CHAPTER SIX

HIERARCHICAL MULTI-CRITERIA UPDATES ORDERING MECHANISM

6.0 Overview

The purpose of updates ordering in our strategy is to provide each host with a unified set of ordered updates for each replicated object. This ordering provides a level of consistency that the research called consistency with ordering guarantee level. In this level, each replica should have identical and ordered values in same manner as its counter part in the master server. This is in contrast to the level of the consistency that is previously discussed, which implies that each replica should be up to date as compared with the replica that is stored in the master server. This level can be achieved only through updates propagation without needing to enforce a unified ordering as in the former level. Accordingly, the research called the latter level consistency without ordering guarantee level. It is clear that the latter imposes low overhead (storage and communication) as compared with the former. The research provides the two levels in order to meet the requirements of different applications.

6.1 Significance of Updates Ordering

The significance of updates ordering in our replication system originates from the existing of large number hosts that replicate and generate large number of updates concurrently on the same replicated data items. These hosts need to agree on a unified ordering for all updates that occurred in their shared data items. This unified ordering ensures the correct semantics of the replicated service system in case of considering the consistency with ordering guarantee level. For example, consider a data item X that is replicated on n replicas in both fixed and mobile hosts and different numbers of update operations are performed on the set of replicas $\{R_1, R_3, R_6, R_8\}$, which contains four replicas out of the n replicas. Those updates are issued on each replica according to the following order (u_{11}, u_{12}, u_{13}, u_{14}, u_{15}), (u_{31}, u_{32}, u_{33}), (u_{61}, u_{62}, u_{63}, u_{64}), and (u_{81}, u_{82}), respectively. This scenario is depicted in Figure 6.1.1.

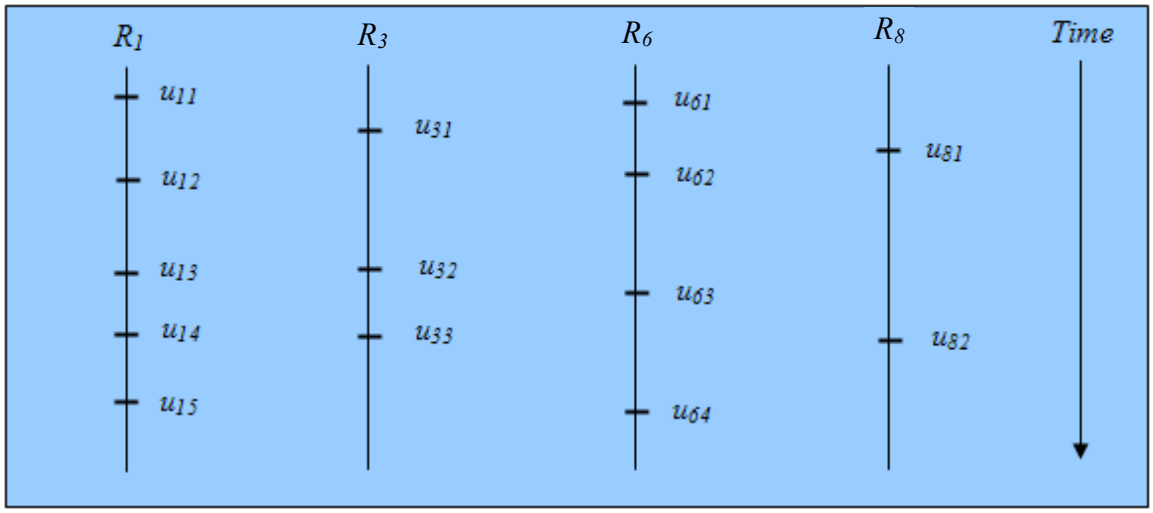

FIGURE 6.1.1 A Scenario of the Local Ordering of the Updates in the Set of Four Replicas {R_1, R_3, R_6, R_8}

Figure 6.1.1 shows the order of updates that occurred in each replica during a specific time period. Each replica of X needs to receive the set of updates that occurred in those four replicas in a unified order. To ensure this unified order, a reasonable total order of those update operations should be defined and enforced over all replicas of X.

Accordingly, our main question in this chapter is: How and in what sequence should the updates be ordered to preserve the data consistency through a unified order if these updates are carried out by a large number of distinct fixed and mobile hosts?. As a response to this question, an ordering mechanism is presented that provides each host on different rims with a set of unified ordered updates. This unified order is produced as a result of participating of all center points in the ordering process through enforcing a hierarchical multi-criteria ordering on updates, which are collected on each center point from underlying hosts in its area. The ordering criteria include timestamp attributes that characterize the events of updates generation and updates propagation. The propagation event can be divided into two sub events, which are: send and receive. The timestamp attributes are associated with the individual updates as well as the messages that propagate these updates to the responsible center points in order to facilitate the ordering process in the higher levels.

The proposed ordering mechanism is based on the causal ordering. However, the constraint of causal ordering is not always necessary to be hold in large scale mobile distributed database systems as proved by the following assertion.

Assertion 6.1.1 Casual ordering constraint that requires the order of receiving two messages should be same as the order of their sending does not hold always in LMDDBSs.

Proof. Consider two updates u_1 and u_2 are sent from *MHi* and *MH*j, respectively, to the same destination *D*, such that $u_1 \rightarrow u_2$ (u_1 is sent before u_2). If sending of u_1 experienced more delay than sending of u_2 on the communication link due to variability and limitation of the available bandwidth, network delays and the distances between both *MHi* and *MHj* and *D*, this increases the probability of receiving u_2 before u_1 at *D* (although u_1 was sent before u_2). Thus, the constraint of causal ordering, which requires that the order of receiving u_1 and u_2 should follow the order of sending them may not be hold in this case.

Accordingly, the proposed ordering mechanism provides new constraints for enforcing causal ordering in LMDDBSs by eliminating the dependency between the orders that are produced by considering both receiving and sending events, as well as updates generation event. Based on these constraints, the ordering is performed through multiple checks by comparing updates based on the occurrences of the three events according to the sequence: generation, sending, and receiving. In this multiple check, updates are compared based on the occurrence of the next event if and only if comparing them using the occurrence of current event has produced the same order. The new constraints are provided in the following section, which are based on the type of event and the location where the event has taken place.

In the proposed mechanism, updates that occurred on the lower levels ore ordered on the higher level when Bottom-Up propagation takes place. On the other hand, the messages that ship the totally ordered updates from the higher level are sent to the lower level in same order. This order follows the order of these messages as were sent by the main center point when top-down propagation takes place.

6.2 Updates ordering in Case of Bottom-Up Propagation

To maintain the level of consistency with ordering guarantee, updates that are generated on the different levels should be totally ordered and then propagated to other replicas. In Bottom-Up propagation, updates that occurred in the underlying levels are totally ordered in the higher levels (i.e. next inner rims) till reaching a unified total order in the highest level (i.e. the main center point) for all updates.

