A High Throughput Chain Replication

Gyanendra Aggarwal

Based on the following papers

A High Throughput Atomic Storage Algorithm

Rachid Guerraoui Dejan Kostic´ Ron R. Levy Vivien Que´ma

Object Storage on CRAQ

High-throughput chain replication for read-mostly workloads

Jeff Terrace and Michael J. Freedman

What is a Chain Replication

It creates a resilient distributed store. It will always be consistent and available despite of server failures (CA).

It assumes that a point-to-point communication is always available between all the servers (partition intolerant). A failure detection mechanism is being setup to detect any server failure.

All server but one can fail and system will still work.

Clients can read and write using any server concurrently thus making it a high throughput store. It will also prevent concurrent update to same object to maintain consistency.

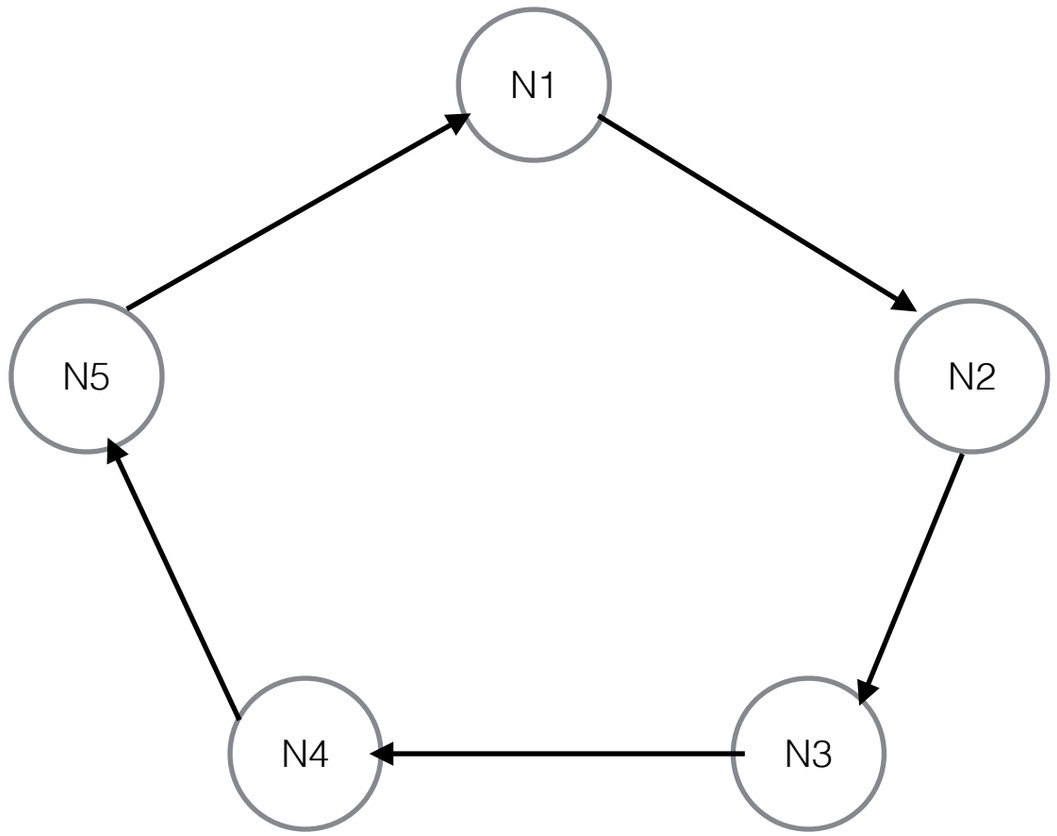
It also solves the read-inversion problem and prevents any read from returning an old value after an earlier read returned a new value.

Let us examine 2 different architectures to implement a Chain Replication.

1. High Throughput Atomic Store (HTAS)

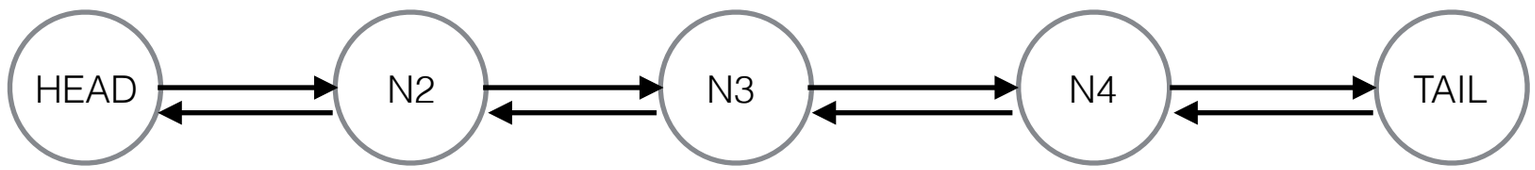
2. Chain Replication Apportioned Query (CRAQ)

