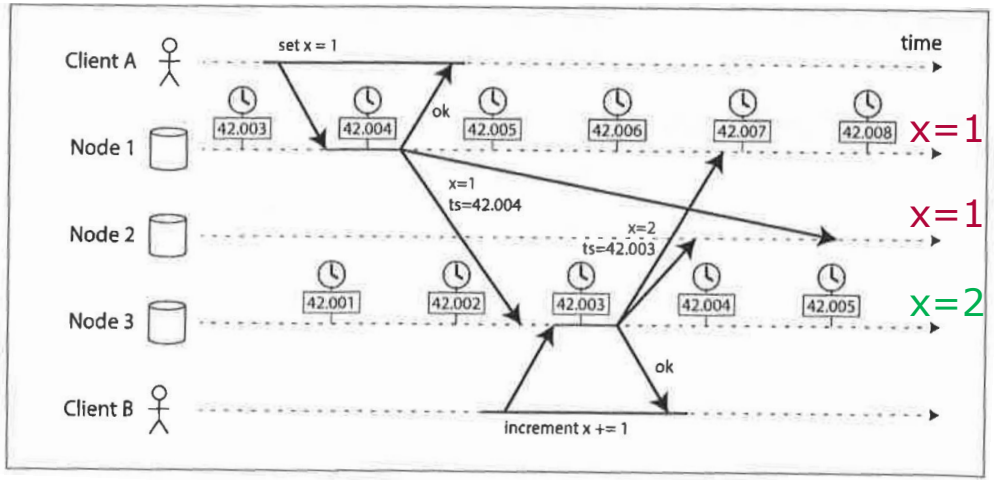
Link layer
ARP · NDP · OSPF · Tunnels (L2TP) · PPP · MAC (Ethernet · DSL · ISDN · FDDI) · *more...*

V · T · E

Risks

Synchronized clocks in distributed DBMSs

- Used often when messages require a global ordering



- Last-Write-Wins (LWW):
  - Writes get a timestamp from the first node that sees them
  - During change propagation, newer writes overwrite older writes
  - ➢ If clocks are out-of-sync, newer writes might get overwritten/dropped

- Snapshot isolation:
  - Transactions get a timestamp from the node that opens them
  - During transaction processing, transactions only see older changes
  - ➢ If clocks are out-of-sync, snapshots might be inconsistent

Unreliable Clocks

Risks

```
while (true) {
   request = getIncomingRequest();
   if (lease.expiryTimeMillis -
            System.currentTimeMillis < 10000) {
      lease = lease.renew();
   }
   if (lease.isValid()) {
      process(request);
   }
}
```

Better not pause here!

Synchronized clocks in distributed DBMSs

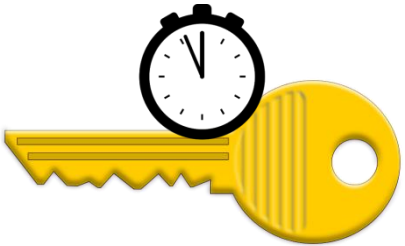
- Used often when messages require a global ordering

- Single-leader lease:
  - In single-leader replication, the leader obtains a lease with a timestamp
  - Lease:
    - Kind of a lock with timeout that can be hold by only one node
    - If leases timeout expires, the leader needs to renew the lease
    - If leader fails and does not renew, another leader can be elected
  - ➢ If clocks are out-of-sync, leader might hold lease for too long (two leader brain split)
  - ➢ If the leader **pauses and resumes** in a critical section, it might process writes without permission

Remember: no mutexes, semaphores, … in distributed systems!

Locking

Leases



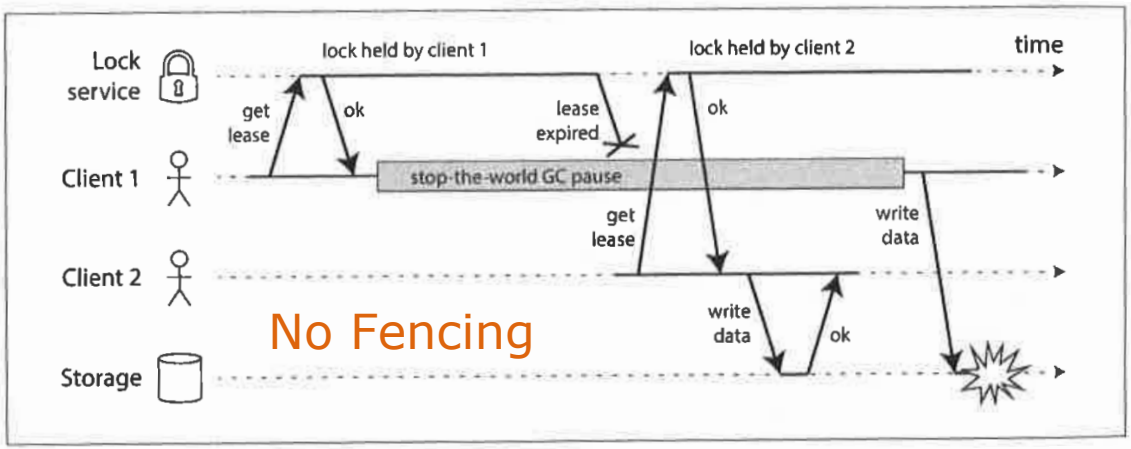
- Necessary if a system requires that there is only one of some thing:
  - One node with a certain permission for a particular resource
  - One node with a particular role in the system (e.g. leader)
- Obtaining a lease grants exclusive rights for a certain time
- Assumption:
  - One node (lock service/server/authority) assigns locks/leases
- Fencing token:
  - A number that increases every time a lock is assigned
  - Handed to the lease owner as part of the lease
  - Lease owner must issue the fencing token with every action
  - Locked resource (!) checks if fence token is up-to-date (e.g. newest)
    - Reject if other node possesses newer fence token

