TDT4225
Chapter 8 – The Trouble with Distributed Systems

Svein Erik Bratsberg
Department of Computer Science (IDI), NTNU

Kunnskap for en bedre verden

Problems in Distributed Systems

- Working with distributed systems is fundamentally different from writing software on a single computer.
- Get a taste of the problems that arise in practice.
- Get an understanding of the things we can and cannot rely on.
- Software running on a single computer has predictable behaviour, unless there are bugs in the software
- Some parts of the system that are broken: partial failure
- This together with non-determinism makes distributed systems hard to work with

Cloud Computing and Supercomputing

- **High-performance computing (HPC).** Supercomputers with thousands of CPUs are typically used for computationally intensive scientific computing tasks
- **Cloud computing**: Multi-tenant datacenters, commodity computers connected with an IP network, elastic/on-demand resource allocation, and metered billing.
- These are different approaches to computing. HPC is much more controlled and is usually batch-oriented

HPC vs Cloud Computing

- CC: online, in the sense that they need to be able to serve users with low latency at any time
- HPC: Specialized hardware, where each node is quite reliable, and nodes communicate through shared memory and remote direct memory access (RDMA)
- CC: networks are often based on IP and Ethernet, arranged in Clos topologies
- CC: it is reasonable to assume that something is always broken. Advantegous if you can run with something broken.
- Network slow on geographically distributed systems
- Building a reliable system from unreliable components

Asynchronous packet networks, no response?

- Your request may have been lost
- Your request may be waiting in a queue and will be delivered later
- The remote node may have failed
- The remote node may have temporarily stopped responding
- The remote node may have processed your request, but the response is lost
- The remote node may have processed your request, but the response has been delayed

No response?

