Pure Operation-Based Replicated Data Types

Gyan Aggarwal

Based on a paper
Pure Operation-Based Replicated Data Types
by
Carlos Baquero, Paulo Sergio Almeida and Ali Shoker

Agenda

- explain in simple terms, how does a CRDT work
- establish requirements for implementing a CRDT
- a framework that is completely based on asynchronous message passing, minimum network traffic, minimum space requirement
- system architecture
- network partition and node (replica) failure
- add, remove replica

Why CRDTs
Conflict-Free Replicated Data Type

Distributed systems designed to serve clients across the world often make use of geo-replication to attain low latency and high availability. Conflict-free Replicated Data Types (CRDTs) allow the design of predictable multi-master replication and support guaranteed eventual consistency of replicas that are allowed to transiently diverge. CRDTs come in two flavors: *state-based,* where a state is changed locally, shipped and merged into other replicas; *operation-based,* where operations are issued and reliably causal broadcast to all other replicas.

state-based CRDTs may require a very large and unpredictable amount to data to be shipped to other replicas and may become impractical to implement, where as pure operation-based CRDTs will require a very small and predictable amount of data to be shipped.

What is a CRDT

A CRDT is a data type with a set of operation and a machinery (?)

Example: a counter (long data type) with INC and DEC operation
a set (collection) with ADD and RMV operation

INC and DEC operation can be applied in any order (commutative) to a counter and still the final result will be same, where as ADD and RMV are non-commutative and the order of application will decide the final result.

CRDTs can be grouped into 2 broad categories, commutative CRDTs and non-commutative CRDTs

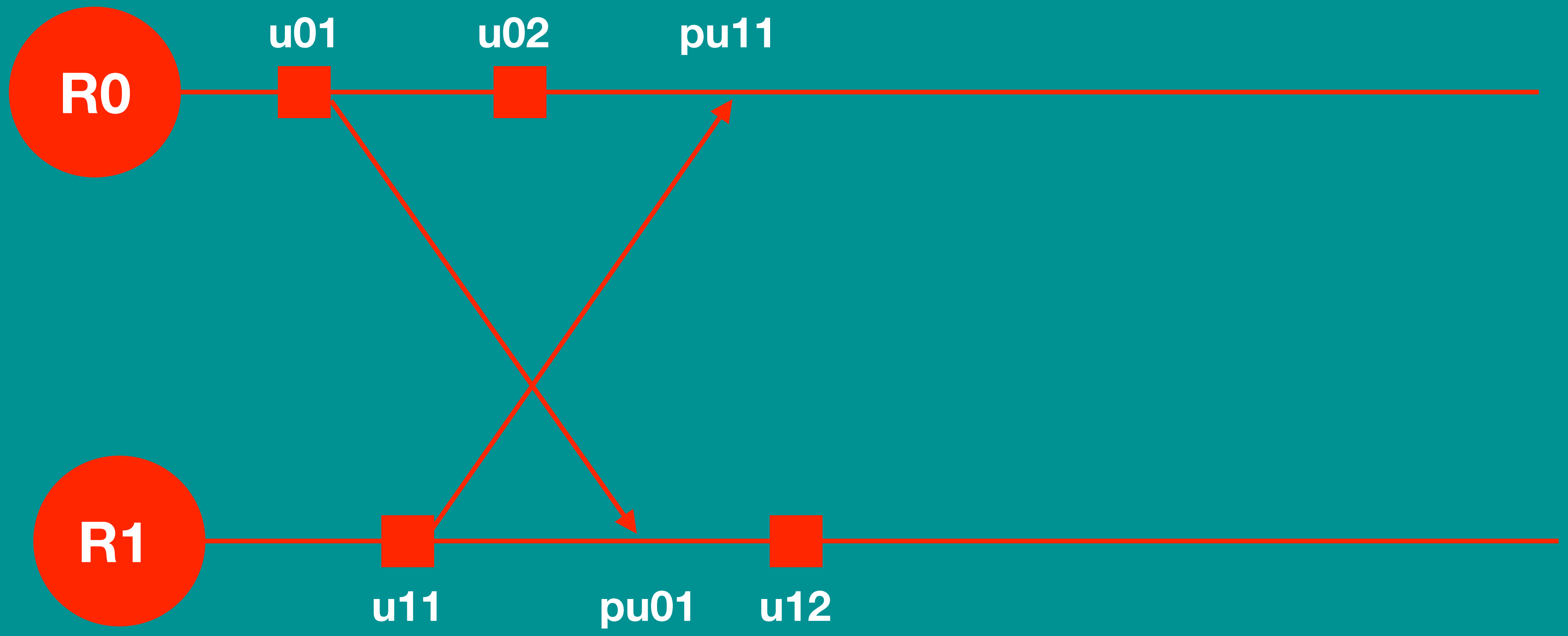
Any data structure with a set of well defined operations can be implemented as CRDT using this approach.

There are many well known CRDT types like PNCounter, GCounter, GSet, AWSet, RWSet, MVRegister etc.

In general, a replica will have multiple instances of different CRDT types.

Let us take one CRDT instance and see how does it work.

How does it work?