HTAS -Overview



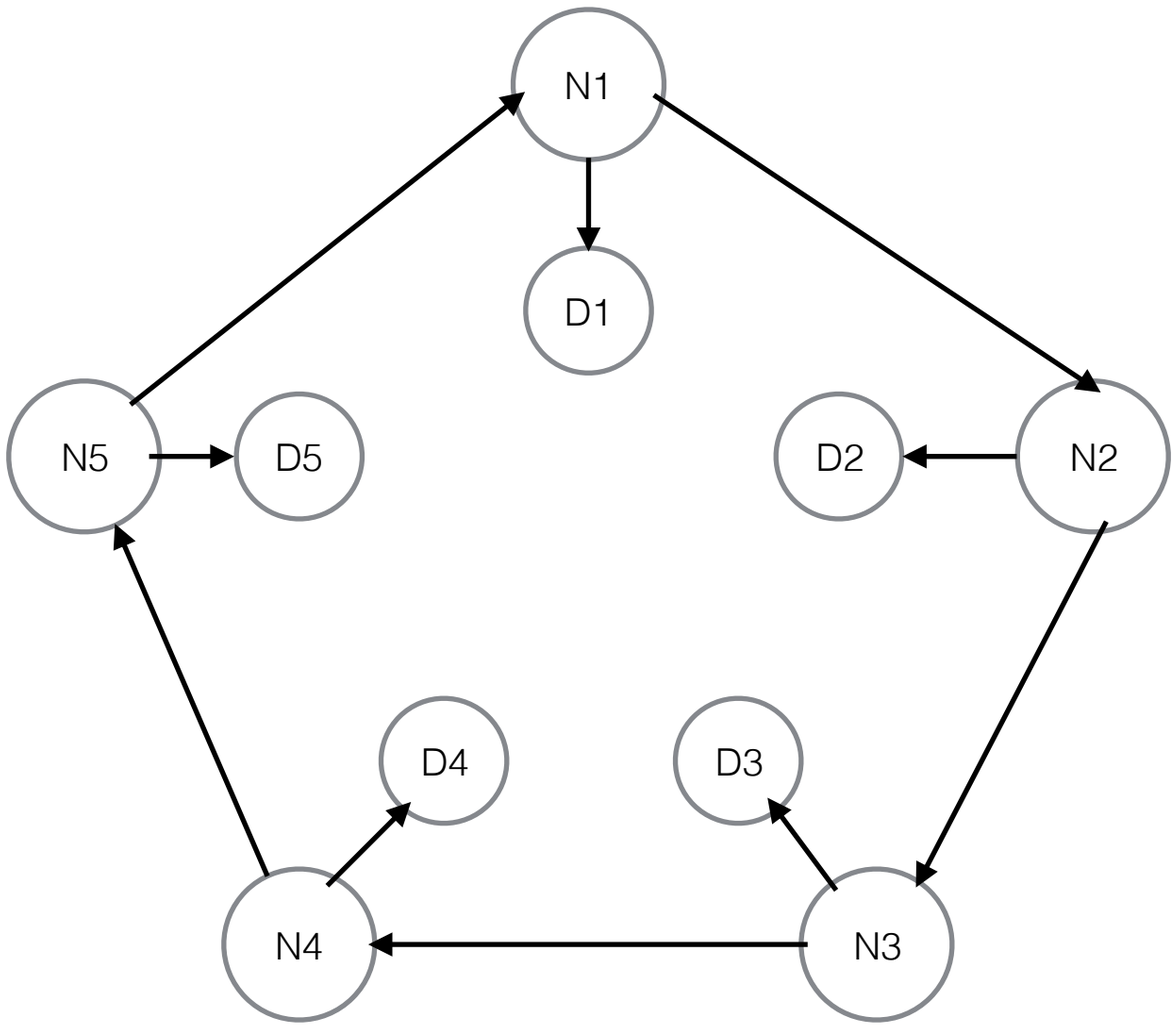
Clients can use any server to read or write.
All the server are logically connected in a ring.
HTAS uses ***read-one write-all*** strategy.
write operation: Client initiates a write operation for object A1 on N1
N1 will start a *pre-write* phase for A1 and send a *pre-write* message to N2 (successor). This message only indicates to N2 that A1 is going to be updated. N2 will forward *pre-write* message to N3 and eventually it will come back to N1. At this point every node will become aware that A1 is going to be updated. Once N1 gets the *pre-write* message, it will persist the change for A1 and start a *write* phase and send a *write* message to N2. N2 will also persist the change and forward *write* message to N3 and again eventually it will come back to N1. Once N1 receives the *write* message, it will acknowledge the client about the completion of write operation. It will take 2N messages to complete an update operation.
read operation: Client initiates a read operation for object A1 on N3. If N3 is not under *pre-write* phase for A1, it will respond to the client with existing value of A1 that it has. But if under *pre-write* phase for A1, it will hold the request and will respond to the client with new value when it persist the change.

