Chain Replication with Apportioned Query
High Throughput Atomic Store

Introduction

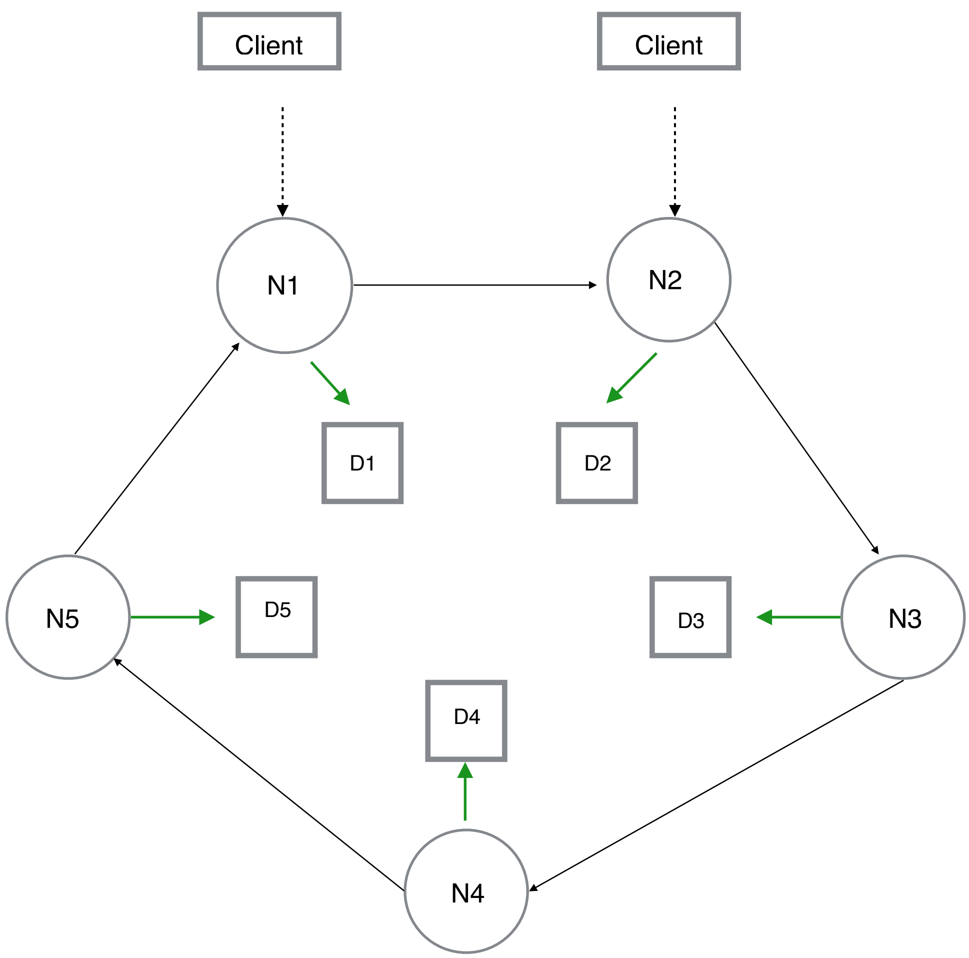
It is an atomic, high throughput, resilient distributed store. It will always be consistent and available despite of server failures (CA).

It assumes that a point-to-point communication is always available between all the servers (partition intolerant). A failure detection mechanism is setup to detect any server failure and system will continue to work as long as at least one server is available.

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

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

Architecture



N1 … Ni are system servers and they maintain all the state information needed to facilitate the CRAQ features.

D1 … Di are data servers and they facilitate the persistence of data. It is a schema-less, column-oriented data store.

Clients can interact with any system server through a given API.

System servers interact with their local data server for query and persistence of data.

System servers are arranged in a ring and they communicate with their successor (clock-wise) in *pre-write* phase and with their predecessor (anti clock-wise) in *write* phase.

