FDPaxos: Server Failure Detection with Paxos

Final Report

Github Repository

Commit ID: `d36d93f`

CPSC 416 2021W2

April 19th, 2022

Members:

Sassan Shokoohi (sassansh)

Naithan Bosse (nbosse)

Johnny Li (johnnybc)

Felipe Caiado (fbcaiado)

Ken Utsunomiya (mikuh)

Jonathan Hirsch (jmhirsch)

Table of Content

Overview of the Problem and Motivation **3**
Background 3
Approach and Solution **3**
Server 4
Fcheck Acknowledging 4
Server Joining Process 4
Failure Monitoring 4
Server API 5
Paxos 5
Intro 5
Initialization 5
paxoslib.go (Networking / Communication) 5
Server-Paxos API 6
Paxos Algorithm 6
Phase 1: Prepare-Promise 6
Phase 2: Propose-Accept 6
Phase 3: Learn 7
Client 7
Cloud 7
What works / What doesn't **7**
Demo: The Part-Time Hospital **9**
References **10**
Appendix **11**
Tracing 11
cTrace 11
sTrace 11
pTrace 11
ShiViz 12
ShiViz Example 12
Bash Script 'cloudrun.sh' 13

Overview of the Problem and Motivation

In a distributed computing system, detecting failure of nodes is one of the key features needed to ensure the system remains operational. Typical failure detection algorithms rely on the response time of heartbeat messages, often by either having nodes send messages to prove they are alive, or having nodes respond to messages asking whether they are alive. However, such algorithms cannot differentiate between a failed node and other network problems that affect measured metrics such as round-trip-time (RTT). Indeed, a node that is temporarily unresponsive may not have completely failed, as it could be busy processing other requests or the network itself may be experiencing congestion or may be partitioned (Figure 1). This problem becomes more complex as nodes are partitioned into separate networks because the number of external, uncontrollable factors that can affect the RTT between two nodes increases greatly. As we have seen with A3, becoming aware of failures in the network, and handling them properly is crucial to ensure data integrity, even in a basic distributed key-value store system. However, as described above, it is clear that simple heartbeat-based failure detection algorithms do not scale well in larger networks.

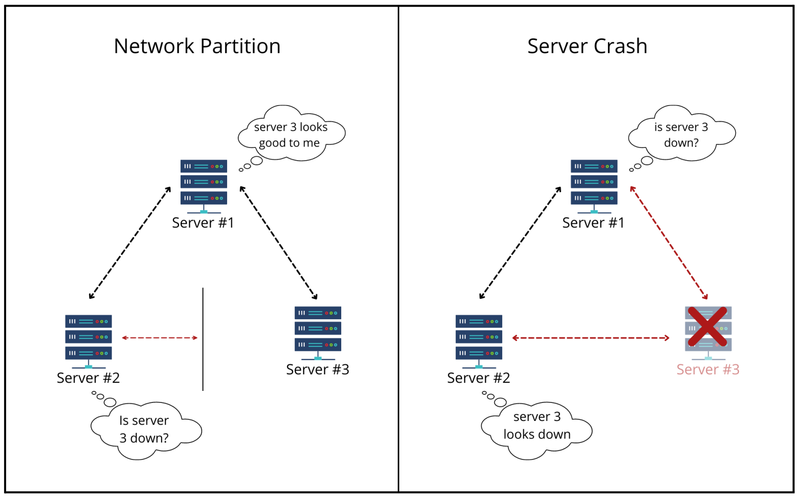


Figure 1. Illustrating a real server crash/failure vs a network partition.

We will explore the Paxos algorithm to understand how a failure-detection algorithm that can better differentiate between real failures and network instabilities can enable systems such as a replicated KVS to scale both in number of nodes and across several subnets, potentially in different geographic areas.

Background

Consensus algorithms rely on multiple nodes agreeing on the state of the network to determine whether certain nodes are alive or have failed. Simple approaches base the state of the network on what is agreed upon by the majority of the nodes responsible for making these decisions. One such algorithm is the Paxos algorithm.

