CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

Database replication has been an area of research for more than thirty years. The first publications on the subject appeared in the later seventies (Thomas, 1979; Gifford, 1979; Stonebraker, 1979). Since then, database replication has been an object of research. An important issue is that while the techniques that were proposed are correct, they have been shown to perform badly if the number of sites increases (Gray et al., 1996). The main reason for this is that those replication strategies were designed to impose minimal changes in the database system environment. Another important issue is how to build an efficient, consistent replicated database is still an open research question (Wiesmann, 2002; Wiesmann et al., 2000; Abawajy, Deris, and Omer, 2006; Tolia, Satyanarayanan, and Wolbach, 2007). To deal with these issues in mobile environments, replication has become an area of interest in the past few years. Accordingly, many replication solutions are proposed to handle these issues in mobile environments. In this chapter, we review some of the previously proposed strategies and we focus more on the solutions that devoted to mobile environments and large-scale systems.

2.2 Interaction between Replicas

A replicated database system is composed of many replicas of databases distributed across different sites. A replica can accept client's requests and interact with other replicas to work cooperatively as a global database system to provide database changes to clients at all sites. Some replicas can act as read-only replicas in that they satisfy only read requests from their clients, while other replicas can satisfy both read and write requests. There are two types of interaction between replicas: Master/Slave and Multi-master (Gray et al., 1996; Pacitt, Minet, and Simon, 2001; Martins, Pacitti, and Valduriez, 2006). Based on these interactions, two types of replications are described in the following subsections.

2.2.1 Master/Slave Replication

This type designates one replica as a master replica and the other replicas are slaves. The master replica accepts update operations, while a slave replica accepts only read operations. In this case, every update is first applied on the master replica and then it is propagated towards the slave replicas.

The centralization of updates at a single replica introduces a potential bottleneck and a single point of failure. Therefore, a failure in the master replica blocks update operations and thus limits data availability, especially when the system experiences frequent updates. Figure 2.2.1.1 shows an example of Master/Slave type with three slaves.

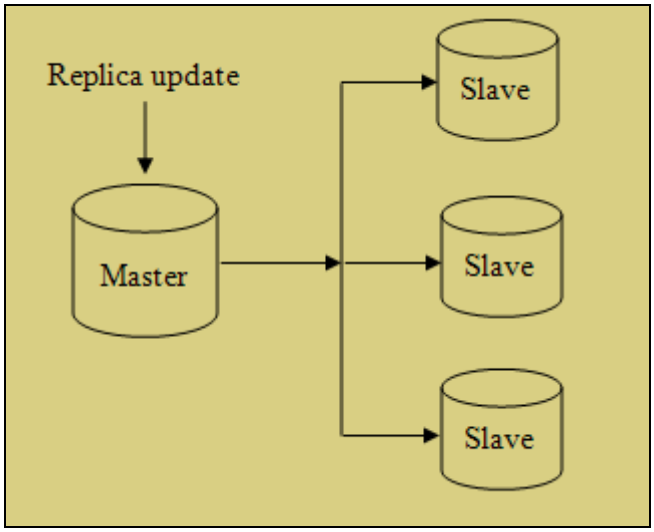

Figure 2.2.1.1 An Example of Master/Slave Replication with Three Slaves

The Master/Slave replication can be implemented using Client/Server architecture (Ekenstam et al., 2001) in which replica updates can be performed only at the server after receiving update requests from the clients. The server node is typically a large, well-connected, fixed (non-nomadic) node, while some or all of the clients are less well-connected mobile nodes. Clients store a read-only copy of the data. For the client to update any data it must connect to the server and submit a request.

A single server can only handle a finite number of clients before performance is adversely affected. If the server is down or the connectivity between the client and the server is lost, then the client cannot perform any updates on the database. This solution clearly poses problems for mobile users.

2.2.2 Multi-master Replication

In the Multi-master approach, multiple sites hold master replicas of the data. All these replicas can be concurrently updated and later the exchanging of updates can occur between them. Distributing updates avoids bottlenecks and single points of failures, thereby improving data availability. However, in order to ensure data consistency, the concurrent updates to different copies must be coordinated or a reconciliation algorithm must be applied to solve replica divergences. Coordinating distributed updates can lead to expensive communication, and on the other hand reconciliation solutions can be complex (Saito and Shapiro, 2005; Martins, Pacitti, and Valduriez, 2006). Figure 2.2.2.1 shows an example of Multi-master replication with four master replicas.

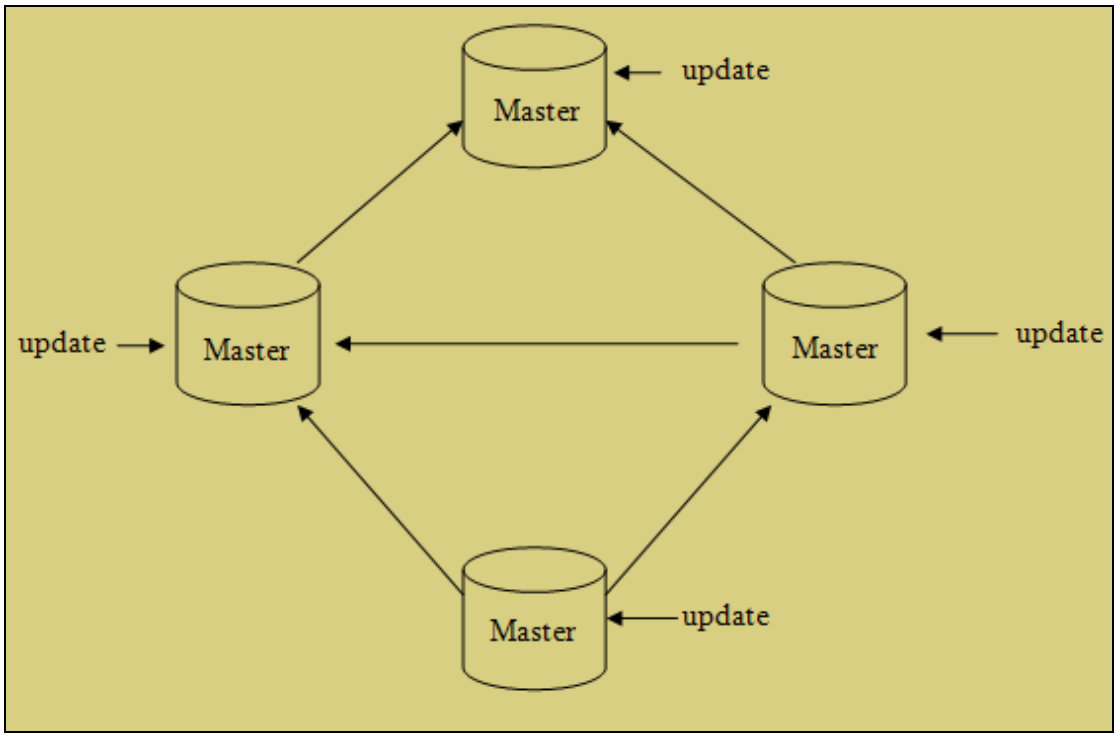

Figure 2.2.2.1 An Example of Multi-master Replication with Four Replicas

Table 2.2.2.1 summarizes the differences between Master/Slave replication and Multi-master replication.