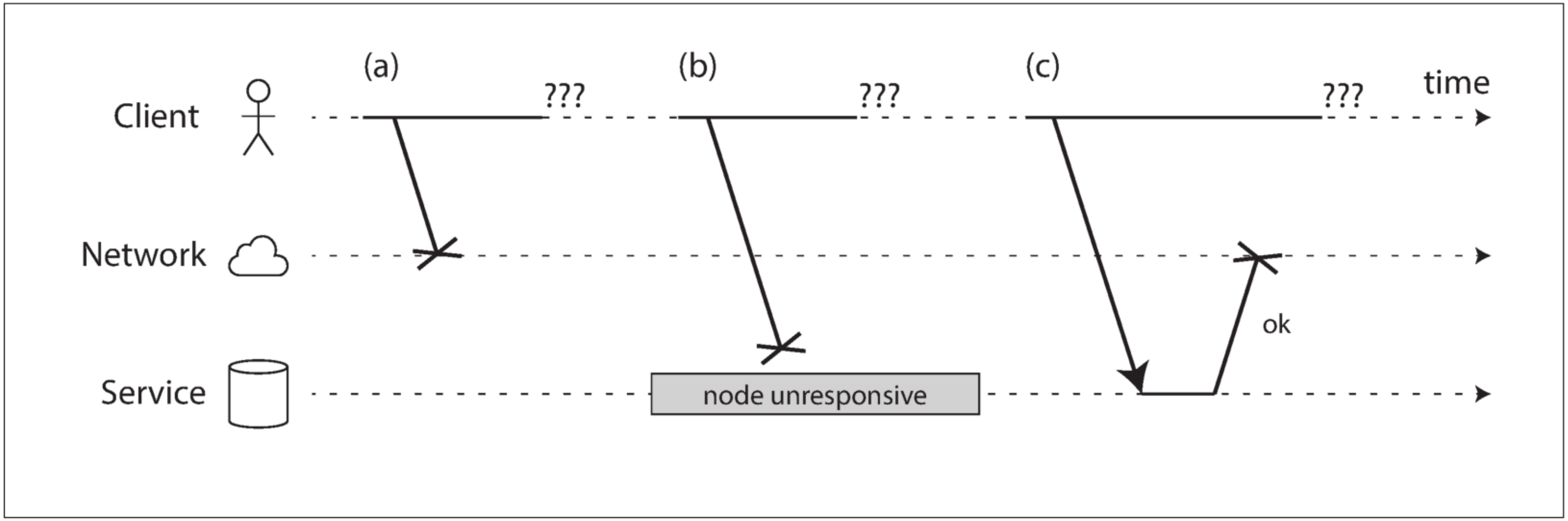


Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

Network Faults in Practice

- Networks are unreliable and all sorts of faults may occur
- Human misconfigurations is the biggest cause
- Fault detection may be necessary
  - A load balancer needs to stop sending requests to a node that is dead
  - In a distributed database with single-leader replication, if the leader fails, one of the followers needs to be promoted
- Some feedback is possible
  - On crash of server process: The OS will helpfully close or refuse TCP connections by sending a RST or FIN reply
  - If a node’s process crashed but the node’s OS is still running, a script can notify other nodes about the crash

Timeouts and Unbounded Delays

- If a timeout is the only sure way of detecting a fault, then how long should the timeout be?
- A long timeout means a long wait until a node is declared dead
- A short timeout detects faults faster, but carries a higher risk of incorrectly declaring a node dead
- When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network.
- Most systems we work with have no guarantees: asynchronous networks have unbounded delays

Network congestion and queueing

- Many computers send packets to the same destination, the network switch must queue them up and feed them into the destination network link one by one

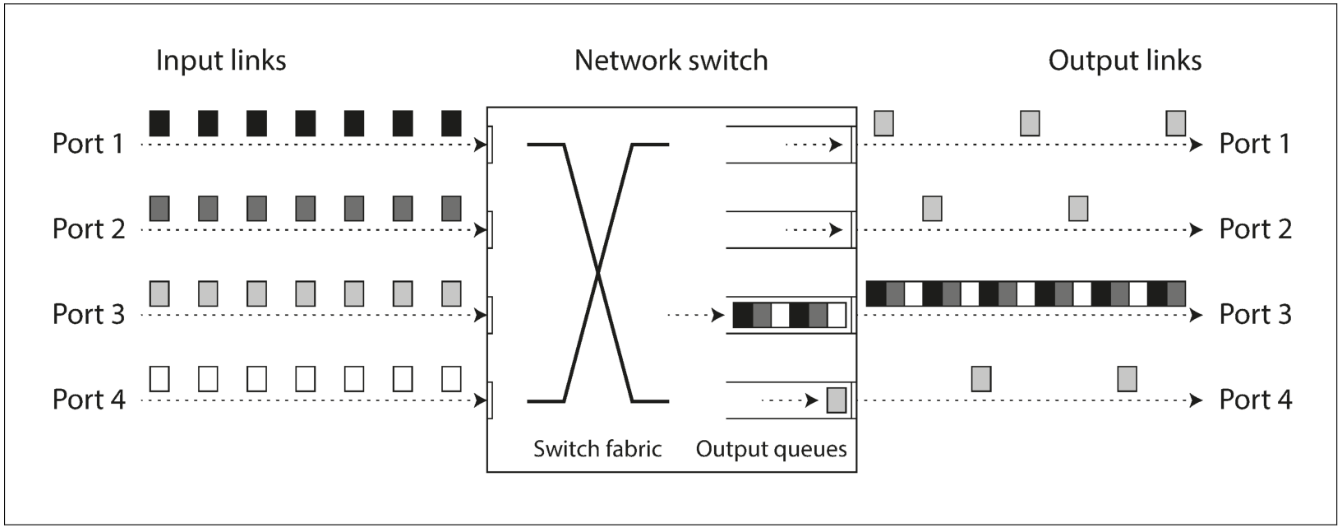


Figure 8-2. If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2, and 4 are all trying to send packets to port 3.

Network congestion and queueing (2)

- If all CPU cores are currently busy, the incoming request from the network is queued by the operating system until the application is ready to handle it.
- In virtualized environments, a running operating system is often paused for tens of milliseconds
- TCP/UDP may add delays at sender node (to wait for more data, or to limit sending data)
- **Timeout**: Systems can continually measure response times, and automatically adjust timeouts according to the observed response time distribution

Predictable networks

- Like the fixed phone lines?
- Packet switching? Optimized for bursty traffic.
- Hard to estimate traffic: If you guess too low (bandwith), the transfer is unnecessarily slow, leaving network capacity unused. If you guess too high, the circuit cannot be set up.
- Variable delays in networks are not a law of nature, but simply the result of a cost / benefit trade-off.