At a high level, Paxos relies on a set of roles called proposers, acceptors, and learners. Proposers propose changes to the network after receiving requests from a client. Acceptors 'cast votes' for proposed changes by accepting a proposed change and rejecting others. Learners then announce accepted changes to clients. Paxos proposers can also elect a leader that can guarantee Paxos will complete in instances where the algorithm is unable to make progress.

It is possible for some or all nodes to take on multiple roles in the network. Paxos is therefore able to combat false server-failure detections resulting from packet losses or delays, or slow servers by requiring that a majority of the nodes, known as a quorum, agree on the state of the network before committing to changes. This increases the confidence that an unresponsive node has truly failed and is not simply busy or slow due to network instabilities.

Approach and Solution

Paxos requires a quorum, or majority, of acceptors to choose a proposed value. Thus, the system consists of 2n+1 servers, where n is the maximum number of failures that the system can tolerate. That is, if n servers fail, the system can continue to operate as long as n+1 servers are running.

Server

Starting the system involves starting all the servers that make up the system. Each server can be started by calling its `Start` method, which commences the following steps:

1. Fcheck acknowledging
2. Server joining process
3. Failure monitoring

All servers must complete the above steps before any client operations may be issued to any server.

Fcheck Acknowledging

Each server will start their own instance of fcheck and allow itself to be monitored by any server in the system. When this step is completed, the server will be responsive to heartbeats. That is, the server will respond with an ack message to the sender of a heartbeat.

Server Joining Process

Each server starts with their own config file, which specifies their unique serverId, the total number of servers `N` in the system, and the IP addresses of those servers. `N` is fixed for the lifetime of the system.

Each server will call the RPC method `ServerRPC.Join` on every other server to confirm that all other servers have started. A server will resend RPC calls until it receives a response from the other server.

The server joining process is complete when `N` servers have started and made contact with each other.

Failure Monitoring

Each server will use fcheck to monitor every other server in the system for failure.

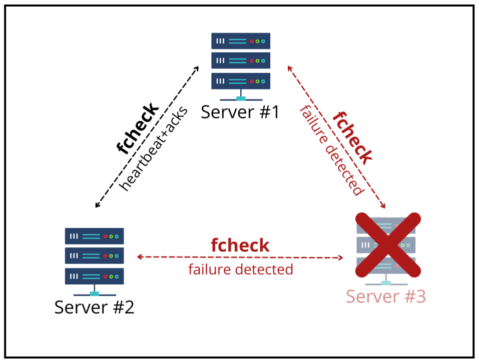


Figure 2. Example system with m=1.
3 servers monitoring each other

When a server's instance of fcheck notifies the server that another server has failed, the server will use Paxos to reach a consensus with the other servers to determine whether the failure is real.

For example, suppose Server A's fcheck instance detected that Server B has failed. Server A will begin the Paxos consensus process. Once consensus has been reached, there are 2 possible outcomes:

1. **A majority of servers consider Server B to have failed**
   - The system will no longer consider Server B to be alive
   - The system state will be updated to include Server B as a failed server. More specifically, the state in each server will be updated to the changed system state
   - The servers in the system will stop monitoring Server B for failure

2. **A majority of servers do not consider Server B to have failed**
   - The system will still consider server B to be alive
   - Server A's fcheck instance will continue to monitor server B for failure

In both scenarios, a server that is queried by a client node will respond with the latest system state. This invariant holds when the servers exist in different network partitions.

Note, the system will assume that server failures are crash-failure. A client will be able to use RPC method calls to query any server in the system for a list of servers that have failed as decided by Paxos.

Server API

`(s *Server) Start(...)`
Starts fcheck acknowledging, starts server joining process, and begins monitoring other servers for failure.

`(s *ServerRPC) Join(req, *res uint8) error`
A server RPC method that a server can call to check if another server has started. If a server has started, the server returns its serverId to the caller of the method. Otherwise, an error is returned. When the system starts, each server will call this method on every other server to check whether or not all N servers have started. N is the total number of servers specified in each server's configuration file.

`(s *ServerRPC) GetFailedServers(req, *res FailedServers) error`
A server RPC method that a client can call. If the server is alive, the server sets `res` to contain the current consensus about all failed servers. This method will be used by a client to query a server for the number of failed servers in the system.