Table 2.2.2.1 Master/Slave Replication vs. Multi-master Replication

Characteristic	Master/Slave	Multi-master
Distinguishing feature	One master replica	Multiple master replicas
Update approach	centralized	Distributed
Update blocking	Master site down	All master sites down
Up-to-date values at	master replica	unknown replica
Possible bottleneck	Yes	No

The Multi-master replication can be implemented as a Client/Server replication or Peer-to-Peer replication (Reiher et al., 1996; Ekenstam et al., 2001). These types are described in the following subsections.

2.2.2.1 Client/Server Replication

In the Client/Server replication, both client and server store a replica of the database and issue updates operations locally on this replica. This represents a distinction from the Client/Server replication in the case of Master/slave replication, where update requests are sent to the server. The clients can only exchange updates with the server, since they can only communicate with one of the servers. Accordingly, updates are required to be sent to the server from where the other clients can then obtain these updates. This type is implemented in replication systems such as Coda (Satyanarayanan, 2002).

The Client/Server replication tremendously simplifies the design of a replicated system, at the cost of limiting its flexibility. This is because all updates must go through the server, since the server acts as a physical synchronization point. This mode of synchronization (i.e. through the server) becomes a problem in mobile environments when the mobile clients are both in a location far from the location of the server. Accordingly, if the two mobile clients are adjacent to each other, requiring them to send their data over large distance to the server is unreasonable.

2.2.2.2 Peer-to-Peer Replication

In this type (Reiher et al., 1996), all nodes holding replicas can synchronize with each other without needing to send their updates to the servers. It is used to avoid the inflexibilities of Client/Server architecture. This type allows mobile peers to synchronize their updates via pairwise synchronization when connectivity is available.

It is implemented in systems such as Ficus (Richard et al., 1990), Bayou (Petersen et al., 1997), Rumor (Richard et al., 1998), and Roam (Ratner, Reiher, and Popek, 2004).

This type, however, has poor scalability because each node must store all replicas information, since every replica is forced to learn about other replica's existence (Ratner et al., 2001). Such an approach consumes a large amount of space at each node and additional communication costs are needed to synchronize all replicas information.

2.3 Abstract Replication Strategy

The complexity of keeping replicas consistent at the presence of update operations is the topic of replication strategies (Ozsu and Valduriez, 1999). A replication strategy can be described abstractly using five generic phases (see figure 2.3.1) (Wiesmann et al., 2000; Barreto, 2003). These phases run on a system that is assumed to be composed of a set of replicas over which operations must be performed. Each node stores a replica is called replica server. The operations are issued by clients. The operation is assumed to be either a single read or write operation. The client can be any node or an application using the database. Communication between different system components (clients and replicas) takes place by exchanging messages.

The five phases are generally involved when an operation request is sent from a client to a replica server. The actual actions that performed on each phase are specific to each particular replication solution (strategy). Moreover, some solutions may skip some phases, order them in a different manner or iterate over some of them. Thus, the protocols can be compared by the way they implement each one of the phases and how they combine the different phases. The five phases are as follows.

Request. The client submits an operation to one (or possibly more) replicas.

Coordination. The replica servers coordinate with each other in order to perform the operation request consistently. Typically, a decision is made to ensure that the replicas agree on the place of the operation in a common execution order that preserves any ordering requirements of the operations.

Execution. The replica servers execute the request upon their replicas. This phase is a good indicator of how each strategy treats and distributes the operations. This phase only represents the actual execution of the operation. The applying of the update is typically done in the Agreement Phase. Thus, in this phase, the effect of the operation

is not applied to the local replica (i.e. updating it in a permanent manner) until a consensus upon this effect is reached in the agreement phase.

Agreement. The replica servers reach a consensus upon the effect of the requested operation.

Response. The outcome of the operation is transmitted back to the client by the node that received the original request or another designated node.

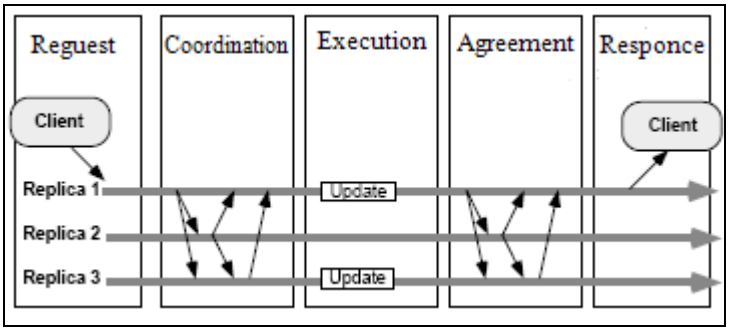

Figure 2.3.1 Phases involved in a replica operation request
(Wiesmann et al., 2000**)**

2.3.1 Classification of Replication Strategies

Replication strategies can be classified into pessimistic and optimistic strategies according to a dimension that concerns the trade-off between consistency and availability (Davidson, Molina, and Skeen, 1985; Barreto, 2003**;** Saito and Shapiro, 2005). A fundamental difference between these types can be seen by considering the order of the five phases described above and the actions that are performed on each phase (Wiesmann et al., 2000). Typically, pessimistic replication follows the five phases in the same order and with same actions as described above. As a consequence, after issuing an operation request, a client has to wait for all the remaining phases to complete before obtaining a response. If a network partition or a failure of some replica server prevents the coordination or agreement phases from performing their distributed algorithms, then the request response will as well be disrupted. As a consequence, the replication system's availability is reduced.

In contrast, to provide a higher availability in comparison to pessimistic replication, the optimistic replication orders the execution and response phases before the coordination and agreement phases (Barreto, 2003). In the execution phase, the operation is executed only at the replica server that received the client's request. After

executing the request, the replica server responds immediately to the client in the response phase. This way, the client issuing the request does not have to wait for the replica server to contact the other, possibly inaccessible, peers in order to complete the coordination and agreement steps. Since, from a client's viewpoint, an operation request is served as soon as the client has received the response from the replica manager, a high availability is accomplished.

A consequence of anticipating the execution and response phases is that inconsistencies may occur if different replicas are updated concurrently. Accordingly, optimistic strategies typically offer weak consistency guarantees.

To restore replica consistency, replica servers must detect update conflicts and, if necessary, resolve them. Hence, the coordination and agreement phases are combined into one phase that is be responsible for dealing with detection and resolution of potential conflicts that may have occurred.

Optimistic replication protocols are beneficial in mobile environments. This is because unreliable and intermittent connectivity between nodes affect the execution of both coordination and the agreement phases. This leads to delay coordination and agreement of client updates until the client reconnects.