Unreliable Clocks

- Clocks used for duration
- Clocks used for point in time
- The time when a message is received is always later than the time when it is sent, but due to variable delays in the network, we don’t know how much later.
- Makes it difficult to determine the order in which things happened when multiple machines are involved.
- Network Time Protocol (NTP) to synchronize clocks
- Servers may use GPS receivers.

Monotonic vs. Time-of-Day Clocks

- Time-of-day clocks
  - Wall-clock time
  - System.currentTimeMillis()
  - clock_gettime(CLOCK_REALTIME)
- Monotonic clocks
  - Suitable for measuring a duration
  - clock_gettime(CLOCK_MONOTONIC)
  - System.nanoTime()
  - Used for measurements
  - Cannot be compared between computers

Clock Synchronization and Accuracy

- The quartz clock in a computer: Drift up to 17 seconds a day
- Local clock with difference from NTP clock: Refuse to synchronize or applications need to accept clock adjustments
- NTP synchronization can only be as good as the network delay
- Some NTP servers are wrong or misconfigured
- Leap seconds
- In virtual machines, the hardware clock is virtualized
- If you run software on devices that you don’t fully control, you cannot trust the clock

Relying on Synchronized Clocks

- Clocks work quite well most of the time, but robust software needs to be prepared to deal with incorrectness.
- Incorrect clocks easily go unnoticed.
- If you use software that requires synchronized clocks, it is essential that you carefully monitor the clock offsets.

Timestamps for ordering events

- A dangerous use of time-of-day clocks in a database with multileader replication: last write wins (LWW)