Then, the main center point propagates ordered updates to underlying levels using Top-Down propagation. The ordering is performed in case of Bottom-up propagation on the level of the updates (i.e. updates are compared instead of messages). The ordering mechanism here is based on the casual order relation, which is specified as in the following subsection. Note that the statement *totally ordered* in this section does not mean the total order relation as in the Top-Down propagation, but it states that total ordering is performed on a set of updates that are collected from all underlying level hosts using causal order relation. This is because the total order relation has a different definition than for the causal order relation.

6.2.1 Causal Ordering Model

This section first provides formal definitions for three types of the causal ordering relation among updates by considering the following sequence of the operations: generation, sending, and receiving. These definitions are produced based on the *happened-before* relation (see section 2.7), and the characteristics of the proposed updates propagation protocol (i.e. updates are propagated through certain paths between the hosts in the different rims). This means that the definition of *happened-before* relation is exploited in defining the three types of causal ordering relation in order to act in accord with the ordering requirements in LMDDBSs. Second, the concepts of the concurrent messages and concurrent updates are provided. Lastly, the definition of *Conditioned causal ordering relation* is given.

Definition 6.2.1.1 *causal ordering relation for generation operation* "$\rightarrow_u$"

Given two updates u and v, generated at site i and j, respectively, then $u \rightarrow_u v$, iff (1) $i = j$ and the generation of u happened before the generation of v, or (2) $i \# j$ and the generation of u at site i happened before the generation of v, or (3) there exists an *update* w, such that $u \rightarrow_u w$ and $w \rightarrow_u v$.

Definition 6.2.1.2 *causal ordering relation for send operation* "$\rightarrow_s$"

Given two messages m_1 and m_2, sent by site i and j, respectively, to the higher level (either to same center point or different center points), then $m_1 \rightarrow_s m_2$, *iff* (1) $i = j$ and the sending of m_1 happened before the sending of m_2, or (2) $i \# j$ and the sending of m_1 from site i happened before the sending of m_2, or (3) there exists a message m_3, such that $m_1 \rightarrow_s m_3$ and $m_3 \rightarrow_s m_2$.

Definition 6.2.1.3 *causal ordering relation for receive operation* "$\rightarrow_r$"

Given two messages m_1 and m_2, received by either same center point or different center points from site i and j, respectively, then $m_1 \rightarrow_r m_2$, *iff* (1) $i = j$ and the receiving of m_1 happened before the receiving of m_2, or (2) $i \# j$ and the receiving of m_1 from site i happened before the receiving of m_2 by any responsible point, or (3) there exists a message m_3, such that $m_1 \rightarrow_r m_3$ and $m_3 \rightarrow_r m_2$.

According to the definitions 6.2.1.2 and 6.2.1.3, the casual relation is defined between messages instead of individual updates. This is because updates information are sent in a form of message. Accordingly, updates that are shipped with these messages inherit both send and receive time attributes of their messages. In definition 6.2.1.3, the causal ordering relation is specified for the receive operation because the second optional condition of lamport's happened-before relation cannot be implemented in this case. This is follows assertion 6.1.1 in that the order of sending two messages does not imply that they will be received in the same order as they were sent.

Definition 6.2.1.4 *Concurrent sent messages "$m_1 ||_s m_2$"*

Two distinct sent messages to the higher level (either to same the center point or different center point) m_1 and m_2 are "concurrent" if $\neg (m_1 \rightarrow_s m_2)$ and $\neg (m_2 \rightarrow_s m_1)$.

Definition 6.2.1.5 *Concurrent received messages "$m_1 ||_r m_2$"*

Two distinct messages received by the higher level (by two different center points) m_1 and m_2 are "concurrent" if $\neg (m_1 \rightarrow_r m_2)$ and $\neg (m_2 \rightarrow_r m_1)$.

This definition does not state that the two messages are received by same center point. This is because we do not expect that two messages satisfy the condition $\neg (m_1 \rightarrow_r m_2)$ and $\neg (m_2 \rightarrow_r m_1)$ exactly if they are received by same center point from underlying hosts.

Definition 6.2.1.6 *Concurrent updates "$u \parallel v$"*

Two distinct updates u and v that are shipped to the higher level via messages m_1 and m_2, respectively, are "concurrent" iff:

(i) $\neg (u \rightarrow_u v)$ and $\neg (v \rightarrow_u u)$ or

(ii) $(\neg (u \rightarrow_u v)$ and $\neg (v \rightarrow_u u))$ and $(\neg (m_1 \rightarrow_s m_2)$ and $\neg (m_2 \rightarrow_s m_1))$ or

(iii)$(\neg (u \rightarrow_u v)$ and $\neg (v \rightarrow_u u))$ and $(\neg (m_1 \rightarrow_r m_2)$ and $\neg (m_2 \rightarrow_r m_1))$ and $(\neg (m_1 \rightarrow_r m_2)$ and $\neg (m_2 \rightarrow_r m_1))$

Definition 6.2.1.7 *Conditioned causal ordering relation* "$\rightarrow_{cu}$"
Given two *updates* u and v, generated at site i and j, respectively, and sent via messages m_1 and m_2, respectively, to the responsible center point C, then $u \rightarrow_{cu} v$, *iff*
(i) ($\neg (u \rightarrow_u v)$ and $\neg (v \rightarrow_u u)$) and $m_1 \rightarrow_s m_2$ or
(ii) ($\neg (u \rightarrow_u v)$ and $\neg (v \rightarrow_u u)$) and ($\neg (m_1 \rightarrow_s m_2)$ and $\neg (m_2 \rightarrow_s m_1)$) and $m_1 \rightarrow_r m_2$ or
(iii) ($\neg (u \rightarrow_u v)$ and $\neg (v \rightarrow_u u)$) and ($\neg (m_1 \rightarrow_r m_2)$ and $\neg (m_2 \rightarrow_r m_1)$) and ($\neg (m_1 \rightarrow_r m_2)$ and $\neg (m_2 \rightarrow_r m_1)$) and $m_3 \rightarrow_s m_4$ in case of u and v are sent to their responsible center points C_1 and C_2 via m_1 and m_2 and then those updates are sent from C_1 and C_2 to their responsible center point Z via m_3 and m_4.

According to the first condition, *the Conditioned causal ordering relation* implies ordering of u and v based on the sent event of their messages to C in case of u and v are concurrent according to the generation event. Similarly, based on the second condition, this relation implies ordering of u and v based on the receive event of their messages by C in case of u and v are concurrent according to both generation and send events. The third condition implies ordering of u and v in the higher level based on the send event in case of that those updates are concurrent according to the three operations generation, send, and receive, which are performed in the lower levels.

Thus, based on the above description, *Conditioned causal ordering relation* acts as a method for resolution of concurrent updates (i.e. assigning them different orders).

6.2.2 Implementing Causal Ordering

The causal ordering in the higher level is carried out based on comparing updates according to a relation called *Treated-before relation*. This relation represents a direct implementation of the *Conditioned causal ordering relation* for performing the multiple checking that is involved in the ordering process. This relation relies on timestamp attributes for both updates and messages that ship these updates to the higher levels. Accordingly, these attributes can be classified as follows:

- The timestamps of the updates generation operations.
- The timestamps of both send and receive operations for the messages.

