AN IMPROVED QUORUM SELECTION ALGORITHM (IQSA)

by

SAMER YOUNES

B.S., Computer Science, Lebanese American University, 2007

Thesis submitted in partial fulfillment of the requirements for the

Degree of Master of Science in Computer Science

Division of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

September 2007

LEBANESE AMERICAN UNIVERSITY

Thesis approval Form

Student Name: Samer Younes I.D.: 199907790

Thesis Title: An Improved Quorum Selection Algorithm (IQSA)

Program : M.S. in Computer Science

Division/Dept : Computer Science and Mathematics

School : Arts and Sciences - Beirut

Approved/Signed by:

Thesis Advisor Dr. Ramzi Haraty [redacted]

Member Dr. Sanaa Sharefeddine [redacted]

Member Dr. Faisal AbuKhzam [redacted]

Date: 24 September, 2007

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Samer Younes

Signature 

Date: 24/09/2007

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

ACKNOWLEDGMENTS

The author wishes to thank all those who made this work possible and bore with me through it all; especially my beautiful wife-to-be, Rana, for all the moral and emotional support.

Abstract

As communication becomes more and more an integral part of our day to day lives, so our need to access information increases as well. Mobility is currently one of the most important factors to consider in our aim to achieve ubiquitous computing, and with it rises the problem of how to manipulate data while maintaining consistency and integrity. Recent years have seen tremendous interest in quorum systems adapted to mobile hosts; however, the more recent topic, of studying the effects of mobile networks on quorum systems, has also been the focus of interest for building quorums aware of their network surroundings. This thesis presents a novel approach in selecting mobile hosts to form epidemic quorum coteries, based on metrics measured by mobile hosts and then transmitted to Base Station servers. The BS constantly maintains a vigil on the state of these mobile hosts to provide higher quorum availability and ultimately higher data accessibility, better integrity and consistency.

Keywords: distributed databases, host selection, mobility, epidemic quorum, heap

Contents

List of Figures

List of Tables

Chapter 1- Introduction

Pervasive computing is a term loosely used to describe the current state of computer technology in modern life. Our reliance on computing mediums increases with the need for mobility, connectivity and data availability.

We often find that the data we need, located on multiple devices and in various locations, is inaccessible directly most of the time. Pervasive computing also encompasses the concepts of data, connectivity and their ubiquitous presence in an individual's daily life.

As an example, in a single day, the average individual can go through a minimum of three different devices to perform various everyday tasks such as checking his/her email account on the desktop computer, calling a family member on the cellphone, listening to some music in the background on his/her personal laptop and syncing all appointments from his/her palm-pilot to his/her email client. Although we take such things for granted, our daily interaction with data keeps increasing, and with it the need to keep our data accessible, well-organized and safe.

1.1- Background

Current tools, such as Google's plethora of desktop search tools, have reduced the divide by centralizing data management, but they do not address issues such as unrelated data repositories, data safeguard and integrity. In addition, the problem of intermittent connectivity through wireless enabled devices is also a major issue in mobility. As such, maintaining data integrity and

consistency in such mobile environments is a challenge; given the diverse factors that influence connectivity, from geography to battery life. Current trends in mobile databases suggest the adoption, among other techniques, of the quorum approach in which multiple mobile hosts perform reads and writes based on the majority vote of hosts selected to the quorum.

Although the quorum algorithm has been extensively studied since its earlier days, adapting it to mobile devices with connectivity issues and providing a solid quality of service (QoS) for quorum members is still in its infancy. The adaptation of quorum consensus to mobile environments to insure a high level of service is one of the main challenges to tackle.

1.2 Scope of the Thesis

The primary aim of this work is to come up with an architecture capable of tracking mobile hosts and providing the foundation for reliable quorum operations. This is achieved by improving quorum host selection techniques, based on a scoring mechanism that combines multiple measurements, such as signal strength, database frequency access, priority and a derived trend calculated by using a weighted linear regression function. This system would complement any existing architecture directly linked to the DBMS, thereby insuring ACID properties by simply plugging into it when quorum consensus is being used. The secondary purpose of this architecture is to attempt to reduce the amount of status messages passed between both mobile and fixed hosts, thus freeing up bandwidth for more critical operations.

1.3- Organization

The next section of this thesis presents a brief overview on the current literature included in its scope. The third section lists and explains the current trends and research in Mobile Database Architecture and discusses the specifics of each approach. Section four presents the contributed architecture, and the mathematical model on which it is based. Section five provides extensive theoretical and model simulation results that support both the viability and added efficiency of the proposed model. Last, we discuss the shortcomings and possible improvement paths to the proposed contribution.

Chapter 2- Literature Review & Related Work

Recent years have seen a much renewed interest in mobile computing, especially with the advent of ubiquitous wireless communication. This interest has also challenged some long held concepts, which applied elegantly to wired networks but fail to apply in wireless networks; more specifically mobile wireless networks.

The architectures for mobile database systems have been varied and diverse; however, all these architectures still adhere to the ACID principals of standard databases systems. Serrano-Alvarado et. al [21] provide an excellent overview of the various mobile database models, the most popular of which, according to [21] and Kumar [11], are the clustering model introduced by Pitoura and Bhargava in [18], explained in chapter 3.2.1, the 2-Tier replication model introduced by Gray et. al in [4], explained in chapter 3.2.2, the HiCoMo model presented in [13] by Lee and Helal, explained in chapter 3.2.3 and the Pro-Motion model by Walborn and Chrysanthis [23] [24], explained in chapter 3.2.4.

With a comprehension of the workings of these various models, recent publications by Holliday et. al [8], [9] and Baretto Ferrero [1], seem to agree that quorum systems are the best suited for a mobile environment. They present a study of how epidemic algorithms can help increase the reliability and availability of mobile database systems. An in-depth look at the model is also tackled in chapter 3.3.

Very recently, the interest in studying the effects of mobile network environments on the performance and availability of quorum systems has spurred interesting publications in this area; most notable of which are, [17] by Peysakhov et. al. that provides a general quorum availability evaluation and Baretto Ferrero [1] that deals more specifically with the performance evaluation of epidemic quorum algorithms.

Other work by Gupta et al. [7] and Golovin et al. [4] were also studied, pertaining to quorum placement and congestion management, but the findings, although very interesting, were left as future improvements on the architecture presented herein.

2.1- Architecture of Mobile Databases & Mobile Transaction Management Models

This chapter starts off by explaining the architecture of a mobile database system and its components to pave the way for a discussion of previous research in the area of mobile transaction management. Definitions of terms that are subsequently used throughout the document will also be covered here.

2.1.1- Mobile Database System Architecture

Mobile database systems can essentially be summarized as large distributed database systems, with the added property of catering for mobile units that may experience connectivity outages depending on their geographical location or data processing capabilities. The acronym MDS (Mobile Database System) is used to refer to them. An MDS comprises of interconnected computers and communication systems, both wired and wireless (typically GSM or 802.11), which allow users to connect to the systems that host the requested data. Typically an MDS comprises of Fixed Host (FH) units, interconnected through a high speed wired network, Base Stations (BS) and Mobile Units (MU) or Mobile Hosts (MH) (Both these terms will be used interchangeably in the text), which typically describe a portable computing device ranging from a laptop to a PDA.

