Distributed Systems

Azza Abouzied

We are moving towards a distributed world

Why?

1. Inter-connected world
2. Too much data
3. Too much computation
4. Backup/Redundancy

Examples of distributed systems

1. Network File Systems
2. Domain Name System (DNS): A hierarchy that maps domain names (nyuad.nyu.edu) to ip addresses (50.19.90.233)
3. HP Clusters & Data Centers

Defining Characteristic: Message Passing

Distributed Systems are hard

Reliability

Time ordering

1. Which write happened first?
2. Writes are not atomic
   - In memory you are sure that a write instruction is atomic!
   - Message Passing (Sending a message takes time)
   - How to get Distributed Mutual Exclusion?

Failures and inconsistencies

Why Mutual Exclusion is not enough? Distributed Commit Protocol

Agreement is hard!

What happens without a centralized coordinator?

Mutual Exclusion

Mutual Exclusion via Central Coordinator

A process sends a request message to the coordinator. The coordinator marks the lock as acquired and responds with a reply message.

When the process is done with the lock, it releases it with a release message.

Bad things happen when:

1. Messages are lost,
2. A process fails.

Advantages

1. Coordinator can implement any scheduling algorithm
2. Only 3 messages are sent per entry into the critical section.

Mutual Exclusion via Token Passing

Some process gets a unique token. When that process is done with the lock (or if it doesn't need it), it passes the token on to some other process. Repeat forever.

Advantage: no central coordinator.

Disadvantages:

1. have to organize the processes into a ring to avoid starvation;
2. token can be lost (or worse, duplicated!);
3. if nobody needs the lock the token still spins through the ring.

Timestamp Algorithm

How to use time?

1. Process *p* generates a timestamp *t* and sends request(*p*, *t*) to all.
2. Each process *q* sends back a reply(*p*, *t*) message **only if**
   (a) *q* is not already in a critical section, and
   (b) *q* is idle or *q* has a higher timestamp than *p*.

   If *q* doesn't send a reply immediately, it queues *p*'s request until it can.

Why this works? For each pair of conflicting machines *p* and *q*, the one with the smaller timestamp won't send a reply to the other until it is done.

Advantage:

1. It guarantees deadlock-freedom and fairness
2. It uses $2(n-1)$ messages per critical section

Disadvantage: Poor scalability — need to know everyone!

Time

Why is time important?

Time lets you know which event happened before the other.

Properties of Distributed Timestamps

1. **Monotonicity**: increasing time
2. **Consistency**: A sends M @ 2:00 PM, B should not receive it @ 1:00 PM

Synchronized Clocks

A clock will always drift from its reference clocks because of different rates of progress.

Time servers use **atomic clocks** as they have little drift as little as a second every 1 million years. GPS clocks are synchronized up to ($\pm 10ns$) (based on satellite atomic clocks)

The Berkeley Algorithm

1. A *coordinator* is chosen via an *election process*.
2. Coordinator polls the site who reply with their time
3. Coordinator observes the *round-trip time* (RTT) of the messages and estimates the time of each site and its own.
4. Coordinator then averages the clock times, ignoring outliers
5. It then sends +ve or -ve update to each site to update its clock.

Synchronized Clocks

We use **Network Time Protocol** to synchronize clocks on the internet (synchronization to within a few milliseconds) and less than 1 ms on local networks.

Read Google's Spanner TrueTime for an even cooler time synchronization strategy!

Logical Clocks

Lamport Clocks

1. Counter that increases by 1 for every event
2. Send (M, t).
3. Receiver updates Clock to max (local time, t) + 1
4. On response, the sender will update its clock as well!

Distributed Atomic Transactions

Distributed Transactions

A transaction either occurs in full (i.e. every participant updates its local state) or not at all (no local state changes).

Mutexes not enough: What happens when a process holding the lock fails in critical section?

Solution: Two-phase commit protocol

1. All the processors participating in a transaction have to agree to commit
2. If protocol fails, rollback of each processor's state to what it was before the transaction started. *Recall Logging!*

2PC: Two Phase Commit

Phase 1

1. Coordinator writes prepare(T) to its log.
2. Coordinator sends prepare(T) message to all the participants in T.
3. Each participant replies by writing fail(T) or ready(T) to its log.
4. Each participant then sends a message fail(T) or ready(T).

Phase 2

1. Coordinator waits to receive replies from all participants or until a timeout expires.
2. If it gets ready(T) from every participant, it may commit the transaction by writing commit(T) to its log and sending commit(T) to all participants; otherwise, it writes and sends abort(T).
3. Each participant records the message it received from the coordinator in its log. In the case of an abort, it also undoes any changes it made to its state as part of the transaction.

2PC & Failures

What happens in 2PC when:

1. A site fails? It reads the redo log and it finds:
   (a) Commit
   (b) Abort
   (c) Ready
   (d) Fail
2. A coordinator fails?
   (a) Temporary failure
   (b) Permanent failure

Fischer-Lynch-Paterson impossibility result: In any distributed commit protocol the permanent failure of some process in an asynchronous system may cause the protocol itself to fail.

The CAP Theorem

Consistency: All nodes see the same data at the same time

Availability: a guarantee that every request receives a response about whether it succeeded or failed

Partition tolerance: the system continues to operate despite arbitrary message loss or failure of part of the system

No distributed system provides all three!

2PC chooses CP … why is A not supported?

1. A distributed system has to guarantee P so most systems tradeoff one of C or A.
2. Systems like twitter and Facebook opt for A instead of C. why? Manifestations of this?

Dealing with Permanent Failure

Agreement Protocols

The key algorithm is **Paxos**

Relies on Failure Detectors Basically, a mechanism that allows us to determine if a participant has died by waiting for a timeout (Don't wait forever!)

Paxos solves the problem of **agreement**,

- All the participants agree on some value
- After some finite amount of time (**termination**),
- This value is proposed by at least one of the participants (**validity**).

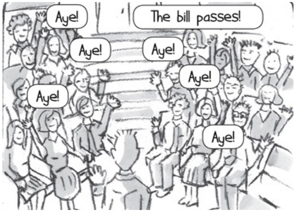
Agreement gives us distributed commit: Agree on commit or abort

Wait! How can we get an agreement if it is impossible for two processes?

The Amazing world of Distributed Systems

Byzantine Generals, Failures, Networks

Gossip Networks, Message Authentication



Quorums

Questions?