The timestamps are assigned locally on each host based on the value of a variable called Real-Like Clock. Updates information is shipped to the higher level

via a message called RU-Message. The contains of each received RU-Message are placed on a queue in the higher level. This queue represents a set of collected updates from underlying level during the updates collection period. Updates on this queue are compared using the *Treated-before relation*.

The following subsections provide the specifications of Real-Like Clock, RU-Message, *Treated-before relation*, and the ordering queue.

6.2.2.1 Real-Like clock

Each host maintains a local variable, called Real-Like clock (RC). It is used for timestamping the three events (generate, send, and receive) that originated on the host. At any time, the current value of this clock should be same as its counter part in the main center point. For achieving this, every center point is responsible also for maintaining the value of RC in its underlying hosts. The RC'*s* value is maintained periodically in each underlying host through a periodical checking algorithm. The method of advancing the value of this variable is assumed to be same as the advancing method of the physical clock. Accordingly, the value of RC is considered as a complex value of certain atomic values that are arranged into two main parts. The first part represents the time and the second part represents the date. The reason behind advancing the value of RC in same manner as the physical clock is to ensure a unified advancing method at all hosts other than advancing its value according to the time of the next event as performed in the logical clock and vector clock (see section 2.7). This follows that the updates occurred in a certain host will be compared with the updates that originated from other hosts. Thus, if we advance the RC's value according to the time of the next event, we find that during the time period between two events there are many updates may occur in the other hosts. Therefore, to provide a fair unified ordering, our ordering mechanism relies on advancing the value of RC in all hosts in the same manner as the physical clock and considering the value of the *RC* at the time when each event occurred.

The rules for advancing RC's value and timestamping events are as follows.

1. **Initially**, Val(*RC*) := Val(*RC*) at the main center point.
2. **When** a tick occurs $\Rightarrow$ Val(*RC*) is advanced in same manner as the physical clock.
3. **Whenever** an *event* occurs $\Rightarrow$ TS(*event*) := Val(*RC*) when event happened.

In these rules, Val(*RC*) is the value of the local RC at a specific host. Initially, the value of RC at each host is identical to the value of RC at the main center point. This value is advanced locally according to the frequencies of ticks as in the physical clock. Each event is timestamped (denoted by TS(*event*)) using the current value of the local RC when the event takes place.

Assertion 6.2.2.1.1 The RC based timestamping scheme as compared to the logical clock based timestamping scheme posses the following property: given two events e and e^- occurred on different hosts H_1 and H_2, respectively, if TS(e) < TS(e^-) then $e \rightarrow e^-$, where TS(e) is the timestamp of e, and TS(e^-) is the timestamp of e^-.

Proof. First, we implement the logical clock (LC) timestamping as follows. Consider an initial event e_0 occurred on both H_1 and H_2 and is assigned same timestamp value on them. If there are many events that occurred in H_1 after e_0 and before an event e, then LC timestamps e with the value: TS(e) = TS(e_0) + E +1, where E is the number of events that occurred between e_0 and e. In contrast, if there are no events occurred in H_2 between e_0 and e-, then the value of the timestamp that is assigned to e- *is* TS(e-) = TS(e_0) +1. This implies that TS(e-) < TS(e), but this statement is not satisfied in the case of the time period between the occurrence of e- and e_0 in H_2 is greater than the time period between the occurrence of e and e_0 in H_1. Accordingly, using LC does not insures that if TS(e) < TS(e-), $e \rightarrow e$-. On the other hand, RC timestamping ensures that each event is timestamped with the current value of RC. Moreover, all RCs in the replication system are always advanced in a unified manner independently of the occurrence of events. This implies that if two events e and e^- occurred on different hosts and TS(e) on H_1 < TS(e^-) on H_2 then $e \rightarrow e^-$.

6.2.2.2 Maintaining the Value of RC

In order to ensure a unified timing system through enforcing the RC's value of the main center point on all underlying hosts, two methods are proposed for correcting mismatched RC values and incorrect timestamps that are assigned to events.

(i) Periodical checking

It represents a Top-Down mechanism for checking and amendment the RC's value and timestamps assigned to events in the lower level hosts. In this method, each responsible center point sends a message called Timing-Value-Message (TVM) that ships the current value of RC to underlying level. TVM migrates periodically (e.g.

every prefixed period or at the beginning of the next collection period for updates from the lower level) to the hosts in the lower level in order to inform the receiver the current value of RC.

Upon arriving at the destination, the shipped value with TVM (i.e. denoted by *S*) is compared with the local value of RC (i.e. denoted by *L*). In case of a mismatch occurred, the following steps are performed.

- Computing the difference between *L* and *S*:

$$D = S - L \qquad (1)$$

- Adjusting the value of *L* to S:

$$L = S \pm D \qquad (2)$$

- Calculating the difference between send and receive timestamps for TVM:

$$D^{--} = \mathrm{TS}(Receive) - \mathrm{TS}(Send) \qquad (3)$$

- Adding the difference to L:

$$L = L + D^{--} + C \qquad (4)$$

- Revising the timestamp assigned to each update event (that is not propagated to the higher level).

In these steps, *C* is the time period that is consumed on calculating aforementioned equations. Equation 4 represents the new value of the system's time on the host in the lower level. The revision of timestamp of each update (*T*) that is not propagated previously is performed using the above four equations by substituting *L* with *T*.

This mechanism is not suitable to check the time value on the mobile hosts due to their inherited frequent disconnections, which hinder the periodical checking process. Therefore, it can be used for the servers in the fixed network, since they have stable and continuous connectivity.

(ii) On-Receiving checking

In this method, the host in the lower level sends TVM to the higher level when the connection is realized. Upon receiving at the higher level, the following steps are carried out.

Step 1 (Determining the exact value of the system time in the host in the lower level (*T*))

1.1 Computing the time period that the migration of TVM has taken (i.e. TS(*send*)-TS(*receive*)).

1.2 Adding the value of the computed period to the shipped value of RC in the same manner as in the abovementioned equations (1 to 4).

Step 2 (Comparing the value of T with the current value of local RC on the host in the higher level (T^{-}))

2.1 If the values of T and T^{-} are not identical, the higher level host will send the current RC's value (T^{-}) to the host in the lower level, which will be responsible for correcting the timing values using above equations.

2.2 The host in the higher level will correct the value of the timestamp of each update received recently from underlying host in addition to the send timestamp for the messages that shipped those updates. This correction is done by adding/removing the difference between T and T^{-} to/from the received value for each update timestamp and send timestamp.

This mechanism is suitable for checking the time on the mobile hosts because it exploits the connection of the mobile host with the fixed network in order to correct its timing values. Also, the load of the correction is shared among the two hosts because the host in the higher level will correct the received timing values, while the host in the lower level will correct its stored values.

6.2.2.3 RU-Message

It is the message that carries the recent updates information from a host to its responsible center point in the higher level. It is defined as a tuple of <*Msg-ID*, *Host-ID*, *Send-TS*, *RU-Update-Vec*, *Receive-TS* >, where *Msg-ID* represents a serial number for messages that send from a specific host, *Host-ID* is the sender, *Send-TS* is the value of sender's local RC at the moment when the message is sent, *RU-Update-Vec* is a vector that contains the information of recent updates that occurred on the sender, and *Receive-TS* is the value of receiver's local RC at the moment when the message is received. Both *Msg-ID* and *RU-Update-Vec* represent the exact contains of *RU-Message*, while the other items represent the sending information.