R0 and *R1* are replicas hosting a CRDT instance

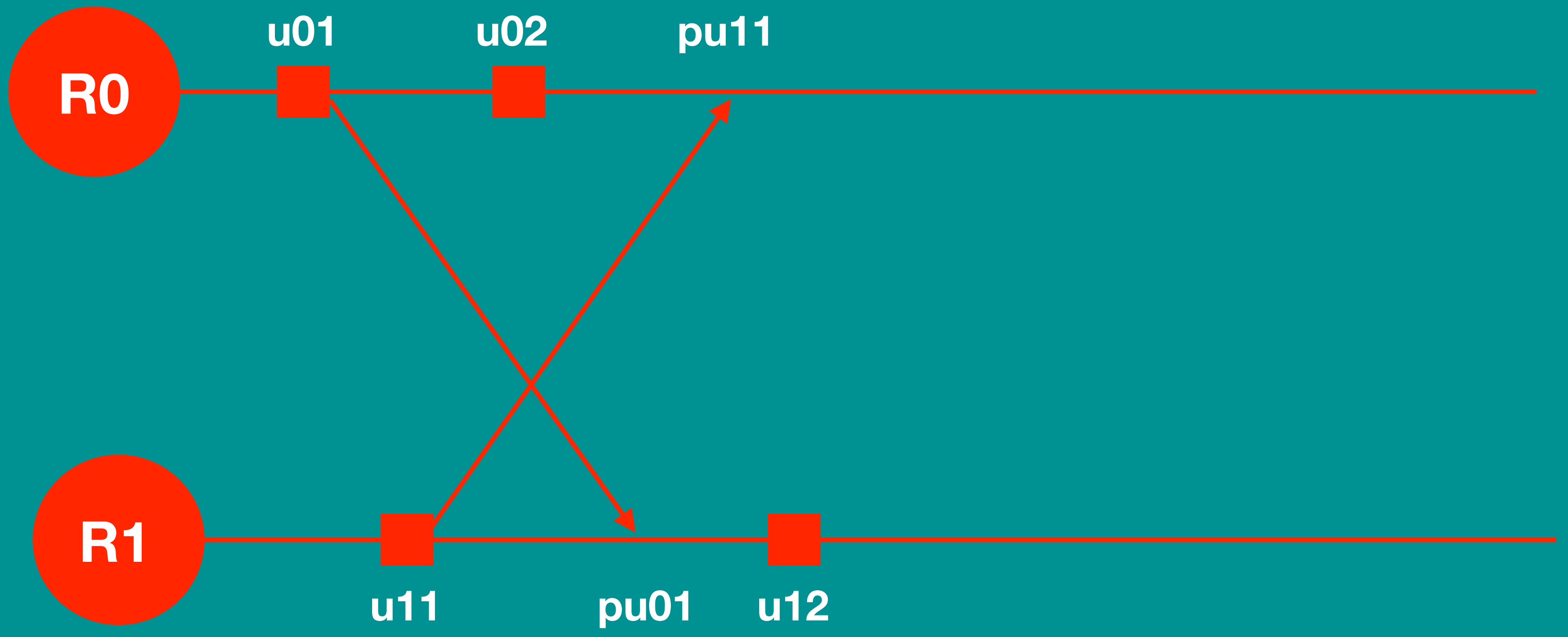
u01, u02 are updates from user on *R0*
u11, u12 are updates from user on *R1*

when an update is made by user on a replica, replica will disseminate update asynchronously to its peer replicas

pu01 is an update made on *R1* that happened because *R0* sent an async message to *R1* corresponding to its update *u01*
pu11 is an update made on *R0* that happened because *R1* sent an async message to *R0* corresponding to its update *u11*