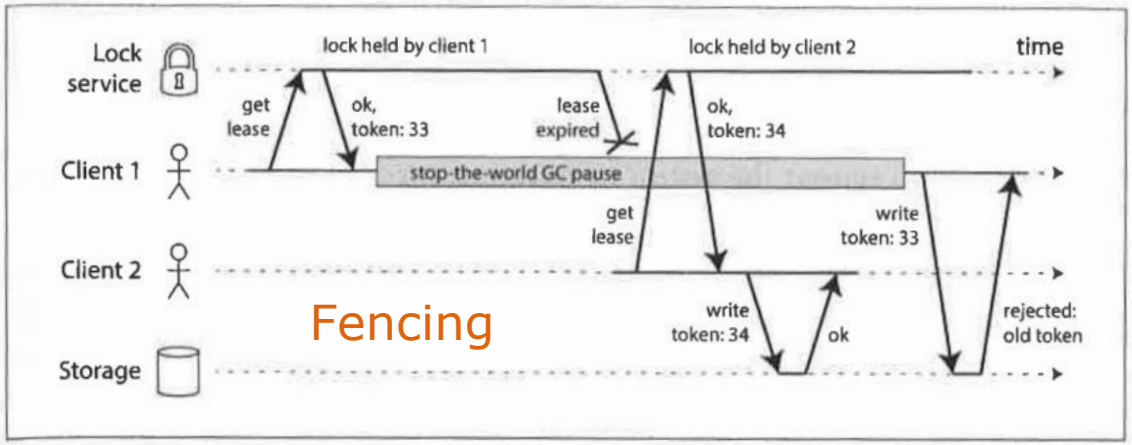
To counter the problem:
A node wrongly thinks
that it has the lock!

Locking

Leases

- Example:



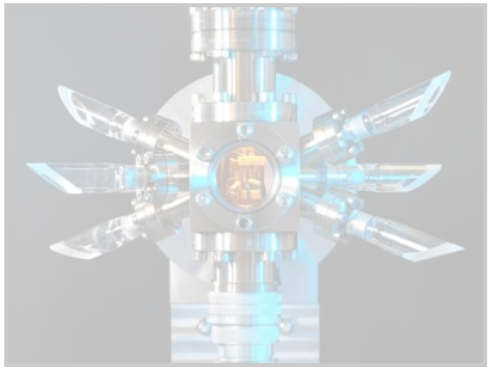


Distributed Systems

Unreliable Networks

A shark raiding an undersea cable

Unreliable Clocks



An atomic clock with minimum drift

Knowledge, Truth, Lies

Students communi-cating their knowledge

Facts: Knowledge and Truth

Knowledge

- A node can know nothing about other nodes for sure
  - Can only make guesses based on received messages

Truth

- Truth is defined by the majority
- Examples:
  - A node looses its connection to the network, but is still alive
    - The majority sees the node disappear and will declare it dead (although the connection and not the node was faulty)
  - A change propagation message gets lost on the network
    - The majority holds an outdated value that is declared valid (although the most recent value is on the node issuing the change)