RU-Update-Vec can be expressed as *RU-Update-Vec* = (<u_1, *Obj-ID, Update-TS*>, …, <u_n, *Obj-ID, Update-TS*>), where u_i (i=1,...,n) is an update generation event that occurred in the sender and *Obj-ID* is the object where the update generation

occurred. The items on *RU-Update-Vec* increase as we move from a level to a higher level. For example, when *RU-Message* is sent from *CS* to *ZS*, additional items are added to *RU-Update-Vec*, which are send and receive timestamps for each *RU-Message* that *CS* received from underlying hosts. This is because each update inherits the attributes of the *RU-Message* that shipped it to the higher level. Accordingly, *RU-Update-Vec* on *RU-Message* that is sent from *CS* to *ZS* can be expressed as *RU-Update-Vec* = (<u_1, *Obj-ID, Update-TS*, *Send-To-CS-TS*, *Receive-By-CS-TS* >, …, < u_n, *Obj-ID, Update-TS*, *Send-To-CS-TS*, *Receive-By-CS-TS* >).

6.2.2.4 Treated-Before Relation

In order to implement the causal ordering, our ordering mechanism is based on a new ordering relation that is called *Treated-Before relation*. This relation acts as an ordering constraint for ordering all updates that occurred in the replication system.

To define *Treated-Before relation* denoted by $<_{TB}$ between two updates in each level, two relations must first be defined as follows.

1. *Generated-Before relation* ($<_{GB}$)

This relation can be defined as follows:

$U_x <_{GB} U_y$ iff $\Omega_{update\text{-}TS}(U_x) < \Omega_{update\text{-}TS}(U_y)$

Where Ω is the selection operator for the selection operation of specific values of attributes that related to the updates U_x and U_y. The ordering is based on these values by comparing corresponding values for both updates. < is the less than relation. *Update-TS* is the value of RC when an update *U* occurred in the system.

This relation is used as a basis for ordering updates that are generated on either same level or different levels.

2. *Transferred-Before relation* ($<_{TRB}$)

This relation can be defined as follows:

$U_x <_{TRB} U_y$ iff

$\Omega_{Underlyinglevel\text{-}Send\text{-}TS,\ Currentlevel\text{-}Receive\text{-}TS}(U_x) <$

$\Omega_{Underlyinglevel\text{-}Send\text{-}TS,\ Currentlevel\text{-}Receive\text{-}TS}(U_y)$

Where *Underlyinglevel-Send-TS* is the value of RC at the sender when the message that ships the information of an update *U* is sent to the higher level.

Currentlevel-Receive-TS is the value of RC when the message is received at the higher level.

This relation is used to compare updates that are received from underlying hosts. The comparing is based on the values of both attributes *Underlyinglevel - Send-TS* and *Currentlevel-Receive-TS*.

Note that Ω specifies only the set of attributes that can be used to compare the two updates. Then, the comparison is performed on this set of attributes in a sequential manner, which means that if the comparison that is based on the first attribute in the set has resulted in that updates are not concurrent (i.e. different values of this attribute for both updates), then there is no need to continue the comparison using the other attributes in the set.

Thus, the *Treated-Before relation* can be defined as a union of $<_{GB}$ and $<_{TRB}$ relations on the servers that exist in the higher levels (*CS*, *ZS*, and *MS*). In the MH level or FH level can be defined as $<_{GB}$ relation only, since there is no need for send and receive operations. Accordingly, based on these relations, the *Treated-Before relation* is defined as follows.

Definition 6.2.2.4.1 *Treated-Before relation* **($<_{TB}$)** is a relation that equivalent to:

I. $U_x <_{GB} U_y$ iff U_x and U_y are generated on either same host or different hosts and Ux and U_y are not concurrent.

II. $(U_x <_{GB} U_y) \cup (U_x <_{TRB} U_y)$ iff U_x and U_y are generated on different hosts and U_x and U_y are concurrent.

According to this definition, concurrent updates can be defined based on *Treated-Before relation* as follows.

Definition 6.2.2.4.2 Two updates U_x and U_y are concurrent if neither $U_x <_{TB} U_y$ nor $U_y <_{TB} U_x$.

This definition can be extended to define *Treated-Before relation* between updates in each level as follows:

1. MH (or FH) level

$U_x <_{TB} U_y$ iff $\Omega_{update\text{-}TS}(U_x) < \Omega_{update\text{-}TS}(U_y)$

2. CS level

$U_x <_{TB} U_y$ iff

$\Omega_{update\text{-}TS,\ Host\text{-}Send\text{-}TS,\ CS\text{-}Receive\text{-}TS}(U_x) <$

$\Omega_{update\text{-}TS,\ Host\text{-}Send\text{-}TS,\ CS\text{-}Receive\text{-}TS}(U_y)$, For each U_x and U_y that are generated on the underlying level.

Where *Host-Send-TS* is the value of RC when the *RU-Message* is sent from either underlying MH or FH. *CS- Receive- TS* is the value of RC at *CS* when *RU-Message* is received from underlying level.

3. ZS level

$U_x <_{TB} U_y$ iff

$\Omega_{update\text{-}TS,\ Host\text{-}Send\text{-}TS,\ CS\text{-}Receive\text{-}TS,\ CS\text{-}Send\text{-}TS,\ ZS\text{-}Receive\text{-}TS}(U_x) <$

$\Omega_{update\text{-}TS,\ Host\text{-}Send\text{-}TS,\ CS\text{-}Receive\text{-}TS,\ CS\text{-}Send\text{-}TS,\ ZS\text{-}Receive\text{-}TS}(U_y)$, For each U_x and U_y that are generated on the underlying level.

4. MS level

$U_x <_{TB} U_y$ iff

$\Omega_{update\text{-}TS,\ Host\text{-}Send\text{-}TS,\ CS\text{-}Receive\text{-}TS,\ CS\text{-}Send\text{-}TS,\ ZS\text{-}Receive\text{-}TS,\ ZS\text{-}Send\text{-}TS,\ MS\text{-}Receive\text{-}TS}(Ux) <$

$\Omega_{update\text{-}TS,\ Host\text{-}Send\text{-}TS,\ CS\text{-}Receive\text{-}TS,\ CS\text{-}Send\text{-}TS,\ ZS\text{-}Receive\text{-}TS,\ ZS\text{-}Send\text{-}TS,\ MS\text{-}Receive\text{-}TS}(Uy)$, For each U_x and U_y that are generated on the underlying level.

Assertion 6.2.2.4.1 $<_{TB}$ is a partial order on a set Q of updates.