Paxos

Intro

In our system, Paxos was implemented as a separate package called "`paxoslib`" which includes the paxos algorithm plus the networking layer needed for communication between the different servers' proposer, acceptor and learner. Some custom functionality was added to support failure detecting and handling. To make things organized, each role was separated into its own go module called `proposer.go`, `acceptor.go`, `learner.go`. Inside these modules, the method signatures are inspired by the following project: https://pkg.go.dev/github.com/kkdai/paxos.

Initialization

The `paxoslib` needs to be imported and initialized by each server when they are starting up. To do this, the server would call `NewPaxos()`to instantiate it. Next, a call is made to `Init(id uint8, serverAddr string, tracer *tracing.Tracer, failedServerCh chan Failed)` with the id of the server, the address to set up the RPC server for the Paxos communication, a tracer object, and a channel to be notified of dead servers. At this stage, the servers perform their normal joining protocol to verify that all the servers are up and joined the system. Once the joining is complete, a call is made to `InitializeState(paxosListenAddrList []string)`with a list of all the other server's addresses in order to initialize the state of all the other servers (alive) and create an RPC client for all of them. At this point `paxoslib` is ready and can be called by the server to start a paxos run to reach consensus on a dead server.

paxoslib.go (Networking / Communication)

Specifically, the `paxoslib.go` contains most of the networking layer between the 3 roles of proposer, acceptor and learner, plus the server-paxos API (covered in the next section). In terms of sending the prepare, propose and accept messages, the `paxoslib` uses asynchronous RPC calls to all other alive servers. However, since RPC calls guarantee an at-least-once semantics, we added a timeout feature in order to allow for a 0-or-more semantic to allow the system to make progress and not wait forever in the case of partitions.

Server-Paxos API

When the server detects a dead server through the use of fcheck, it calls `UpdateCurrentState(failedServer Failed)` with the detected failed server and the method return either true for confirming that consensus was reached and the server is considered dead by majority of servers in the system, or false for when consensus is not reached.

Another API call from the server to the `paxoslib` is the call to `GetGlobalState()`, which returns the most recent consensus-reached state about the livelihood of all the servers. This call is used for example when a client queries a server for a list of the failed servers.

Paxos Algorithm

Our paxos algorithm is broken into 3 phases. Phase 1, prepare-promise, is run when a failure is initially detected by one of the servers. Phase 2, propose-accept, begins only if a proposer receives a majority of promises. Phase 3, learn, begins only when the majority of acceptors agree with the state being proposed.

Phase 1: Prepare-Promise

When fcheck detects a potential failure, it is forwarded from the server to the paxoslib through `` `UpdateCurrentState(failedServer Failed) bool ``. This function will update it's local state to mark the server as failed, asynchronously call prepare on the local server's proposer, and start a 10 second timer. When the timer is done, it will check the state of the server in the global state. If the global state is unchanged, the local state will be reverted, and the server will be returned to fcheck.

The asynchronous call to the paxos algorithm begins in the `prepare(s State)` function of the proposer. The proposer will send a prepare call to every acceptor its paxoslib knows about, include their current SequenceNumber and wait for a response. If, during that time, it is found that the state it plans to propose is identical to its global state, prepare will be aborted and the paxos algorithm will return.

Acceptors receive prepare statements in their `receivePrepare(prepare Message) Promise` function. They will return a promise if the received SequenceNumber is greater than any other prepare statements they have seen before, or they will reject the promise with the highest sequence number they have seen otherwise.
The proposer then receives the promises. These are counted. In the event that the proposer does not receive a majority of promises, it will update its sequence number to the highest it has received, wait a random amount of time determined by the following random exponentially increasing backoff function `waitTime:= (2^p.numRejected)*100 + randIntn((300-25+1)+25)`, and attempt to send prepare statements again.

If the proposer receives a majority of promises, it will move on to Phase 2.

Phase 2: Propose-Accept

At the beginning of Phase 2, a proposer that has received a majority of promises will send a proposal to all acceptors from whom it has received a promise. This message will include the proposed state and the sequence number the proposer received promises for.