2.4 Pessimistic Strategies

Pessimistic strategies prevent inconsistencies by limiting availability. They operate under a pessimistic assumption that if an inconsistency can occur, it will occur (Davidson, Molina, and Skeen, 1985). Therefore, these strategies restrict updates to a single replica or a group of replicas by locking access to these replicas during the processing of update requests. Then, updates are applied to all other replicas (Saito and Shapiro, 2005). Accordingly, the pessimistic strategies provide the strong consistency guarantee that is called one-copy equivalence (Ozsu and Valduriez, 1999). This consistency guarantee requires that all replicas be mutually consistent (i.e. have identical values for all shared data items) at the end of each update operation.

According to Bernstein, Hadzilacos, and Goodman (1987), A replicated database system is one-copy serializable, if it ensures both one-copy equivalence and the serializable execution of transactions (i.e. accesses to the database). One-copy-serializability is considered as the correctness criterion of the replicated database system. It requires that an execution of a set of accesses (read and write) on a replicated database is equivalent to a serial execution of these accesses on a non-

replicated database. A simple protocol for ensuring one-copy serializability is ROWA (see section 2.4.1), which requires a transaction to execute every write operation by writing all copies of the object and a read operation to read any copy.

This section provides a description for four well-known pessimistic replication strategies that are proposed to guarantee strong consistency for applications.

2.4.1 Read One Write All (ROWA)

In this strategy (Bernstein, Hadzilacos, and Goodman, 1987; Helal, Heddaya, and Bhargava, 1996), an update operation is applied to all the replicas. Reads, on the other hand, can be performed on any single replica. This achieved by converting a read operation on any a data item to one read operation on any a single replica, and a write operation to *n* writes, one at each replica. Thus, when the write operation commits, all of the replicas have the same value.

The obvious advantages of this approach are its simplicity and its ability to process reads despite site or communication failures, so long as at least one site remains up and reachable. But, in the event of even one replica being down or unreachable, the protocol would have to block all write operations until the failure is repaired, which means that the update operation cannot be terminated. Accordingly, ROWA fails in meeting one of the fundamental goals of replication, namely providing higher availability (Ozsu and Valduriez, 1999).

2.4.2 Primary-copy Approach

In this approach, a specific copy of a data item is designated as the primary copy (Stonebraker, 1979; Breitbart and Korth, 1997). The remaining copies are called backups. A write operation is carried out at the primary copy and all operational backups while a read operation is executed only at the primary copy (this represents a difference between this strategy and ROWA). An update that writes the replicated item is allowed to commit only after the primary and all operational backups have successfully recorded the write operation.

The advantage of this replication strategy is that at least one replica of each data item (i.e. the primary) exists, which has all updates. Moreover, ordering of updates is easy to achieve, since all updates are directed to the primary. However, the primary replica might become overloaded. While a crash of backup replica does not

require specific actions by the replication protocol, a crash of the primary replica requires reconfiguration since a new primary needs to be promoted.

2.4.3 Tokens Approach

This approach is very similar to the primary-copy approach except that the primary copy of an item can change for reasons other than site failure (Minoura and Wiederhold, 1982). In this approach, each data item has an associated token, which allows the replica holding it to access the item's replicated data. Whenever a replica needs to access specific data item to perform a write operation, it locates and obtains the token from the replica that is currently holding it. When a network partition takes place, only the partition which includes the token holder will thus be able to access the corresponding data item. One disadvantage of this approach lies in the fact that the token can be lost as a result of a communication or replica server failure.

2.4.4 Voting

This strategy generalizes ROWA scheme by trading off read and write availability (Gifford, 1979). The fundamental idea is to synchronize a quorum of servers to perform an operation prior to performing it and this done by requesting servers to vote for or against performing a certain operation. Quorums are formed such that conflicting operations require overlapping quorums (i.e. quorum intersection property), ensuring that no two conflicting operations can be executed concurrently. The benefit of this approach is that it enables higher availability and fault-tolerance than ROWA approach. A drawback of this approach is that multiple sites need be contacted even for a read operation.

According to the description of the aforementioned strategies, it can generally be said that these strategies rely on two factors as follows. The first factor is a reliable and constant communication between replicas. The second factor is the coordination between the replica servers that are involved in the performing of operations, which may require the blocking of access to these servers during performing the operations. However, these factors are not feasible in mobile environments where frequent disconnections is common, which makes these pessimistic strategies are not suitable for maintaining consistency of replicated data in such environments, especially when some (or all) mobile nodes act as replica servers (i.e. they hold an updatable replica).

2.5 Optimistic strategies

In contrast with pessimistic strategies, optimistic strategies do not limit availability. Operations (read and write) may be executed on any replica. These strategies operate under the optimistic assumption that inconsistencies, even if possible, rarely occur (Saito and Shapiro, 2005). At reconnection time, the system must first detect inconsistencies and then resolve them. Accordingly, optimistic replication allows replicas to access and update data independent from one another by delaying consistency checks. Then, replicas exchange updates with one another in a process called reconciliation which occurred periodically (Parker and Ramos, 1982; Ekenstam et al., 2001; Ratner, Reiher, and Popek, 2004). During the reconciliation process, two replicas exchange all updates that occurred since the last reconciliation and employ mechanisms for update conflicts detection and resolution. Optimistic replication (Ekenstam et al., 2001) can use a Client/Server model or a Peer-to-Peer model, which are described above.

Optimistic replication is often used to increase database availability in systems where communication is unreliable or nodes require access to data while disconnected from the network (e.g. mobile environments) (Barreto, 2003). However, optimistic replication strategies face the challenge of keeping replicas consistent. This challenge is complicated, because these strategies let updates to be issued at multiple replicas at the same time. Accordingly, optimistic replication strategies cannot guarantee strong consistency through achieving one-copy equivalence (and thus one-copy serializability). Instead of that, they provide a weak type of replica consistency guarantee called eventual consistency (Yu and Vahdat, 2000; Saito and Shapiro, 2005).

Eventual consistency guarantees that the contents of all the replicas become identical eventually. Eventual consistency is important because it is the minimal requisite of a replication strategy; without this guarantee, the replica contents may remain corrupted forever, making the system practically useless. To achieve eventual consistency, optimistic replication algorithms should provide mechanisms for quick propagation (dissemination) of updates among replicas in order to minimize the divergence between them.

Update propagation (which occurs during the reconciliation process) involves a site accumulating changes while being isolated from others, detecting when it can

communicate with another, computing the set of changes to be transferred to the other site to make the two replicas consistent with each other, and transferring the changes quickly. This propagation ensures that replicas are up to date by minimizing the divergence between them. However, updates propagation does not ensure that all replicas sort and apply the received updates in a well-defined order. Accordingly, a total ordering for updates can be implemented to achieve eventual consistency in a manner that forces all replicas to apply updates in same order (see section 2.7, which contains updates ordering in optimistic replication).