In this implementation, there is no fixed *head/tail* node but for every update message, the originator node becomes the *head* node and the predecessor of the originator node becomes *tail* node thus clients can use any node to make updates.

Originating (*head*) node of the update message starts a *pre-write* phase and send a *pre-write* message to its successor (clock-wise). Eventually, the *pre-write* will reach the predecessor (*tail*) of originating node. Once, predecessor of originating node gets the *pre-write* message, it starts a *write* phase and sends a *write* message to its predecessor (anti clock-wise) and eventually the *write* message will come back to originating node (*head*). Once, originating node receives a *write* message, it responds back to the client about update.

It also supports mini-transactions. Clients can update multiple objects is a single invocation. Incase of write conflicts, the whole transaction will be aborted.

Users can configure the system to use either lager logging framework or gen_event/OTP logging framework.

Write Conflict

Write conflict will occur if 2 or more nodes are trying to update the same object. If the same object is a part of concurrent updates of mini-transactions, only one update/mini-transaction will be allowed to complete and rest of updates/mini-transactions will be aborted.

If all the concurrent updates are in *pre-write* phase then update message with highest originating (*head*) node value wins.

If concurrent updates are in the mix of *pre-write* and *write* phase (only one update can be in *write* phase) then update message in *write* phase wins.

Data Checkpoint and Data file versioning

After a number of update entries, current data file is checked for data corruption. If no data corruption is found, the current data file is closed and new versioned data file is created. If there is any data corruption found, the current data file is truncated at the right place and that CRAQ node is removed from the ring. User needs to add that CRAQ node back to the ring using *add_node* function. Once that CRAQ node is added back, it gets an incremental snapshot from its successor based on how much data it has lost and it becomes current again.

sys.config

replication_type	: It is only for documentation purpose. It is not being used.
node_order	: It takes a value ***sorted*** or ***user_defined*** ***sorted*** means the replicated ring is created using the sorted list of node values ***user_defined*** means, user provides the ring order
failure_detector	: Name of failure detection module that sets up the failure detection feature
repl_data_manager	: Name of module for interacting with data_server
storage_data	: Name of module which decides the file format of data where data_server persists data
write_conflict_resolver	: Name of module used for write conflict resolution
unique_id_generator	: Name of module used to generate unique id for any async request
query_handler	: Name of the module that is being used to handle a query request for an object if that object is in *pre-write* phase. One can use any of the following modules: ***eh_wait_query_handler_api*** system_server will wait for new value to persist and only then it will respond to the client ***eh_no_wait_query_handler_api*** system will not wait and respond immediately saying that the object is being updated ***eh_dirty_read_query_handler_api*** system_server will respond immediately with existing old value (dirty_read) ***eh_aq_query_handler_api*** query request will be send to *tail* to respond.
event_logger	: User can configure the system to user either lager logging framework (assign value ***lager_event***) or gen_event/OTP logging framework (assign value ***gen_event***).
data_checkpoint	: system will check the data file for data corruption after the number of update entries specified through this parameter. If the the data file is not corrupted then system will close the current data file and create a new versioned data file.
data_dir	: path where data and log files will be persisted if left blank then data and log files will be persisted in erlang_htas home directory (explained later)
file_repl_data	: this will be added as suffix to the node name to create actual data file name
file_repl_data_suffix	: when a versioned data file is created, this parameter is used to create a padded data file version number that is appended to file name.

file_repl_log	: this can be either set to standard_io or a file suffix. If this value is set to standard_io then all the log messages will be displayed on console otherwise they will be written in file with file name where this value will be suffix to node name
debug_mode	: log messages will appear only when this value is set to true.
sup_restart_intensity sup_restart_period sup_child_shutdown	: these 3 parameters are used by OTP supervisor

All the modules names that appear above are based on behaviors and can be changed if one wants to handle the these functionality in a different way.