Acceptors will receive proposals in `ReceivePropose(proposeMsg Message) bool`. At this point in the Paxos algorithm, we chose to deviate slightly from the generic Paxos algorithm. Acceptors will only accept a proposal if the promise's sequence number is the highest they have seen and their local state matches the proposed state. The latter is necessary because Paxos is being used both to ensure that the state is identical across all servers and to ensure that the failure is a true failure seen by the majority of servers, as opposed to simply a partition or temporary failure witnessed by a minority of the servers. If the Acceptor accepts the proposal, it will be broadcast using `paxoslib.endBroadCast(m Message)` to all known learners. Otherwise, it will ignore the proposal and do nothing. The return value is used for tracing purposes.

Phase 3: Learn

The final phase begins when an acceptor broadcasts a state. Learners will receive the broadcast in their `ReceiveAccept(acceptMsg Message)` function. Here, acceptors will verify that the received sequenceNumber is higher than any other seen before sequenceNumber from the acceptor sending the broadcast, if so they will tally the number of broadcasts received for that sequenceNumber as they arrive from all the acceptors. Once the count reaches majority for that sequence number, a call is made to `paxoslib.setLearnedState(state State, failedServer Failed)` to update the global and local state.

Client

The client acts as an outside observer that queries the system to learn which servers have failed. When the client script is executed, the main function instantiates a new client and begins using the client to query the system for failed nodes using the GetSystemState client API method in a loop. The loop continues either until GetSystemState returns an error, or the script is terminated.

The client API consists of the following methods:

`NewClient(tracer *tracing.Tracer, clientId string, ServerAddrList []string) *Client`
Constructs a new client. The Client uses the `ServerAddressList` to query servers in the `getSystemState` method and to perform failover.

`(c *Client) GetSystemState() ([]Failed, error)`
Queries a server for a list of failures by calling the `GetFailedServers` Server API RPC method. If the call to the queried server returns an error, then the client queries the next server, and so on until the client either receives a response, in which case it returns the result and a nil error value, or it is unsuccessful at reaching any server in the system, in which case it returns an error. If the server receives a list of failures, it updates a local variable with the results and will skip these servers if it performs a failover in a future call.

(c ***Client**) `CloseConnection`() ([]**Failed**, error)
Close any existing RPC clients.

Cloud

Our team created 13 virtual machines (VMs) in our [416 Azure resource group](), 1 VM for a client, 1 VM for tracing, and 11 VMs for independent servers. All of our VMs were created using an Ubuntu image and installing only the apt make, golang, and unzip packages. To enable our VMs to connect to each other, we have allowed all inbound and outbound connections. We created independent configurations for local and cloud deployment to enable quick development iterations. VMs are deployed, started, and stopped through Azure. We have also implemented a Bash script that handles initializing servers, uploading code to the servers, starting and killing servers, the tracing servers and the cloud client, and also handles partitioning servers from each other. The script is explored in more detail in the appendix and in the README file in our repository.

What works / What doesn't

We have successfully accomplished all of the features stated in the final proposal

- Exactly 11 servers nodes can join the system
- The system can operate normally with at exactly 11 server nodes

- Definition of normal system operation:
    - All relevant client and server events/actions should be logged in the terminal
    - All server nodes in the system are live
    - Each server node uses fcheck to monitor the status of all other server nodes, by sending heartbeat messages to check if a server node has failed or not.
    - Each server node will allow itself to be monitored by the other server nodes and respond with an ack message upon receiving a heartbeat.
    - When a client queries a live server for the system state, the server should return an empty list of failed servers

- Server node failures/partitions can be emulated on the terminal with a bash script
- Expected system response to server node failure:
    - All relevant client and server events/actions should be logged in the terminal
    - The local failure history in the server node that had been monitoring the failed node should be updated to include the failed node
    - After Paxos consensus, the global failure history in all live server nodes should be updated to include the failed node. In addition, the server node being queried by the client should return a list of all failed server nodes
    - The failed server node's local failure history would be lost. However, the global failure history in all live nodes should be consistent

