Distributed Systems

11. Consensus

Paul Krzyzanowski
pxk@cs.rutgers.edu
Consensus Goal

- Allow a group of processes to agree on a result
  - All processes must agree on the same value
  - The value must be one that was submitted by at least one process (the consensus algorithm cannot just make up a value)
We saw this!

- Mutual exclusion
- Election algorithms
- Other uses:
  - Manage group membership
  - Distributed commit
  - Synchronize state

- General problem:
  - Get unanimous agreement on a given value
Achieving consensus is easy!

• Designate a system-wide coordinator to determine outcome

• BUT … this assumes there are no failures … or we are willing to wait indefinitely for recovery

• Example:
  – 2-phase commit protocol (indefinite wait)
  – All of the mutual exclusion algorithms we studied
Faults

• Three categories
  – transient faults
  – intermittent faults
  – permanent faults

• Any fault may be
  – fail-silent (fail-stop)
  – Byzantine

• synchronous system vs. asynchronous system
  – Synchronous = system responds to a message in a bounded time
  – E.g., IP packet versus serial port transmission
Agreement in faulty systems

Two army problem

- Good processors - faulty communication lines
- Coordinated attack
- Infinite acknowledgement problem
Agreement in faulty systems

Byzantine Generals problem

– reliable communication lines - faulty processors
– $n$ generals head different divisions
– $m$ generals are traitors and are trying to prevent others from reaching agreement
  • 4 generals agree to attack
  • 4 generals agree to retreat
  • 1 traitor tells the 1st group that he’ll attack and tells the 2nd group that he’ll retreat
– can the loyal generals reach agreement?
Agreement in faulty systems

Byzantine Generals problem

- Solutions require:
  - \( \geq (3m+1) \) participants for \( m \) traitors (\( 2m+1 \) loyal generals)
  - \( m+1 \) rounds of message exchanges
  - \( O(m^2) \) messages
- Costly solution!

- Variation: use signed messages
  - Messages from loyal generals cannot be forged/altered
  - Traitors can still lie
  - Consensus can be achieved with \( \geq (m+2) \) loyal generals
Agreement in faulty systems

• It is impossible to achieve consensus with asynchronous faulty processes
  – There is no way to check whether a process failed or is alive but not communicating (or communicating quickly enough)

• We have to live with this
Consensus Goal

• Create a fault-tolerant consensus algorithm that does not block if a majority of processes are working

• Goal: agree on one result among a group of participants
  – Processors may fail (some may need stable storage)
  – Messages may be lost, out of order, or duplicated
  – If delivered, messages are not corrupted

• The algorithm needs \((2P+1)\) processors survive the simultaneous failure of \(P\) processors
Paxos
Paxos

• Fault-tolerant distributed consensus algorithm
• Goal: all machines agree on a proposed value
  – The value can be associated with an event
  – Paxos ensures that no other machine associates the value with another event
Paxos players

• **Client**: makes a request

• **Proposers**:  
  – Get a request from a client and run the protocol  
  – **Leader**: elected coordinator among the proposers

• **Acceptors**:  
  – multiple processes that remember the state of the protocol  
  – Quorum = any majority of acceptors

• **Learners**:  
  – When agreement has been reached by acceptors, a Learner executes the request and/or sends a response back to the client
Paxos in action

Goal: have all acceptors agree to a value $v$ associated with a proposal

Paxos nodes: one machine may serve several roles
Paxos in action: Phase 0

Client sends a request to a proposer

Client sends a request to a proposer
Paxos in action: Phase 1a

Proposer (leader) creates a proposal #N (N acts like a Lamport time stamp), where N is greater than any previous proposal number used by this proposer.

Send to Quorum.
Paxos in action: Phase 1b

Acceptor: if proposer’s ID > any previous proposal
   reply with highest past proposal # & promise to ignore all IDs < N
   else ignore (return NACK – proposal is rejected)
Paxos in action: Phase 2a

Proposer: if proposer receives enough promises
Set a value $v$ to the proposal. $v$ can be $\max(\text{previous values}) + 1$
Send $\text{Accept Request}$ to quorum with the chosen value
Acceptor: if the promise still holds, then register the value $v$
  Send `Accepted` message to Proposer and every Learner else ignore the message (or send `NACK`)

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```

```
Accept Request(N, v)
```
Paxos in action: Phase 3

Learner: Respond to client and/or take action on the request
Paxos: Keep trying

• A proposal may fail
  – because the acceptor may have made a new promise to ignore all proposals less than some value >N
  – Because there are conflicts in Prepare messages from different proposers
  – Because a proposer does not receive a quorum of responses: either promise (phase 1b) or accept (phase 2b)

• Algorithm then has to be restarted with a higher proposal #
Paxos summary

• Paxos allows us to ensure consistent (total) ordering over a set of events in a group of machines
  – Events = commands, actions, state updates

• Each machine will have the latest state or a previous version of the state
Paxos summary

• To make a change to the system
  – Tell the proposer (leader) the event/command you want to add
  – Note: these requests may occur concurrently

  – The proposer picks its next highest event ID and asks all the acceptors to reserve that event ID
    • If any acceptor sees it’s not the latest event ID, it rejects the proposal; the proposer will have to try again with another event ID

  – When the majority of acceptors accept the proposal, accepted events are sent to learners, which can act on them (e.g., update system state)
    • Fault tolerant: need $2k+1$ servers for $k$ fault tolerance
Replicated state machines

• We want high scalability and high availability
• Achieve via redundancy
• High availability means replicated functioning components will take place of ones that stop working
  – Active-passive: replicated components are standing by
  – Active-active: replicated components are working
• Model system as a state machine – and replicate it
  – Input to a specific state produces deterministic output and a transition to a new state
  – To ensure correct execution & high availability
    • Each process must see the same inputs
    • Obtain consensus at each state transition
Leasing versus Locking

• Common approach:
  – Get a lock for exclusive access to a resource

• But: locks are not fault-tolerant

• It’s safer to use a lock that expires instead
  – Lease = lock with a time limit
  – Example:
    • three-phase commit vs. two-phase commit
    • Remote objects in .NET or Java

• Trade-off
  – Long lease with possibility of long wait after failure
  – Or Short leases that need to be renewed frequently
Hierarchical Leases

• For fault tolerance, leases should be granted by consensus
• Consensus isn’t super-efficient
• Compromise
  – Use consensus as an election algorithm to elect a coordinator
  – Coordinator is granted a lease on a large set of resources
  – Coordinator hands out sub-leases on those resources
• When the coordinator’s lease expires
  – Consensus algorithm is run again
The End