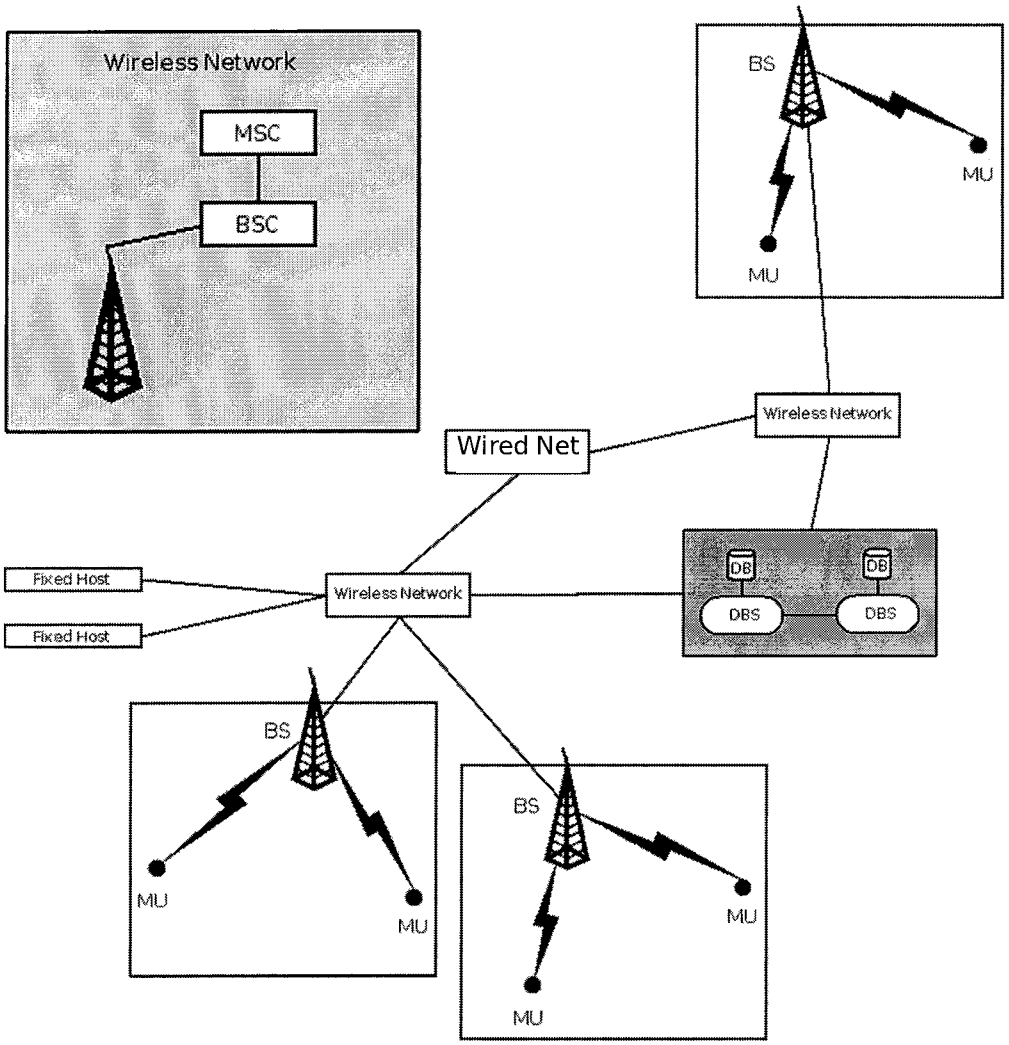


Figure 2.1 Mobile Database System (MDS) Architecture

Figure 2.1 depicts the various components of an MDS and the interconnectivity among its various components. The mobile unit, as stated earlier, refers to any hand held device that can be carried by its owner and is able to communicate with other computing devices. These mobile units are typically considered outlying components of the system and are inter-networked

through the wireless network infrastructure. This wireless infrastructure can be a typical 802.11 wireless network, a GSM network or a hybrid of both, and comprises of a Base Station (BS) with which the MU communicates directly. The BS, in turn, communicates with a Base Station Controller (BSC), which work in tandem to control and coordinate traffic among the various BS in a given geographical region.

Communications from the MU are then routed either through a standard wired network infrastructure or directly to the Database Server, which handles fetching requested information and sending information back to the MU that originated the request.

An MDS with the above architecture would provide the following basic properties:

Geographical Mobility: Essentially allowing a mobile host to move around and still have access to their data without affecting connectivity.

Connection/Disconnection: This is again a feature of a mobile unit, through which an MU may opt to disconnect at any time and reconnect to access the needed data, through any available server.

Data Processing: Use of data clustering and data partitioning in an MDS would greatly enhance the response time of MU, by reducing the amount of hops to get to a particular piece of data. This would also minimize the amount of data that needs to be transported. Typically, a MU would have relevant data subsets stored locally These data subsets may be frequently accessed data or relevant data in the context that MU is operating in. Whereas a FH would

have a replica of the entire database from which MUs are served depending on geographical proximity or database load.

Network Connectivity Infrastructure: Which allows mobile hosts as well as all the other elements of the MDS to communicate with each other, typically through wired or wireless networks.

Transparency and Scalability: Insures that different types of communications can coexist without one interfering with another and that the infrastructure is adaptable enough to add or remove clients at any time from the network.

It's also worth noting that an MDS would comprise of multiple DBSs and various DB configurations, which could be in various geographical regions and contain, either replicated data or spatial data pertinent to the geographical location where that DBS is located. The architecture is also dynamic enough to allow semantically related data to be clustered together. The above three DBS types (replicated, spatial, semantic) may also coexist together and provide a hybrid MDS incorporating all of these features. The choice for such configurations is usually related to data availability and redundancy considerations. In the case of geographically related data this distribution could also serve to provide users with data relevant to their current geographical location as well.

As such we can define two broad categories of replication, spatial replication for location dependent data or temporal replication for traditional databases. The main difference being that temporal replication provides a single, unique, consistent value for any accessed data, from any replication site, whereas spatial replication (although provides the same DB structure on all sites) provides a single unique value of requested data, depending

on the geographical location where that request has been made. Thus in spatial replication data in various geographical locations may have different values that are unique to a particular geographical context. Note that temporal replicas of spatial data in a particular geographical location is also possible.

2.1.2- Transaction Execution in MDS

Transactions are the basic atomic units, that carry requests and data between the DBS and the Client (MU or FH). In an MDS, the distributed nature of data and the nature of requests made by clients, involves a lot of parallel processing, both to improve system performance as well as provide the necessary data. In short transactions are no longer treated as atomic units in distributed databases, but are themselves amenable to being divided into sub-transactions, that are spread and sent to the DBS containing the requested data. The entity responsible for breaking down a transaction into subcomponents is called the coordinator. A coordinator is usually a system that is aware of the network within its coverage zone, spatial location of requested data and geographical location of associated DBSs, making it a crucial component in request dissemination (see Figure 2.2). As such a coordinator is a system that should satisfy, at the very least, the following two properties:

Continuous Connectivity: A coordinator should (in theory) maintain connections with the rest of the system with no downtime or intermittent failures.

Continuous Availability: It should also be accessible at any time

with no downtime and provide comparatively large storage capabilities for cached data.

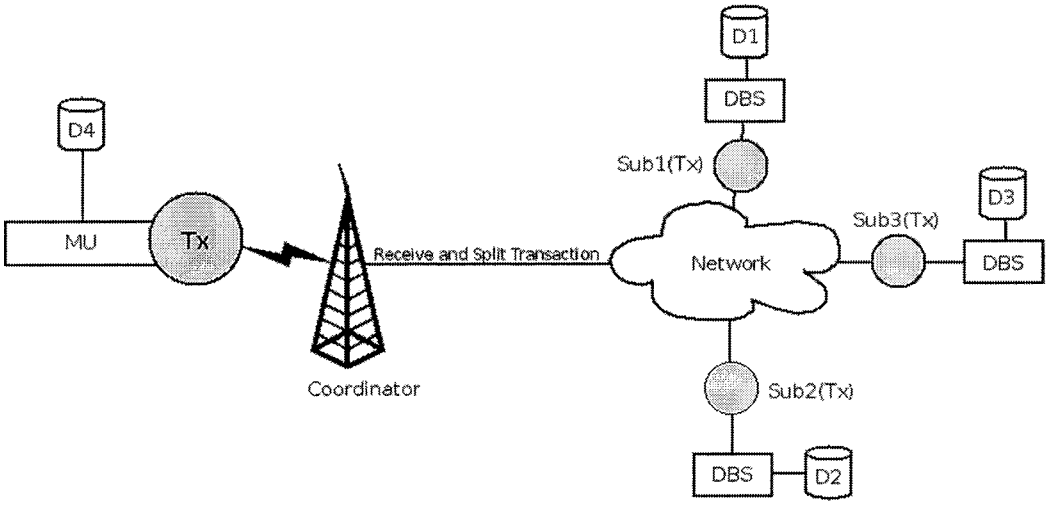


Figure 2.2 Transaction Execution in MDS

According to [11] the most suitable entity (as represented in the above architecture) satisfying the above requirements, would be the BS acting as the coordinator, as its features include, in addition to the points mentioned above, direct communication with MUs in its coverage zone.

Communication between the MU and the DBMS typically involves transaction exchanges, that can be initiated by the DBS, the MU or both. The processing of these transactions can also take place exclusively at the MU where it was instantiated, exclusively at the DBS, or at both locations simultaneously. In a multi-node setup the following foreseeable scenarios may arise:

Transaction Originating at the MU: Transactions originating at an MU can be processed entirely at an MU, with no requirements

from the DBS. If the data is not available on the MU, then the transactions may be entirely processed at the DBS or execution may be split among the MU and the DBS. In all cases the final result is returned to the initiating MU.

Transaction Originating at the DBS: Such transactions are usually executed either locally on the DBS or may be split across multiple DBSs. The final result is returned to the initiating DBS.

Given that mobile units may move from one coverage area to another, changing the BS/Coordinator to which it is attached, a transaction may complete in one of the following scenarios:

Static MU: the MU establishes a connection with a given BS, which is designated as its coordinator. If the MU does not move then the transactions initiated by the MU will complete through the same coordinator it was initiated from. If the MU is on the move and finds itself outside the coverage area of its designated coordinator, then all the transactions executed or requested by the MU will still be processed by the primary designated coordinator even if the BS changes.

Dynamic MU: In this scheme the designated coordinator of an MU can change depending on the location of the MU and the coverage area of the BS. Thus, when an MU leaves one BS's coverage area for another, its the MU's responsibility to signal the new BS the identity of its previous coordinator so that any transactions initiated by the MU may be handled by the new coordinator. Migration from one coordinator to another may happen using various schemes, two of which are worth mentioning.

The first scheme, implies a residency limit, wherein the

coordinator of the MU is changed once that MU has stayed in a new BS's coverage area longer than the specified residency limit. Other parameters in this routine that may affect the designation of a new coordinator to the MU may include, congestion, traffic load and other network parameters that may prohibit such a migration.