- Expected system response to server joins:
    - All relevant client, server, and fcheck events and actions should be logged in the terminal
    - Server nodes should ignore client requests for system state until all server nodes have joined the system
    - When the server joining process is complete, the total number of live server nodes in the system should be equal to N, where N is the expected number of servers, specified in the configuration file
    - The system should be live and responsive to client requests for system state once the expected number of server nodes have joined. A server node is considered to have joined the system when it is monitoring all other nodes for failure using fcheck heartbeats and replying with acks in response to heartbeats from other server nodes

Demo: The Part-Time Hospital

We will present FDPaxos as a system modeling the steps taken by Paxon doctors to diagnose their patients as "deceased". We will demonstrate the following aspects of the system:

1. Cloud deployment
   - Brief overview of how the system is deployed on the cloud. (E.g. number of servers, client, tracing server) and introduction to the cloud shell script.
2. Join process
   - Demonstrate N servers joining the system at start up using tracing actions printed to the console output.
   - Demonstrate that the client is not able to query the servers before the join process has completed
3. Normal operation
   - Start the client and demonstrate that the client is now able to query the server for a list of failed nodes. The list will be empty at this point.
   - Show that the appropriate client tracing actions are printed to the console
   - Briefly describe the role of fcheck in our system.
4. Basic failure detection
   - Kill server 1 and demonstrate (via the console) that the node is detected as having failed.
   - Show that the client fails over to server 2 and that future query responses include the failed node.
5. Consensus with network partitions
   - Create a network partition between two nodes and show that neither node is detected as failed.
   - Create a network partition between one node and a majority of other nodes and show that the one partitioned node is detected as failed by consensus.
6. Killing n+1 servers
   - Slowly kill n+1 of the 2n+1 total servers and demonstrate that only the first n failed servers are detected as having failed.
7. Tracing
   - Present a newly created ShiViz diagram and tracing logs and show that the recorded tracing actions match our semantics.

References

- Failure Detectors and Extended Paxos for k-Set Agreement:
https://www.microsoft.com/en-us/research/wp-content/uploads/2007/05/prdc07-kSet.pdf
- Paxos Made Simple: http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf
- ShiViz: https://bestchai.bitbucket.io/shivi

Appendix

Tracing

Tracing v.2

We have introduced tracing semantics using tracing library, and we have defined three 3 types of traces and some remarkable actions recorded in each type of trace:

cTrace

cTrace records client actions. In particular, cTrace records an action whenever the client sends or receives a query and whenever the server receives or replies to this query. It also records actions related to failover in the event of a connection error or timeout.

- `GetFailedServersReq`: `recorded when client issues a query to one of the servers`
- `GetFailedServersResRecvd:` `recorded when client receives the response to the query`

sTrace

Each server in the system receives an sTrace. sTrace records actions related to server initialization (e.g. on server start, at the beginning of the join process, when a server has joined, and when all servers have joined) and records an action whenever a server's local fcheck instance has identified a failure.

- `ServerJoining:` `recorded when a server is about to join the system`
- `ServerJoined:` `recorded when a server has joined the system`
- `AllServersJoined:` `recorded when a server is connected to every other server`
- `ServerFailureDetected:` `recorded when fcheck detects and notifies a server failure`

pTrace

pTrace records actions related to Paxos. Actions are recorded when a proposer sends a prepare or a proposal, when an acceptor makes a promise, chooses or rejects a proposal, or broadcasts a consensus result, and when a learner learns a result.

