**CSE 610: Project Report**

**Project: Implement Avalanche Consensus Protocols in PAXI Framework**

**Github Link:** <https://github.com/anmol372/paxi>

**Video Demonstration Link:** <https://youtu.be/1cFiCsfskYo>

**PAXI Framework:**

Paxi is a framework used to implement and benchmark consensus protocols. From the many protocols implemented using it such as WPaxos and other Paxos variants its clear that the framework is adept at meeting replication protocol needs, including network communication, state machine of a key-value store, client API and multiple types of quorum systems.

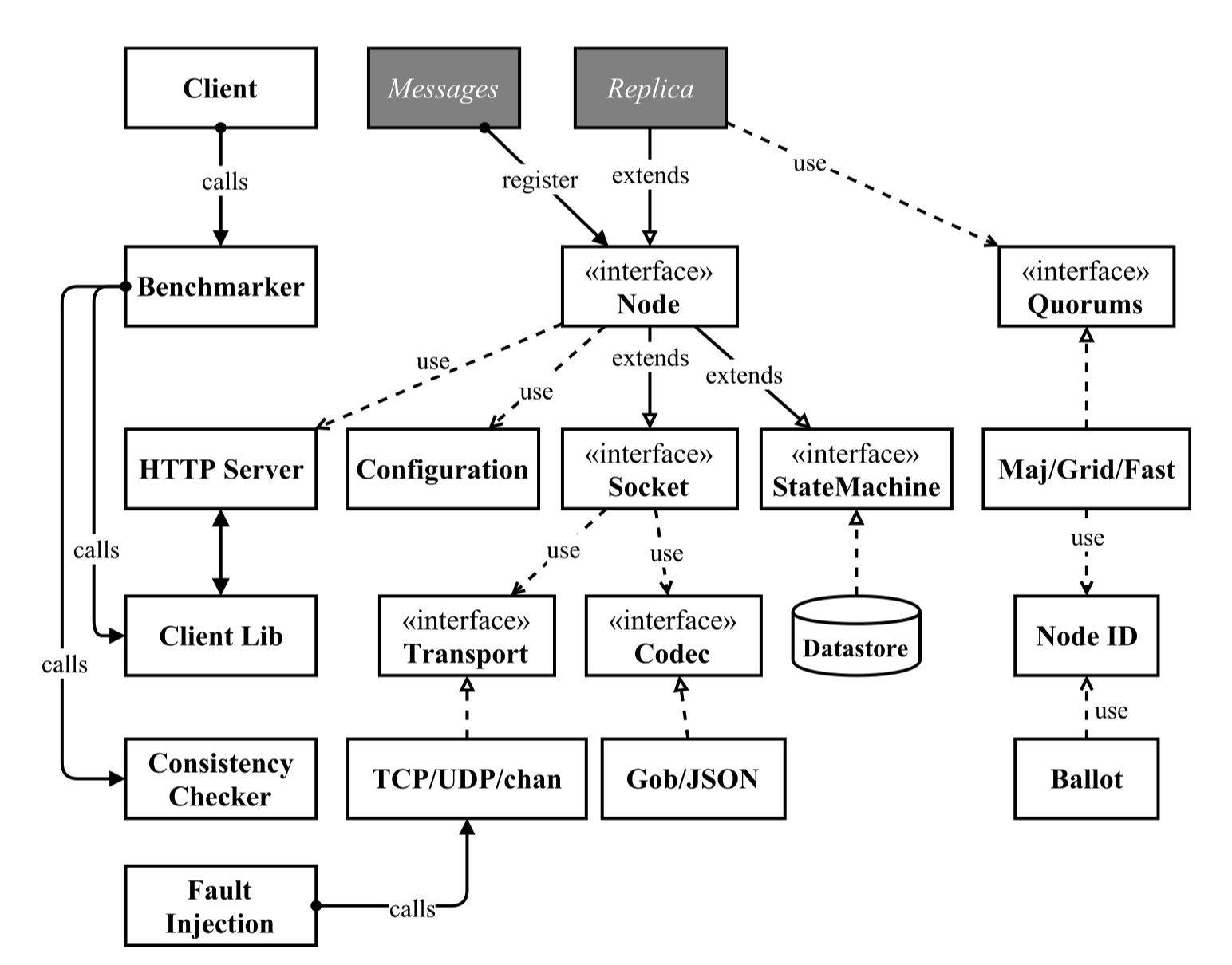


Figure:1 PAXI Framework ( <https://dl.acm.org/doi/pdf/10.1145/3299869.3319893> )

To implement any consensus protocol a developer must implement the Messages and Replica types (shaded in Figure 1). In Messages type we must define the structure of messages our protocol will deal with to communicate amongst themselves, after receiving a client request. In Replica type we must define the properties that we expect our servers to have, and also write the logic of how the server should react to each type of message defined in Messages type. Both these interfaces are implemented in a package, and the replica type must be registered for servers and clients, this allows us to create servers and clients for different types of protocols.