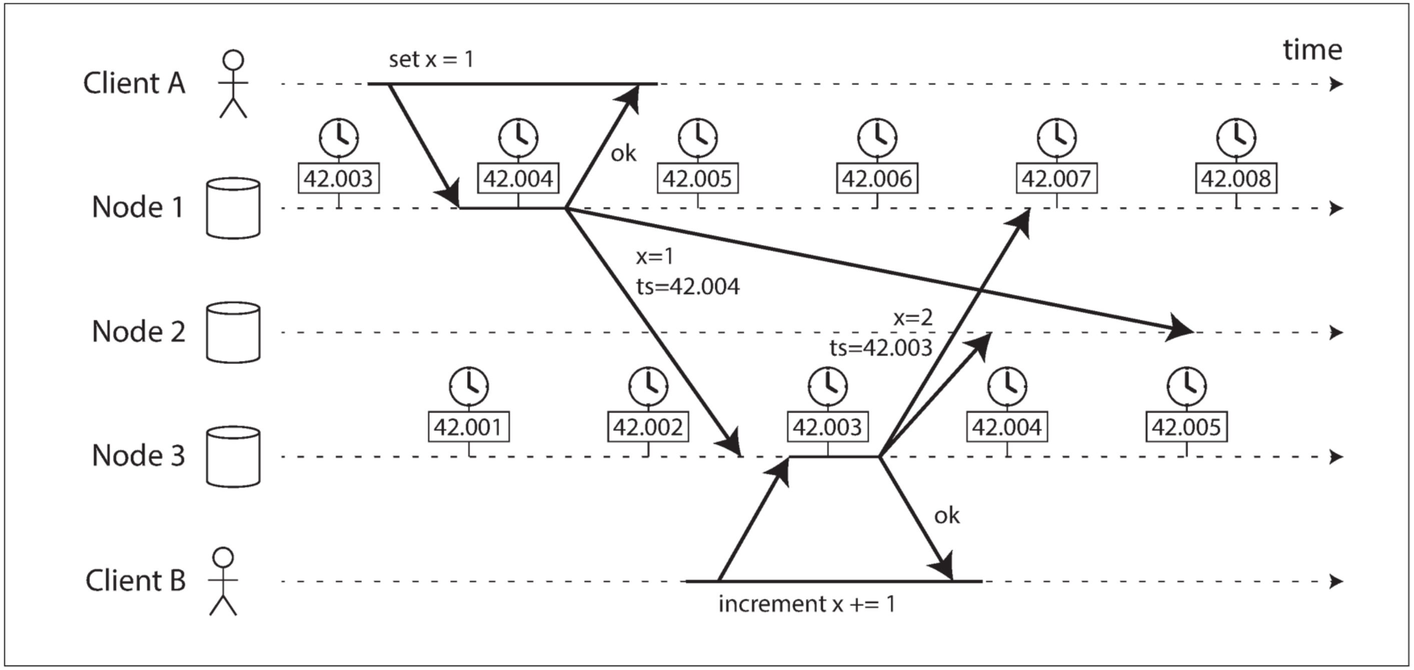


Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

Accuracy of last write wins

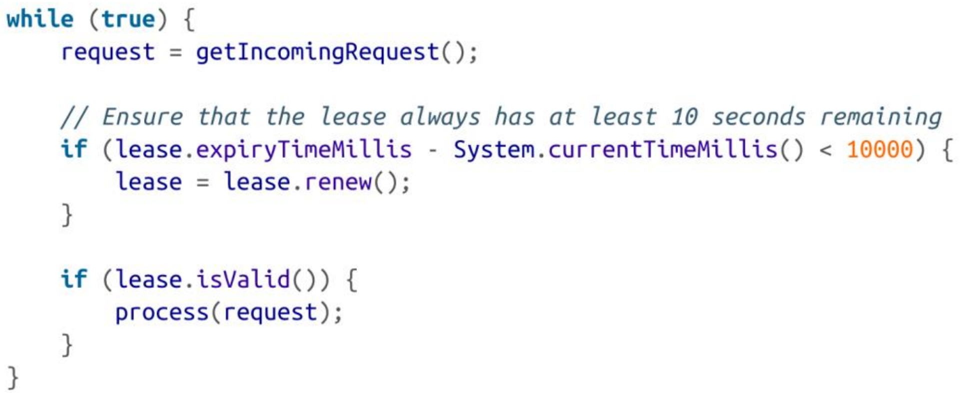
- Could NTP synchronization be made accurate enough that such incorrect orderings cannot occur?
- Probably not, because NTP’s synchronization accuracy is itself limited by the network round-trip time.
- Logical clocks based on incrementing counters are a safer alternative for ordering events.
- Clock readings have a confidence interval
- NTP server gives best possible accuracy is tens of milliseconds, and the error may easily spike to over 100 ms when there is network congestion
- Most systems don’t expose clock uncertainty

Synchronized clocks for global snapshots and Google Spanner

- Snapshot isolation: Allows read-only transactions to see the database in a consistent state at a particular point in time
- Distribution: A global, monotonically increasing transaction ID is difficult to generate.
- With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck.
- Spanner implements snapshot isolation across datacenters using its TrueTime API.
- Using overlapping intervals as undefined order.