in general, a CRDT instance on a replica will get updated by users and peers

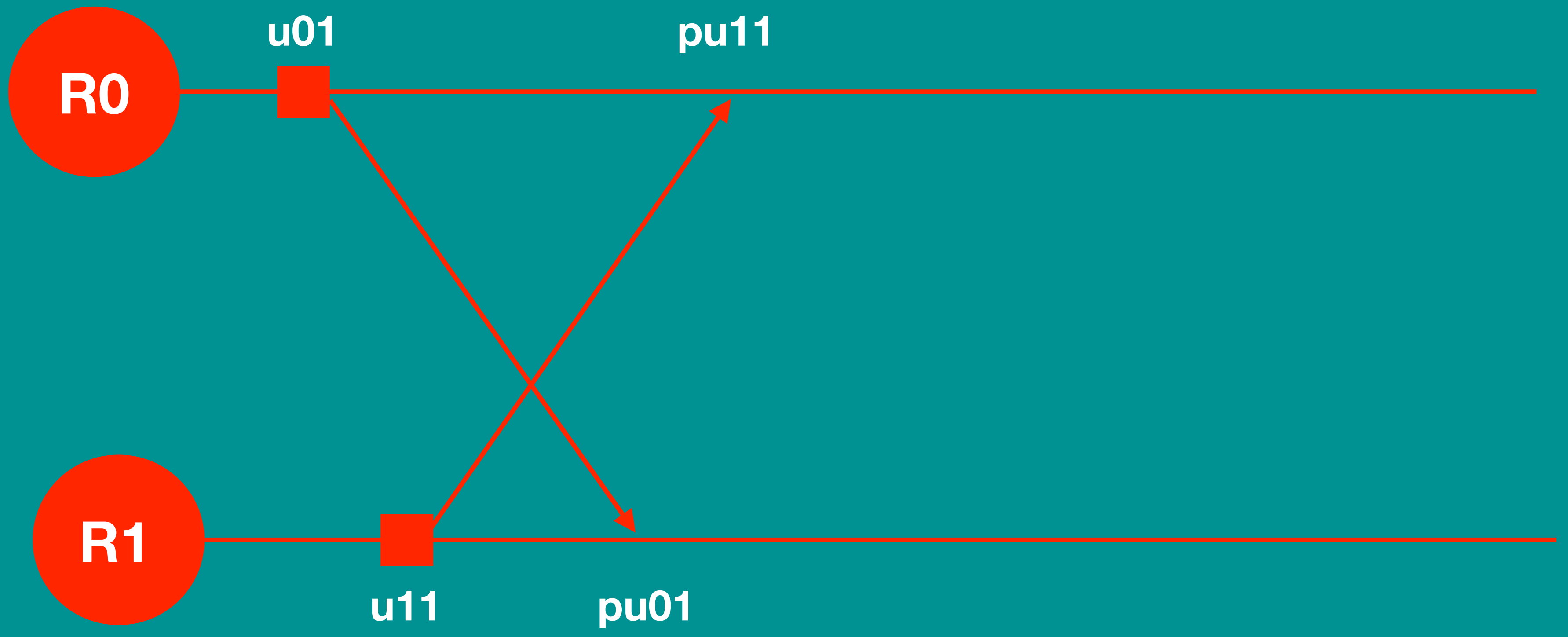
Causal Order



u02 happened after *u01*
u12 happened after *u11*

what about *(u01, pu11)*, *(u02, pu11)*, *(u11, pu01)* and *(u12, pu01)*

Causal Order

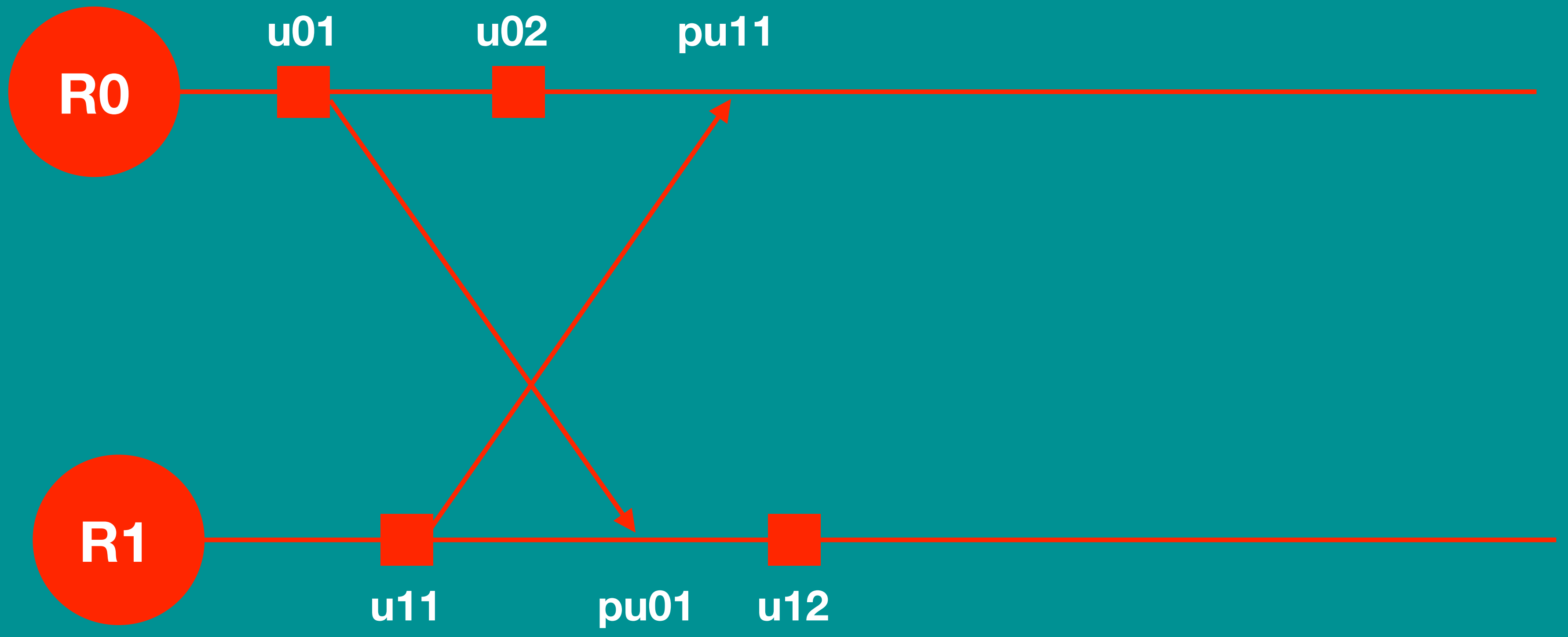


what if CRDT instance is a *counter* and *u01* is INC on counter and *u11* is DEC on counter

what if CRDT instance is a set and *u01* is ADD(5) on set and *u11* is RMV(5) on set.In such cases we define concurrent operation semantics, ADD wins over RMV (AWSet) or RMV wins over ADD (RWSet)

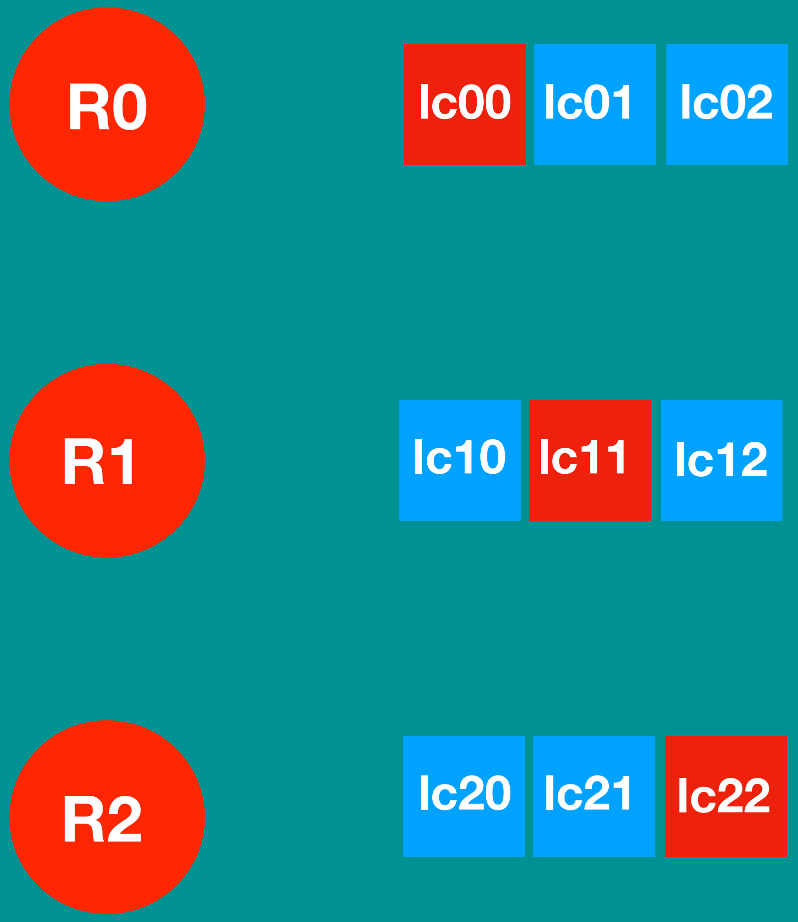
one may expect that at R0, u01 will be executed followed by pu11(u11)
and at R1, u11 will be executed followed by pu01(u01)

Implementation Requirement



- every replica must receive all the updates from its peer replicas *exactly once* in a *causal order*
- we know that network is unreliable and messages can be lost, replica can receive messages out-of-order and it can also receive duplicate messages
- a replica may loose connection with a peer replica because of network partition or peer replica is down
- even with such unreliable network and failure of replica, we can establish a *reliable causal broadcast* that will ensure *at-least once delivery*
- if we have *at-least once delivery* then receiving replica will be required to handle duplicate messages and out-of-order messages to achieve *exactly once delivery* in a *causal order*