The second scheme involves keeping the MU attached to a given coordinator as long as the number of hops required to get to its coordinator is one. This is also referred to as adjacency migration where, as long as the MU is within cells adjacent to its coordinator, that coordinator will remain assigned to the MU given the high probability of that MU returning to the coordinator.

2.2- Mobile Transaction Models

This section covers the most recent mobile transaction management and execution models by giving an informative overview of their features and mode of operation, as well as comparing their high-points and low-points. All discussed models adhere (to varying extent) to, or extend, the ACID transaction execution model. This model has been the generic underlying of all transaction execution models because of its proven reliability, insofar as the properties it provides guarantee that transactions are processed in a reliable fashion. A brief explanation of ACID properties follows:

Atomicity: This property guarantees that any executing transaction is treated as an entity that may not be fragmented into any smaller components. This property insures that any operations within a transaction either all complete successfully, or, if any of the operations should fail, the entire transaction fails. This subscribes to the all or nothing execution paradigm.

Consistency: Insures that all integrity constraints (regardless of granularity) is maintained to provide an accurate and timely view of data. This property incorporates transaction conflict resolution models as well as data constraints, whether at the level of an entity or related entities.

Isolation: This property ensures that no data may be viewed in an intermediate state by one transaction while another transaction is operating on that data. Essentially this means that data can be viewed in a single state and never in two (or more) simultaneous states. The formal adjective describing this state is referred to as a serializable state. This property complements the consistency property mentioned above.

Durability: Refers to the persistence of the operations of a transaction after successful execution, so that any modifications carried out by that transaction on a data item will not be rolled back.

As mentioned in the previous chapter, mobility introduces new challenges to the way data is handled and presented, because of its spatial quality, the same data in two different geographical locations will have different values. Therefore, maintaining consistency also becomes more complex as the spatial component gets factored in. Most MDS transaction execution models introduce the concept of spatial consistency. The idea consists of providing an MU consistent data, based on its current location, in a way that the owner of the MU can use. As an example, assume a user requests information about a particular restaurant which is a mile from his current location. If the returned answer was given back after that user has passed the restaurant, then that request is

no longer relevant; the user having left the geographical area where this information would have been useful. The reason for getting a belated response could be due to factors mentioned prior such as bandwidth limitation, MU disconnections, or query processing time.

Following are the most current mobile transaction models according to [21], each which will be described in detail with accompanying figures, where applicable.

2.2.1- Clustering

This model introduced in [18] and extended in [19], assumes a fully distributed system where data is clustered based on a set of dynamic semantic proximity. These clusters are created and merged dynamically based on either global conditions, or on conditions set by mobile users, which would allow them to cluster frequently accessed data in a way that minimizes access time.

Data in a specific cluster is required to be fully consistent, in so far as various versions of the same data cannot co-exist. Different clusters may exhibit what [19] refer to as bounded inconsistency wherein data items may have different values in different clusters based on a certain set of predefined metrics. The metrics include the number of different copies of the data in all clusters. Once this limit is reached, a reconciliation function takes care of minimizing the value of that metric back to a lower threshold.

The model also defines two types of consistency, an inter-cluster consistency and an intra-cluster consistency. Inter-cluster consistency, is the equivalent of global consistency in traditional

database models, with the difference that inter-cluster states may contain irregularities that fall within the values of the bounded inconsistency threshold. The cluster is referred to as being m-degree consistent, where the degree refers to the divergence in the value of the chosen bounded inconsistency. Intra-cluster consistency, on the other hand, is equivalent to strict consistency in traditional database models, whereas no data in the specified cluster may have more than one value associated with it.

To achieve this the model introduces two types of transactions, weak transactions and strict transactions. Weak and strict operations are also introduced in terms or reads and writes. As a rule, strict transactions may apply to a single cluster or be inter-cluster operations leaving the database in a globally consistent state. On the other hand, weak transactions may only be executed within a particular cluster only and modifications by a write operation become permanent once the scheduled reconciliation function is run. In general, strict reads will only read values written by strict writes, and weak reads will read values written by a weak write. A strict transaction becomes a set of strict operations (reads and writes), whereas weak transactions refer to a set of weak reads and writes.

It also worth mentioning that in this approach MHs, when disconnected, become individual clusters on their own. Only weak transactions are allowed to be performed by the MH on its dataset. Once that MH reconnects, a synchronization operation process is executed to bring the database back to a globally consistent state. Table 2.1 summarizes properties and mechanism of the clustering model to maintain ACID properties:

Table 2.1 Clustering Model ACID Properties

	Atomicity	Consistency	Isolation	Durability
Clustering	MH Disconnected: weak operations are performed and locally committed. MH Connected: strict operations are performed using 2 Phase Commit (2PC) DB Server: reconciliation function commits or rolls back transactions in case of conflict	Intra-cluster consistency: For the two types of data values either weak or strict, a single value for each may exist for each no data my value may have multiple values for a specific type. Inter-cluster consistency: Is maintained by insuring that divergence doesn't exceed the specified degree of inconsistency.	Uses Strict 2PL for concurrency control and introduces 4 lock tables one for each operation type (WR, WW, SR, SW). Intermediate values are not visible to transactions. but locally committed values are visible to local transactions on a specific MH. Strict versions of data are replicated using a quorum consensus protocol, whereas weak versions are propagating according to the degree of inconsistency.	No guarantees on durability. Dependent on the degree of inconsistency in the cluster.

Figure 2.3 illustrates the functioning of this model:

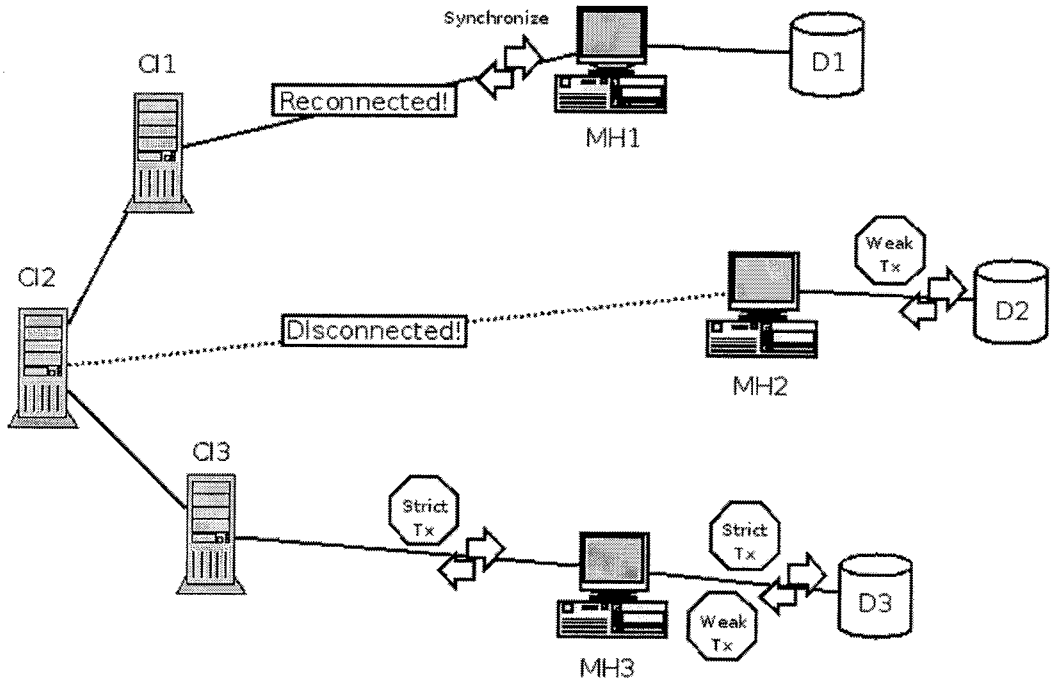


Figure 2.3 The Clustering Model Architecture

Two of the main drawbacks in this approach involve issues with the architecture itself. First, the fact that clustering maintains two different types of data, makes replication a very complex operation. Compounded to that, the model does not fully adhere to the durability property of the ACID model, as locally committed transactions may be rolled back because of reconciliation conflicts, leading to a higher degree of cascaded aborts.

2.2.2- Two-Tier Replication

Two tier replication [4], is another MDS model that relies on a lazy replication mechanism geared towards mobile environments. The model introduces the concept of Master copies to which fully replicated copies are associated. As the clustering model, it also classifies transactions in two categories, base

transactions, which operate on Master copies of data, and tentative transactions, which operate on the replicated copies when a MH is disconnected. As long as the mobile host is connected, it participates in all operations using base transactions to modify stored master copies of the data and propagate these changes through a lazy replication scheme that guarantees one copy serializability. When an MH is disconnected, it no longer has access to stored master data and may only operate on tentative copies of the data instead, using tentative transactions. Once the connection is reestablished, the MH re-executes tentative transactions as base transactions to update its master copy and propagate the data changes. The acceptance of the re-executed operation for final commit is dependent on the predefined acceptance criteria. In case of conflicts, the initiating tentative transaction is aborted. This model allows for semantic divergence between tentative and base data, and reconciliation is done by the re-execution of tentative transactions as base transactions. The adherence of this model to the ACID properties can be found in Table 2.2:

Table 2.2 Two-Tier Replication Model ACID Properties

	Atomicity	Consistency	Isolation	Durability
Two-tier Replication	MH Disconnected: tentative operations are performed and locally committed. MH Connected: Base operations are performed using an atomic commit protocol DB Server: reconciliation is performed by re-executing tentative transactions as base transactions.	Is maintained through acceptance criteria and commutative tentative transactions.	Uses a 2PL variant for concurrency control. Intermediate values are not visible to transactions. Locally committed values are visible to local transactions on a specific MH. Base versions of data are replicated through a lazy replication scheme,. Tentative versions remain local to the MH that generated them.	No guarantees on durability. Dependent on acceptance criteria during re-execution.