CRAQ - Overview



Clients can use any server to read but it can initiate write only on head.
All the server are logically connected in a chain.
CRAQ uses ***read-one (apportioned query) write-all*** strategy.
write operation: Client initiates a write operation for object A1 on HEAD.
HEAD will start a *write* phase for A1, HEAD will make update for A1 but this update will not be committed, which means, the updated value of A1 will not be available for query.
It will also and send a *write* message to N2 (successor). Similarly, N2 will also make uncommitted update for A1 and it will send *write* message to its successor N3. Eventually *write* message will reach TAIL. TAIL will make an update for A1 and also commit.
It will start a *commit* phase. TAIL will send a *commit* message to its predecessor (N4). N4 will commit its update and send the *commit* message to its predecessor and eventually HEAD will receive a *commit* message. HEAD will also commit the update and will acknowledge the client about the completion of write operation.
Other nodes can have an uncommitted update but TAIL will always have committed update.
It will take only 2N-2 messages to complete an update operation.
read operation: Client can initiate a read operation for an object on any node. If client initiate a read operation for A1 on N3. If N3 has a committed value of A1, it will respond to client's request with value of A1. If N3 has uncommitted value for A1, it will get the committed value for A1 from the TAIL and respond to client's request.

High Throughput Chain Replication
Overview



update
When a client starts an update with a node (originating node), that node becomes HEAD for that specific update message and the predecessor of that node becomes TAIL for that specific update message.

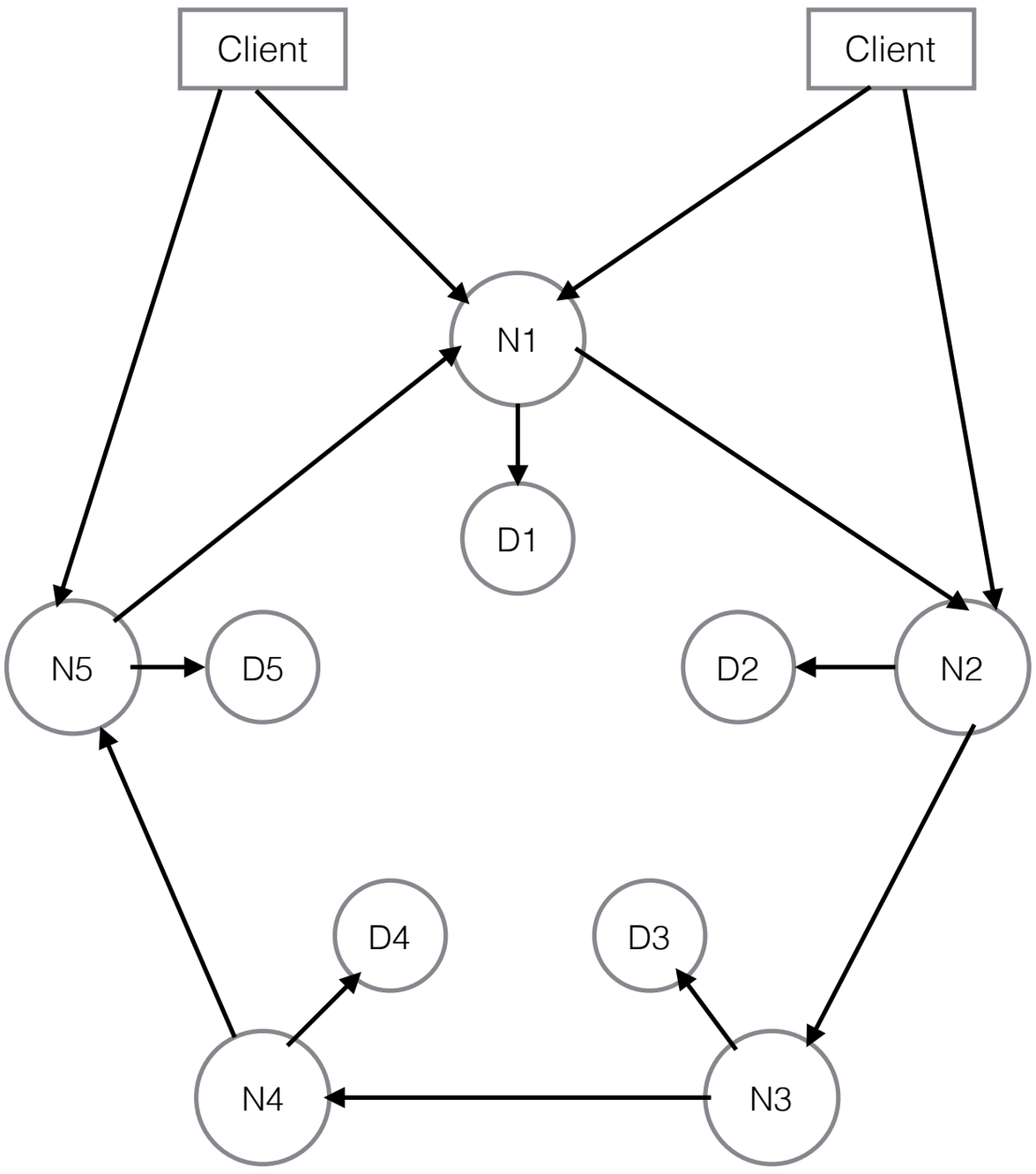
HEAD will start the *pre-write* phase (clock-wise) for update message and when that message reaches TAIL, it will persist the update and start a *write* phase (anti clock-wise). *write* message will reach every node and eventually the HEAD node. Every node (including HEAD) will persist the update and HEAD will acknowledge the client about completion of update. It allows client to use any node for update and it takes only 2N-2 messages to complete an update.

query
When a client makes a query for an object and if the object is not under *pre-write* phase then the node will respond immediately with value.
If the object is under *pre-write* phase, there are 4 choices (all the 4 implementation are available).

