Conflict-free Replicated Data Type

CRDTs are maintained as state by gen_servers(replicas). A replica can support and maintain many CRDTs (of same type or different types). Each replica has a Recursive Composite Map CRDT as its state to compose CRDTs that clients are interested in. CRDTs are automatically created with mutation operation. Inside a replica, a CRDT is identified by type and name pair. All the replicas should have same value for type and name if they want to share their mutations. CRDT types are given below (inside bracket) and name can be any Erlang term. All the CRDTs support some mutations and query operation.

Currently, the following CRDTs and mutation and query operation are supported.

1. **Multi-Valued Register (type : ec_mvregister):** It supports only 1 mutation operation {val, term()}. If there are concurrent mutation operations, it maintains all the values and user resolves the conflict. Query operation does not require any query criteria, so user can send empty list [] as query criteria.

2. **Grow-Only Counter (type : ec_gcounter):** It support only 1 mutation operation {inc, non_neg_integer()}. If there are concurrent mutation operations then it applies both the operations. Query operation does not require any query criteria, so user can send empty list [] as query criteria.

3. **PN-Counter (type : ec_pncounter):** it supports 2 mutation operations {inc, non_neg_integer()} and {dec, non_neg_integer()}. If there are concurrent mutation operations then it applies both the operations. Query operation does not require any query criteria, so user can send empty list [] as query criteria.

4. **Enable-Wins Flag (type : ec_ewflag):** it supports only 1 operation {val, true | false}. If there are concurrent mutation operations and one of the mutation is {val, true} then value is set to true. Query operation does not require any query criteria, so user can send empty list [] as query criteria.

5. **Disable-Wins Flag (type : ec_dwflag):** it supports only 1 mutation operation {val, true | false}. If there are concurrent mutation operations and one of the mutation is {val, false} then value is set to false. Query operation does not require any query criteria, so user can send empty list [] as query criteria.

6. **Add-Wins Observe-Remove Set (type : ec_aworset):** it supports 2 mutation operations {add, term()} and {rmv, term()}. Because of observe-remove behavior, a {rmv, Element} will be a valid mutation operation only if Element has already been added to the set by any previous {add, Element} mutation operation. If there are concurrent mutation operations {add, Element} and {rmv, Element} (add and rmv for same Element) then Element will be added to the set. If you want to see the whole set then query operation does not require any query criteria and user can send empty list [] as query criteria. If you just want to see whether an Element exists or not, then use [Element] as your query criteria.

7. **Remove-Wins Observe-Remove Set (type : ec_rworset):** it supports 2 mutation operations {add, term()} and {rmv, term()}. Because of observe-remove behavior, a {rmv, Element} will be a valid mutation operation only if Element has already been added to the

set by any previous {add, Element} mutation operation. If there are concurrent mutation operations {add, Element} and {rmv, Element} (add and rmv for same Element) then Element will be removed from the set. If you want to see the whole set then query operation does not require any query criteria and user can send empty list [] as query criteria. If you just want to see whether an Element exists or not, then use [Element] as your query criteria.

8. **Put-Wins Observe-Remove Map (type : ec_pwormap):** it supports 2 mutation operations {put, {term(), term()}} ({term(), term()} is a key-value pair) and {rmv, term()}. Because of observe-remove behavior, a {rmv, Key} will be a valid mutation operation only if Key has already been added to the map by any previous {put, {Key, Value}} mutation operation. If there are concurrent mutation operations {put, {Key, Value}} and {rmv, Key} (put and rmv for same Key), then {Key, Value} will be added to the map. If you want to see the whole map then query operation does not require any query criteria and user can send empty list [] as query criteria. If you just want to get a value for a particular Key, then use [Key] as your query criteria.

9. **Remove-Wins Observe-Remove Map (type : ec_rwormap):** it supports 2 mutation operations {put, {term(), term()}} ({term(), term()} is a key-value pair) and {rmv, term()}. Because of observe-remove behavior, a {rmv, Key} will be a valid mutation operation only if Key has already been added to the map by any previous {put, {Key, Value}} mutation operation. If there are concurrent mutation operations {put, {Key, Value}} and {rmv, Key} (put and rmv for same Key), then Key will be removed from the map. If you want to see the whole map then query operation does not require any query criteria and user can send empty list [] as query criteria. If you just want to get a value for a particular Key, then use [Key] as your query criteria.