Yu and Vahdat (2000) consider eventual consistency (i.e. provided by optimistic replication) and strong consistency (i.e. provided by pessimistic replication) to clarify the relationship between consistency, availability, and performance, which is depicted in Figure 2.5.1. In moving from strong consistency to eventual consistency, application performance and availability increase. This benefit comes at the expense of an increasing probability that individual accesses will return inconsistent results, e.g., stale/dirty reads. Accordingly, to achieve increased performance, applications must tolerate a corresponding increase in inconsistent accesses.

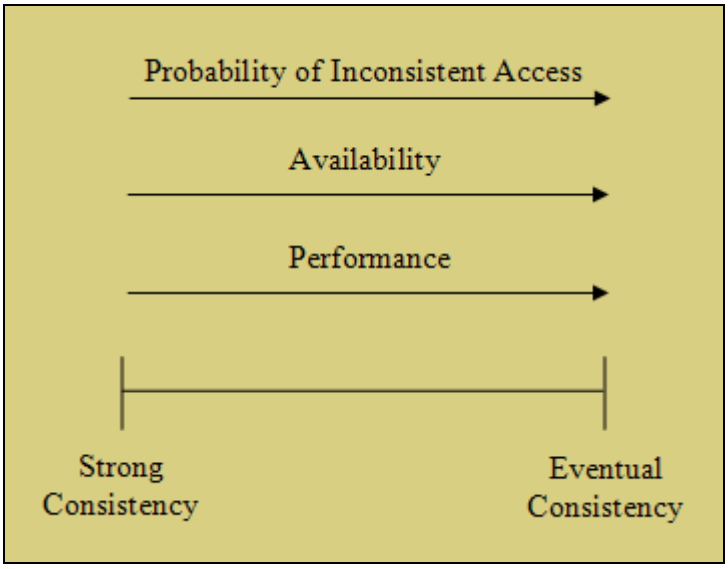

Figure 2.5.1 The Spectrum between Strong and Eventual Consistency as measured by a bound on the Probability of Inconsistent Access [adapted from Yu and Vahdat (2000)]

Based on the descriptions of both pessimistic and optimistic replication strategies, the main differences between them are summarized in Table 2.5.1 by considering some characteristics of both of them.

Table 2.5.1 Optimistic Replication vs. Pessimistic Replication

Characteristics	Pessimistic replication	Optimistic replication
Distinguishing feature	Updates occur in certain replica (or replicas), then immediate synchronization with other replicas.	Updates occur in any replica and then are propagated to other replicas.
Consistency guarantee	Strong consistency	Eventual consistency
Availability	Weak	Strong
Local reads	Return up to date values	No guarantees
Scalability	A few tens or hundreds of sites	Larger number of sites
Size of applications	Small or medium	Large scale
Environment	Local Area Network (LAN)	Anywhere

Source: Martins, Pacitti, and Valduriez (2006)

Optimistic replication is used in several solutions for handling data replication issues in mobile environments. This is because it meets the goal of providing higher availability. In the following subsections, a brief description of some representative optimistic strategies is provided.

2.5.1 Cedar

Cedar (Tolia, Satyanarayanan, and Wolbach, 2007) is a replication strategy focuses on preserving eventual consistency with acceptable performance under conditions of weak connectivity. Cedar uses a simple Client/Server design in which a central server holds the master copy of the database. Cedar's organizing principle is that even a stale client replica can be used to reduce data transmission volume. The volume reduction is greatest when the client replica and the master copy are identical. At infrequent intervals when a client has excellent connectivity to the server (which may occur hours or days apart), its replica is refreshed from the master copy. The using of a central server to hold the master copy in this schema, limits the implementation of this

schema in environments that are characterized by large number of updatable replicas and highly mobile users.

2.5.2 Read-any/Write-any Scheme

A multi-master scheme is used in Monteiro, Brayner, and Lifschitz (2007), that is, read-any/write-any. To reach an eventual consistency in which the servers converge to an identical copy, an adaptation in the primary commit scheme is used. In this adaptation, a server chosen as a primary has the responsibility to synchronize and commit the updates. The committed updates are propagated to the other servers. This strategy inherits the drawbacks of primary-copy approach, since it relies on a selected server that is responsible for synchronizing all updates between the different replicas.

2.5.3 Hybrid replication Strategy

To cope with the limitations of mobile environments, a hybrid replication strategy is presented in Abawajy, Deris, and Omer (2006) that has different ways of replicating and managing data on fixed and mobile networks. This strategy is based on a combination of pessimistic and optimistic replication. It replicates data pessimistically on the fixed network in a manner of logical three dimensional grid structure while data is optimistically replicated on mobile network based on commonly visited sites for each user, which defined as the most frequent site that request the same data at the fixed network.

This strategy does not provide the required mechanism for exchanging large number of recent updates among the hosts in both fixed and mobile networks. Also, it hinders the scalability requirement for LMDDBSs. This is because it replicates data pessimistically on the fixed network, and this is does not valid when the fixed network is to be scaled to a wide area fixed networks.

2.5.4 Transaction-Level Result-Set Propagation (TLRSP)

A mobile database replication strategy called Transaction-Level Result-Set Propagation is proposed in Zhiming, Xiaofeng, and Shan (2002). Each fixed and mobile nodes store a replica of the data. The mobile node is allowed to update its local replica. However, updates locally committed at the mobile nodes need to be verified at the fixed node before they can be globally committed. A mobile node can go through three different states:

- **Consistent State.** When the data in both mobile and fixed nodes are consistent, a mobile node is said to be operating in consistent state. The mobile node enters this state at the instant of time when a synchronization process with the fixed node is over and all differences between the fixed node and the mobile node have been reconciled.
- **Accumulating State.** The mobile node enters into this state when it begins to update the local replica of the database.
- **Resolving State.** When the mobile node is reconnected with the fixed node and starts a synchronization process, it enters into resolving state. In this state, the mobile node sends the locally committed updates to the fixed node for conflict detection. The fixed host updates those transactions that passed the validation test and the recently updated copies of the objects are forwarded to the mobile host to refresh its local copies.

To maintain the consistency of replicated data on fixed nodes, ROWA is used to perform the write operation on all fixed hosts as one logical entity. Accordingly, this strategy inherits the drawbacks of ROWA, which restricts the availability of write operations, since they cannot be executed at the failure of any replica. Moreover, this strategy suffers from the overhead that is involved in the resolving process for large number of updates.

2.5.5 Three Modules Based Replication System

Beloued et al. (2005) proposed a replication system that contains three principal modules: a replica planner, a localization manager and a consistency manager. The replica planner is responsible for the creation and placement of replicas on nodes. Next, the localization manager locates replicas for read/write operations and then performs these operations. Finally, the consistency manager ensures replica consistency by exchanging update messages after each write operation and resolving update conflicts.