1. Send the request to TAIL of the *pre-write* message
2. Hold the message and respond when *write* is performed
3. Respond with existing committed value with a tag *dirty_read*
4. Send an error response saying the object is *being_updated*

Erlang Implementation (erlang_craq)

Architecture



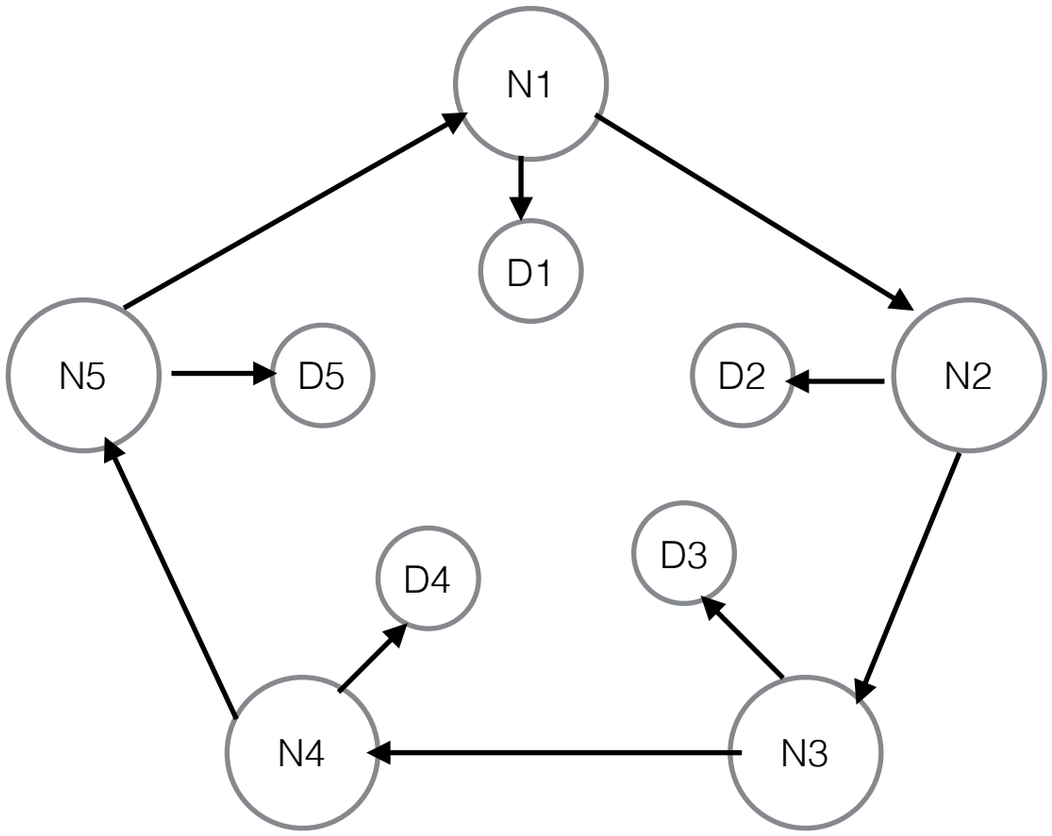
client
communicate (async) with any node (system_server)
API : setup_repl, query, update, delete, add_node

system_server
setup failure detection
maintain all the state information needed for CRAQ operation
communicate (sync) with data_server to persist/retrieve data

data_server
schema-less, column-oriented persistent data store

state information maintained by system_server
- node_status
- repl_ring_order
- repl_ring
- successor
- predecessor
- timestamp (logical clock)
- pre_msg_data
- msg_data
- pending_pre_msg_data
- completed_set
- query_data