Logical Clock and Vector Clock



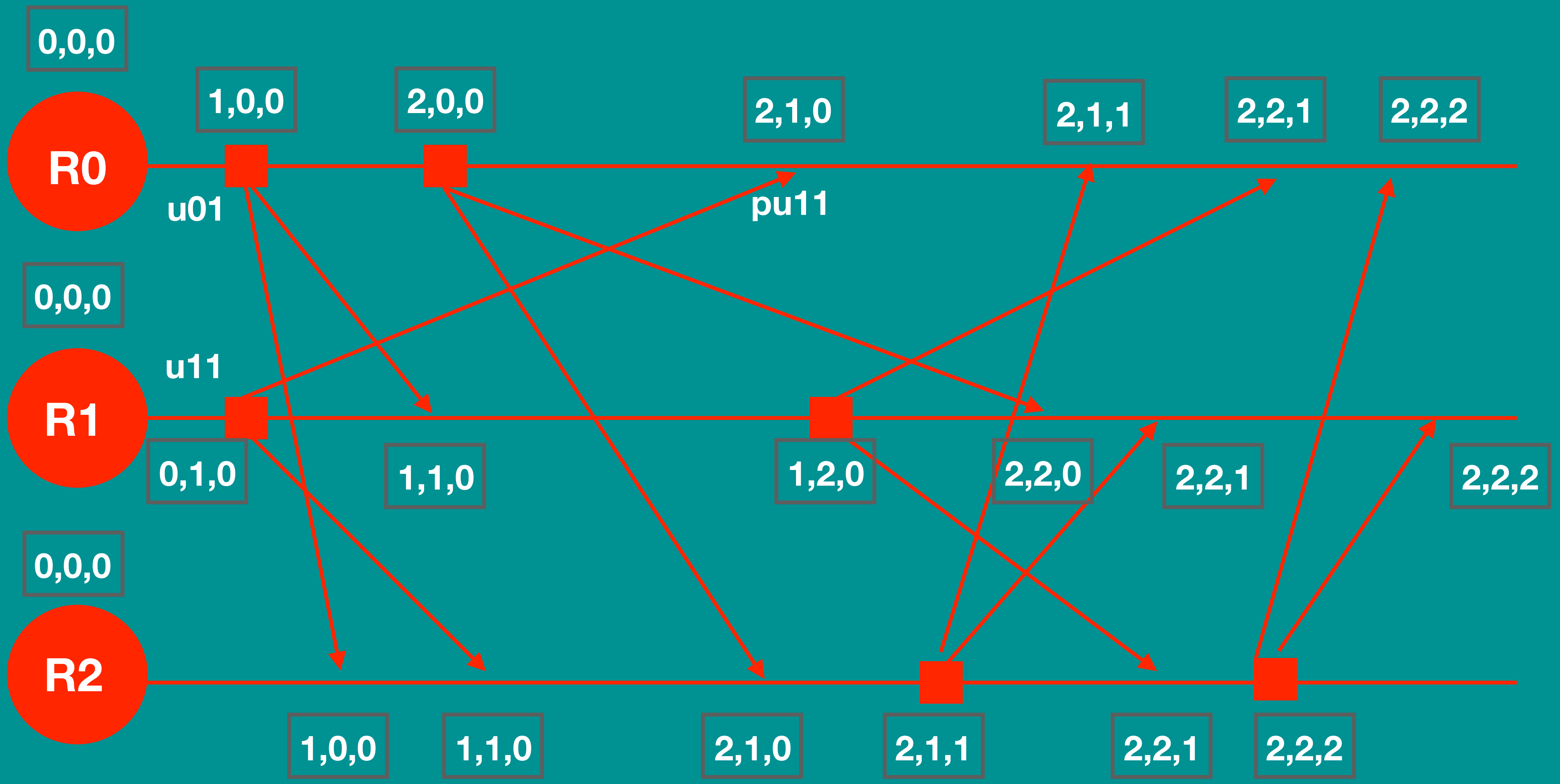
- lc00 is the value of logical clock of R0
- lc01 is the best known value of logical clock of R1 known to R0
- lc02 is the best known value of logical clock of R2 known to R0

- lc11 is the value of logical clock of R1
- lc10 is the best known value of logical clock of R0 known to R1
- lc12 is the best known value of logical clock of R2 known to R1

- lc22 is the value of logical clock of R2
- lc20 is the best known value of logical clock of R0 known to R2
- lc21 is the best known value of logical clock of R1 known to R2

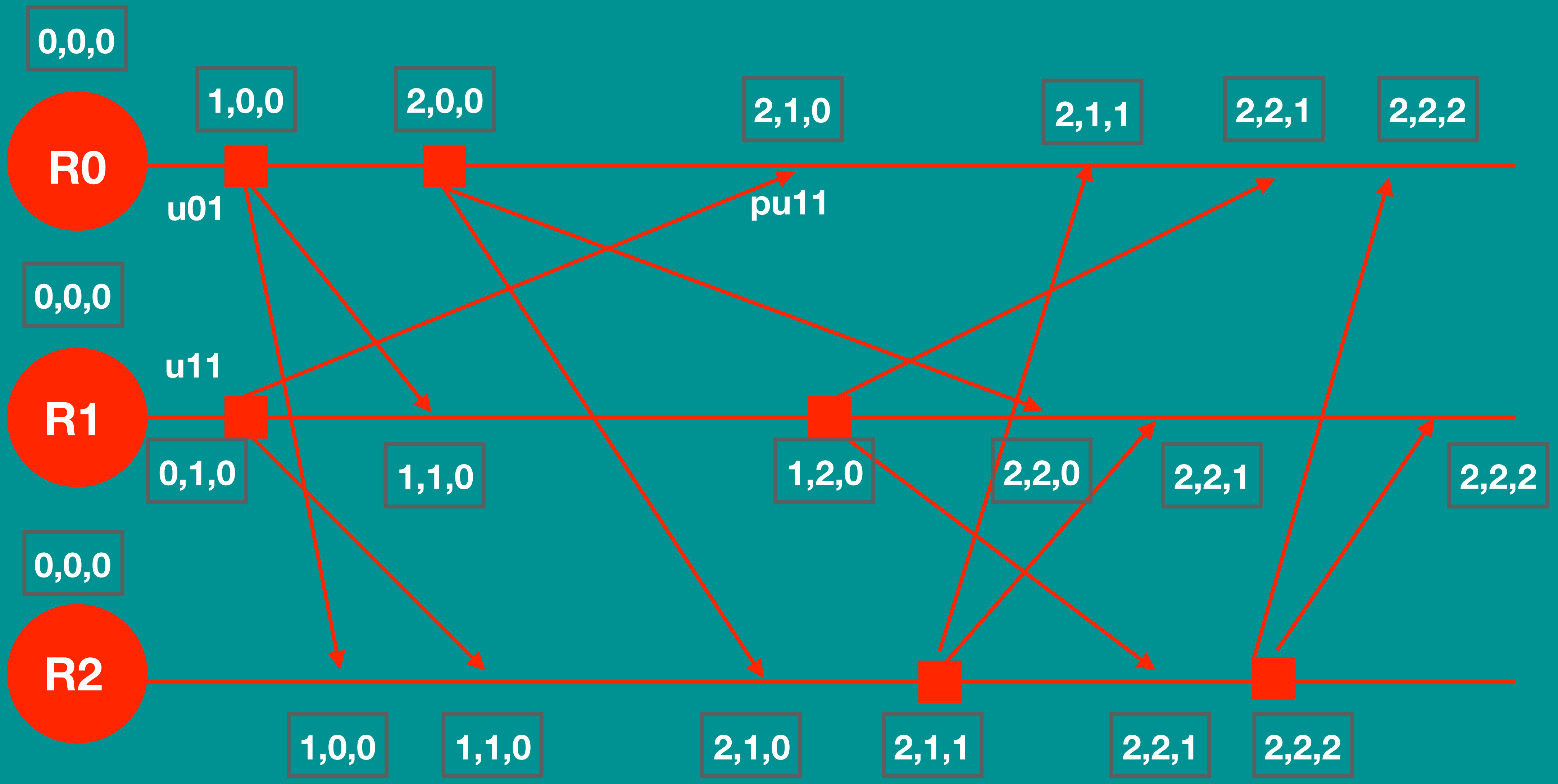
- a *logical clock* is just an integer number that is being incremented with every update operation
- a *vector clock* is a collection of *logical clocks,* the size of *vector clock* is same as number of replicas that are there in the system
- each replica has a middleware component that maintains a *vector clock* (for each CRDT instance), middleware uses *vector clock* to maintain value of its own *logical clock* and peer replica’s *logical clock* (best known value)

