Lecture 24: Formalizing 2-Phase Commit & Reliable Transport (reintro)

November 21, 2019
Instructor: David E. Culler
https://cs162.eecs.berkeley.edu

* Back from the lofty heights of Kubernetes across the cloud
Recall: Cloud Native

Merging Two Kinds of Containers

**Docker**
- It's about **packaging**
- Control:
  - packages
  - versions
  - (some config)
- Layered file system
- => Prod matches testing

**Linux Containers**
- It's about **isolation**
  - performance isolation
- not security isolation
  - use VMs for that
- Manage CPUs, memory, bandwidth, …
- Nested groups

Google has been developing and using **containers** to manage our applications for over 10 years.

Google has launched 4B containers per week:
- simplifies management
- performance isolation
- efficiency

- Always-On, Always evolving, System of systems
- Upon elastic sea of resources
- Namespaces, their management, isolation, sharing
**Recall: Kubernetes**

### Kubernetes API

- **Nodes** are the virtual (or physical) machines that Kubernetes can schedule containers onto.
- **Pods** are the smallest unit of execution. It is a set of one or more containers that is always deployed together and onto the same node.
- **Deployments** are an object that manages a set of identical pods. It is comprised of a specification for a pod and the number of replicas.
- **Services** expose pods to the network and act as a load balancer for pods registered under it.

### Capture design patterns of gigantic-scale services in declarative fashion

- In Distributed Key-Value store

---

**Reconciliation**: Components keep a local version of the objects and watches the database every period for updates. Then changes local / actual state to match the desired state.
Recall: Key-Value store

• Key Value Store: Simple put and get operations
  – Fault tolerance: replication
  – Scalability: Add nodes, balance load, no central directory
  – Consistency: Quorum consensus for better performance

• Consensus Goal: Everyone agrees on the state of the distributed system
  – Doesn’t depend who you ask
  – Doesn’t matter if nodes go down

• Example: etcd in kubernetes
Recall: KV Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures

Master/Directory

<table>
<thead>
<tr>
<th></th>
<th>N1, N3</th>
<th>N1, N3</th>
<th>N50</th>
</tr>
</thead>
<tbody>
<tr>
<td>K5</td>
<td>N2</td>
<td>K14</td>
<td>N1, N3</td>
</tr>
<tr>
<td>K105</td>
<td>N50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N1, N3: put(K14, V14)

N1: put(K14, V14), N1

N50: put(K14, V14)
Recall: Scaling Up Directory

- Directory contains number of entries equal to number of key/value pairs in entire system
  - Could be tens or hundreds of billions of pairs

- Solution: Consistent Hashing
  - The set of storage nodes may change dynamically
    » fail, enter, leave
  - Assign each node a unique ID in large namespace \([0..2^m-1]\)
    » \(m\) bit namespace, s.r., \(M \ll 2^m\)
    » Each node can pick its ID at random!
  - hash keys in a manner that everyone assigns same range of IDs to a node
  - Each (key, value) stored at node with smallest ID larger than \(\text{hash(key)}\)

- Important property: Adding a new bucket doesn't require moving lots of existing values to new buckets
Recall: Replication & Consistency

• Replication is essential for fault tolerance and performance
  – But introduces inherent challenges
• Need to make sure a value is replicated correctly

• How do you know a value is replicated on every expected node?

• \textit{Wait} for acknowledgements from all expected nodes ???
  – What if they “fail”
Recall: General’s Paradox

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
- No, even if all messages go through

General 1

11 am ok?
Yes, 11 works

So, 11 it is?

Yeah, but what if you
Don’t get this ack?

General 2
Recall: Two-Phase Commit

• We can’t solve the General’s Paradox
  – No simultaneous action
  – But we can solve a related problem

• Distributed Transaction: Two (or more) machines agree to do something or not do it atomically

• Extra tool: Persistent Log
  – If machine fails, it will remember what happened
  – Assume log itself can’t be corrupted
2-Phase Commit Algorithm

• One coordinator & N workers (replicas)

• First Phase: Coord Initiates & Workers Decide
  – Coordinator “asks” all workers if they can commit
  – Sends VOTE-REQ to all workers
  – Each worker decides to accept or abort, records its decision in its log, then sends “VOTE-COMMIT” or “VOTE-ABORT” to coordinator

• Second Phase:
  – If coordinator receives “VOTE-COMMIT” from all workers, it broadcasts “GLOBAL-COMMIT” (unanimous approval)
    Otherwise coordinator broadcasts “GLOBAL-ABORT”
  – Workers obey the GLOBAL messages
Algorithm – at a glance

Coordinator Algorithm

- Coordinator sends VOTE-REQ to all workers
- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn’t receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