Setup

Use craq_erl.sh to start a erlang distributed node. This script uses *sname* to create distributed node, but you can change it to *name* if *sname* does not work for you.

Create a home directory for erlang_craq.
Create a sub-directory (folder) ebin under this home directory and copy all the *.beam and *.app files here.
Create a sub-directory (folder) craq_test/ebin under this home directory and copy all the *.beam files for automated testing.
Change the first line of this bash script to so that you cd into your erlang_craq home directory.

Use ./craq_erl.sh <sname> to create a erlang distributed node and then use the following command to start erlang_craq server.

```
erlang_craq:start().
```

In case, you want to shut down your erlang_craq server (application) then you can type the following command at each distributed erlang server node.

```
erlang_craq:stop().
```

erlang_craq:start() will create only 1 erlang_craq server, you would normally require more than 1 server so repeat this step in different terminal windows to create more servers.

A typical example for creating 3 node erlang_craq distributed nodes will be to execute the following commands in 3 different terminals as follows:

```
terminal 1 :    ./craq_erl.sh    ec_n1
> erlang_craq:start().

terminal 2 :    ./craq_erl.sh    ec_n2
> erlang_craq:start().
```

terminal 3 :   ./craq_erl.sh   ec_n3
> erlang_craq:start().

ERLANG_CRAQ API

We define an ***object*** as a pair of *object_type* and *object_id.* Example {person, 10}.

Here in most of the examples we are going to use atoms and integers but they can be any erlang term.

All the erlang_craq api commands are executed from a distributed client node. So create a new distributed client erlang node (do not start any erlang server on client node) using craq_erl.sh bash script.

A typical example of creating a distributed node for client will be:

terminal 4:   ./craq_erl.sh   ec_c1

You can execute any of the following commands from this client node.

setup_repl

erlang_craq:setup_repl(NodeList).

This is the first command you are going to execute before you can do anything meaningful. This command connects all the erlang server that you have created earlier in a ring and also sets the failure detection.

Here, NodeList is a list of erlang distributed nodes that you had created earlier.
A typical NodeList will be (for my computer)

N1 = ‘eh_n1@Gyanendras-MacBook-Pro’.
N2 = ‘eh_n2@Gyanendras-MacBook-Pro’.
N3 = ‘eh_n3@Gyanendras-MacBook-Pro’.

NodeList = [N1, N2, N3].

update

This command is used to insert a new value or update an existing value.

erlang_craq:update(Node, ObjectType, ObjectId, [{Col1, Val1}, ..,{Coln, Valn}]).

This command will send and update message to Node.

Example

erlang_craq:update(N1, person, 10, [{name, john}, {age, 30}, {gender, male}]).

If you want to update the age of {person, 10}, you can give following command
on the same node or any other node in the ring.

erlang_craq:update(N2, person, 10, [{age, 40}]).

You also use this to update and delete columns in a single command.
It will be as follows:

erlang_craq:update(Node, ObjectType, ObjectId, UpdateColList, DeleteColList).

Example

We want to update name to john_smith and remove age, so our command will
be as follows:
erlang_craq:update(N3, person, 10, [{name, john_smith}], [age]).

We can perform mini-transaction (update multiple objects) in a single update command

erlang_craq:update(Node, UpdateList).

Example

```
erlang_craq:update(N1, [{candidate, 10, [{name,donald_trump},{party.republican}]},
                        {person, 10, [{name,john_smith},{age,40}]}]).
```

delete

This command is used to either delete columns from an object
or entire object.

Deleting columns

erlang_craq:delete(Node, ObjectType, ObjectId, DeleteColList).

Example

erflang_craq:delete(N1, person, 10, [gender]).

Deleting entire object

erlang_craq:delete(Node, ObjectType, ObjectId).

Example

erlang_craq:delete(N1, person, 10).

query

This command is used to query an object.

erlang_craq:query(Node, ObjectType, ObjectId).