User updates a CRDT instance



- user update message consists of CRDT Instance and operation/args (crdt_inst, ops_args)
- user sends user update message to middleware component of a replica
- middleware updates its logical clock in the vector clock and adds its node_id and vector clock to user message and makes it a replica message (node_id, vector_clock, crdt_inst, ops_args)
- middleware immediately delivers replica message to its local replica
- middleware sends replica message to all the peer middleware

Messages at each Replica

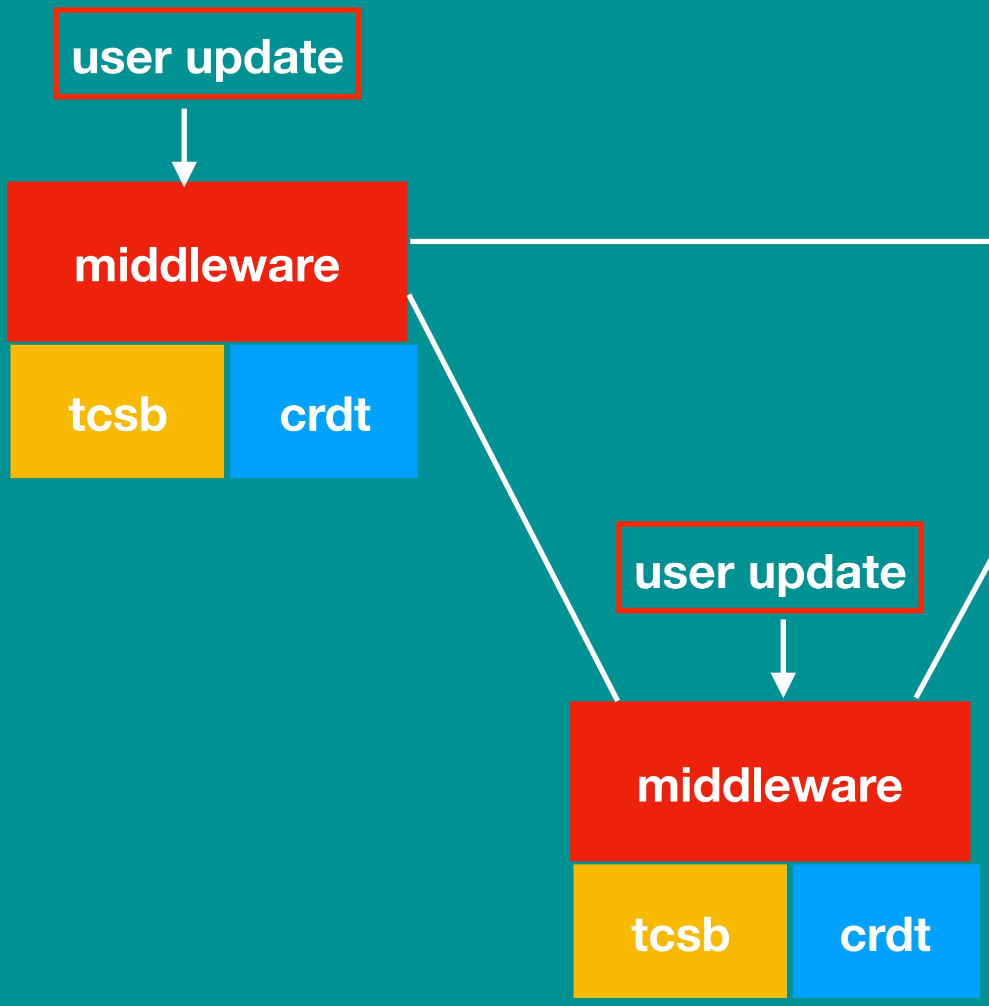


R0	R1	R2
1,0,0	0,1,0	1,0,0
2,0,0	1,0,0	0,1,0
0,1,0	1,2,0	2,0,0
2,1,1	2,0,0	2,1,1
1,2,0	2,1,1	1,2,0
2,2,2	2,2,2	2,2,2

➡

1,0,0 0,1,0
2,0,0 1,2,0
2,1,1
2,2,2

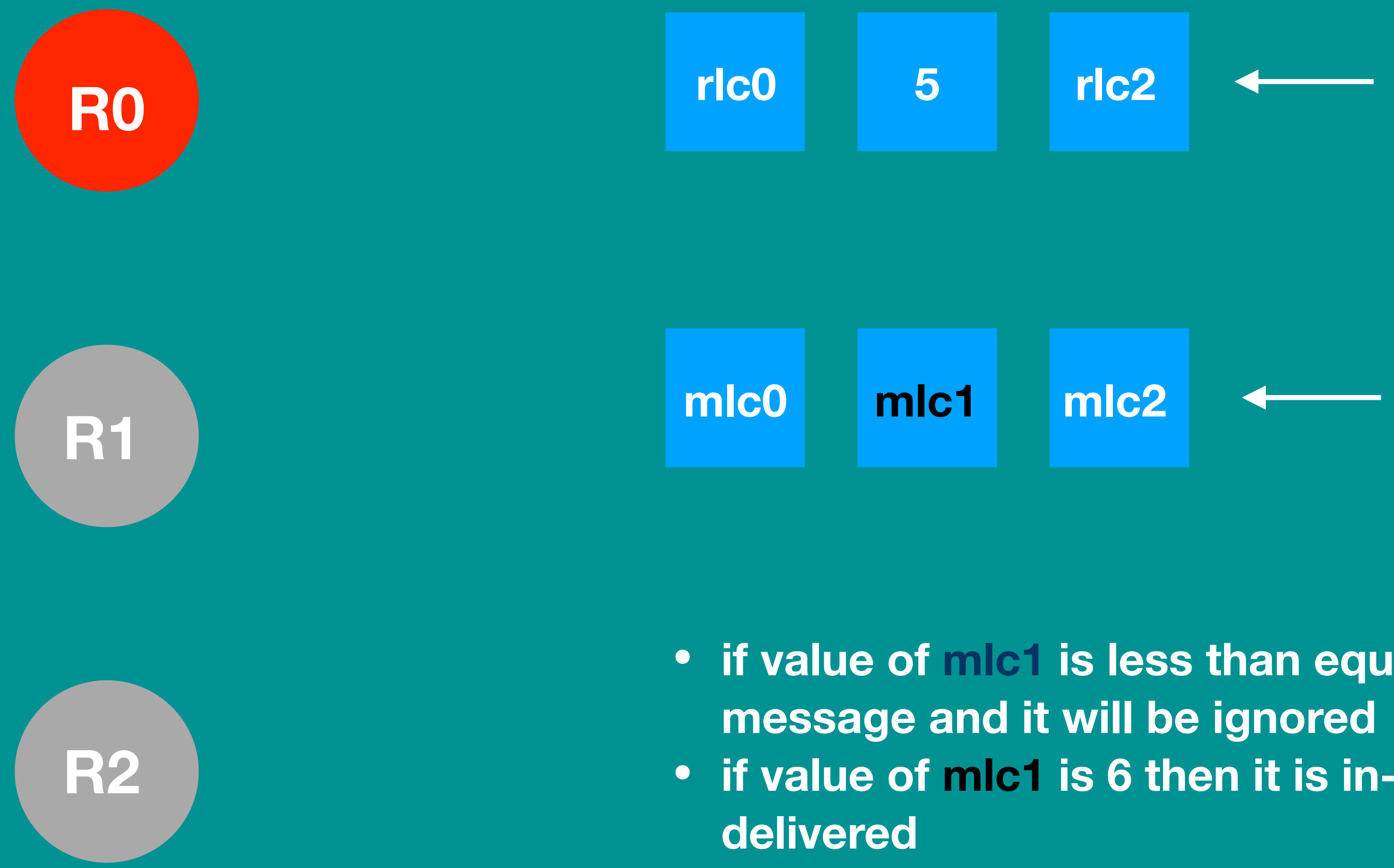
System Architecture



- middleware and tcsb form the CRDT machinery
- crdt is crdt data (set, interger) and a PO-Log (partially ordered log) that temporarily holds messages
- user sends update request to middleware
- middleware validates the user request and ignores invalid request (user request with ADD operation to a PNCounter CRDT instance is invalid request)