Proof. For two updates U_x and U_y that belong to Q, $<_{TB}$ is antisymmetric because if both $U_x <_{TB} U_y$ and $U_y <_{TB} U_x$ hold, then necessarily either $U_x = U_y$ or U_x and U_y are concurrent. Furthermore, $<_{TB}$ is reflexive since every update is treated before itself previously in the underlying level (i.e. it is sent to the current higher level before sending it to the next higher level). $<_{TB}$ is transitive, since if U_x is treated before U_y and U_y is treated before U_z then U_x is treated before U_z.

6.2.2.5 Ordering Queue

An ordering queue in the proposed ordering mechanism is defined as follows.

Definition 6.2.2.5.1 An ordering queue (q) is a tuple $<U, <_{TB}>$, where U is a set of updates that have joined q and $<_{TB}$ is the *Treated-Before relation* for comparing updates in q.

When running the relation $<_{TB}$ on q, it will result in a total ordering for all updates within q. This total ordering is different from one level to another. For example, in *CS* level, it represents a total ordering for all updates that are propagated to specific *CS*, while in the *ZS* level it acts as a total ordering for all updates received by specific *ZS*.

The proposed ordering mechanism relies on a set of N ordering queues denoted Q. Each queue is located and maintained by a responsible center point in order to encompass all received updates from underlying hosts during the updates collection period. Accordingly, these queues can be classified into different sets as follows.

1. The set of queues in cell servers (Q_{CSQ})

$Q_{CSQ} = \{Q_{CSk} : 1 \leq k \leq M, M \equiv$ the total number of CSs in the replication system$\}$

2. The set of queues in zone servers (Q_{ZSQ})

$Q_{ZSQ} = \{Q_{ZSl}: 1 \leq l \leq L, L \equiv$ the total number of ZSs in the replication system$\}$

3. The queue in the master server (Q_{MSQ})

$Q_{MSQ} = \{Q_{MS}\}$

Thus, the set Q and the total number of queues (i.e. its elements) denoted N can be written as:

$$Q = Q_{CSQ} \cup Q_{ZSQ} \cup Q_{MSQ}$$

$$N = M + L + 1$$

Each $q \in Q$ is populated during the collection period with the received updates from the underlying level and is emptied after propagating the previously ordered updates to the higher level (after the elapsing of the collection period).

The previously ordered updates in each q are propagated to the ordering queue in the higher level in a unidirectional manner for further ordering in the higher level. This further ordering involves comparing all collected ordered updates from all

queues in the underlying level. Therefore, we can describe our queuing system as a tandem queuing system.

According to the aforementioned description, the structure of the queuing system can be viewed as a directed tree-like structure (see Figure 6.2.2.5.1) in which vertices represent the hosts and the edges represent the links. Each server is seen as a queue in which an update is ordered on its level and then sent to the next higher level for further ordering.

The MS represents the root node for this tree, because it acts as the main collection point for all updates that have been ordered in the lower levels and need further ordering. Each server in underlying levels (i.e. CS and ZS) acts as a representative for the MS in its level in performing updates ordering. Thus, each server acts as a root node for its level. This results in multiple sub trees that their number equals to the number of both cell servers and zone servers. The queues in the cell level are not fixed since the mobile hosts can move from one cell to another. But the queues in the both the zone server and the master server are fixed.

The state of each queue q at any point is defined to be the sequence of updates existing in that queue and are waiting for ordering. The elements of each queue may experience a finite delay in their processing by the higher level until collecting all updates from underlying levels, but this delay is limited by elapsing of the collecting period, which is specified according to the consistency and availability requirements of the application.

Note that this tree can be converted to bidirectional graph by considering the opposite flow of updates to the lower level (in a form of *ReRU-Message* as shown in section 6.3) after they are ordered by MS.

An update in each queue is represented by a data structure called Update Descriptor (UD). The UD contains the required information for comparing of updates using *Treated-before relation*. The UD is an incremental data structure since its information increases in the higher levels. Figure 6.2.2.5.2 represents the structure of UD in each level.