10. **Recursive Composite Map (type : ec_compmap):** This CRDT is used to compose multiple CRDTs into a single CRDT. It supports only 1 mutation operation {mutate, {{Type, Name}, Ops}}. {Type, Name} identifies the CRDT that you want to mutate using Ops. Type will be one of the types that we have mentioned here (including ec_compmap, you can have a Composite Map inside another Composite Map) and Name can be any term(). CRDT identified by {Type, Name} need not exist before you want to perform a mutation on CRDT {Type, Name}, it will be added dynamically with the mutation operation. Query operation requires a query criteria of the form [{Type, Name}] for query operation. For Set and Map CRDT, users can use query criteria [{Type, Name}, term()] to further refine their query. Here term() will be either Element (for set) or Key (for map).

System Configuration

timeout_period: Replica runs its anti_entropy logic when timeout occurs. This parameter is used by replica to get a timeout period. Its value is {time_unit, {Min, Max}}. time_unit can be seconds or milli_seconds. If time_unit is seconds then value of Min and Max is converted to mill_seconds. Replica chooses a random value between Min and Max as its timeout period.

crdt_spec: Normally, this value will be {ec_compmap, undefined} so that a replica can have a Recursive Composite Map (CRDT) as its state but if one wants to use it only for a specific CRDT, they can specify a different value.

data_manager: This parameter specifies the api module name for data server, you need to change this only if you are using a different data server.

storage_data: This parameter specifies the module name that is used to persist data in a specific format. If you want to use a different format for data persistence then specify here the new module name.

optimized_anti_entropy: This parameter is set to either true or false. If this is set true then replica will send its causal history along with delta interval. This will make replicas to converge faster. If this parameter is set to false then a replica will send it causal history only when it does not have any delta interval to be send.

data_dir: This parameter specifies the name of directory where data will be persisted.

file_delta_mutation: This parameter is used to create the name of file where data for a replica is persisted for all the delta mutations generated by client and delta interval received from other replicas (any data that changes its state). System prepends the node's sname to this parameter to create the file name.

file_delta_interval: This parameter is used to create the name of file where data for a replica is persisted for all delta interval sent by this replica to other replicas. We need this because if a receiving replica missed a delta interval because of some glitch, replica will need to re-send missed delta-intervals. System prepends the node's sname to this parameter to create a file name.

API

erlang_crdt module implements client API. It has the following functions

1. **start()** : to start a replica

2. **stop()** : to stop a replica

3. **setup_repl(NodeList)** : this sets up the replica cluster, users can start mutation operations only after replica cluster has been setup.

4. **mutate(Node, Ops)** : this sends a mutation request (specified in Ops) to a Node to mutate a CRDT in a replica (specified by Node).

5. **query(Node, Criteria)** : User can query a Replica (specified by Node) for a CRDT (specified by Criteria).

6. **resume(Node, NodeList)** : After a down node is recovered and started again, use this command to join the down node (specified by Node) to the replica cluster.

Manual Testing

1. Start 3 replicas on distributed Erlang Nodes (r1, r2 and r3) on same machine or different machines.
2. Start 1 distributed Erlang Node c1 (client node) to issue mutation and query commands.
3. Create a replica list RL ([R1, R2, R3]) on client node and issue the following commands from client node.
4. Issue a command erlang_crdt:setup_replica(RL) to create a replica cluster.
5. Issue ec_gen_crdt_test:mutation01(RL) to send mutations to all the 3 replicas and wait for them to timeout so that you can see that all the replicas send their delta-interval to other replicas for them to converge.
6. Once you have seen the merge operation, issue command ec_gen_crdt_test:mutate02(RL) command. This command will send mutations to all the replicas and also bring down the replica R3. Again wait for replica R1 and R2 to timeout. R3 will not timeout because it went down. We do not need to start R3 again because it is brought back by its supervisor. At this point, R3 is up but still not a part of cluster. Here, R1 and R2 will timeout and you will see that R1 and R2 send their delta-interval to each other. At this stage, R1 and R2 have converged with each other but diverged from R3.
7. Issue a command ec_gen_crdt_test:mutate03(RL) to send more mutations to R1 and R2 and wait for them to converge.
8. Issue a command ec_gen_crdt_test:muttate04(RL) to send mutations to R2 only and wait for R2 to send its delta-interval to R1. At this stage, R1 and R2 have converged with each other but they are missing 1 delta-interval from R3 because R3 went down before sending its delta-interval. R3 is also missing some delta-intervals from R1 and R2 because R3 has still not joined the replica cluster.
9. Issue a command erlang_crdt:resume(R3, RL) to make R3 as part of replica cluster and wait for R1, R2 and R3 to send other replicas their delta-intervals to that all them converge.
10. What I have mentioned here is a most likely scenario that is expected to happen, but because of randomness in the timeout for each replica and also the time when you issue your mutation commands, you might see that replicas may send their delta-intervals in different order and that does not matter. You will see regardless of how and when delta-interval are sent, all the replicas will converge. You can issue query commands to check if all the replicas have same values for CRDTs or not.