- for a valid user request, middleware updates the local vector clock, adds the node_id and its vector lock to update request.
- update request immediately delivered to local replica and sent to middleware of all the peer replicas

- when a replica middleware receive an update request from peer, message is checked for duplicate and out-of-order message. A duplicate message is ignored, an out-of-order message is added to a pending msg log and not delivered to replica

Validation and Exactly Once Delivery at one replica

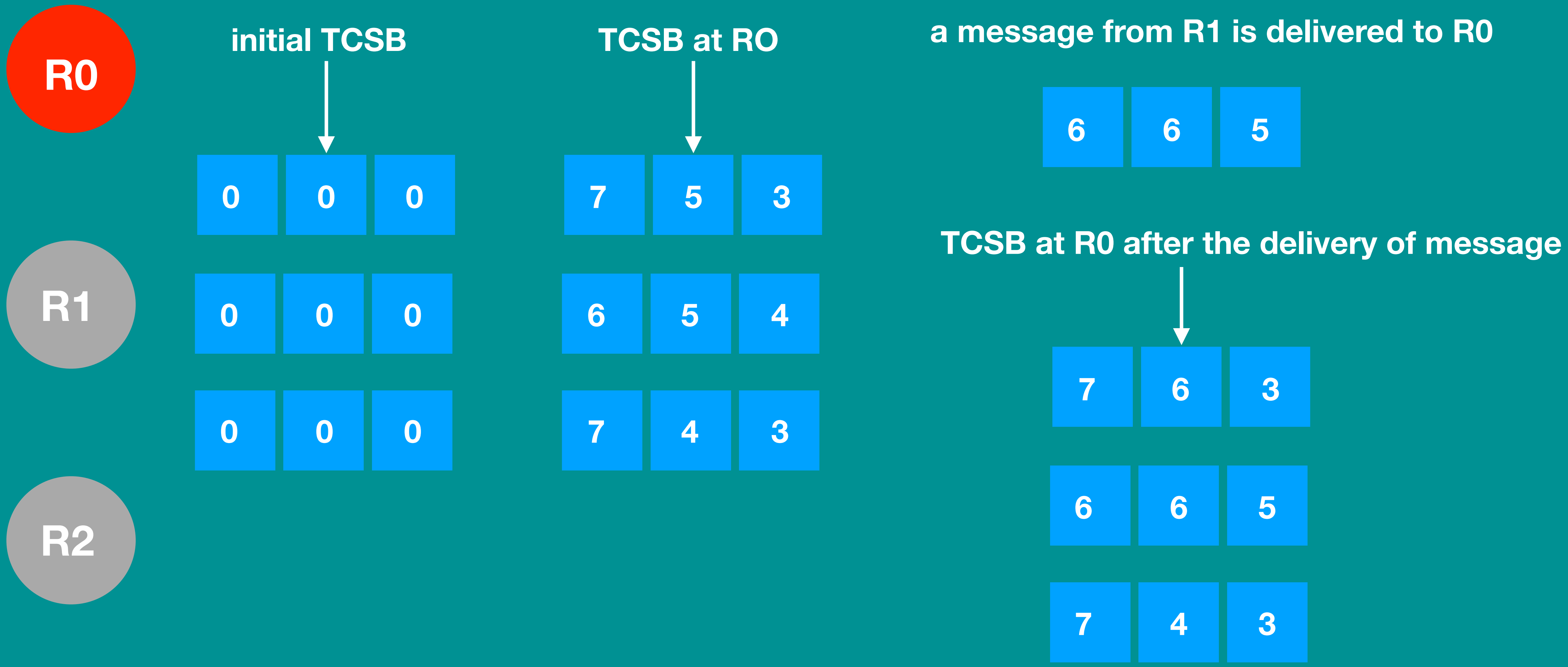


R0 vector clock indicates the it has been delivered 5 messages from R1

vector clock of a new update message from R1 received by middleware of R0

- if value of mlc1 is less than equal to 5 then it is a duplicate message and it will be ignored
- if value of mlc1 is 6 then it is in-order message and it will be delivered
- once a message is delivered, replica will update the value of logical clock at sender replica’s logical clock position
- if value of mlc1 is more than 6 then it is out-of-order message and it will be added to pending message log
- if we have a at-least once delivery machinery, this validation will ensure exactly once delivery