**My Understanding of the protocols for Implementation:**

All nodes are the same, no nodes have any special privileges or functionalities.

1. **Slush** 
   1. **Logic: Node with client request**
      1. A node receives a query with a color Red/Blue
      2. If the node is uncolored it adopts the color of the client request, else it keeps its color.
      3. Then the node creates a query for the client request color and forwards it to some ‘K’ random nodes with a timeout ‘T’ on replies received.
      4. Then the node waits for the timeout period, during which if it receives expected number of replies from the random nodes, the timer ‘T’ for this iteration is stoped.
      5. If timeout occurs before the node has received expected number of replies, then the node sends the same query to additional random nodes that will ensure that it receives appropriate number of responses.
      6. Once the node receives the expected number of replies, it tabulates the votes on Red/Blue.
      7. If higher number of votes in a color does not result in a color flip, it replies to the client with the color as consensus decision.
      8. If higher number of votes in a color result in a color flip, the node starts M iterations of steps 2.1.2 - 2.1.6, with an additional timer of ‘MT’ time for M rounds. (MT = M iterations \* T timeout)
      9. The server then responds to the client with the result it has at end of M iterations, or if ‘MT’ timeout occurs before the ‘M’ iterations are complete, it responds to the client with the result it has at the time. And it adopts the color of the response.
   2. **Logic: Node(s) with server request**
      1. The random nodes on receiving any query follow steps 2.2-2.9, with two exceptions:
         1. They don’t send any replies to the client, instead simply set their color to the color of the response.
         2. They only query other nodes, if their color is undecided.
2. **Snowflake Logic:**
   1. Snowball strengthens Slush’s conviction in its color, by adding a counter to the nodes. This also introduces BFT in the protocol
   2. Each node now maintains a counter for every query. This counter is incremented for every consecutive query that results in response same as the color of the query.
   3. The node counter is reset as soon as the response differs.
   4. When the counter for any one color exceeds a certain value Beta for a query, that color is accepted by the node as its color, and the consensus result is sent to the client.
3. **Snowball Logic:**
   1. Snowball replaces Snowflake’s counters with confidence counters for each color.
   2. Upon each successful query, the confidence of resulting color (>50% of K) is incremented.
   3. Each node flips its color based on its confidence counters.
   4. Finally the client is sent a response once the nodes confidence in a color does not flip, for some consecutive queries.

**My implementation logic of Slush to Snowball:**

1. A server ‘S’ receives a client request
   1. If S is uncolored it adopts the color of the query.
   2. Timer Step: Server stores the time of query generation as Unix time, the number of nanoseconds elapsed since January 1, 1970 UTC.
   3. Then the server sends out a request to k-sample servers.
      1. In my implementation, this k-sample is assumed to be some percentage(60%) of the total number of nodes defined in the config file.
2. All the servers in the K-sample receive the query request from S.
   1. If receiving server is colored, it responds to the query with their color.
   2. If receiving server is uncolored, it responds to the query with the color of query and starts its own query request to a k-sample of nodes.
   3. Timer Step: Server stores the time of query generation as Unix time, the number of nanoseconds elapsed since January 1, 1970 UTC.
3. Servers that receive replies do the following tasks:
   1. Based on query reply maintain counter for Red/Blue per query per iteration.
   2. Once threshold replies have been received for the query the following tasks are done:
      1. Store result for the iteration of the query.
      2. Increment servers confidence in Red/Blue per query.
      3. Set the color with higher confidence as the node’s color.
      4. Keep count of consecutive Red/Blue results.
      5. If the node has achieved enough confidence in a color (beta consecutive query results are for the same color) it can stop sending further iterations of the query. At this point if the node was originally sent the query from the client, it sends the client the consensus decision.
      6. Finally on each query reply received the server checks the number of replies received for each query still in play.
         1. If the query has passed a specific timeout and still not received threshold replies, the server sends that query to some other random nodes, enough to achieve threshold replies.

**How K-Sampling is implemented:**

For K-Sampling I wrote an additional method for the Socket interface in the PAXI framework called ‘MulticastRandom’.