In this system, exchanging update messages after each write operation and resolving update conflicts represents an overhead when the number of updates is increased. Also, there is an overhead results from that the system changes some replica locations in order to avoid the use of weak bandwidth links. These overheads hinder the replication system to scaling well in case of there are large number of nodes that are involved in the replication process.

2.5.6. Bengal

Bengal (Ekenstam et al., 2001) represents a Peer-to-Peer optimistic database replication system that allows disconnected operation by mobile peers on a distributed database. It is claimed that Client/Server replication does not match the requirements of mainly disconnected users and thus Bengal uses a Peer-to-Peer relationship.

Updates are reconciled between replicas when connectivity is available. Version vectors (see section 2.7.2) are used to compare and exchange update information. If two replicas have completely identical version vectors, then each replica has seen all updates that the other replica has seen. Accordingly, there are no updates need to be propagated. If one replica's version vector contains one or more elements that are larger than the matching elements in the other replica's, the replica with the larger version vector elements has seen more updates, and its data should be propagated to the other replica. If, however, each version vector has at least one element larger than the matching element in the other version vector, then each replica has seen at least one update unknown to the other replica. In this strategy, maintaining version vectors and using it in the reconciliation process represents an overhead when the number of hosts is increased in both fixed and mobile networks.

2.5.7 Two-Tier Replication

Gray et al. (1996) proposed a two-tier replication strategy that allows mobile (disconnected) applications to propose tentative updates while they are disconnected. Upon reconnection with the fixed network, the produced tentative updates are applied to the primary copies of data objects (called object masters), which reside at certain sites. In this strategy, the first tier consists of mobile nodes, which are frequently disconnected from the fixed network. The second tier consists of base nodes, which are stably connected to each other through the fixed network.

Two types of update transactions are supported: base and tentative transactions. Base transactions access master objects whereas tentative transactions access local copies of data objects and they run on the first tier. When the connection is established, tentative transactions are sent to the base nodes to be re-processed as base transactions to reach global consistency.

Two-Tier Replication ensures convergence of the replicas, but it does not allow reconciliation between mobile replicas. It can suffer from heavy re-processing overhead for tentative transactions when the number of these transactions is increased.

2.5.8. Epidemic Update Propagation Protocols

Optimistic replication can be implemented using epidemic update propagation protocols. In these protocols (Demers et al., 1987), any site communicates with any other and transfers both its own updates and those received from other sites (i.e. any site can send updates to any other site). Firstly, update operations are executed locally at any single site. Later, sites communicate to exchange up-to-date information. In this way, updates pass through the system like an infectious disease, since they spread through random, pair-wise exchanges, hence the name epidemic. Thus, users perform updates on a single site without waiting for communication and the system can schedule communication at a later convenient time. These algorithms rely on the application-specific update operations being commutative and maintain the causal ordering that exists between operations. Examples of such protocols include Rabinovich, Gehani, and Kononov (1996), Agrawal, El Abbadi, and Steinke (1997), and Holliday et al., (2003). Anti-entropy propagation mechanism (Golding, 1992; Petersen et al., 1997) is an example for implementing epidemic protocols. In this mechanism, each site periodically reconciles with a randomly chosen site.

2.5.9 Deno

Deno (Keleher, 1999; Keleher and Cetintemel, 2000; Cetintemel et al., 2003) is a replicated database strategy that is targeted for weakly connected environments. Deno adopts an optimistic strategy that allows updates to be received at any replica in order to provide a highly available service. To achieve eventual consistency, Deno relies on a voting approach that is implemented through pairwise epidemic information flow.

In the voting approach, when a tentative update is issued at a given replica it is placed in a queue of candidate updates waiting to be voted. To commit these tentative updates, Deno regards update commitment as a series of elections. Each election decides, amongst a collection of concurrent tentative updates, which one of them should be committed while the remaining updates are aborted. Each replica acts as a voter in such elections. Similarly, each tentative update acts as a candidate for one

election. Once an election is over, one candidate wins the election and such winner is the same at every replica. Then a new election is started.

The performance and network usage overheads imposed by the voting scheme are its main disadvantages. Moreover, the number of aborted updates may increase in case of the presence of a higher rate of updates generation.

2.5.10 Bayou

Bayou (Terry et al., 1995; Petersen et al., 1996; Petersen et al., 1997) is a mobile database system that is proposed to meet the requirements of mobile computing applications. This system is concerned on ensuring high data availability with weak data consistency guarantees. Bayou used read any/write any replication strategy that allows a user to access the data from any node.

The system satisfies eventual consistency, which only guarantees that all replicas eventually receive all updates. Update propagation only relies on occasional pairwise communications between servers, which called anti-entropy sessions. Pair-wise communication supports the reconciliation of any two replicas independently of which other replicas may be available and of how the network connection between the servers is established. A replica can choose its anti-entropy partner at random or based on other knowledge, like network characteristics.

Bayou requires applications to provide conflict detection and resolution instructions along with each data update they make. These instructions are designated as dependency checks and merge procedures, specified by the application which issued the update. A replica executes an update's dependency check before applying it. If the dependency check detects a conflict, the update's merge procedure is called.

Bayou's performance is significantly constrained by the overhead resulting from the application of dependency checks and merge procedures for conflicts detection and resolution. The system does not provide replication transparency, since the application explicitly participates in conflict detection and resolution.

2.5.11 Coda System

Coda (Satyanarayanan, 2002) is a Client/Server based replication system in which an optimistic consistency strategy is used to enable the disconnected client to read and update the data in its cache. A client can be in one of three distinct states throughout its execution: hoarding, emulation and reintegration. The client is normally in the

hoarding state, when it is connected to the server infrastructure and relies on its replication services. Upon disconnection, it enters the emulation phase, during which update operations to the cached objects are logged. When a connection is again available, the reintegration occurs, in which the update log is synchronized with the objects stored in the servers' disks. The hoarding state is then entered. However, disconnected operation of clients depends strongly on the server infrastructure, since the updates made during disconnection will only be available to other clients after reconciliation with the server. This makes Coda's disconnected operation model is inadequate for applications that impose exchanging of updates among mobile hosts.

2.5.12 Client-Oriented Approach

Gollmick (2003) described a client-oriented service for replication in mobile database environments based on the requirements of application developers and also administrators. The client interface allows mobile applications to (re)define the data and functionality, which they want to be available offline, on demand.

The proposed replication service provides a descriptive interface (SQLlike command set for replication definition/control and conflict management) to mobile applications and administrators. Using the interface, applications can select data from the server database for replication into a local database on demand.

This replication approach requires the administrator only to define things that cannot be defined at the application level (e.g. certain conflict resolution options). Therefore the replication definition is divided into two steps: replication schema definition and replica definition. The replication schema is created by the administrator and describes the subset of the source database schema, which is visible to all mobile clients for later replica definition.