Tagged Causal Stable Broadcast (TCSB) at one replica



R0

R1

R2

initial TCSB

0	0	0
0	0	0
0	0	0

TCSB at RO

7	5	3
6	5	4
7	4	3

a message from R1 is delivered to R0

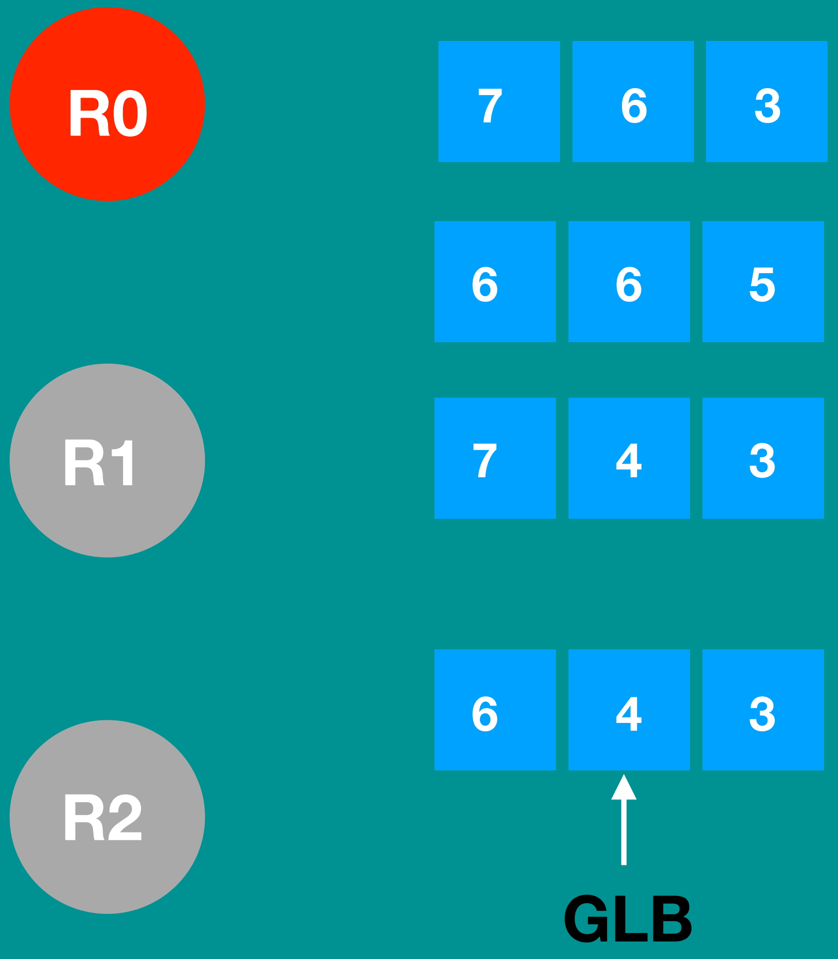
6	6	5

TCSB at R0 after the delivery of message

7	6	3
6	6	5
7	4	3

- each replica has TCSB data that consists of its own vector clock and copy of vector clock from the last message received from its peer replicas
- initially, all the logical clock values in every vector clock will be 0.