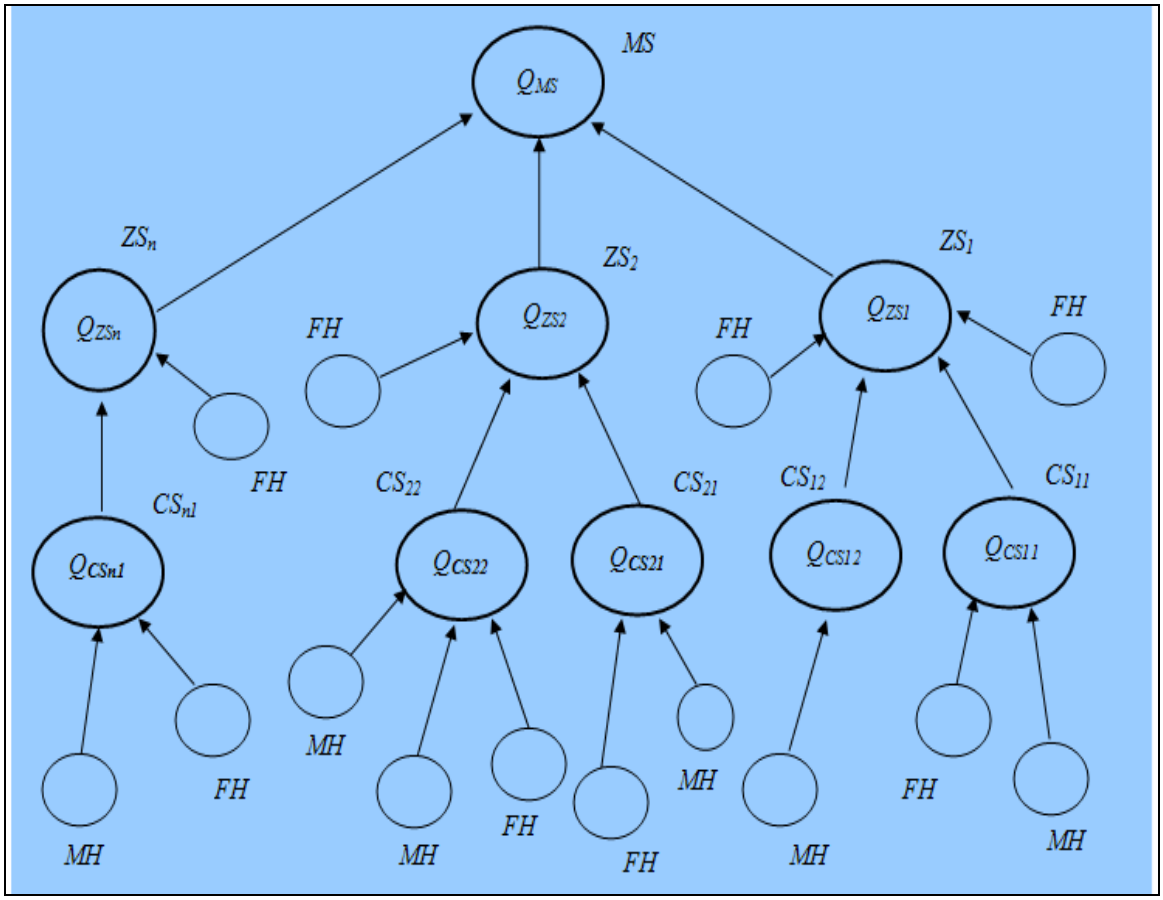

Figure 6.2.2.5.1 The Directed Tree for the Ordering System

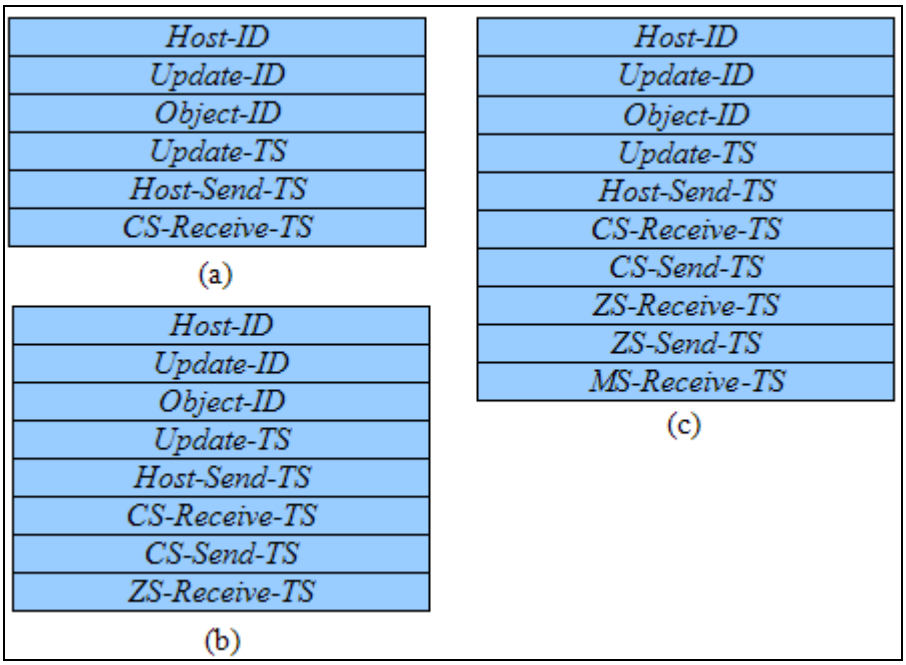

Figure 6.2.2.5.2 The Contains of UD in (a) CS (b) ZS (c) MS

In this figure, *Update-ID* is an incremental number assigned to each update on each object. *Object-ID* is the number of object where update occurred. The other attributes are as mentioned above.

6.2.2.6 Casual Ordering Algorithm

In this algorithm, each server in the higher level uses the *Treated-Before relation* for comparing between updates that are currently placed in its local queue. The comparing between two updates is performed first based on the *Update-TS* attribute of both updates, and in case of concurrent updates are detected, the resolution (i.e. reordering) will be performed based on the *Send-TS* and *Receive-TS* attributes of messages that shipped updates information to the higher level.

After updates are totally ordered in the highest level (i.e. master level), *MS* assigns *Global ID* for each update based on the order of that update. The value of this *ID* represents the order of the update on the level of the whole system and it is assigned in a sequential manner.

In this algorithm, *Host-ID* is the ID assigned to the sender. *i* is the sequence number of *RU-Message* at the sender . *Dest* is a variable that represents the ID of the responsible server in the higher level. *k* represents the ordering key, which is a set of attributes that is used to compare updates. *Max* is a variable that specifies the maximum value for the period of collecting updates from underlying hosts.

The causal ordering algorithm is as follows.

(i) At each host generates and sends updates:
∀ generated update: Update-TS = RC's value when update takes place
Assign *Send-TS to RU-Message: Send-TS = RC's value when send takes place*
Send *(Host-ID, i, RU-Message, Dest, Send-TS)*

(ii) At each host receives updates:
Assign *Receive-TS to each received message: Receive-TS = RC's value*
Place *the received Message at the end of the local ordering queue*
Set *Max = T*
While *Collecting-Period < T*
∀ updates in the queue
Compare *updates using timestamp attributes in RU-Update-Vec*
If *Host-type = 'CS'* ***then***
k = {Update-TS}/ Use the set {Update-TS} as ordering key*
If *¬ (u →u v) and ¬ (v →u u)* ***then*** */* if conflict occurs or concurrent updates u || v */*
/ Compare updates using send timestamp attribute of RU-Message */*
k = {Send-To-CS-TS}

```
            If ¬ (m1→s m2) and ¬ (m2→s m1) then
                /* Compare updates using receive timestamp attribute of RU-Message
          */
                k = {Receive-By-CS-TS}
    Else If Host-type = 'ZS' then
      /* Compare updates using received timestamp information from underlying CSs
*/
      k = {Update-TS, Send-To-CS-TS, Receive-By-CS-TS}
          If ((¬ (u →u v) and ¬ (v →u u)) and (¬ (m1→r m2) and ¬ (m2→r m1)) and
            (¬ (m1→r m2) and ¬ (m2→r m1))) then
             /* Compare updates using send timestamp attribute of RU-Message to ZS
             */.
             k = {Send-To-ZS-TS}
               If u || v then
                 /* Compare updates using receive timestamp attribute of RU-Message
             by ZS */.
                k = {Receive-By-ZS-TS}
     Else If Host-type = 'MS' then
      k = {Update-TS, Send-To-CS-TS, Receive-By-CS-TS, Send-To-ZS-TS,
      Receive-By-ZS-TS}
        If u || v then
            k = {Send-To-MS-TS}
              If u || v then
                 k = {Receive-By-MS-TS}
             Else
                  Order u before v
                  Assign Global ID for both u and v
End {While}
```

6.2.2.7 Correctness Proof for the Casual Ordering Algorithm

To prove that the proposed causal ordering algorithm is a correct solution to the problem of ordering updates in LMDDBSs, there is a need to show that it has both safety and liveness properties.

The original and general purpose of liveness is to guarantee that every message is eventually delivered to its destination (s). For our case, we prove that every *RU-Message* that is sent from underlying level, its contains (i.e. updates) will be ordered (delivered) in the current level and then will be sent to the next higher level.

Assertion 6.2.2.7.1 (Liveness)

The causal ordering algorithm ensures that all received updates from underlying level will be ordered and sent to the higher level.

Proof.

Let *S* be the ordering process in each higher level host. *S* can be viewed as a transition system that has three states: {*C*: Collecting, *O*: Ordering, *SE*: Sending} and two transitions: {t_1: elapsing of current collecting period and running of causal ordering, t_2: completion of ordering process} as shown in the following figure that describes ordering process as a state machine.