Figure 2.4 illustrates the architecture:

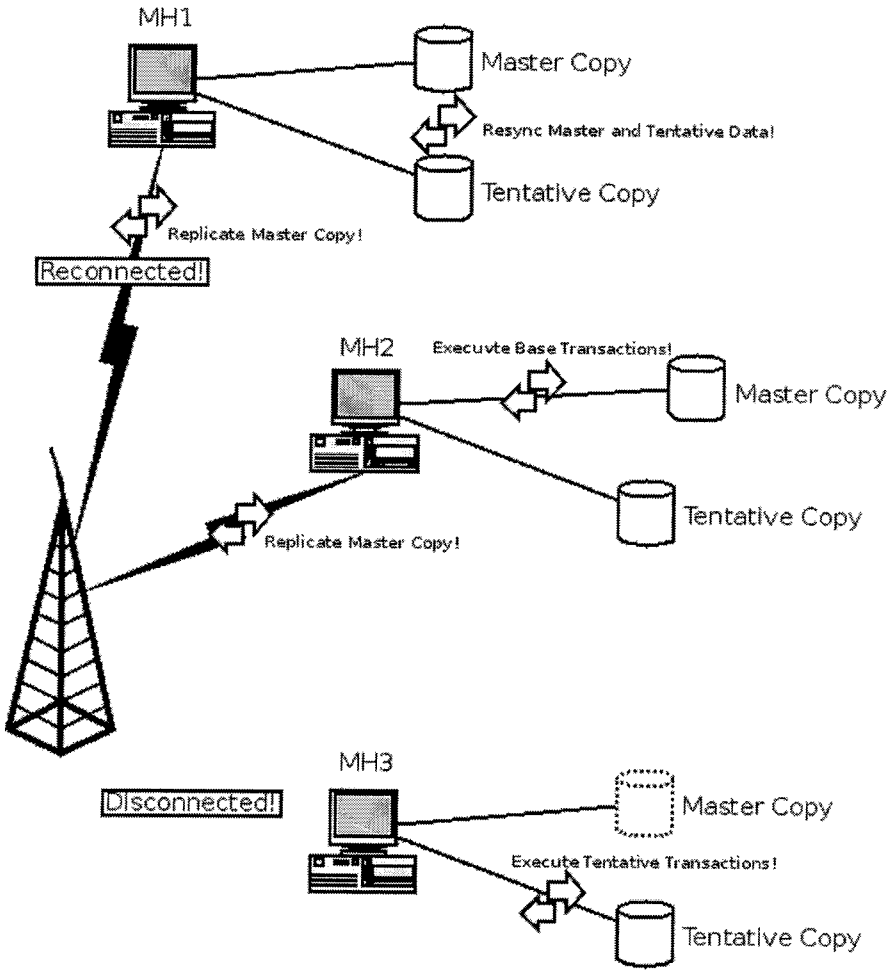


Figure 2.4 The 2-Tier Replication Model Architecture

2.2.3- HiCoMo

Introduced by [13], is yet another novel approach to managing transactions in highly mobile environments. Like the two preceding models, it distinguishes between two types of transactions, base transactions and HiCoMo transactions. The difference being that HiCoMo mobile hosts do not operate on base data but on aggregate data (summation, counts, minimum, maximum, average, etc...) which are obtained from base tables.

HiCoMo transactions are executed when the MH is operating in disconnected mode. Upon reconnection any modifications performed on the aggregate tables is then re-synced with the base tables using commutative inference and semantics functions, which allow HiCoMo transactions to be transformed into base transactions. A divergence threshold is also tolerated between base and HiCoMo transactions.

The ACID properties of this scheme are maintained through the following features: Atomicity is guaranteed in the use of an extended nested transaction model in which base transactions are treated as sub-transactions and organized in a tree like structure with parent-child dependencies. What is most striking about this approach is its flexible commit model that allows base transactions (for the same HiCoMo transaction) to be re-executed within a preset error margin (divergence criteria). Once that error margin is exceeded the transaction is aborted. The triggering of the error margin retry, due to a conflict between base transactions, results from the transformation of the HiCoMo transactions into the original base table data (i.e, a constraint on the maximum value of a data field). The re-execution of transactions is allowed in this model due to the commutative nature of the original aggregate data made available to the MH. This commutative feature also provides the model with a high level consistency. In terms of isolation, intermediate values (locally committed results) of data are only made visible to local transactions on the MH, whereas concurrency is maintained using an optimistic concurrency control strategy that uses timestamps to order operations and avoid conflict between base and HiCoMo transactions. In terms of replication, correctness is maintained through a convergence scheme that guarantees that data between HiCoMo and base tables always remains within the specific error margin. This

condition is guaranteed by the data commitment process.

One of the main drawbacks of this approach is inherent in its design. The use of aggregate data, although improving local transaction commit times, also makes the conversion process (from HiCoMo to Base transactions) rather complex, and limits the possibilities of data manipulation to commutative operations only. Table 2.3 summarizes the ACID properties of the HiCoMo model.

Table 2.3 HiCoMo Model ACID Properties

	Atomicity	Consistency	Isolation	Durability
HiCoMo	MH: HiCoMo transactions are locally committed and manipulate only aggregate data. DB Server: reconciliation is performed by re-executing HiCoMo transactions as base transactions, taking into account the predefined error margin. Transformed HiCoMo transactions are aborted if ,during re-execution, the error margin is not exceeded.	Is maintained through the generation of aggregate tables, commutative transactions, and predefined error margins.	Uses an optimistic timestamp ordering concurrency control. Intermediate values are not visible to transactions. But locally committed values are visible to local HiCoMo transactions on a specific MH. A convergence scheme maintains consistency among replicated aggregate and base tables.	Data items are committed locally directly after the execution of a HiCoMo transaction. However final commit is dependent on the successful re-execution of HiCoMo transactions as Base transactions.

Figure 2.5 illustrates the general architecture of HiCoMo:

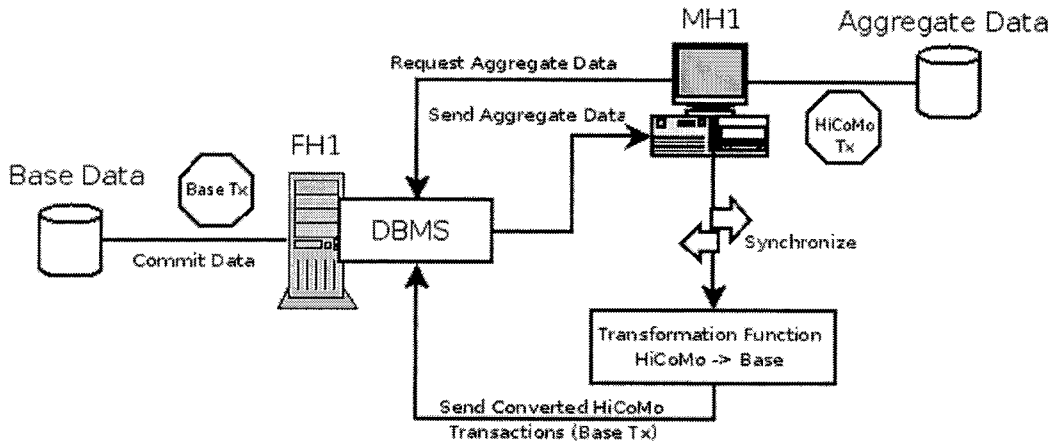


Figure 2.5 The HiCoMo Replication Model Architecture

2.2.4- Pro-Motion

One of the most innovative approaches to mobile databases was introduced in [23] [24] and is mainly a data caching scheme that allows for consistent local transaction processing. The main innovation of this scheme is the introduction of the concept of compacts. Compacts, as the name suggests, are an agglomeration of data, operations and constraints, which form the basic caching and control mechanisms among MHs and FHs, and considers all operations executed on mobile systems as a very long transaction executed on the server. In other words, all operations are executed on the MH and are synchronized later with the server. A compact manager takes care of the generation and management of compacts on the server, whereas a compact agent takes care of caching and processing transactions on the MH. Interaction between various MHs and FHs are done through the mobility manager in charge of data exchange between various compact agents. The model extends the standard transaction execution model with some specialized methods pertaining directly to the manipulation and querying of compacts to support data

manipulation and concurrency schemes. As such, compacts can be inquired about using the "Inquire" method and notified of any changes in the state of the MH using the "Notify" method. Commit and Abort operations are also used to validate or invalidate data changes performed by compact transactions. A special "Dispatch" method is used to process operations initiated by the compact agent, which are to be locally committed.