2.5.13 Configured Replication Approach

Lubinski and Heuer (2001) described an approach to tailor a suitable replication strategy corresponding to the mobile environment. This approach is based on a verity of syntactic and semantic knowledge about the mobile environment including its technical and infrastructural conditions as well as its user-forced regulations.

This approach allows for configured replication based on various environmental and data characteristics. Three steps lead to the configured replication are outlined as follows.

1. Set the importance of the three aims consistency, availability, and minimal costs, perhaps with the help of a scale
2. Evaluate the available information about the network, device, application, data, and user
3. Model strategies depending on the available knowledge for all possible states (site and Connection states).

In the first step, the application is placed into the triangle of the three replication objectives. The importance of one aim establishes a smaller importance of the other aims. The aim's importance determines the applicable strategy. In the second step, all of the available data are collected or acquired, respectively. The more data are available the better the replication mechanisms can be tailored. In the third step, requirements meet available conditions. The following decisions can be made. The replication strategy (optimistic, pessimistic, hybrid) is selected based on the selected objective (consistency, availability, minimal costs, or scalability). The strategy, application, and data characteristics decide placing of data and metadata (like replication schema).

However, tailoring approach requires more management overhead. Moreover, the framework emphases softening the replication transparency for applications and users in order to inform them about possible inconsistencies, waiting periods or necessary communications.

2.6 Implementing Optimistic Replication in Large Scale Environments

A few optimistic replication strategies have been introduced in the literature to handle data replication issues in systems that consist of large numbers of hosts. The common approach taken by these strategies is relaxing consistency, in trade for higher performance and availability. Some of these strategies are presented in the following subsections.

2.6.1 Roam

Roam (Ratner, 1998; Ratner, Reiher, and Popek, 2004) is an optimistic replication system that is proposed for providing a scalable replication solution for mobile environments. ROAM allows any replica to serve operation requests, without the need of accessing a centralized server. ROAM is based on the Ward Model (Ratner et al., 2001). Ward model incorporates elements of both Client/Server and peer-to-peer

solutions to handle replica management and update distribution. Replicas are grouped into wards (wide area replication domains) (see figure 2.6.1.1). A ward is a collection of nearby nodes. All ward members are peers, allowing any pair of ward members to directly synchronize and communicate.

Although all members of the ward are equal peers, the ward has a designated ward master, similar to a server in a Client/Server model but with several differences that include:

- Any two ward members can directly synchronize with one another. Typical Client/Server solutions do not allow client-to-client synchronization.
- Since all ward members are peers, any ward member can serve as the ward master. Automatic re-election and ward-master reconfiguration can occur should the ward master fail or become unavailable.

The ward master is the ward's only link with other wards; that is, only the ward master is aware of other replicas outside the ward. This is one manner in which the ward model achieves good scaling by limiting the amount of knowledge stored at individual replicas, since replicas are only knowledgeable about the other replicas within their own ward.

All ward masters belong to a higher-level ward, forming a two-level hierarchical model. Ward masters act on their ward's behalf by bringing new updates into the ward, exporting others out of the ward, and gossiping about all known updates. Consistency is maintained across all replicas by having ward masters communicate directly with each other and allowing information to propagate independently within each ward.

Updates are exchanged within each ward (i.e. between ward members) and among wards (i.e. between ward masters) using ring topology. Such ring topology imposes that each ward member reconciles only with the next ring member. The ring is adaptive, in the sense that it reconfigures itself in response to changes in the ward composition. The authors call their replica synchronization process reconciliation. Reconciliation never directly involves more than two replicas. Reconciliation is pull-only process, new information is propagated in one direction.

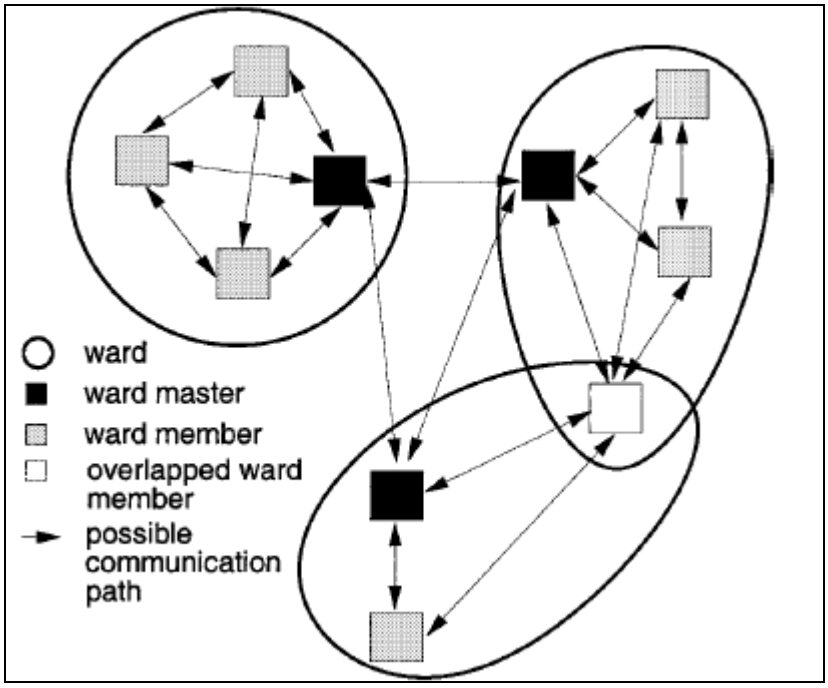

Figure 2.6.1.1 The Basic Ward Architecture
(Ratner et al., 2001)

ROAM uses version vectors to track updates and compare data versions. Each replica is assigned a version vector. Periodically, each replica reconciles its data versions with another replica, according to the reconciliation topology imposed by the ward model.

Accordingly, Roam employs optimistic replica control mechanism that ensures an eventual convergence for replica updates to maintain the consistency within each ward and among wards. ROAM tries to provide high scalability without discussing a mechanism of ensuring fast propagation of large numbers of updates that can be performed in replicas that are distributed over wide geographic areas.

A significant drawback is the consistency guarantees that ROAM provides. Since consistency relies on an epidemic propagation of updates between replicas, every read request that a client may issue will only return tentative data. This aspect restricts ROAM's applicability to applications whose correctness criteria are sufficiently relaxed to tolerate dealing with tentative data. In our case, each update is considered as stable update (i.e. it is applied permanently in the replica) and just it takes a global order on the level of the system. This means that the application always

reach a part of the stable data that is only stored on the host where application runs. These stored data may represent all stable updates that are received from other hosts.

ROAM tries to provide high scalability without discussing in details the mechanisms of handling high update rates (i.e. large number of updates that can be performed on each replica) and supporting large number of fixed and mobile replicas that distributed among wide geographic areas.