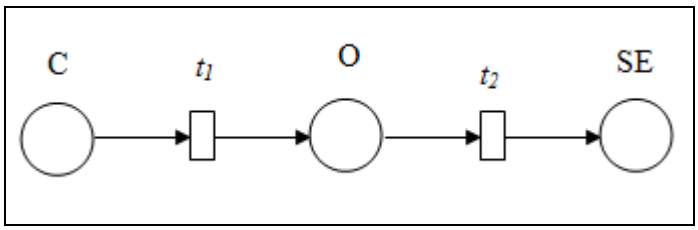

Figure 6.2.2.7.1 A state Machine for the Description of the Ordering Process

S always receives updates from underlying hosts through *RU-Message*. Let *Q* be the set of contains of all *RU-Message* that are received from underlying level during the current *C* state. When *S* enters *O* state (i.e. elapsing of collecting period), this implies servicing current updates in *Q* and storing all the contains of the messages that come during the period of *O* state in a waiting queue (*W*). When the *S* enters *SE*, this implies emptying *Q* and *S* transits to the *C* state by filling *Q* with the contains of each *message* belongs to *W* and all new arrived messages. Accordingly, $\forall\ m \notin Q$ (i.e. *m* is a new arrived *message*), *m* will be serviced (through placing its contains in *Q* and then ordering them) when *S* transits from *SE* to *C* and then from *C* to *O* once again. Thus, there is no deadlock or blocked updates.

Assertion 6.2.2.7.2 (Safety)

The causal ordering is never violated at any higher level.

Proof.

Assume that two updates *u* and *v* are generated on hosts h_1 and h_2, respectively. If *u* is generated before *v*, this implies that *Update-TS*(*u*) < *Update-TS*(*v*) based on the values that are assigned by RC on h_1 and h_2. Accordingly, *Treated-Before Relation* ensures that if h_1 and h_2 in same cell, same zone, or different zones, then each receiving server in the higher level will order *u* before *v*. In case of *u* and *v* are concurrent based on the generation event, the *Treated-Before Relation* implies that comparing *u* and *v*

using *Send-TS* value. Accordingly, if u is sent before v to the higher level $\Rightarrow$ *Send-TS*(u) < *Send-TS*(v). Thus, each server in the higher level will order u before v. Same discussion is hold to prove maintaining of causal ordering in case of u and v are concurrent based on both generation and sending events.

6.2.2.8 Priority-Based Causal Ordering

Instead of using timestamp information to provide the unified ordering as described above, a priority based ordering can be employed. This involves specifying priority rules to assign priorities to different updates according to their types and relationships. These priority rules can be specified by the application designer according to the semantic of the data that is handled by a given application. This semantic determines the relationships between updates.

This means that the semantic of the data can be used to determine which operations on each object should be given the priority to be ordered before other operations on that object and this can be specified on the database design stage. For example, in the products object, the priority rule can be specified as: Add quantity operation should be given a priority to be ordered before withdraw quantity operation.

Based on the priority, the definition of concurrent updates can be as follows

Definition 6.2.2.8.1 *Priority-based Concurrent updates "u* $||_p$ *v"*

Two distinct updates u and v that are shipped to the higher level via messages m_1 and m_2, respectively, are" *Priority-based Concurrent*" iff:

(i) *u and v* have same priority or

(ii) *u and v* have same priority and ($\neg$ ($m_1 \rightarrow_s m_2$) and $\neg$ ($m_2 \rightarrow_s m_1$)) or

(iii) *u and v* have same priority and ($\neg$ ($m_1 \rightarrow_r m_2$) and $\neg$ ($m_2 \rightarrow_r m_1$)) and ($\neg$ ($m_1 \rightarrow_r m_2$) and $\neg$ ($m_2 \rightarrow_r m_1$))

Accordingly, *Conditioned causal ordering relation* "$\rightarrow_{cu}$" can be modified to encompass Priority-based concurrent updates as follows.

Definition 6.2.2.8.2 *Priority-based conditioned causal ordering relation* "$\rightarrow_{pcu}$"

Given two *updates u* and *v,* generated at site *i* and *j,* respectively, and sent via messages m_1 and m_2, respectively, to the responsible center point *C*, then $u \rightarrow_{pcu} v$, *iff*

(i) (u $||_p$ v) and $m_1 \rightarrow_s m_2$ or

(ii) (u $||_p$ v) and ($\neg$ ($m_1 \rightarrow_s m_2$) and $\neg$ ($m_2 \rightarrow_s m_1$)) and $m_1 \rightarrow_r m_2$ or

(iii) ($u \,||_p\, v$) and ($\neg\, (m_1 \rightarrow_r m_2)$ and $\neg\, (m_2 \rightarrow_r m_1)$) and ($\neg\, (m_1 \rightarrow_r m_2)$ and $\neg\, (m_2 \rightarrow_r m_1)$) and $m_3 \rightarrow_s m_4$ in case of u and v are sent to their responsible center points C_1 and C_2 via m_1 and m_2 and then those updates are sent from C_1 and C_2 to their responsible center point Z via m_3 and m_4.

Thus, the priority of updates is incorporated as one of the conditions for the conditioned causal ordering relation between u and v. This incorporation is valid only in the case of both u and v have same priorities, Which means that if two updates are concurrent with regard to they have same priority, then we can rely on the sending and receiving times of their messages.

By considering the priority of updates instead of the timestamp information of their generation, the only change that is needed in the proposed causal algorithm is that each receiver performs reordering of priority-based concurrent updates based on the definition of *Priority-based conditioned causal ordering* relation.

To implement this type, there is a need to determine the type of each update (e.g. Add or withdraw) and its relationship with other updates in order to assign its priority. This requires both storing the update priority as a data item in the object and shipping it with the update information to the higher level.

6.2.2.9 Performance Issues

This section presents performance issues that related to implementing the proposed mechanism and previous version vectors based mechanisms in LMDDBs.

1. The size of timestamp information

The casual ordering algorithm presented here cuts down the size of the timestamp information appended to each message to be smaller than the size that is imposed by version vectors based algorithms (see section 2.7.2). These algorithms append a timestamp information of size $O(n)$, where n is the number of replicas. As a consequence, they impose a high communication overhead, especially for a large number of replicas. The size of timestamp information that is imposed by the proposed algorithm is provided by the following assertion.

Assertion 6.2.2.9.1 The size of timestamp information appended to each *RU-message* is:

1. $O(k+1)$, if it is sent from a server to a server in the higher level, where k is the number of underlying hosts that propagate their updates during the last collection period.

Or:

2. $O(1)$, if it is sent from a host (MH or FH) to its responsible server.

Proof. For the first case, since for each message received from underlying level, the receiver server in the higher level will assign *Receive-TS*. This means that there are *K Receive-TSs*. Accordingly, when these messages are incorporated in one message to be sent to the server in the higher level, the receiver will add *send-TS* to the incorporated message. Thus, the size of the timestamp information that will be appended to *RU-Message* represents the size of the timestamps that are added by the servers, which is *Receive-TS* for each message + *Send-TS* for the incorporated message. For the second case (from a host to its responsible server), the size is $O(1)$ because the host appends only *Send-TS* to each *RU-Message*.

2. The size of the clock

The size of the information that related to the clock's time is $O(1)$ for the RC, since the host only keeps track of its own time. This is in contrast to the vector clock, which its time information imposes a size of $O(n)$, where n is number of replicas. This implies that vector clock occupies storage space that is proportional to the total number of replicas in the replication system.