The interaction between compact agent and manager is confined to four types of transaction processing activities, which involves the agent and/or manager. When the MH is connected to the network, it always attempts to store compacts for an eventual disconnection. This constant storage of compacts is referred to in the model as hoarding. Compacts are stored in a compact registry.

While connected and performing its hoarding function, the MH will continually process transaction with the compact manager. Although the model does not differentiate between connected and disconnected modes, when interacting directly with the compact manager, the MH will be operating at a more optimized level than in disconnected mode due to the quick turnaround of operations. This is referred to as connected execution. When disconnected, the MH will revert to local processing of transactions and maintains a log of operations that can be replayed later for either recovery or resynchronization.

After the MH re-establishes a connection with the network, a synchronization process takes place to reconcile the locally committed transactions with the permanent data store on an FH. If no conflicts are detected, all updates are performed. In the eventuality of a conflict or data expiry, the following venues may be undertaken by the system: In case of data expiry, the compact

agent will attempt to get a renewal on the data item from the compact manager, pending no other transaction (from another MH) has modified that data after the expiry date. A contingency procedure associated with the compact may be triggered to attempt to remedy the problem. In case both of these operations fail, compacts that have failed to reinstate their changes with the compact manager and incorporate their changes with the permanent data store, are aborted, and all associated compacts are invalidated. Once a list of valid compacts is generated, these are allowed to issue a dispatch event to the server, replaying the operations that have been locally committed on the database server for final commitment.

As far as the model's adherence to ACID properties, Table 2.4 summarizes the various mechanisms and schemes used by PRO-MOTION to abide to them.

Table 2.4 PRO-MOTION Model ACID Properties

	Atomicity	Consistency	Isolation	Durability
PRO-MOTION	MH: Local commit using 2PC DB Server: synchronization operation to reconcile the MH's data store with the permanent data store.	Is maintained through constraints and state information in the compact upon dispatch to the MH.	Uses isolation levels (0-10) applied individually to each generated compact	Data expiry and reconciliation conflicts may rollback locally committed transactions

An overview of the PRO-MOTION architecture is depicted in the Figure 2.6:

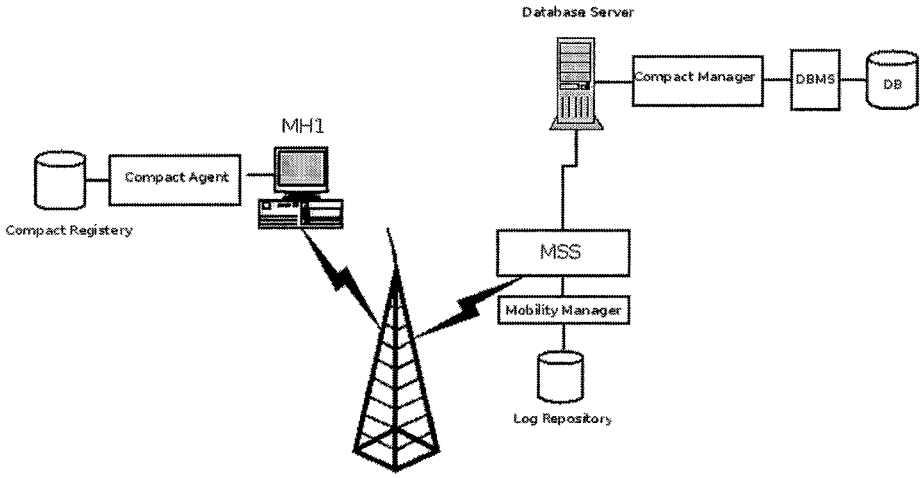


Figure 2.6 The PRO-MOTION Replication Model Architecture

2.2.5- Semantics Based

The architecture described in [25], exploits semantic relationships between data objects to split large data stores into smaller manageable chunks that can be cached and manipulated by MHs. The approach focuses on object fragmentation to provide better concurrency and optimize communication. Objects in this model refer to aggregate data or other data structures, such as queues, stacks and sets. Every cached object at the MH is exclusively locked for that MH. Hence, multiple access to the same data object is not allowed under this model. The architecture is very reminiscent of the PRO-MOTION model (albeit being its predecessor) in some aspects, in terms of MH information caching. As such the semantics model uses two parameters: one indicating which data should be cached (the selection criteria), the second

specifying the constraints that apply to the selection in order to maintain global consistency (consistency conditions). When an MH is disconnected, all transactions are locally committed until the MH regains its connectivity, at which point a reintegration process, merging both MH and FH data fragments, takes place. Inconsistencies are avoided by applying consistency conditions originally imposed on the data fragment cached on the MH. The model's adherence to the ACID properties are summarized in Table 2.5:

Table 2.5 Semantics Based Model ACID Properties

	Atomicity	Consistency	Isolation	Durability
Semantics Based	MH: Local commit. DB Server: Merging of data fragments between MH and FH.	Is maintained through constraints conditions that were originally loaded with the object fragments on the MH.	Uses 2PL to control concurrent access to locally cached fragments.	Because if data fragmentation data availability becomes limited when MHs hoard data for extended periods of time.

2.2.6- Pre-Write

Another interesting approach to transaction execution in mobile environments is Pre-Write [14]. The main purpose of this architecture is to improve the availability of data for MHs and FHs, by splitting operations into two categories: pre-operations and normal operations. Transaction execution is divided between the MH and the BS, to allow for higher concurrency and to reduce transaction blocking. To achieve this, the model uses an optimistic concurrency control scheme with no cascading aborts. Through the division of transaction execution, the data becomes available on both the MH and the DBS, even before final commit, which increases data availability. Interaction between the MH and BS is

as follows: the MH first requests locks on data items on the DBS through its transaction manager. Once these locks are successfully acquired by the BS's data manager, the MH is free to disconnect. When the MH finishes executing pre-operations, the final operation that is executed on the MH is a pre-commit operation. Once a transaction is pre-committed, it cannot be aborted. These pre-writes are then sent to the BS and made visible to all subsequent transactions that require that data item. A log of operations is kept by the BS as well. To make them permanent, the BS's data manager transforms pre-commits into normal commits and stores the changes made by the MH. Adherence to the ACID properties can be summarized in Table 3.6:

Table 2.6 Pre-Write Model ACID Properties

	Atomicity	Consistency	Isolation	Durability
Pre-Write	MH: Local commit. DB Server: Locally committed data on the MH are made permanent on the DBS	Is maintained by the separation of operation into pre ops and normal ops	Locally committed operations are viewable by all hosts. Uses relaxed 2PL extended by one conflict table and new lock types	Data is committed permanently after local commit. However involves heavy message exchange.

Other transaction management schemes also include IOT, Kangaroo, MDSTPM, MoFlex and Pre-Serialization. Overviews of these architectures can be found in [11] and [21].

2.3- Epidemic Quorum Algorithms

This section introduces a particular type of distributed mobile transaction propagation and availability model called epidemic quorum algorithms.

Traditional quorum algorithms have been extensively used to insure a certain degree of reliability in distributed database systems by providing checks on data replication and implementing mutual exclusion [16]. However their applicability to mobile networks is not adequate, given the promiscuous nature of connectivity in such environments, which would lead to disconnections in the quorum partition and disruptions of quorum agreements.

Epidemic quorum systems, on the other hand, allow for data propagation along unconnected quorums by initiating a finite number of elections. As the results of these elections are decided, the votes are propagated across the different quorums, leading eventually to a full propagation across all quorums. This is achieved through comparing the MH's local state to the current quorum state and modifying its state to reach one consistent with the quorum's. Epidemic algorithms usually operate similarly to traditional quorum algorithms in so far as agreeing on the propagation of a certain data item through vote results from quorum groups. The following paragraph presents an epidemic algorithm model on whose architecture, studies and quorum selection model this thesis is based on.

In general terms, epidemic algorithms [9] rely on two fundamental conditions to insure that data is properly propagated across all sites. The first condition insures that all local data, to a particular site, are totally ordered and form a one-copy serialization sequence, whereas the second condition states that the global order of inter-site events are causally ordered, so that if event c_1 and c_2 happened at consecutive time t_1 and t_2 where $t_1 < t_2$, then the causal order of events is maintained if $c_1 \rightarrow c_2$.

Epidemic algorithms maintain cross site timestamps using vector clocks (variants of the standard Lamport clocks) to keep causal order across sites.

Epidemic Quorum Algorithms (which will be referred to as eQuorums from hereon), are a particular breed of epidemic algorithms which, like the normal quorum algorithm, operate on the same basis with some particular features that make them fit for distributed environments. eQuorums are used as substitutes to the standard pessimistic epidemic algorithm (ROWA) in environments that require high system throughput, by allowing one transaction to commit from each set of conflicting transactions. As mentioned previously, transactions in eQuorums are serialized in a causal fashion, so that out of each pair of conflicting transactions, one and only one transaction will commit through a *yes, no,* vote by site quorums. Vote results are stored in a log that indicates the local site's time, vote result and the identification number of that site. This log entry is then propagated to other sites through eQuorum messages until all sites have received the vote results. Vote results are sent from all sites to all sites and are propagated according to availability, among other factors. When a particular site receives a positive vote for a particular transaction, it automatically commits the data for that transaction. When a particular transaction commits at a certain site, all other conflicting transactions at that site are aborted.