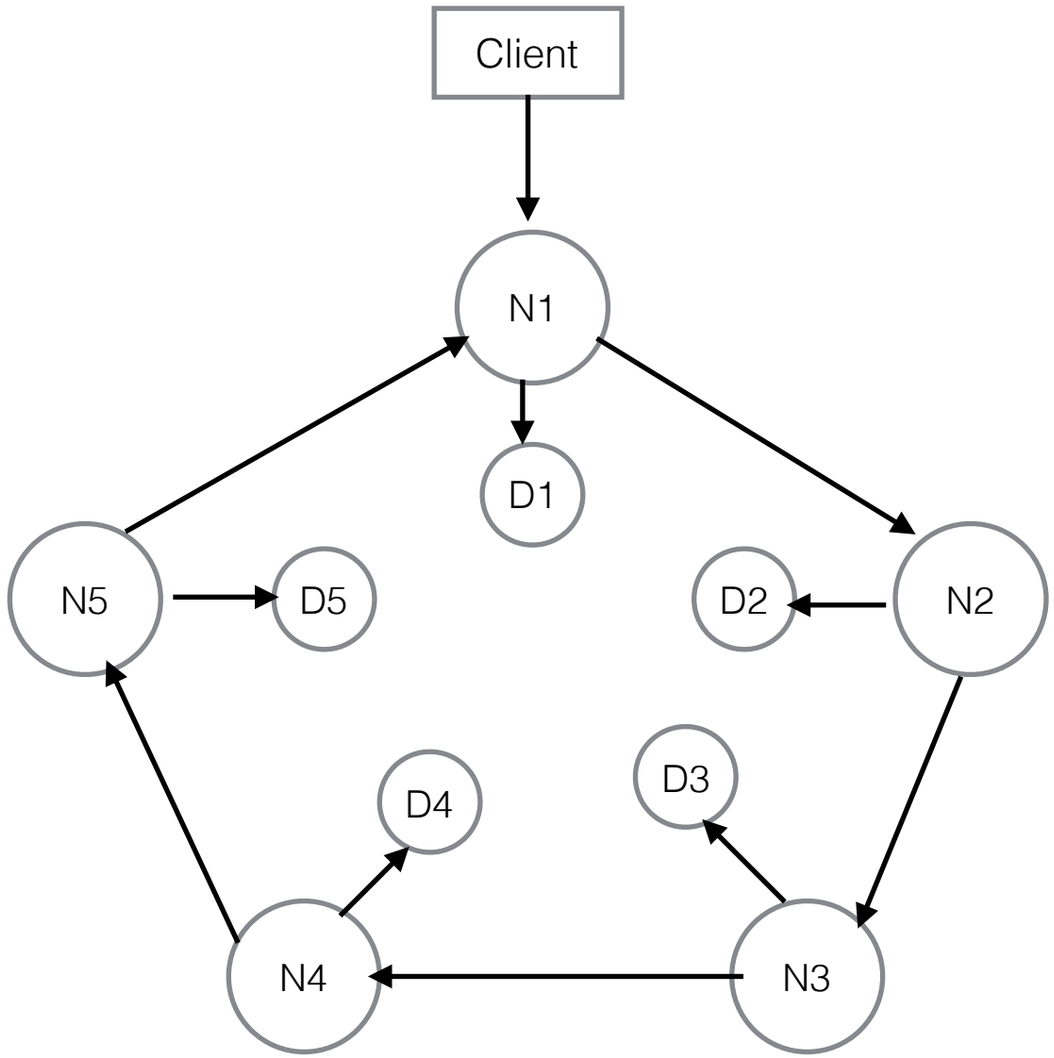
erlang_craq - setup_repl



setup_repl
setup failure detection
set the following server state information

- node_status
- repl_ring_order
- repl_ring
- successor
- predecessor
- timestamp

erlang_craq - update



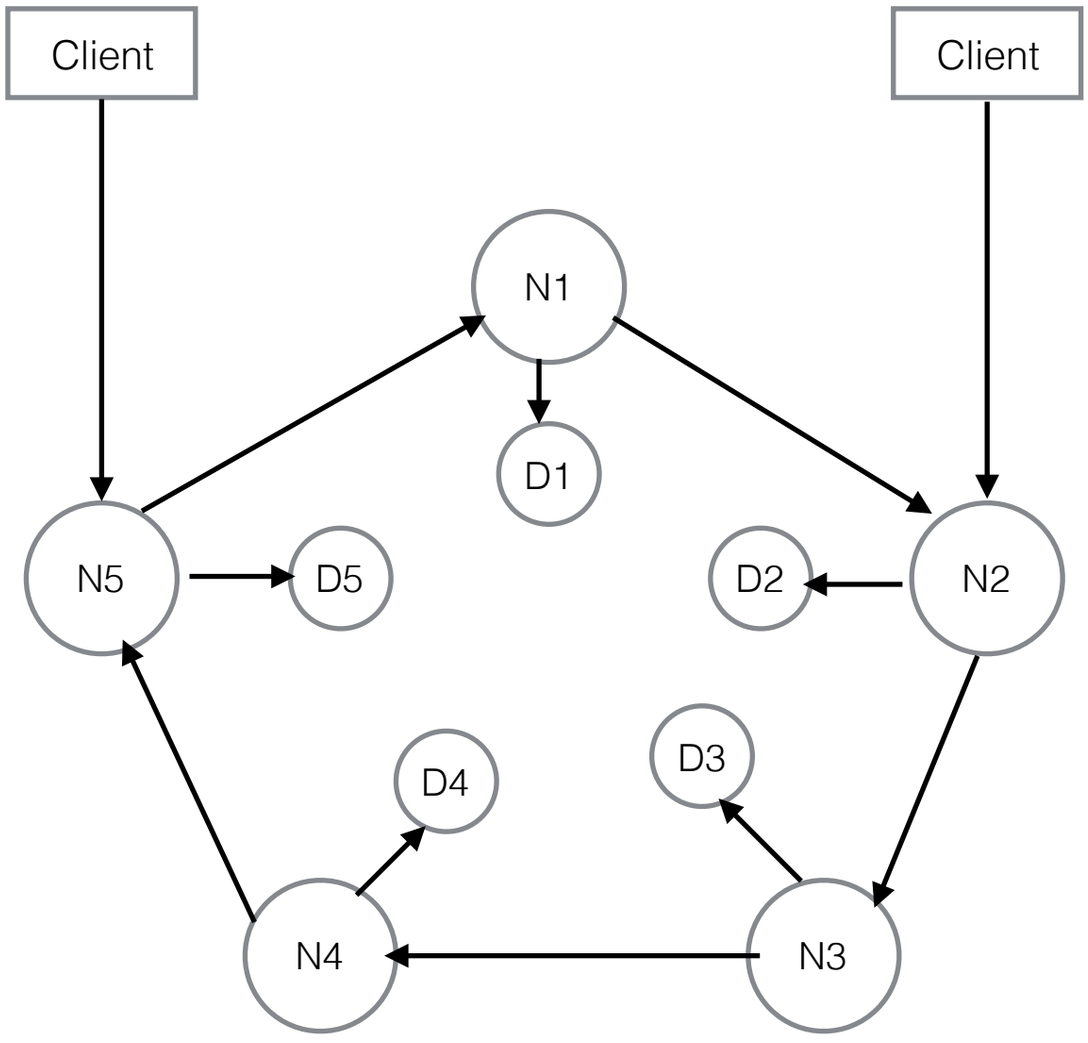
When N4 receives a *write* message, it persists the data locally using D4, removes it from its **pre_msg_data**, adds it to its **msg_data** and sends a *write* message to its successor.
When N1 receives a *write* message, it persist the data locally using D1, since N1 is the HEAD of the message, it removes the message from **pre_msg_data**, adds only the update_msg_key to **completed_set** and sends a response to client.
What happens to **msg_data** and **completed_set**? Will it keep growing?