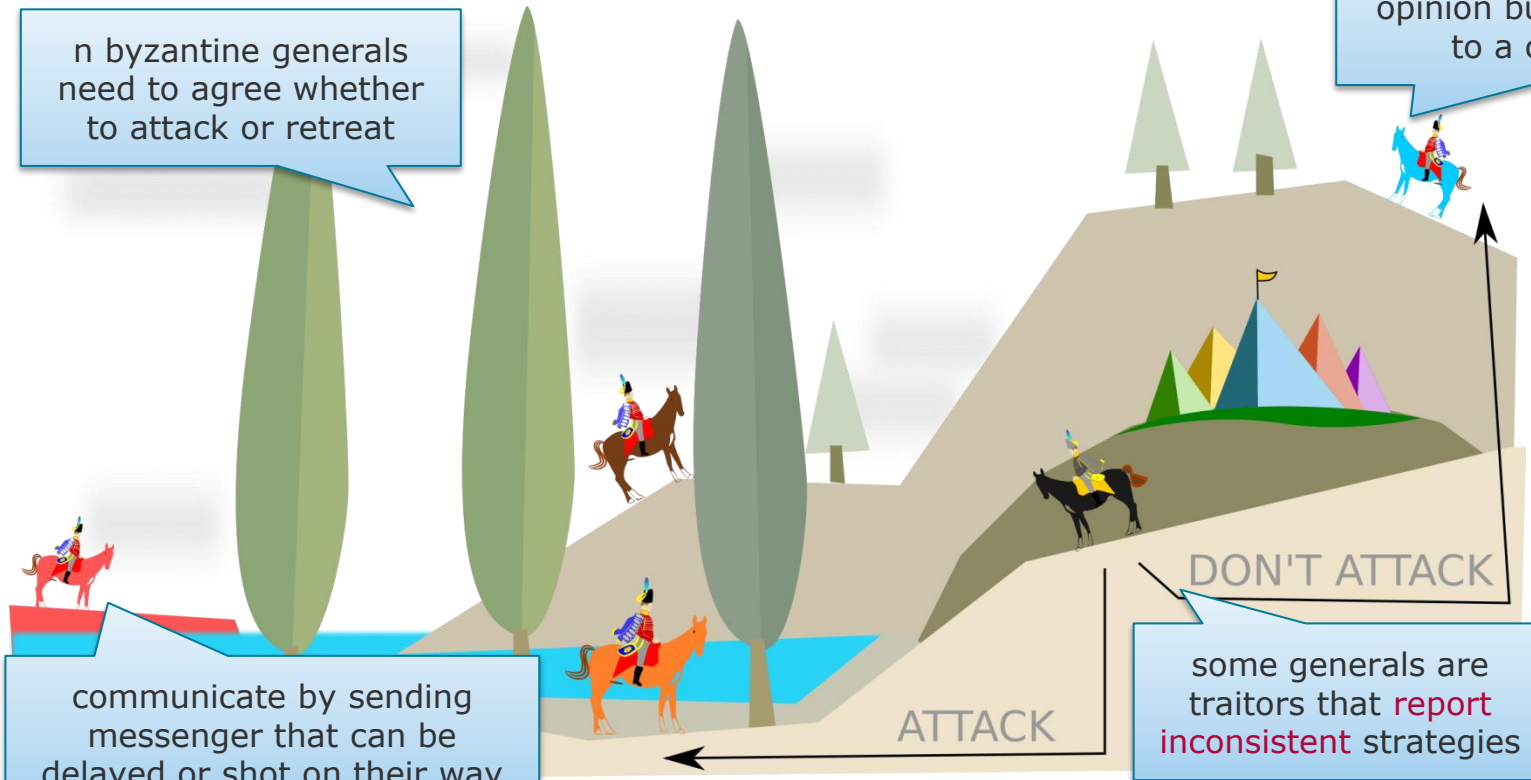
Determining Facts

Remember quorum reads and writes (quorum consistency) from chapter replication!