Transactions in an eQuorum can usually be in three states, two of which are trivial. A group of transactions can be in conflict, or in an uncertain state, whereas one and only one transaction from that group can be in a committed state. A group of transactions are said to be in an uncertain state if a site S_i has not

received enough information to commit or abort a transaction. This usually occurs when the outcome of a vote at that site has yet to be determined. Transactions in an uncertain state usually obtain locks on a certain data item X. Unlike the ROWA algorithm, an item X can have locks on it by more than one transaction that need to access X. However, these write locks have been adapted from the Pre-Write model [14], where the concept of Intention to Write (*IW*) is introduced to mitigate the problem of multiple locks through the addition of an extra conflict table for *IW*. It is also worth mentioning that local transactions, wishing to commit a write operation, are also forced into converting any write request to an *IW*; thus, mimicking a remote transaction. This mechanism prevents local transactions from accessing data items before pre-committing. When two conflicting transactions possess an *IW* lock on a data item, if either one should abort, then the data item remains inaccessible until the fate of the remaining transaction is decided. The performance results presented in [9], apply to a fixed network, and as such do not provide any insight that may be extended to wireless networks.

The property of eQuorums that this thesis is most intent on dealing with, is the availability of quorums at various sites. The goal is to maximize the availability of quorums and to increase the probability of the system, eventually reaching a global consensus, and thus agreeing on a given value. [1] provides a good overview of the performance of eQuorums and provides tangible metrics through which performance can be measured. [1] also presents a unified framework to compare and measure various epidemic quorum algorithms. As mentioned earlier, the main goal of this thesis is to insure that the availability of quorums is maximized, providing improved overall system performance. To better understand the metrics used in [1], it is essential that we clarify

some details on the model presented in the earlier chapter pertaining to the voting process. eQuorums propagate data items based on per site quorum votes. However, given that some sites may be unavailable at times, or may require more information (in the case of uncertain vote outcome), eQuorums perform a finite number of voting rounds; the outcome of which may be a second round of votes (if uncertain) or a decision. The work refers to these two metrics as probabilistic values represented by rep_ε for the repeat probability condition and dec_ε for the decision probability condition.

The probabilistic properties of rep_ε and dec_ε are as follows:

- $rep_\varepsilon(n)+dec_\varepsilon(n)\leqslant 1$
- and $\forall n: rep_\varepsilon(n)<1$

Following the above workings of eQuorum votes and its probabilistic constraints, [1] has defined the availability of an eQuorum by the formula given in (1):

$$\sum_{n=0}^{y} \frac{\binom{y}{n}(1-p_f)^n p_f^{(y-n)} dec_\varepsilon(n)}{1-rep_\varepsilon(n)} \tag{1}$$

Where p_f is the probability of failure of a given host to vote, part of the numerator expression $\binom{y}{n}(1-p_f)^n p_f^{(y-n)} dec_\varepsilon(n)$ represents the probability of having n correct processes out of y, and $\frac{dec_\varepsilon(n)}{1-rep_\varepsilon(n)}$ represents the probability of consecutive repeat votes followed by a decide vote.

Although [1] assumes p_f to be constant and uniform, in

reality, given the volatile nature of wireless networks, the probability of failure cannot be fixed or defined ahead of time, as disconnections may occur randomly and without prior precursors. However, a general behavioral pattern for p_f can be deduced and applied to equation (1). The model presented in chapter 4 discusses this matter in more detail. The goal is to minimize the probability of failure p_f to maximize availability. It is also worth mentioning that as rep_ε and dec_ε grow, availability grows

Although the sensing and incorporation of network states in quorums is a very recent topic of discussion, the most notable work in this area of research has been done by Peysakhov et al. [17] and Gupta et al. [6][7]. The approach brought forth by [17] uses the same general principals, as detailed later. However, instead of using the standard client/server model, [17] uses migrating agents applied to standard quorums. As for the metrics evaluated in [17], they use a probability density function (equations (2) and (3)), similar to equation (1), of positive versus total number of votes, to calculate a confidence factor.

$$\binom{y}{n}(1-p_f)^n p_f^{(y-n)} \tag{2}$$

and

$$F(C)=\sum_{k\epsilon C} f(x=k)\geqslant 0.9 \tag{3}$$

Where C is a pre-selected confidence interval and F(C) denotes the combined probabilities of all the members of C. The process of measurement works by continually collecting votes until such time as the uncertainty threshold (in the above example 0.9 ~ 90%), defined as values lying outside of the interval C, is reached. On a given site these values would tend to indicate that a certainty

for either a positive or negative outcome of the votes has been reached by the quorum. Although [17]'s method enhances the general confidence in a quorum, the presented approach deals only with the quality of the host as measured prior to quorum selection. Furthermore, an agent approach to data collection suffers in weakly connected networks. The measures presented in this work would also help agents find better hosts to collect data from, reducing the amount of failed visits an agent may encounter.

The following chapter presents an idea for adapting quorum host selection in wireless networks, by adding an extra middleware layer, allowing the systems involved to be aware of the state of hosts in a particular site. When quorums are formed, this information would lead to minimized delays and less disconnections that often occur in wireless networks.

Chapter 3- A Quorum Selection Architecture for Wireless Networks

3.1- The Architecture

This chapter explains the architecture and various modules used to achieve better quorum host selection, based on metrics measured by the MH pertaining to its state, and sent periodically to the BS. The agent approach was used to minimize the amount of messages passed from client to server, as well as provide a distributed architecture to circumvent any single point of failure that may arise.

As mentioned in chapter 2.1.1, the base station is the most suited element in a mobile network to keep track of mobile hosts in its current orbit. Base Stations (BS), as adapted from the mobile phone architecture, serve to keep registration information about current mobile hosts in their area of coverage, thus making them strategically placed to carry out the function delineated in this work. The new role alloted to a base station will be to gleam the chosen performance metrics from mobile hosts, currently registered with that base station. As such, we introduce the following metrics and variables to measure the performance, connectivity and health of a particular mobile host.

- Signal Strength: Defines the current signal strength of a MH, currently registered with a BS. This measure is usually in decibels (dB), and is directly measured by the MH's wireless network card (PCMCIA card or Centrino wireless chip), accessible through the card's device driver framework. In general, the signal strength can be

expressed as a percentage of the maximum speed available on a particular wireless network. Typically an 802.11 b/g network would operate at 54 Mbps, and as such, the signal strength is measured as the current speed achieved by the mobile host as a percentage of 54 Mbps (representing 100% signal strength). Typically this value depends on the geographical environment the mobile host is located in, as well as factors such as battery autonomy and I/O conditions on the MH.

- Host Priority: Depending on the current signal strength of the MH, a priority number is given to that MH. The higher the signal strength, the higher the priority. This can be set directly by the MH, or if selected as part of the quorum in a given BS, the BS may assign that MH a priority, based on measurements by the BS. The priority can also be set based on the level of criticality of data currently waiting to be read or written by a particular MH. The more important the data, the higher the priority. This is usually set and agreed upon when the MH first checks out the necessary data items it wishes to work on offline. The DBMS can set priorities on various data slices which, when checked out by an MH, sets its priority metric.

- Host Trend: We also define a derived metric based on the performance of a MH's signal strength with time. The trend of an MH's signal strength is calculated using a standard weighted linear regression, which shows the trend over time of the aggregates of both aforementioned metrics (Signal Strength & Priority). This metric would help amortize discrete signal strength values (whether

high or low) and provide the BS with a better picture of the overall performance of that MH.

- Other Variables: These include various variables such as host identification and a variable that always contains the last BS a particular host was registered with.

Any MH currently registered with a particular BS will have an agent running that sends or receives metrics to and from that BS, keeping that BS up to date on its current status.

The BS on the other hand, once it receives this data from a MH, will classify it in a heap structure that allows the BS to pick the best performing MHs when a quorum is being built. The heap structure on the BS is constantly maintained as a max heap. Every new mobile host addition to the MH, will force the BS to restructure the max heap to take into account this new addition. Figure 4.1 shows a general overview of the proposed architecture and illustrates another use of the data in conjunction with Layer 7 switches. The Layer 7 switch can be used to provide traffic priority to MHs which have been selected to act as quorum members. This is especially important when selected MHs are present in a congested BS. By providing higher traffic priorities to quorum members, the Layer 7 switch insures that quorum traffic gets the highest priority over other traffic when the quorum is casting votes and deciding the outcome of a read or write operation.