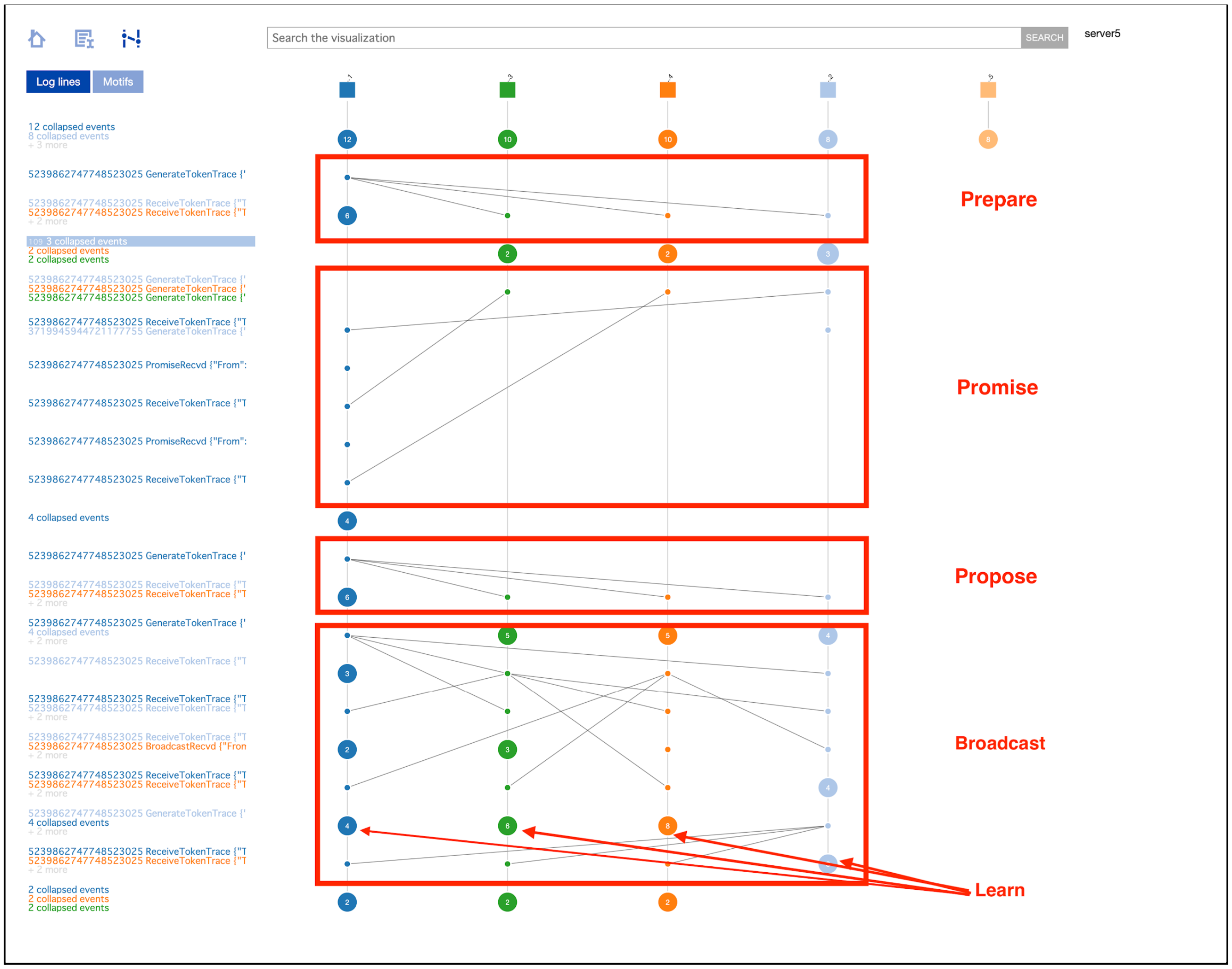
- `PaxosInit:` `recorded when Paxos algorithm is triggered`
- `PrepareReq:` `recorded when a proposer sends Prepare message to every other server`
- `PromiseReq:` `recorded when Paxos an acceptor sends Promise message to the proposer`
- `ProposeReq:` `recorded when a proposer sends Propose message to every acceptor`
- `BroadcastReq:`
  `recorded when every server sends Broadcast message to every other server`
- `ProposalChosen:`
  `recorded when an Acceptor accepts the proposal`
- `ProposalRejected:`
  `recorded when an Acceptor rejects the proposal`
- `Learn:` `recorded when each server updates its local and global state of servers`

*For a full description of the above trace actions and their constraints visit the following tracing action documentation.

ShiViz

To visualize our trace log, we have utilized [ShiViz](#) .