PO-Log, Causal Stable Messages at one replica



R0

7 6 3

6 6 5

R1

7 4 3

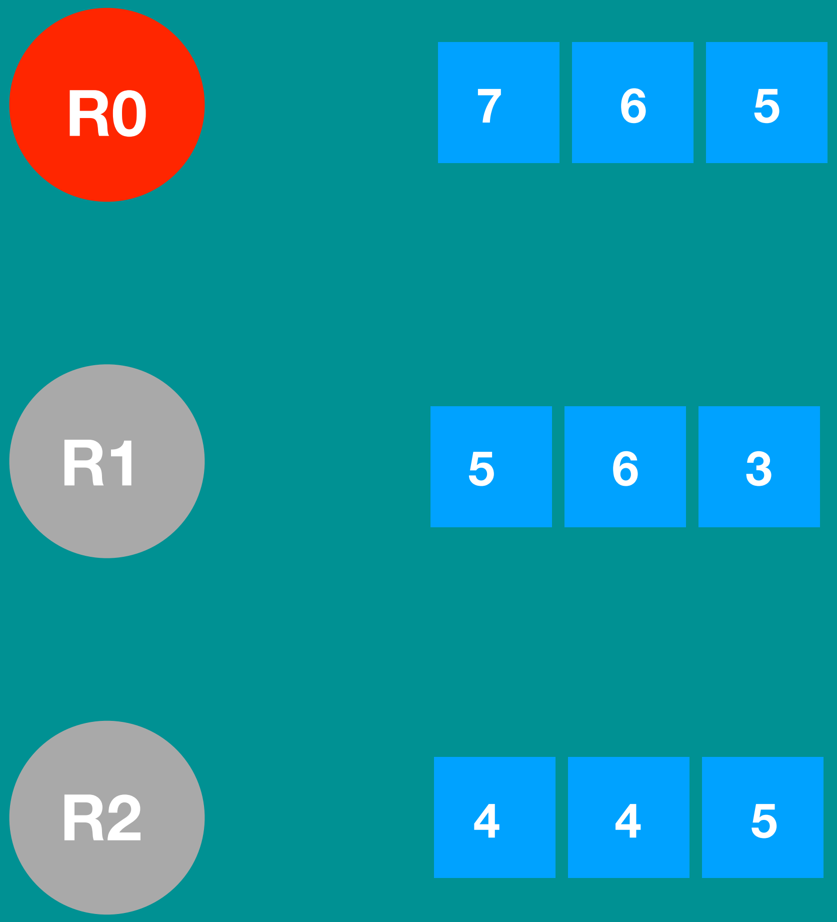
R2

6 4 3

GLB

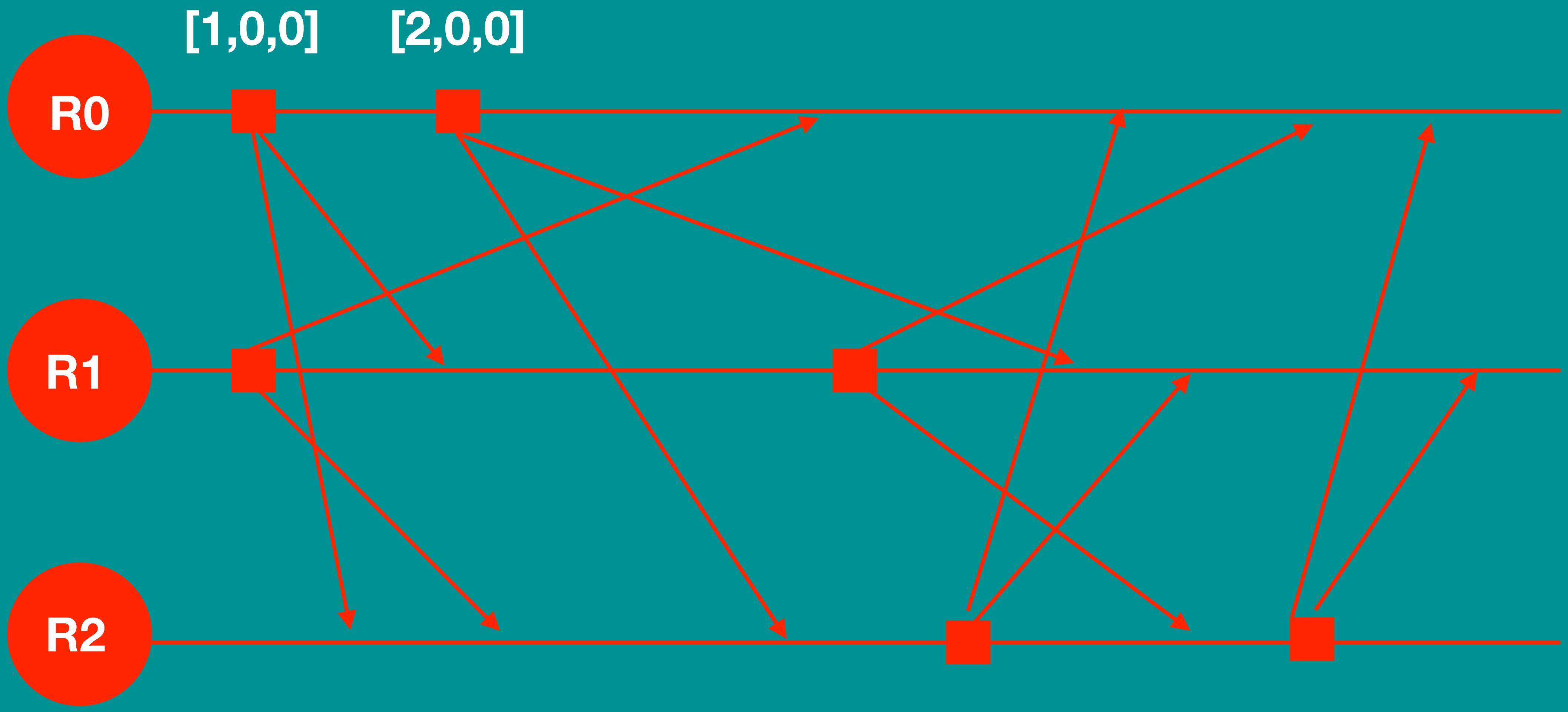
- once a message is delivered to a replica, in addition to updating TCSB, message is added to PO-Log
- a replica uses PO-Log and TCSB to create undelivered message list for a peer replica
- for a commutative type of CRDT, a message is applied to CRDT
- every time a message is delivered to a replica, it computes a **Greatest Lower Bound** of TCSB
- for non-commutative CRDT, we use **GLB** and PO-Log to determine if message is causally stable (will not have any concurrent message in the future) and if a message is causally stable, it is applied to CRDT
- A GLB value [6,4,3] says that every replica has been delivered 6 messages from R0, 4 messages from R1 and 3 messages from R2, so any message in PO-Log that has a vector clock that is <= [6,4,3] will not be required when undelivered message list is created and it can be removed
- any message that has applied to CRDT and it has delivered to every peer replica is removed from PO-Log

At-Least Once Delivery and handling of failure at one replica



- a replica makes undelivered message list for all its peer replicas after each user update
- TCSB value at R0 after an user update
- R0 will add update message 6 and 7 (made by user on R0) in the undelivered message list for R1
- in addition to this, R0 will also include update message 4 and 5 (made by user on R2 and these messages are available with R0 in its PO-Log)
- R1 may also receive same user update message (made by user on R2) and this is OK because every replica can handle duplicate messages
- advantage of sending additional messages 4 and 5 from R0 on behalf R2 is that if R2 replica is down then R1 will not receive these messages as long as R2 is down and replica R0 and R1 will not converge

Vector Clock Message



- user makes only 2 updates on replica R0, the vector clock of last message delivered to R1 and R2 from R0 will be [2, 0, 0]
- but R0 is receiving peer updates from R1 and R2 and vector clock of R0 is getting updated with every peer update
- R1 can not know if its user update made it to R0 because it not reflected in the last message received by R1 from R0. R1 will keep sending its user updates R0
- to handle this kind of situation, in the absence of user updates R0 will periodically send its vector clock to all its peer replicas

Add, Remove Replica and Cluster detail

- every replica has a CLUSTER_DETAIL object in its state
- a CLUSTER_DETAIL object consists of cluster_id, ver_num and node_list
- every replica in a replica cluster must have same cluster_id in their CLUSTER_DETAIL object to participate as a cluster member and communicate with peer replicas
- when a new replica is added to the cluster or an existing replica is removed from cluster, a new CLUSTER_DETAIL is created that has same cluster_id but ver_num is incremented and node_list is changed based on whether a replica is added or removed
- new CLUSTER_DETAIL is sent to all the existing (surviving) replicas and replicas adopt to new CLUSTER_DETAIL as long as one replica receives new CLUSTER_DETAIL
- when a replica receives a new CLUSTER_DETAIL, it upgrades itself by changing its own state
- when a replica prepares undelivered message list for its peer replicas, it also includes its own CLUSTER_DETAIL object with undelivered message list
- when a replica receive undelivered message list, it checks the CLUSTER_DETAIL of message with its own CLUSTER_DETAIL
- it accepts the undelivered message if they have same cluster_id
- if ver_num of both the CLUSTER_DETAIL objects are same then business as usual
- if the ver_num of CLUSTER_DETAIL with undelivered message list is lower than CLUSTER_DETAIL of replica then undelivered message list is upgraded before it is applied to replica
- if the ver_num of CLUSTER_DETAIL with undelivered message list is higher than CLUSTER_DETAIL of replica then replica is upgraded before undelivered message list is applied to replica

Q & A

Thank you

code is available at github.com/gyanaggarwal/pure_ops_crdt