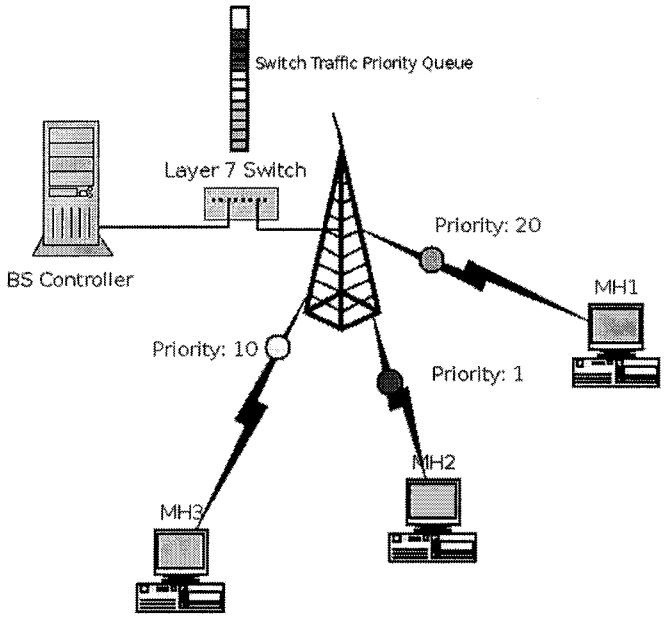


Figure 3.1 IQSA General Architectural Overview

Figure 3.2 provides more insight into the architecture, by showing how packets get arranged in the BS' max heap structure. The max heap keeps a record of all currently registered MHs with it, and orders the list, based on the metrics mentioned prior, from largest (most reliable host) to smallest (least reliable host). This max heap is constantly updated (Every five minute interval) whenever a host is added or removed from the heap, providing the BS with the most up to date state of MHs currently registered with it. The priority metric is used for the comparison. However, during selection, the BS may also use the trend data sent by the MH as an added measure during the selection process. No mechanism for selection has been implemented in the current release, as it serves more as a proof of concept of basic functioning rather than a full fledge application.

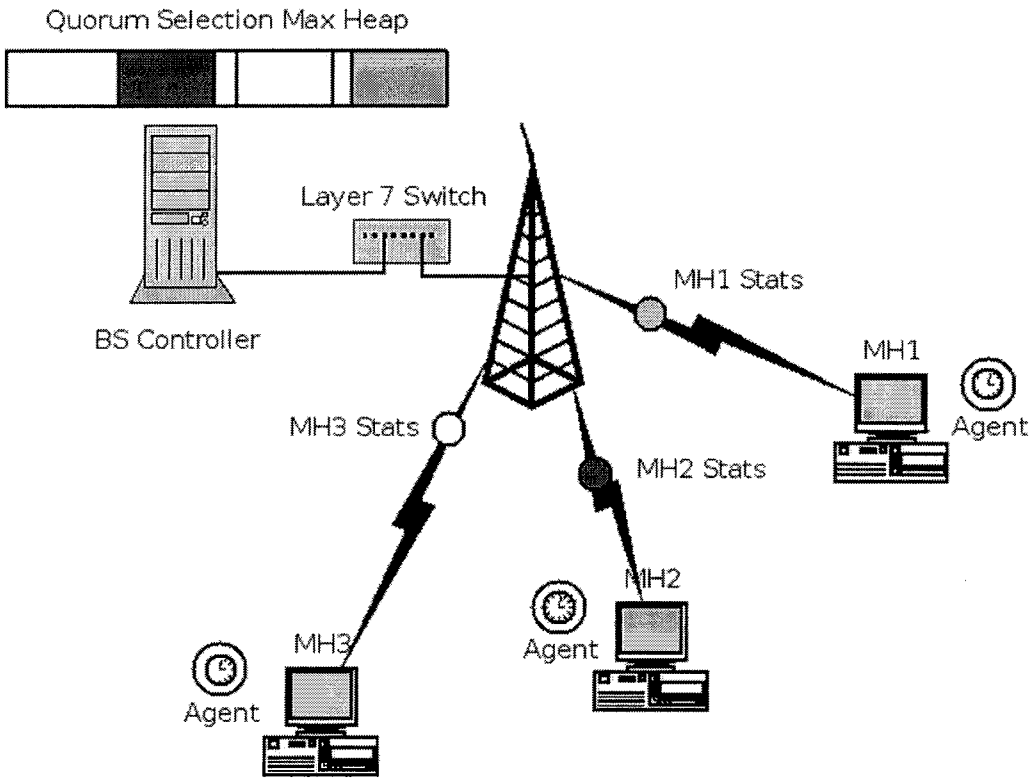


Figure 3.2 IQSA System Architecture: MH Agents - Max Heap Interaction

The architecture also introduces the means to migrate MHs from one BS to another. Since the MH is essentially a mobile unit (cellphone, PDA, Laptop) with a moving user, for example, this user may travel through various BSs while going to work, and as such may travel and register with various base stations (similar again to the mobile phone network). So as not to clutter every BS with stale data, the MH will automatically be unregistered from a BS once that MH leaves its area of coverage. When a user registers with a new BS, the MH will automatically send the new BS the previous BS's address it was registered with. This allows both BSs to communicate with each other and migrate the entry from the previous BS to the new BS. In some cases, mobile users

may swing between two or more BSs, creating heavy migration traffic. In these cases, the MH may opt to remain registered with a particular BS as long as the cell that the user was registered in, is adjacent, in terms of area of coverage, to the the cell he/she's currently in. Figure 3.3 illustrates such a scenario, where MH7 migrates from a BS's coverage of cell C7 to a BS covering cell C6. The entry for MH7 is automatically migrated from C7 to C6 by communication between the BS's of both cells, as soon as the MH provides the new BS with the address of the previous BS. The adjacency scenario can also be inferred from Figure 4.3 as well, where C7 has adjacent cells C6, C5 and C4 through which the BS, if present in any of the three, may use them as a bridges to communicate directly with C7, without re-registering with either C4, C5 or C6.

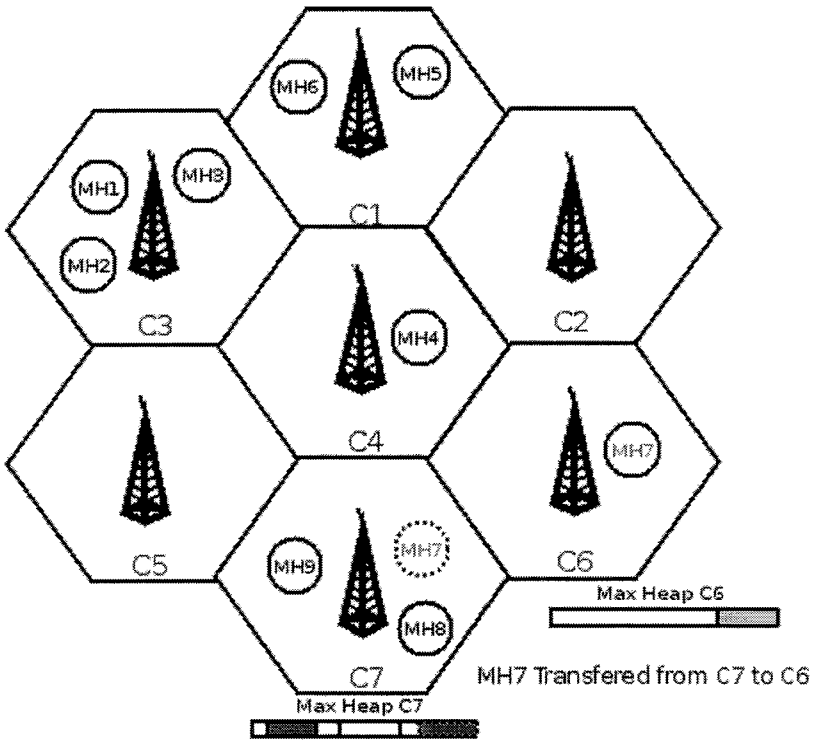


Figure 3.3 IQSA Architecture - Mobile Host Migration Process

3.2- The Algorithm

In this chapter we discuss the high level algorithms that need to be implemented, in order to mimic the architecture discussed in the previous chapter. Although the core parts of it have been implemented in C++, this chapter presents the various modules in a more high level descriptive language. Where necessary, references to actual code will be made. Appendix A contains all the detailed implementation of a proof of concept code that lays the foundations for the above mentioned architecture in its general form. A simulation model of the architecture has been implemented using the NS2 network simulator package to test the performance of the algorithm. We leave the details of performance and evaluation to the next chapter.

A general sectioning of the various modules needed will be presented, as well as a detailed description and pseudo-code for selected parts. Figure 3.4 shows the various high level parts and their high level interaction with one another.