- Wait for VOTE-REQ from coordinator
- If ready, send VOTE-COMMIT to coordinator
- If not ready, send VOTE-ABORT to coordinator
  - And immediately abort
- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort
Formalizing Two-Phase Commit

• *N* workers (replicas): actually perform transactions
  – *Local decision is committed to the log*
  – *Message reflecting log entry may be sent multiple times*

• One coordinator (may also serve as a worker)
  – Asks each worker to vote on transaction
  – Tells every machine result of the vote (workers don’t need to ask each other)
Messages in Two-Phase Commit

Coordinator → Worker
- VOTE-REQ

Worker → Coordinator
- VOTE-COMMIT
- VOTE-ABORT

Coordinator → Worker
- GLOBAL-COMMIT
- GLOBAL-ABORT

No taking back: always logged before sending

Actual result of transaction attempt
State Machines

• Distributed systems are hard to reason about
• Want a *precise* way to express each node’s behavior that is also *easy to reason about*

• One approach: State Machine
  – Every node is in a *state*
  – When the node receives a message (or timeout),
  – it *transitions* to another state
  – Taking some actions and
  – Sends zero or more messages
Aside: State Machines in SW…

• Are everywhere in building systems
  – Socket Protocol State Machine
  – WC: if (in_word) {handle char; may terminate word} else {…}
  – Page Fault Handling
  – Buffer Cache Handling
  – …

• Loop over “events”
• Case dispatch on state for each event
• Each arm handles what to with events when in that state
  – Take action
  – Transition to a next state

• In hardware the state transition is atomic by design, in software we use mechanisms like a log to provide atomicity
Coordinator’s State Machine

INIT
  Recv: START
  Send: VOTE-REQ

WAIT
  Recv: VOTE-ABORT
  Send: GLOBAL-ABORT
  Recv: all VOTE-COMMIT
  Send: GLOBAL-COMMIT

ABORT

COMMIT
Coordinator’s State Machine

1. INIT
   - Send: VOTE-REQ

2. WAIT
   - Recev: START
     - Send: VOTE-REQ
   - Recev: VOTE-ABORT
     - Send: GLOBAL-ABORT
   - Recev: all VOTE-COMMIT
     - Send: GLOBAL-COMMIT

Triggers change of state
Coordinator’s State Machine

- After completing commit/abort, transition to INIT to await next START (arc not shown)
Worker’s State Machine

- After completing commit/abort, transition to INIT to await next VOTE-REQ (arc not shown)
Protocol: cooperating state machines

- **INIT**
  - Recv: START
  - Send: VOTE-REQ
  - Recv: all VOTE-COMMIT
  - Send: GLOBAL-COMMIT

- **WAIT**
  - Recv: VOTE-REQ
  - Send: VOTE-ABORT

- **ABORT**

- **COMMIT**

- **READY**
  - Recv: VOTE-REQ
  - Send: VOTE-COMMIT
  - Recv: GLOBAL-ABORT
  - Recv: GLOBAL-COMMIT

- **INIT**
  - Recv: VOTE-REQ
  - Send: VOTE-COMMIT
  - Recv: GLOBAL-ABORT
  - Recv: GLOBAL-COMMIT

- **READY**
  - Recv: VOTE-REQ
  - Send: VOTE-COMMIT
  - Recv: GLOBAL-ABORT
  - Recv: GLOBAL-COMMIT
Example: Failure-Free 2PC (w/ state)

Coordinator

Worker 1

Worker 2

Worker 3

Time
Dealing with Worker Failures

- Failure only affects states in which the coordinator is *waiting* for messages.
- In WAIT, if coordinator doesn’t receive \( N \) votes, it times out and sends GLOBAL-ABORT.

**Diagram:**

- INIT
  - Recv: START
    - Send: VOTE-REQ (start timer)

- WAIT
  - Recv any: VOTE-ABORT or T.O.
    - Send: GLOBAL-ABORT
  - Recv all: VOTE-COMMIT
    - Send: GLOBAL-COMMIT

- ABORT
- COMMIT
Example of Worker Failure (w/ state)

- Coordinator
- Worker 1
- Worker 2
- Worker 3

Diagram:
- INIT
- WAIT
  - ABORT
  - COMM
- VOTE-REQ
- VOTE-COMMIT
- GLOBAL-ABORT
- Timeout
- Time
Dealing with Coordinator Failure

- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
  - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send GLOBAL_* message

```
Recv: VOTE-REQ
Send: VOTE-ABORT

Recv: VOTE-REQ
Send: VOTE-COMMIT

Recv: GLOBAL-ABORT

Recv: GLOBAL-COMMIT

INIT

READY

ABORT

COMMIT
```
Example Coordinator Failure: W Ready