state information
- successor
- predecessor
- timestamp (monotonically increasing number)
- pre_msg_data (map)
- msg_data (map)
- completed_set (set)

update message
timestamp
object_type
object_id
update_data
client_id
node_id
msg_ref

When N1 (HEAD) receives a client request for update, it increments its timestamp and creates an update message with new timestamp. It adds this message to its **pre_msg_data** and sends a *pre-write* message to N2 (successor)
When N2 receives a *pre-write* message, it updates its timestamp with the timestamp value of message, adds this message to its own **pre_msg_data** and sends a *pre-write* message to its successor.
When N5 (TAIL) receives a *pre-write* message, since N5 is the TAIL of this message, it persists the data locally using D5, adds it to **msg_data** and sends *write* message to N4 (predecessor)

erlang_craq - update - write conflict

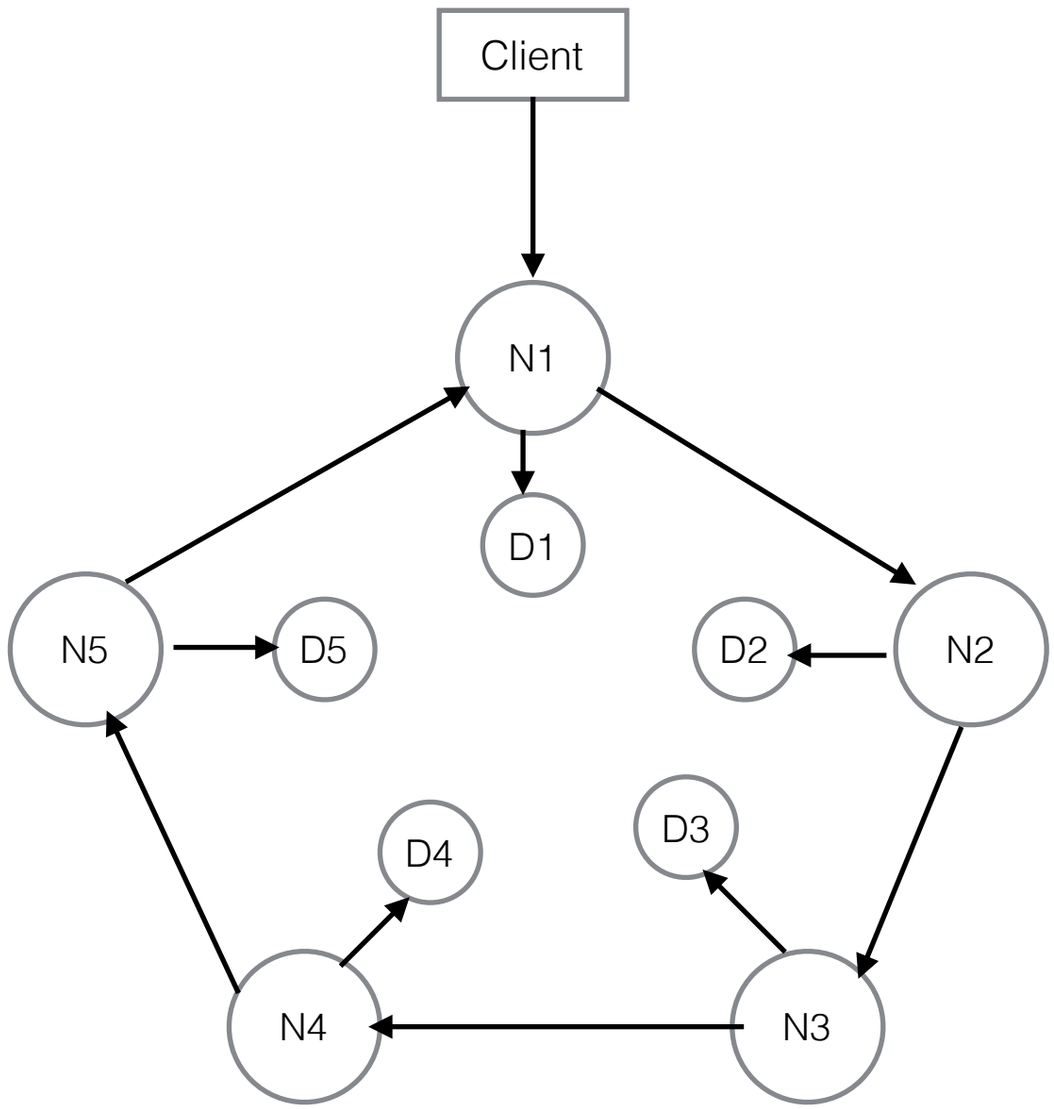


Let us assume that all the nodes have a timestamp value 10 and clients send an update message for object {candidate, 70} to nodes N2 and N5. The message key generates for these update message will be **{11, candidate, 70}**