3. Storage, computation, and communication overheads

The proposed ordering mechanism does not impose any overhead on mobile hosts as well as fixed hosts, since there is no need for collecting or ordering of updates that come from other hosts. Thus, the proposed ordering mechanism placed the overhead of both storage and computation on the servers that exist in the higher levels, since these servers are assumed that they have more storage and processing capabilities than both fixed and mobile hosts. Moreover, it does not impose large (i.e. significant) overheads on those servers according to the following reasons.

1. Not all hosts in the underlying level are expected to propagate their updates during each updates collection period from underlying hosts. This is according to the fact that not all mobile hosts stay connected to the fixed network at all

times. Moreover, each server receives only the recent generated updates from its underlying hosts.

2. The ordering process occurs only when the collection period elapsed. This means that the CPU does not service ordering process at all times.
3. The small size of RC, since each host needs only to keep track of its time rather than the times of all other hosts as compared with version vectors.
4. Only the information that is needed to implement the causal ordering is sent with each message to the higher level as follows. Each host (MH or FH) ships only with *RU-Message* the attributes *Update-TS* and *Send-TS*. The CS adds only small amount of information, which are *Receive-TS* and *Send-TS* to the messages that are received from underlying hosts. Also, both ZS and MS add small amount of information in same manner.

On the other hand, the mechanisms that are based on version vectors suffer from both storage and communication overheads that originated from the size of version vectors, which increases as the number of sites increase and the large amount of information that should be sent and stored for carrying out causal ordering. Accordingly, these mechanisms are not suitable for LMDDBSs, since those overheads hinder the scalability. Thus, the proposed mechanism is more suitable to be implemented in LMDDBSs, since the size of RC is not affected by the number of replicas. Also, as mentioned, only the needed information for implementing causal ordering in the higher level is shipped via *RU-Message*. Furthermore, according to the proposed updates propagation protocol, each host propagates and receives updates from a few hosts only. This leads to a low space overhead on the timestamp information associated with each message.

Moreover, the proposed mechanism supports the characteristics of LMDDBs through exploiting the updates collection periods to perform the unified hierarchical ordering for recent updates. These collection periods provide mobile hosts with a reasonable time periods (i.e. can be determined according to the requirements of the application) for propagating their updates to the higher levels in order to be ordered with other updates that come from fixed hosts. This support both mobility and disconnections of mobile hosts, since the mobile hosts can disconnect or move to another cell during the same collection period at the higher level.

4. Overhead of time divergence amendment

The communication and computation overheads that are resulted from sending MVT and amendment process are not sensitive according to the following reasons:

1. The TVM ships only the current value of RC to the other host. This means that the size of this message is small.
2. The steps that are required for amendment process involve only simple calculations that do not need large computation overhead.

6.3 Messages Ordering in the Case of Top-Down Propagation

In the Top-Down propagation, the messages that carry ReRU (which are totally ordered by the master server) are sent to multiple underlying hosts from their responsible center point. Accordingly, it is desirable that all messages are delivered to all underlying hosts in same order as they generated on the master server. Thus, in Top-Down propagation mechanism, the ordering type can be specified as a hybrid of both casual order and total order as presented in the following subsection.

6.3.1 Ordering Model

The different concepts that are involved in the ordering of messages in Top-Down Propagation are as follows.

Defintion 6.3.1.1 (*ReRU-Message*). It is the message that carries the recent resolved updates information from a server to underlying hosts and is defined as a tuple of <*Msg-ID*, *ReRU-Update-Vec*>, where *Msg-ID* is the ID assigned serially by the master server, and *ReRU-Update-Vec* contains the details of the recently ordered updates and can be expressed as Update-Vec =(<u_1, *Generated-Host, Global-ID* >, …, <u_n, *Generated-Host, Global-ID* >)

Definition 6.3.1.2 (*Casual ordering*). Given two *ReRU-Message* *x* and *y*, if *x* is generated at the master server before *y*, then each underlying center point should:

1. receive *x* before *y*
2. send *x* before *y*.

Definition 6.3.1.3 (***Total Order***) For all messages *ReRU-*m_1 and *ReRU-*m_2 that are sent by center point *C* and all underlying hosts H_1 and H_2, if *ReRU-*m_1 is received at *C* before *ReRU-*m_2 from its responsible center point, then *ReRU-*m_2 is not received before *ReRU-*m_1 at both H_1 and H_2.

Accordingly, the hybrid of both causal ordering and total ordering is defined as follows.

Definition 6.3.1.4 (***Hybrid Casual-Total Ordering*** ⌂) Given two *ReRU-Message x* and *y*, then *x* ⌂ *y* iff:

a. *x* is generated at the master server before *y*
b. each underlying host receive *x* before *y* from its responsible center point
c. Each underlying host send *x* before *y* to its underlying hosts.

According to this definition, every lower level host should receive and send the messages that are received from the higher level in the same order as they produced by the highest level host (e.g. MS). For example, if ZS has received m_{12} from MS and then received m_{16}, it will not deliver those messages to the lower level unless it gets m_{13}, m_{14}, and m_{15}. Moreover, if ZS has received m_{20} and the message that is received before it is m_{17}, it will not send it to underlying cell servers unless it gets the messages m_{18} and m_{19}. Same discussion is hold for sending *ReRU-Message* from each CS to underlying fixed and mobile hosts.

6.3.2 Hybrid Casual-Total Ordering Algorithm

The ordering of messages is based on the ID of the *ReRU-Message* that is assigned by the master server. Each server has a data structure called *ReRU-Messages-Tracking* to store the IDs of all ReRU-Message that are received previously from the higher level. *L* is used to denote last Received *ReRU-Message* from the higher level and *N* to denote New Received *ReRU-Message*.

At the master server
Attach *ordered updates to ReRU-Message*
ID(ReRU-Message) = ID(Last ReRU-Message) + 1

At each server in underlying level
When *receiving ReRU-Message from higher level* ***Do***
Store *ID(N)* ***on*** *ReRU-Messages-Tracking*
Retrieve *ID(L)* ***from*** *ReRU-Messages-Tracking*
If *ID (N) = ID (L) +1* ***then***
Deliver *N to all underlying hosts.*
/ this ensures that if a message x is delivered before y to the lower level by the master server, then y will not be received before x at any lower level host*/*
Else
Send ID *(L) to the higher level /* this is to send right messages*/*

6.4 Summary

This chapter has presented the updates ordering mechanism that provides a unified order for all generated updates on the different levels of the replication architecture. In this mechanism, each update is timestamped with the information that is only needed to implement the causal ordering on the higher levels. This reduces the communication overhead, which is imposed by the size of the messages that ship updates information to the higher levels.

The ordering mechanism exploits the proposed updates propagation protocol in that each host sends and receives messages from a few hosts only. This leads to a low space and computation overheads that are imposed by the timestamp information that is associated with each message.

The overall order operation for recent updates is committed at the master server. This committing is finalized by assigning each update the Global ID that represents a sequence number assigned by the master server. This ID is incremented with each update.

In this chapter, the research interested only in the ordering of the messages that carry updates information to the higher levels as well as the messages that carry ReRU from the higher levels to the lower levels. Accordingly, the ordering of the other messages such as acknowledgement messages or TVM is not considered.