Example

erlang_craq:query(N1, person, 10).

stop

This command has been created mainly for testing purpose. It should never be used otherwise. This command is used to bring down a server from a client node. The server does go down.Since it works under OTP supervision so supervisor bring it back. Server does come back but it does not become part of ring automatically, so for all practical purpose, it remains down for the ring.

erlang_craq:stop(Node).

This command will bring down Node.

Example

erlang_htas:stop(N1).

add_node

This command is used to add a node back to the ring.

erlang_craq:add_node(Node, NodeList, NodeOrderList).

NodeList should contain list of all the nodes (including the Node that is being added). NodeOrderList is list of all the nodes including any down node. This list must be in correct node order if the node order has been defined as *user_defined*.

Most likely, the node that has been added to ring will be behind other nodes in the ring in terms of the data it contains. So newly added node, always get an incremental data snapshot from its successor to become current.

Example

erlang_craq:add_node(N1, [N1, N2], [N1, N2, N3]).

Manual Testing

Remove *.data files from the data_dir or erlang_craq directory to ensure that we perform this test with a consistent state.

Start 3 erlang_craq servers and a client node as follows.

terminal 1 : ./craq_erl.sh ec_n1
erlang_craq:start().

terminal 2 : ./craq_erl.sh ec_n2
erlang_craq:start().

terminal 3 : ./craq_erl.sh ec_n3
erlang_craq:start().

terminal 4 : ./craq_erl.sh ec_c1

Now on terminal 4 which is a client node, create a node list as follows.

```
> N1 = ‘ec_n1@<your_host_name>’.
> N2 = ‘ec_n2@<your_host_name>’.
> N3 = ‘ec_n3@<your_host_name>’.
> NL = [N1, N2, N3].
```

setup the replication ring

```
> erlang_craq:setup_repl(NL).
```

insert data using N1

```
> erlang_craq:update(N1, candidate, 10, [{name,donald_trump},{party,republican}]).
```

and you should see the following response:

[{ok, {[{candidate, 10}], updated}}]

query N1, N2 and N3 to make sure that data has been inserted in all 3 nodes.

```
> erlang_craq:query(N1, candidate, 10).
> erlang_craq:query(N2, candidate, 10).
> erlang_craq:query(N3, candidate, 10).
```

and you should see the following response for each query:

[{ok, {candidate,10,[{party,republican},{name,donald_trump}]}}]

Let us make some more update entries

```
> erlang_craq:update(N2, candidate, 20, [{name,hillary_clinton},{party,democrat}]).
> erlang_craq:update(N3, candidate, 30, [{name,ted_cruz},{party,republican}]).
```

A mini-transaction

```
> erlang_craq:update(N1, [{candidate, 40, [{name,bernie_sanders},{party,democrat}]},
                          {candidate, 50, [{name,marco_rubio},{party,republican}]}]).
```

Now, let us do a concurrent mini-transactions update on different objects,
this is done using a multi_update api.
This api has been provide to simulate concurrent update from a single client.

```
> erlang_craq:multi_update([N1, N2, N3],
                           [[{candidate,10,[{name,donald_trump},{party,republican}]},
                             {candidate,20,[{name,hillary_clinton},{party,democrat}]}],
                            [{candidate,30,[{name,ted_cruz},{party,republican}]},
                             {candidate,40,[{name,bernie_sanders},{party,democrat}]}],
                            [{candidate,50,[{name,marco_rubio},{party,republican}]},
                             {candidate,60,[{name,ben_carson},{party,republican}]}]]).
```

both the updates should succeed and you should see the following response:

*[{ok,{[{candidate,10},{candidate,20}],updated}},
{ok,{[{candidate,30},{candidate,40}],updated}},
{ok,{[{candidate,50},{candidate,60}],updated}}]*

Let us do concurrent update on a same object.

```
> erlang_craq:multi_update([N1, N2, N3],
                           [[{candidate,10,[{name,donald_trump},{party,republican}]},
                             {candidate,20,[{name,hillary_clinton},{party,democrat}]}],
                            [{candidate,30,[{name,ted_cruz},{party,republican}]},
                             {candidate,20,[{name,hillary_clinton},{party,democrat}]}],
                            [{candidate,50,[{name,marco_rubio},{party,republican}]},
                             {candidate,20,[{name,hillary_clinton},{party,democrat}]}]]).
```

It should permit only one of the updates to succeed,
hence you should see the following response.