2.6.2 HARP

The hierarchical asynchronous replication protocol (HARP) (Adly, Nagi, and Bacon, 1993; Adly and Kumar, 1994; Adly, 1995) uses an optimistic update propagation scheme based on organizing the replicas into a logical, multilevel hierarchy. In this hierarchy, nodes are grouped into clusters, and clusters are organized into a tree, such that each cluster is assigned a father node in its parent cluster. Nodes in the same cluster should have efficient communication between them as well as with their father. A node *i*, originating a message, sends it to its neighbors, father and sons, and then waits for their acknowledgments. Each receiving node *j* sends an acknowledgment to the sender, then it passes the message to the next level as follows: if the message is coming from a neighbor or from the father, then j sends the message to its sons; else, if it is coming from a son then j sends it to its neighbors, its father and to its sons of clusters other than the one the message is coming from. This works recursively and a message originating at any site will eventually propagate everywhere.

In HARP, The Fast_read and Fast_write operations support relaxed consistency, while the Slow_read and Slow_write operations support strong consistency. The Fast_read and Fast_write can be performed on any replica. In the case of new updates occurred as a result of performing Fast_write, the propagation protocol relies on the abovementioned hierarchical structure for sending messages to other replicas. The Slow_read and Slow_write operations can be initiated at any replica, but they require assembling quorums from replicas of the top cluster of the hierarchy. Accordingly, the Slow_read and Slow_write operations need reliable connectivity between the replicas that participates in the quorum.

HARP only addresses single read/write operations. It propagates each update individually by relying on reliable communication. However, this reliable communication is not available in mobile environments.

2.6.3 N-ary Tree Based Updates Propagation

This strategy (Hara et al., 2005; Watanabe et al., 2007; Watanabe et al., 2008) assumed an environment where update information is immediately sent to all peers holding replicas when an update occurs. The proposed strategy creates an *N*-ary tree, whose root is the owner of the original data while the other nodes are peers holding its replicas, and propagates the update information according to the tree. Each peer in the tree records its parent and children, and by using this information, the location of a newly participating peer in the tree is autonomously determined.

However, in this strategy, the updated data must be propagated to all replica holders and this causes heavy traffic for update propagation. Accordingly, this strategy is not suitable for propagating updates to replicas that are stored on mobile hosts, since these hosts frequently change their location and disconnect from the network. Also, there is an overhead originated from the need of each mobile host to manage information about the parent and children peers.

2.6.4 Timestamp Anti-Entropy Protocol

Golding (1992) proposed a weak consistency replication strategy called timestamp anti-entropy protocol (TSAE). The TSAE protocol allows updates to be processed by a single replica, then propagated through messages from one replica to another in the background, causing replicas to temporarily diverge.

When a replica wishes to send a message, it stamps the message with the current time and the identity of the replica, then writes the message to a log. From time to time, a replica will select another replica (either randomly or deterministic), and the two will exchange the contents of their message logs in an anti-entropy session. At the end of the session, both replicas have received the same set of messages.

Each replica maintains a summary timestamp vector, indexed by replica identifier, containing the greatest timestamp it has received from other replicas. An anti-entropy session begins with two replicas exchanging their summary vectors. Each replica can determine what messages its partner has not yet received by comparing its summary vector to that of its partner. Once both replicas have received their messages, they can update their summary vector.

This protocol cannot scale to a large number of replicas, since it is assumed that any replica can communicate with any other replica. Further, it imposes a space overhead for maintaining logs and timestamp vectors.

2.7 Updates Ordering in Optimistic Replication

Updates ordering enables optimistic replication strategies to be employed (Barreto, 2003; Saito and Shapiro, 2005) by providing a relaxed consistency guarantee. This guarantee does not require that operations are executed according to a canonical order. Instead of that, replicas are allowed to execute operations in different orders as long as the relaxed consistency guarantee is maintained.

The update ordering data consistency model requires placing ordering constraints on update operations so that updates occurred at different replicas are ordered (Birman, 1993; Zhou, Wang, and Jia, 2004). Ordering constraints can be categorized into three types: FIFO, causal, and total (Jia and Zho, 2005) to reflect semantical requirements of both group of replicas and individual hosts. Generally, from the replica group point of view, as long as updates are handled (e.g. ordered or delivered) at all replicas in the same order, the data consistency is guaranteed among replicas. On the other hand, from the client point of view, it may require updates sent from the same client to be handled in the generation or sending order at all replicas.

FIFO constraint is defined between one sender and a set of receivers. It requires that any two updates that are originated and sent from the same replica *Ri* are handled by any receiver in the same order as they were generated in *Ri*.

Casual constraint is a generalization of the FIFO constraint (Schiper, Eggli, and Sandoz, 1989; Agrawal, El Abbadi, and Steinke, 1997) by considering different senders. It is based on the happened-before relation (denoted→) that is introduced by Lamport (1978). This constraint requires that if two messages m_1 and m_2 sent from two different hosts be related such that sending of m_1 happened before sending of m_2, then m_2 cannot be received before m_1 by any receiver.

Total constraint requires that for all messages m_1 and m_2 and all replica R_i and R_j, if m_1 is received at R_i before m_2, then m_2 is not received before m_1 at R_j (Vijay, 2002). Accordingly, the total ordering implies m_1 and m_2 to be received either in the order of (m_1, m_2) or (m_2, m_1), as long as the ordering is consistent at all replicas.

As compared with the other constraints, causal ordering is beneficial for maintaining consistency in many distributed applications. These applications include news systems, weather forecasting networks, stock trading, monitoring a distributed system, etc. (Raynal, Schiper, and Toueg, 1991; Adelstein and Singhal, 1995; Adly and Nagi, 1995 ; Alagar and Venkatesan, 1997).

The mechanisms for implementing causal ordering include logical clocks and version vectors (Barreto, 2003; Ghosh, 2006). The purpose of these mechanisms is to timestamp events that occurred on each replica in order to determine the casual order for each event, detect conflicts, and determine the set of updates to be exchanged between replicas. These mechanisms are as follows.

2.7.1 Logical clocks (Lamport clocks)

Lamport (1978) suggested that each process in a distributed environment implements a logical clock that is used to assign logical timestamps to local events.

According to Lamport's point of view, a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. More precisely, a clock C_i for each process P_i is defined to be a function which assigns a number $C_i(a)$ to any event a in that process. The entire system of clocks is represented by the function C which assigns to any event b the number $C(b)$, where $C(b) = C_j(b)$ if b is an event in process P_j. There is no assumption is made about the relation of the numbers $C_i(a)$ to the physical time, so the clocks C_i can be thought as logical rather than physical clocks. They may be implemented by counters with no actual timing mechanism.

A clock C_i in process P_i is initially set to 0 and advanced according to the following rules:

(1) Each time a local event takes place in P_i, $C_i := C_i + 1$

(2) When sending a message, append the value of C_i to the message

(3) When receiving a message, $C_i := 1 + \max(C_i, C_m)$

Where C_i is the local value, and C_m is the value appended with the incoming message from another process P_j.