Process Pauses

- Leaders obtain a lease with a timeout from the other nodes



```
while (true) {
    request = getIncomingRequest();

    // Ensure that the lease always has at least 10 seconds remaining
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

- It’s relying on synchronized clocks.
- The code assumes little time passes between the point that it checks the time and the time when the request is processed.

Response time guarantees

- Hard real-time systems: There is a specified deadline
- A real-time operating system (RTOS) that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals is needed.
- C programs
- Developing real-time systems is very expensive, and they are most commonly used in safety-critical embedded devices
- Garbage collection is a challenge in real-time

Knowledge, Truth, and Lies

- Distributed systems: No shared memory, only message passing via an unreliable network with variable delays, and the systems may suffer from partial failures, unreliable clocks, and processing pauses.
- The truth is defined by the majority
- A node cannot necessarily trust its own judgment of a situation
- Quorum: voting among the nodes. Consensus algorithms.

The leader and the lock

- Some situations require only one node to be the leader, hold a lock to a particular resource

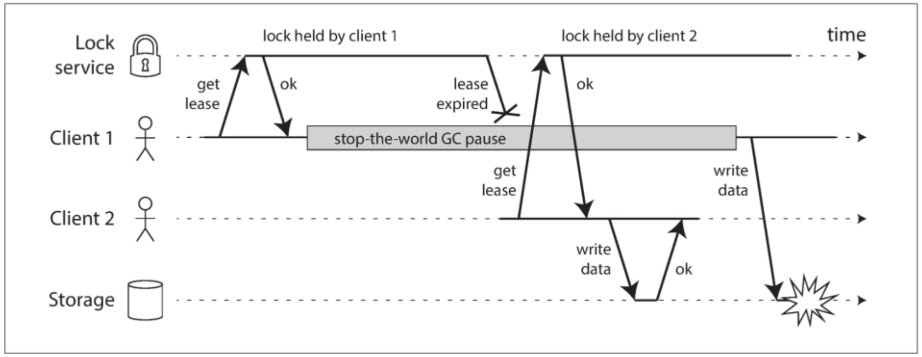


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

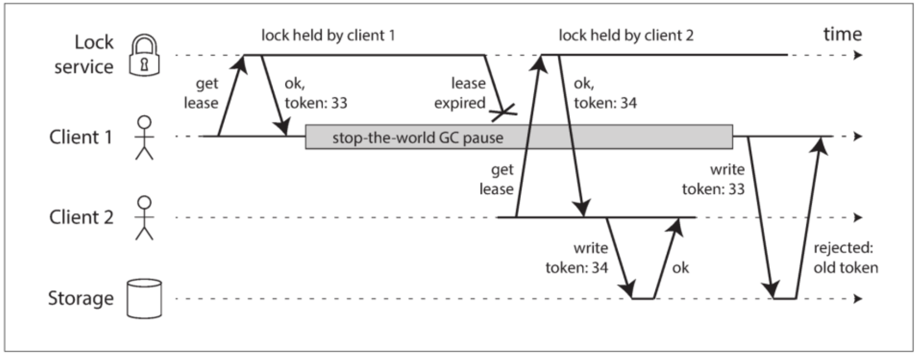


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

- Fencing tokens: An increasing number is get every time a lease is given
- ZooKeeper offers such properties

Byzantine faults

- We assume that nodes are unreliable, but honest: they may be slow or never respond, and their state may be outdated. But they never “lie”.
- If a node tells a “lie”, everything becomes harder: Byzantine faults.
- Aerospace applications need to tolerate Byzantine faults (cosmic radiation)
- Bitcoin / blockchains need to tolerate Byzantine faults (fraud)
- A bug in the software could be regarded as a Byzantine fault, but if you deploy the same software to all nodes, then a Byzantine fault-tolerant algorithm cannot save you.

Handling «weak forms of lying»

- *Errors due to software or hardware bugs*:
- Checksums on messages
- Checking input values
- NTP clients must connect to multiple servers to estimate errors in time

System Model and Reality - timing

- Synchronous model: Bounded network time, bounded clock error, bounded process pauses (not used because assumptions do not hold)
- Partially synchronous model: Usually synchronous, but may experience and tolerate glitches. Mostly used.
- Asynchronous model: No timing asymptions.

System Model and Node failures

- Crash-stop faults: Node fails by crashing
- Crash-recovery faults: Node fails, but recovers from persistent medium
- Byzantine faults: Nodes may do absolutely anything.

Correctness of an algorithm

- **Example**: Lock using fencing token. Properties:
- *Uniqueness*: No two requests for a fencing token return the same value.
- *Monotonic sequence*: If request **x** returned token **tx** , and request **y** returned token **ty** , and **x** completed before **y** began, then **tx < ty**
- *Availability*: A node that requests a fencing token and does not crash, eventually receives a response.

Safety and liveness (math properties)

- **Safety**: Nothing bad happens
- **Liveness**: Something good eventually happens.
- If a safety property is violated, we can point at a particular point in time at which it was broken. After a safety property has been violated, the violation cannot be undone—the damage is already done.
- A liveness property may not hold at some point in time, but may eventually hold
- For distributed algorithms, it is common to require that safety properties always hold
- Liveness: A request needs to receive a response only if a majority of nodes have not crashed.