*[{error,{[{candidate,10},{candidate,20}],being_updated}},
{error,{[{candidate,30},{candidate,20}],being_updated}},
{ok,{[{candidate,50},{candidate,20}],updated}}]*

Now, let us stop N1 and do more updates using N2 and N3.

```
> erlang_craq:stop(N1).

> erlang_craq:update(N3, candidate, 80, [{name,chris_christie},{party,republican}]).
> erlang_craq:update(N2, candidate, 90, [{name,jeb_bush},{party,republican}]).
```

At this stage check the file size of data file and you will see that ec_n1_repl.data
is smaller compare to ec_n2_repl.data and ec_n3_repl.data.
ec_n2_repl.data and ec_n3_repl.data should be of same size.
Now, let us bring back N1.

> erlang_craq:add_node(N1, NL, NL).

Now, check the file size again and you will see that all the 3 data files are of same size. Also to ensure that N1 has data for candidate 80 and candidate 90, execute the following commands.

> erlang_craq:query(N1, candidate, 80).
> erlang_craq:query(N1, candidate, 90).

Automated Testing

To use this automated testing tool, we will need multiple clients.

One client will randomly bring down or bring up a erlang_craq server every 5 seconds. This client will perform this activity about 15 to 20 times and if at the end of its run ,some of the nodes are still down, it will bring them back one at a time.

Other clients (at least 2 more) will keep entering data. They will make about 400 to 500 entries at rate of 5 entries every second. They will choose a node randomly from a list of nodes to enter the data. It does not matter if that node is down because client will timeout for that data entry. Client will also choose randomly a object_type, object_id, column_name and column_value for a data entry. Once, all the clients finish their job we can validate if our erlang_craq servers are in consistent state or not. These are the steps required to perform our automated test.

1. remove all the data files
2. start 6 terminals and follow the instruction given below.

terminal 1 : ./craq_erl.sh   ec_n1
> erlang_craq:start().

terminal 2 : ./craq_erl.sh   ec_n2
> erlang_craq:start().

terminal 3 : ./craq_erl.sh   ec_n3
> erlang_craq:start().

terminal 4 : ./craq_erl.sh   ec_c1
> NL = [‘ec_n1@<host_name>’, ‘ec_n2@<host_name>’, ‘ec_n3@<host_name>’].

Now keep the following command ready but do not execute it
> erlang_craq_test:data_entries(NL).

terminal 5 : ./craq_erl.sh   ec_c2
> NL = [‘ec_n1@<host_name>’, ‘ec_n2@<host_name>’, ‘ec_n3@<host_name>’].

Now keep the following command ready but do not execute it
> erlang_craq_test:data_entries(NL).

terminal 6 : ./craq_erl.sh   ec_c3
> NL = [‘ec_n1@<host_name>’, ‘ec_n2@<host_name>’, ‘ec_n3@<host_name>’].

Now execute the following commands on this terminal.

> erlang_craq:setup_repl(NL).

> erlang_craq_test:node_change(NL).

Now, go back to terminal 4 and terminal 5 and execute data_entries command.

Once, all the 3 clients finish their job, execute the following command on terminal 6 to validate if erlang_htas servers are in consistent state or not.

> erlang_craq_validate:check_data(NL).

and you should see response as

valid

Any other response will indicate that system has failed the test.