MulticastRandom accepts 4 parameters:

1. nodePercent: Percentage of nodes to send message to from the total nodes available in the config files.
2. numNodes: Number of nodes to send message to from the total nodes available in the config files.
3. mident: This is an identifier that the function expects to distinguish between different queries, so it can randomly pick unique nodes, especially when resends are required during timeouts.
4. m: The message interface to send on go based channels between the servers.

The function ignores the nodePercent parameter, if numNodes is greater than 0. This was done to allow developer to send additional ‘Query’ messages to fixed additional sample, if timeout occurs before threshold replies have been achieved.

The method selects random nodes by iterating over all the nodes available in the config and choosing them based on output of Random Number Generator library “math/rand”. To get optimum randomness I choose to provide seed value as the current time, converted to int64.

The method also maintains a global hash map of all the nodes that were sent request per query per iteration. This map is used to avoid sending same nodes a query that it has already been sent. This functionality comes into play, when a node needs to send query to additional sample after a timeout.

A problem with this method is, that if all nodes in the config file have been sent a particular query, for a particular iteration, the method would be unable to choose nodes to send queries. This problem has not been addressed in the Avalanche paper as well, so I have ignored it for now. Further this condition should not arrive as the number of servers increase. In my implementation, I simply return from the function when such a situation arises.

**Journey With Timer Implementation:**

Currently the protocols timer implementation is not accurate. In essence the actual protocol needs an asynchronous timer for each query, and if the timeout occurs it requires that the server resend the particular query to additional unique random nodes and reset the timer, till the time threshold replies for the query have been received.

I tried a few ways to make it work:

1. Including a concurrent timer whenever a new query is sent to other nodes. With this implementation the problem was that once a timeout occurred on the concurrent go thread, that thread would interrupt the flow of the normal server, resend query if required and finally exit. This exit of the thread disrupted the flow of the server’s main thread, and the server would simply stop executing.
2. My initial assessment of the previous timer problem was that if I could keep the thread running indefinitely, the problem could be solved. This solution resulted in goroutine leaks. Leaking goroutine is basically a type of memory leak. You start a goroutine but that will never terminate, forever occupying a memory it has reserved. I learned that while go does extensive memory management, it does not like goroutines that run indefinitely. Goroutine leaks can be caused by empty select statements, infinite loops, and open sender or receiver channels. The only way to avoid this is to send a “close()” command to the goroutine, but implementing this had 2 scenarios:
   1. In happy path execution when a server receives threshold replies before timeout, the server could easily close the goroutine.
   2. In cases where a timeout occurs, we end up with a hung server or a goroutine leak.
3. The next timer implementation I tried was a Publisher Subscriber model. I basically started a separate goroutine watcher on the server log’s query timestamp key at the time of the servers initialisation. Till the server receives, its first query, the watcher simply waits for an entry. Once the server would receive a query, it would add it to the log, along with a timestamp as to when the query was multicasted. Then the server would update the number of replies received for that query as soon as it receives them. The watcher during this time would continuously loop over all logs, that have not received threshold replies yet and check their timestamp, with the current timestamp, to determine if some new k-sampling is required. If a timeout occurred, this implementation too caused the server to stop all execution.

All three of these implementations are present as comments in the current implementation, to allow me to debug the problem in detail. I suspect that if I can change the Paxi framework to make it open new goroutines for every function it calls through reflection and closure on receiving a message from other servers, I may be able to create a parallel timer. Since in this scenario, all processing would be done in concurrent threads and not a single main thread. But I did not implement this since my understanding of PAXI is still a little limited and this would definitely have introduced unwanted bugs, that could be too big to handle at this time.

So finally I ended up with a silly timer implementation, which is highly inaccurate. The current timer is called in the main thread, whenever the server receives a reply. It then iterates over all queries that have not received threshold replies. It then checks the time difference between the current time and time when query was initially generated, if ‘timeout’ time has elapsed, it resends query to additional sample.

**Outputs:**

Tested with 5 servers, and concurrencies: 1, 5, 20, 40, 100, 200.