Quorum

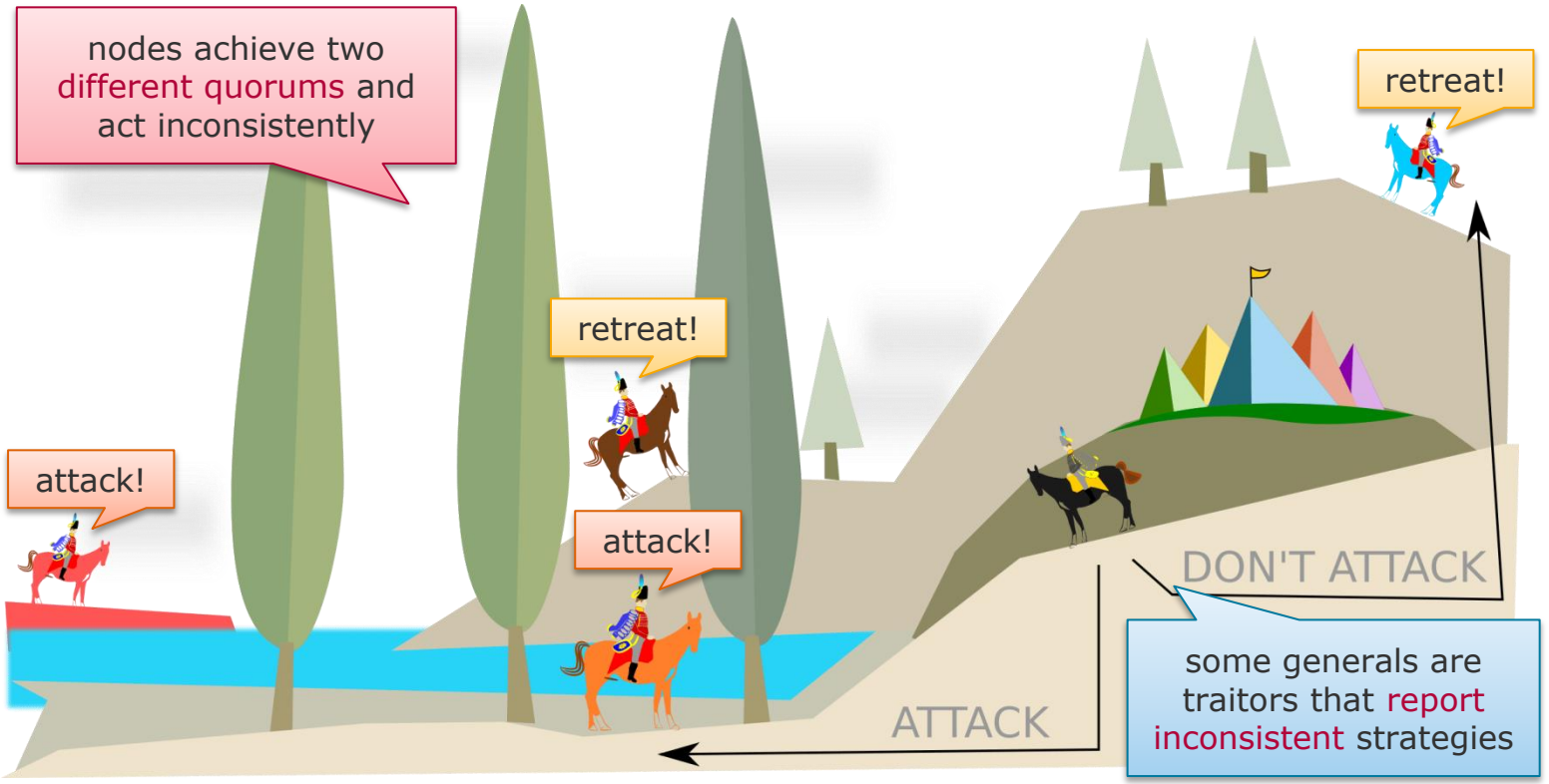
- Minimum number of nodes that must agree on a statement to be accepted
- Typically the majority (>50% of the nodes)
  - A majority quorum is safe, because there can be only one majority

Lies: Byzantine Problem



©https://blog.cdemi.io/byzantine-fault-tolerance/

Lies: Byzantine Fault



©https://blog.cdemi.io/byzantine-fault-tolerance/

Lies

Weak Lies

- Nodes accidentally send invalid information (with no bad intention):
  - outdated, miss-calculated, damaged, lost, …
- Reasons:
  - software bugs, signal interference, misconfiguration, hardware faults, …
- Protection:
  - checksums (e.g. TCP), redundancy (e.g. NTP), quorums (e.g. Cassandra), sanity checks (application), …

Byzantine Lies

- Nodes systematically send invalid information (usually with bad intention)
- Reasons:
  - hardware faults, security compromises, malicious attacks, …
- Protection:
  - complicated, often inefficient consensus protocols
    - hardware-based, multiple-consensus-rounds, consensus-hierarchies, proof of work …

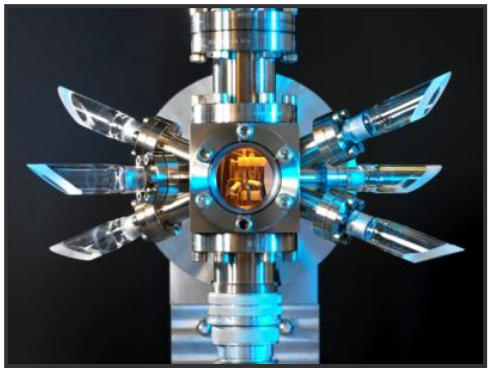
No issue in OLAP systems!

Summary

Unreliable Networks

A shark raiding an undersea cable

Unreliable Clocks



An atomic clock with minimum drift

Knowledge, Truth, Lies

Students communi-cating their knowledge

Unreliable Networks

- Messages can be lost, reordered, duplicated, and arbitrarily delayed

Unreliable Clocks

- Time is approximate at best, unsynchronized, and can pause


Harsh Winds of Reality
Clock Error
Process Pauses
Unbounded Network Delay
Distributed Systems