[ShiViz Example (5 servers + server No.5 failure)](#) ([Click here to view larger image](#))



```
12 collapsed events
8 collapsed events
+ 3 more

5239862747748523025 GenerateTokenTrace {'

5239862747748523025 ReceiveTokenTrace {"T
5239862747748523025 ReceiveTokenTrace {"T
+ 2 more

109 3 collapsed events
2 collapsed events
2 collapsed events

5239862747748523025 GenerateTokenTrace {'
5239862747748523025 GenerateTokenTrace {'
5239862747748523025 GenerateTokenTrace {'

5239862747748523025 ReceiveTokenTrace {"T
3719945944721177755 GenerateTokenTrace {'

5239862747748523025 PromiseRecvd {"From":

5239862747748523025 ReceiveTokenTrace {"T

5239862747748523025 PromiseRecvd {"From":

5239862747748523025 ReceiveTokenTrace {"T

4 collapsed events

5239862747748523025 GenerateTokenTrace {'

5239862747748523025 ReceiveTokenTrace {"T
5239862747748523025 ReceiveTokenTrace {"T
+ 2 more

5239862747748523025 GenerateTokenTrace {'
4 collapsed events
+ 2 more

5239862747748523025 ReceiveTokenTrace {"T

5239862747748523025 ReceiveTokenTrace {"T
5239862747748523025 ReceiveTokenTrace {"T
+ 2 more

5239862747748523025 ReceiveTokenTrace {"T
5239862747748523025 BroadcastRecvd {"From
+ 2 more

5239862747748523025 ReceiveTokenTrace {"T
5239862747748523025 ReceiveTokenTrace {"T
+ 2 more

5239862747748523025 GenerateTokenTrace {'
4 collapsed events
+ 2 more

5239862747748523025 ReceiveTokenTrace {"T
5239862747748523025 ReceiveTokenTrace {"T
+ 2 more

2 collapsed events
2 collapsed events
2 collapsed events
```

Figure 3. Shiviz diagram depicting a 5 server system, detecting the failure of server 5 and reaching consensus. ([Link to shiviz log file](#))

Above, we can see at the very top the servers all join the system (inside of the 8-12 combined events). Next, we see that server 5's traces are stopped, this is because server 5 was terminated. Immediately after, server 1 detects this failure first and starts to send out 4 **prepare** messages (highlighted in a red box with prepare label beside it), one to itself and to the other 3 servers (servers 2, 3 and 4). All of them respond back with promises as highlighted in the next red box. Since this server now has majority promises, it starts its **propose** phase to all the servers that gave a promise. This is highlighted in the next red box with the propose label. Next, all the servers that received this propose, begin their **broadcast** (accept) to all the nodes. Lastly, the **learn** phase is taking place, as the servers receive majority broadcasts, they learn that server 5 is dead.

Bash Script ‘cloudrun.sh’

The script ‘`cloudrun.sh`’ can be run in single-command mode or in interactive mode. The interactive mode is more comprehensive and can be accessed using the ‘`-r`’ or ‘`--run`’ options. To view the commands in the single-command mode, please run the script with the ‘`-h`’ or ‘`--help`’ options. Below are the interactive commands supported by the script:

```
#separate server numbers in <[servers]> with a space#

help | h - prints this help message
setup | i <[servers]> - sets up specified servers
ssh <server> - ssh into server, 'c' for client, 't' for tracing server
upload | u <path> - uploads folder at <path> to all servers"
exit | q - exits interactive mode
showPorts | sp <[server]> - shows ports used by specified server, 'c' for cloud client,
      't' for tracing server. Runs until canceled

client | c - start the cloud client
start | s - <[servers]> starts specified servers
startAll | sa - starts all servers except the client
startAndTrace | st <[servers]> - starts the tracing server and specified servers
trace | t - starts tracing on server
gettrace | gt - copies the traces from the tracing server to the current directory

kill | k <[servers]> - kills specified servers
killall | ka - kills all servers
killClient | kc - kill the cloud client
killTrace | kt - kills tracing server

partition | p <server1> <[servers]> - partitions server <server1> from server
<[servers]>
unpartition | up <server1> <[servers]> - unpartitions server <server1> from <[servers]>
unpartitionall | upa - unpartitions all servers by resetting their IPTables
```