Once N2 generated message (*pre-write*) reaches N5, it will see the conflict and since N5 is a higher node, it will stop the N2 generated message.

Once N5 generated message (*pre-write*) reaches N2, it will also see the conflict and since N5 is a higher node, it will replace its own generated message with N5 generated message and send an error response to client saying that object is being updated.

erlang_craq - query

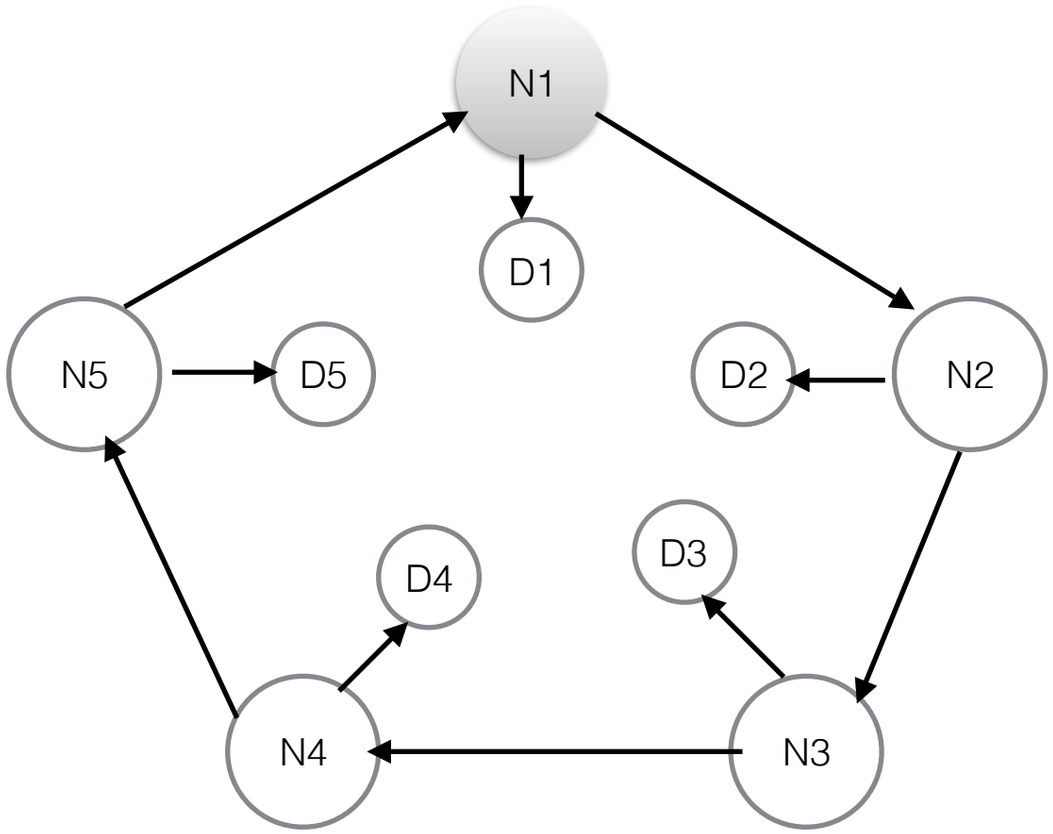


state information
- pre_msg_data
- query_data

When N1 receives a query request from a client for an object **{candidate, 10}**, it will check if there is any entry for **{candidate, 10}** in **pre_msg_data**. If no, then N1 does not know if **{candidate, 10}** is being updated or not, so it will retrieve the data locally using D1 and respond to the client.
If there is an entry in **pre_msg_data**, then N1 knows that **{candidate, 10}** is being updated and N1 does have the latest updated value locally. In this case, it will do one of the following things.