Concurrency Vs Throughput

**Log Results:**

**Concurrency: 1**

[INFO] 2020/05/20 16:26:54.909353 benchmark.go:179: Concurrency = 1

[INFO] 2020/05/20 16:26:54.909833 benchmark.go:180: Write Ratio = 0.500000

[INFO] 2020/05/20 16:26:54.909855 benchmark.go:181: Number of Keys = 1000

[INFO] 2020/05/20 16:26:54.909868 benchmark.go:182: Benchmark Time = 1m0.000106985s

[INFO] 2020/05/20 16:26:54.909876 benchmark.go:183: Throughput = 997.414888

[INFO] 2020/05/20 16:26:54.911014 benchmark.go:184: size = 59843

mean = 1.256382

min = 0.000000

max = 45.602357

median = 0.973740

p95 = 3.122406

p99 = 7.860116

p999 = 9.429003

**Concurrency: 5**

[INFO] 2020/05/20 16:32:06.315030 benchmark.go:179: Concurrency = 5

[INFO] 2020/05/20 16:32:06.316329 benchmark.go:180: Write Ratio = 0.500000

[INFO] 2020/05/20 16:32:06.316350 benchmark.go:181: Number of Keys = 1000

[INFO] 2020/05/20 16:32:06.316362 benchmark.go:182: Benchmark Time = 1m0.000317555s

[INFO] 2020/05/20 16:32:06.316369 benchmark.go:183: Throughput = 999.311378

[INFO] 2020/05/20 16:32:06.317444 benchmark.go:184: size = 59957

mean = 2.002804

min = 0.000000

max = 45.939131

median = 1.096994

p95 = 7.865115

p99 = 10.931630

p999 = 14.166514

**Concurrency: 20**

[INFO] 2020/05/20 16:34:55.897236 benchmark.go:179: Concurrency = 20

[INFO] 2020/05/20 16:34:55.898667 benchmark.go:180: Write Ratio = 0.500000

[INFO] 2020/05/20 16:34:55.898691 benchmark.go:181: Number of Keys = 1000

[INFO] 2020/05/20 16:34:55.898703 benchmark.go:182: Benchmark Time = 1m0.001294775s

[INFO] 2020/05/20 16:34:55.898711 benchmark.go:183: Throughput = 1000.011753

[INFO] 2020/05/20 16:34:55.899547 benchmark.go:184: size = 60001

mean = 1.910015

min = 0.000000

max = 56.452908

median = 1.148572

p95 = 6.187723

p99 = 9.210815

p999 = 24.468040

**Concurrency: 40**

[INFO] 2020/05/20 16:38:00.383474 benchmark.go:179: Concurrency = 40

[INFO] 2020/05/20 16:38:00.383980 benchmark.go:180: Write Ratio = 0.500000

[INFO] 2020/05/20 16:38:00.384003 benchmark.go:181: Number of Keys = 1000

[INFO] 2020/05/20 16:38:00.384017 benchmark.go:182: Benchmark Time = 1m0.001337211s

[INFO] 2020/05/20 16:38:00.384029 benchmark.go:183: Throughput = 1000.011046

[INFO] 2020/05/20 16:38:00.384947 benchmark.go:184: size = 60001

mean = 1.862545

min = 0.000000

max = 86.770812

median = 1.125671

p95 = 5.655187

p99 = 7.451708

p999 = 35.338283

**Concurrency: 100**

[INFO] 2020/05/20 16:40:09.588228 benchmark.go:179: Concurrency = 100

[INFO] 2020/05/20 16:40:09.589988 benchmark.go:180: Write Ratio = 0.500000

[INFO] 2020/05/20 16:40:09.590020 benchmark.go:181: Number of Keys = 1000

[INFO] 2020/05/20 16:40:09.590033 benchmark.go:182: Benchmark Time = 1m0.002299623s

[INFO] 2020/05/20 16:40:09.590042 benchmark.go:183: Throughput = 999.995006

[INFO] 2020/05/20 16:40:09.591483 benchmark.go:184: size = 59998

mean = 2.353691

min = 0.000000

max = 186.989837

median = 1.166066

p95 = 5.245885

p99 = 10.879842

p999 = 150.653480

**Concurrency: 200**

[INFO] 2020/05/20 16:42:23.982170 benchmark.go:179: Concurrency = 200

[INFO] 2020/05/20 16:42:23.982974 benchmark.go:180: Write Ratio = 0.500000

[INFO] 2020/05/20 16:42:23.982995 benchmark.go:181: Number of Keys = 1000

[INFO] 2020/05/20 16:42:23.983004 benchmark.go:182: Benchmark Time = 1m0.000050155s

[INFO] 2020/05/20 16:42:23.983009 benchmark.go:183: Throughput = 1000.015831

[INFO] 2020/05/20 16:42:23.983859 benchmark.go:184: size = 59999

mean = 1.860581

min = 0.000000

max = 88.449865

median = 1.197362

p95 = 4.905315

p99 = 7.392321

p999 = 44.923282