Based on the value of the logical clock C_i, the logical timestamp $L(e)$ of an event e occurring in process P_i is the reading of clock C_i when e occurs. A logical clock is correct if it holds for any two events e_i and e_j that $L(e_i) < L(e_j)$ if $e_i \rightarrow e_j$. However, the opposite does not hold; $L(e_i) < L(e_j)$ does not necessarily imply that e_i

$\rightarrow e_j$. This means that the timestamp values of two events cannot reveal if they are causally related. Therefore, timestamps provided by the logical clock is consistent with causality, but does not characterize causality and can not be used to prove that events are not causally related. This represents a weakness of logical clocks.

Sun and Maheshwari (1996) used logical clocks in their ordering mechanism.

2.7.2 Version Vectors

A version vector (Mattern, 1989; Fidge, 1991; Ghosh, 2006) (also known as a vector clock) is a vector of counters, one for each replica in the system. Version vectors overcome the weakness of logical clocks. Their goal is to detect causality. They define a mapping V from events to integer arrays, and an order < such that for any pair of events *a*, *b*: a happened before b is equivalent to $V[a] < V[b]$.

In a replication system containing N replicas 0, 1, 2, ..., N-1, for every replica i, the version vector V is an integer vector of length N that consists of a set of timestamps, one for each replica. Like the logical clock, the version vector is also event-driven. Each element of V is a logical clock that is updated by the events local to that replica only.

The version vector V_i of a replica R_i is maintained according to the following rules:

(1) Initially, $V_i[k]:=0$, for $k=1,\ldots, N$ replicas.

(2) On each internal event e, replica R_i increments V_i as follows: $V_i[i] := V_i[i]+1$.

(3) On sending message m, R_i updates V_i as in (2), and attaches the new vector to m.

(4) On receiving a message m with attached version vector $V(m)$, R_i increments V_i as in (2). Next R_i updates its current V_i as follows: $V_i:=\max\{V_i, V(m)\}$.

Two version vectors can be compared to assert if there exists a happened-before relationship between them. Given two version vectors, V_1 and V_2, V_1 causally precedes V_2, meaning that a happened-before relationship links V_1 to V_2, if and only if, the value of each entry in V_2 is greater or equal than the corresponding entry in V_1. If this condition is verified, V_2 is said to dominate V_1. If neither V_1 dominates V_2, nor V_2 dominates V_1, V_1 and V_2 are conflicting versions.

The size occupied by version vectors is linearly dependent on the number of replicas in the system. This is a significant scalability obstacle when concerning

systems with a high number of replicas. This is because all messages of a distributed computation have to be tagged with a timestamp of size equals to the number of replicas in order to maintain the version vectors on other replicas.

There are many strategies used version vectors for implementing causal ordering including Ladin et al. (1992), Singhal and Kshemkalyani (1992), Adly and Nagi (1995), Prakash et al. (1996), Prakash et al. (1997), Satyanarayanan (2002), and Ratner, Reiher, and Popek (2004). However, these algorithms do not act accord with the characteristics of LMDDBSs (host mobility, large no of replicas, large number of updates, etc.). Their implementation in such systems implies both communication and storage overheads. This is because each message in these algorithms carries large amount of information to implement causal ordering.

On the other hand, in our proposed ordering mechanism, causal ordering is implemented by assigning timestamps for local events using a variable called Real-Like clock. This variable ensures a unified assignment for timestamps according to the exact time when the event occurred in each replica. The size of this variable is not affected by the number of replicas. This leads to ensuring sufficient unified ordering of updates without needing for each node to keep track for the time information in all other nodes as in the version vector method.

2.8 SPN Background

Stochastic Petri Nets (SPNs) are used in this research to model and analyze the stochastic behavior of the replication system using the proposed strategy. SPNs are derived from standard Petri nets (PNs). PNs are an important graphical and mathematical tool used to study the behavior of many systems. They are very well-suited for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, and stochastic (Ajmone, Balbo, and Conte, 1986; Murata, 1989). A PN is a directed bipartite graph that consists of two types of nodes called places (represented by circles) and transitions (represented by bars). Directed arcs connect places to transitions and transitions to places. Places may contain tokens (represented by dots).

The state of a PN is defined by the number of tokens contained in each place and is denoted by a vector M, whose i^{th} component represents the number of tokens in the i^{th} place. The PN state is usually called the PN marking. The definition of a PN requires the specification of the initial marking M'. A place is an input to a transition

if an arc exists from the place to the transition. A place is an output from a transition if an arc exists from the transition to the place. A transition is said to be enabled at a marking *M* when all of its input places contain at least one token. A transition may fire if it is enabled. The firing of a transition t at marking *M* removes one token from each input place and placing one token in each output place. Each firing of a transition modifies the distribution of tokens on places and thus produces a new marking for the PN.

In a PN with a given initial marking *M′*, the reachability set (*RS*) is defined as the set of all markings that can be "reached" from *M′* by means of a sequence of transition firings. The *RS* does not contain information about the transition sequences fired to reach each marking. This information is contained in the reachability graph, where each node represents a reachable state, and there is an arc from M_1 to M_2 if the marking M_2 *is* directly reachable from M_1. If the firing of *t* led to changing M_1 to M_2, the arc is labeled with *t*. Note that more than one arc can connect two nodes (it is indeed possible for two transitions to be enabled in the same marking and to produce the same state change), so that the reachability graph is actually a multigraph.

SPNs are derived from standard Petri nets by associating with each transition in a PN an exponentially distributed firing time (Ajmone et al., 1995; Bause and Kritzinger, 2002). These nets are isomorphic to continuous-time Markov chains (CTMCs) due to the memoryless property of exponential distributions. This property allows for the analysis of SPNs and the derivation of useful performance measures. The states of the CTMC are the markings of the reachability graph, and the state transition rates are the exponential firing rates of the transitions in the SPN. The steady-state solution of the equivalent finite CTMC can be obtained by solving a set of algebraic equations.

2.9 Summary

This chapter described a background material for this thesis and outlined representative replication strategies that are devoted for both distributed and mobile environments. Section 2.2 focused on the relationship between replicas and the role that each replica plays in the replication system regarding update operations. Section 2.3 discussed the abstract replication strategy and classification of replication strategies into pessimistic and optimistic strategies. Section 2.4 provided a brief description for four pessimistic strategies. Sections 2.5 and 2.6 outlined representative

optimistic replication strategies that are devoted to mobile computing environments and large scale systems. Section 2.7 discussed the mechanisms that are used by optimistic replication strategies for implementing updates ordering.

In summary, we argue that existing replication strategies are not coping well with the characteristics of large-scale mobile systems containing large number of geographically distant replicas that experience large number of updates. Accordingly, such systems demand new solutions for addressing data consistency through ensuring fast propagation of recent updates among replicas as well as supporting scalability for encompassing new replicas when the replication system covers new geographic areas. Moreover, these solutions should provide mechanisms for updates ordering that impose low communication and storage overheads.