It will find the TAIL node for that object from **pre_msg_data** and send the request to the TAIL node to respond.
it will add this query request to **query_data**. N1 also knows that since **{candidate, 10}** is being updated, once N1 receives such a *write* message, it will update/persist the data for **{candidate, 10}.** it will remove that request from **query_data** and respond to the client with latest updates.
It will read the old (committed) value locally and respond with *dirty_read* tag.
It will send an error response saying that object is *being_updated.*

erlang_craq - failure - node down



Similarly, client may also receive a duplicate response and its should know how to deal with such duplicate responses. So it is also possible that a client can send a update request to one node (N1) and receive response from another node (N2). This is the reason for having a async communication between client and server.

What happens if N1goes down.

Now we will have only 4 nodes (N2, N3, N4, N5) in the ring. So every surviving node has to change the state information for repl_ring. In addition to this, now N2 will be successor of N5 and N5 will predecessor of N2.

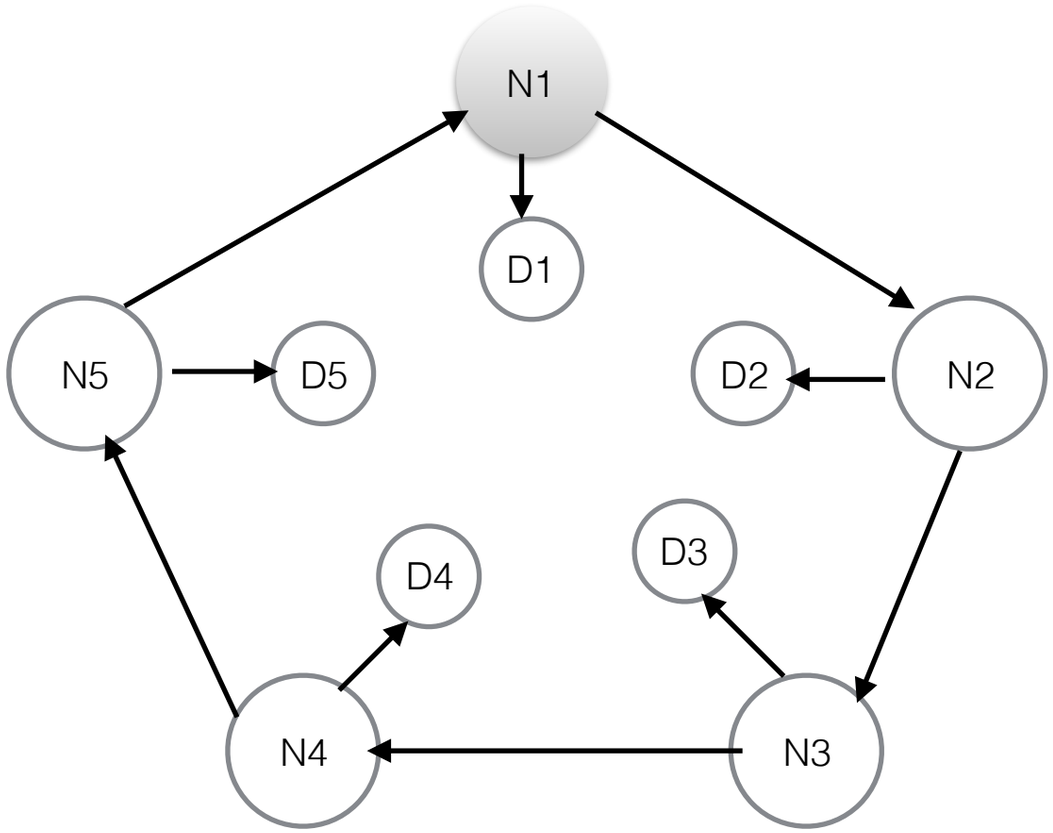
We also need to consider the possibility of loss of message that were stopped from further processing. So here are the possible scenarios.

N5 has a *pre-write* message
scenario 1: N2 was not the HEAD for this message. N5 will send a *pre-write* message to N2.
scenario 2: N2 was the HEAD for this message. N5 becomes effective TAIL for this message and N5 will persist this message and sends a *write* message to N4.

N2 has a *write* message
scenario 3: N1 was not the originating node for this message. N2 will send a *write* message to N5.
scenario 4: N1 was the originating node for this message. N2 becomes effective HEAD for this message and sends a reply to client.

So it is possible that N2/N5 may receive same message more than once. Fortunately, N2/N5 has enough information in their state to know if a message is duplicate or not.

erlang_craq - adding node



What happens when N1 is added back into ring

Every node will receive this information and they will update repl_ring state information and also setup failure detection for N1.
In addition to this, N5 will make N1 as its successor and N2 will make N1 as its predecessor.

N1 will update its repl_ring state information and make N2 as its successor and N5 as its predecessor.
It will also setup failure detection
for all other nodes (N2, N3, N4, N5).
N1 may also be lagging behind other nodes in terms of data that it has. So it get the missing updates from its successor (N2) and become current.

corner case : N2 has some updates in *pre-write* phase. Will N1 receive a *write* message?

erlang_craq - additional features

- sorted or user-defined node order
- mini-transaction for updating multiple object
- Check pointing data file to detect file corruption
- Splitting the data file to manage file size

source code : github.com/gyanaggarwal/erlang_craq.git