- **INIT**
- **READY**
- **ABORT**
- **COMM**

Coordinator

Worker 1

Worker 2

Worker 3

Votes:
- VOTE-REQ
- VOTE-COMMIT

Global:
- GLOBAL-ABORT

Block waiting for coordinator

Restarted
Failure Recovery

• Nodes need to know what state they are in when they come back from a failure
• How? Log events on local disk, SSD, NVRAM
• Then we have the following recovery rules:
  – Coordinator *aborts* transaction if it was in the INIT, WAIT, or ABORT states
  – Coordinator *commits* transaction if it was in COMMIT
  – Worker *aborts* if in INIT or ABORT states
  – Worker *commits* if it was in COMMIT state
  – Worker “*asks*” coordinator what to do if in READY state
• A worker waiting for global decision can ask fellow workers about their state
  – If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
    » Thus, worker can safely abort or commit, respectively
  – If another worker is still in INIT state then both workers can decide to abort
    » At least one will log & vote ABORT
  – If all workers are in ready, need to BLOCK
    » don’t know if coordinator decided to abort or commit, i.e., what’s in its log?
Example Coordinator Failure – W INIT

Coordinator

Worker 1

Worker 2

Worker 3

INIT

READY

ABORT

COMM

VOTE-REQ

timeout

VOTE-ABORT

timeout

timeout
Blocking for Coordinator

• What if *both* coordinator and a worker fail?
• The remaining workers can still consult each other
• But they can’t reach a conclusion on what to do!

Why?
• If all workers in INIT, we still don’t know state of failed worker \( w \)
• \( w \) may have been first to be notified of a commit, and then coordinator and \( w \) crashed

• This problem motivated *Three Phase Commit*
Distributed Consensus

- Two- (and Three-) Phase commit make a decentralized decision
- Example: Changing the value of a key among all replicas for the key
- But they are hardly the only solutions to this problem
Better Agreement in Face of Failure

• Idea: If a majority of nodes agree, commit
• If a minority don’t participate, ignore them

• Assumes a *fail-stop* model: if a machine breaks, it won’t send/receive messages

• **Algorithms that do this:** Paxos, Raft
  – Very tricky (but some will tell you it’s “Simple”)
Why a Majority?

- Key property: Overlap
- Suppose use transactions to track a value, initially 0

\[
\begin{array}{cccccc}
A: & 0 & B: & 0 & C: & 0 \\
D: & 0 & E: & 0 & & \\
\end{array}
\]

- We run transaction “+2” while D, E are down

\[
\begin{array}{cccccc}
A: & 2 & B: & 2 & C: & 2 \\
D: & 0 & E: & 0 & & \\
\end{array}
\]
Why a Majority?

• Now $D, E$ come back up, $A, B$ go down

• Our overlap in this case is $C$
  – Guaranteed by choice of majority

• Overlap prevents us from losing transactions
  – Means every node is responsible for resending missed updates
  – Not just the value, but the transaction ID so we know we’re behind

A: 2  B: 2  C: 2  D: 0  D: 0
Beyond Fail-Stop

• What if nodes don’t just stop talking when they fail?
• What if they send incorrect information?
• Or what if nodes are actively malicious?

• This is called the Byzantine Failure Model
• Do Paxos/Raft still work even if just a minority of nodes exhibit Byzantine failures? No
• One general, \( N-1 \) lieutenants
• Some number \((F)\) want chaos
  – Can say contradictory things
• Goal: General sends order and
  1. All non-insane lieutenants obey the \textit{same order}
  2. If the commanding general is sane, they obey his/her order
Byzantine Generals: Impossibility

• No solution when $N = 3$

• General Theorem: Must Have $N \geq 3F + 1$
Byzantine Generals: Solutions

• There are protocols that solve Byzantine generals for $N \geq 3F + 1$ (the lower bound)
• Original Algorithm: $O(2^N)$ messages!

• Castro and Liskov (1999): “Practical Byzantine Fault Tolerance”: $O(N^2)$ messages
  – Still a lot worse than Paxos/Raft
  – Also a lot more complicated
Summary: Two-Phase Commit

• Consensus Goal: Everyone agrees on the state of the distributed system
  – Doesn’t depend who you ask
  – Doesn’t matter if nodes go down

• Distributed Transactions
  – Atomic, can’t revert once agreement is reached

• Voting protocol requires unanimity

• Transaction committed if and only if: all workers and coordinator vote to commit

• Nodes never take back their vote
  – Logged for durability

• Nodes work in lock step (for an item)
  – Don’t perform new transactions until old one is resolved
  – Stall until transaction is resolved