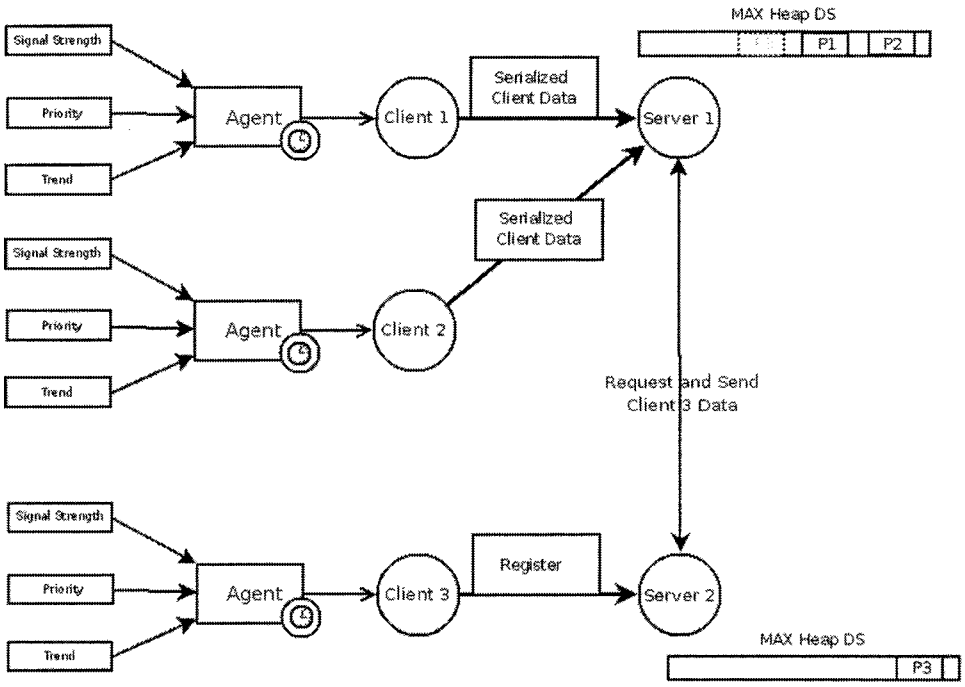


Figure 3.4 IQSA High Level Modules Interaction

As mentioned earlier on, quorum host selection is based on client/server interaction. The agent modules running on the various MHs are responsible for collecting metrics (signal strength, priority, etc.). The client module is responsible for serializing the data and sending it over the wire to the server. When a server receives a packet from a client, it stores and classifies the client data into a max heap structure, based on the priority metric measured at the client end. The last module involves the inter-server data exchange system, which allows servers to exchange information about clients when they migrate from one server to the other and complete registration.

The next chapter present the pseudo-code of the various modules with an in-depth discussion of each one.

3.2.1- Client Agent Procedures

This particular piece of pseudo-code show how an agent collects and measures the various metrics involved before sending them to the client module for transfer across the wire to the server. The snippet below in Figure 3.5 shows the record structure of data gathered by the client.

```
    type: HostData
          record
                SignalStrength          :     int
                Hostname                :     string
                PreviousBS              :     string
                Priority                :     float
                Trend                   :     double
                Time                    :     vector of
double
                PrioSig                 :     vector of
double
```

Figure 3.5 Host Data Record Structure.

Of the aforementioned variables, most should be familiar

from previous chapters, with the notable exceptions of the *Time* vector structure and the `PrioSig` vector structure. The `Time` structure, as the name indicates, holds a timestamp for every measure of both the `SignalStrength` and `Priority` performed by the agent, at specific intervals metrics. Every time these two metrics are measured, a timestamp is added to the dynamic array. This also holds true for the first time the structure is being initialized as well. The `PrioSig` structure on the other hand, holds an aggregate weighted measure of both the `Priority` metric and `SignalStrength` metric. These two vector structures are used to store historical data, needed by the linear regression function, which measures the host's trend metric.

```
    calculateTrend(PrioSig, Time) :
    begin:
          if { sizeof(PrioSig) ≥ 2 } then
             Linear A =
CalculateLinearReg(sizeof(Time), PrioSig,
                          Time);
                return getSlopeOf(A);
          else
                return 0;
    end;

    calculateWeightedAggregate(SignalStrength,
Priority) :
    begin:
          signalWeight ← signalImportanceFactor;
          priorityWeight ← priorityImportanceFactor;
          return
(SignalStrength*signalWeight)+(Priority*priorityWeight)
;
    end;

    Initialize(HostData) :
    begin:
          SignalStrength = measured(SignalStrength);
          Hostname ← gethostbyname();
          Priority ← measured(Priority);
          Time[] ← push { time(now) };
          PrioSig[] ← push {
weightedLinearDist(SignalStrength,
                              Priority) };
          trend ← calculateTrend(PrioSig, Time);
    end;
```

```
updateMetrics(SignalStrength, Priority) :
begin:
      Time[] ← push { time(now) };
      PrioSig[] ← push {
weightedLinearDist(SignalStrength,
                              Priority) };
end;

updateTrend() :
begin:
      trend ← calculateTrend(PrioSig, Time);
end;
```

Figure 3.6 Host Metrics Initialization and Update Procedures

The calling of the procedures in Figure 3.6 takes place as follows: when a MH first connects, it initializes all its metrics with either the most current measure taken (signal strength and priority), or to default values (trend metric). The trend metric is initially set to zero due to the unavailability of data to perform regression calculation. A minimum of two datum points are needed to perform a linear approximation, hence the greater than or equal to 2 condition in the *calculateTrend* procedure in Figure 3.6. Calculations of a weighted aggregate value combining both signal strength and priority metrics is also performed. The aggregation of these two metrics and its subsequent use in the linear approximation, would allow the environment to control which metric should have a higher importance, by assigning it a variable weight factor. The ensuing aggregate value is then associated with a timestamp in order to calculate the trend metric over time. This calculation is performed every time the client agent gets an updated measure on the basic metrics: signal strength and priority. The updates for the signal strength, priority metrics, and trend metric have been split into two procedures to make allow greater control over when and in which order these two procedures get called. Once all the data has been updated, the client serializes the

host data and sends it over the wire to the BS.

3.2.2- Server Side Procedures

On the receiving end, is the base station, running a server listening for registration and update requests sent by MH agents. The BS is responsible for maintaining an organized max heap structure of all currently registered MHs that fall under its zone of coverage. Once a registration or update packet is received, the following procedures take place to insure the careful addition/update of that MH's information into the BS's heap structure.

Step1: The BS would first check if the record for that MH is currently registered with it as a BS, if not the BS will retrieve that data from the last BS that MH was registered with and ask the previous BS to delete the entry of that MH in its heap. The BS will then send the MH its credentials to update its BS information and proceed to add the MH's record to its max heap structure.

Step2: If the MH's record already exists then the BS will consider the incoming data as an update and proceed to search and update its heap structure with the new data sent by the MH agent.

Step 3: The last case the server considers is when no previous BS record has been sent by the MH and no data for that MH has been found in the server's heap structure. In this case the server will assume that this is a first time sign on by a new client and proceed as outlined in step 1.

Following, is the pseudocode of the above mentioned steps and their associated procedures, most notable of which are the heap building and sorting procedures. Every procedure will be discussed in detail on its own, followed by a sequencing of these

procedures as described in the previous paragraph.

```
    maxHeapify ( heapStruct[] , i ) :
    begin:
          l ← left(i);
          r ← right(i);
          if ( l ≤ length[ heapStruct ] and
l.HostData.Priority >
    heapStruct[i].HostData.Priority  then
          largest ← l;
          else largest ← r;
          if ( r ≤ length[ heapStruct ] and
r.HostData.Priority >
    heapStruct[i].HostData.Priority ) then
          largest ← r;
          if ( largest ≠ i ) then
                exchange heapStruct[i] ↔
heapStruct[largest]
                     maxHeapify ( heapStruct [], i );
    end;
```

Figure 3.7 The maxHeapify Procedure.

```
    buildHeap ( heapStruct[] ) :
    begin:
          for i ← ⌊length[ heapStruct ] / 2⌋ downto 1
                do maxHeapify( heapStruct [] , i );
    end;
```

Figure 3.8 The buildHeap Procedure.

```
    sortHeap ( heapStruct[], length [ heapStruct ] ) :
    begin:
          buildHeap( heapStruct[] );
          for i ← length [ heapStruct ] downto 2
                do exchange heapStruct[12]
heapStruct[i];
                length [ heapStruct ] ← length
[ heapStruct − 1 ];
                maxHeapify( heapStruct[] , 1);
    end;
```

Figure 3.9 The sortHeap Procedure.

```
    searchForElement( heapStruct[] , Element) :
    begin:
          foreach Element in heapStruct[] as Temp
                if ( Element.HostID eq Temp.HostID )
then
                      return position of Element;
                else
                      return -1;
    end;
```

Figure 3.10 The Iterative Search Procedure.

```
    migrateRecord( Element.serverID, Element.HostID ) :
    begin:
          buff [] ← send(Element.serverID,
Element.HostID,
"Migrate");
          /* The above procedure, sends out a request
to the last BS          Element was registered with,
retrieves the data and asks                the previous
BS to delete its records of BS */

          new ( Element ) ← unserialize ( buff[] );
          heapStruct[] ← push ( Element );
          buildHeap ( heapStruct[] );
          sortHeap ( heapStruct[], length[ heapStruct ]
);
    end;
```

Figure 3.11 The Migrate Procedure.

```
    updateElement( heapStruct[], Element, Position ) :
    begin:
          if ( isdefined(Position) and Position ≠ -1 )
then
                heapStruct[ Position ] = Element;
                buildHeap ( heapStruct [] );
                sortHeap ( heapStruct[], length[
heapStruct ] );
    end;
```

Figure 3.12 The Update Procedure.

```
    insertMaxHeap( heapStruct[], HostData Element )  :
    begin:
          if( Element.serverID neq serverID ) then
                migrateRecord ( Element.serverID,
Element.HostID );
          else
                if ( position ← searchForElement
( heapStruct[],
                                                  Element ) ≠
-1 ) then
                      updateElement ( heapStruct[],
Element,

    position  );
                else
                  heapStruct[] ← push ( Element );
                  buildHeap ( heapStruct [] );
                  sortHeap ( heapStruct[], length[
heapStruct ] );
    end;
```

Figure 3.13 The Insert Procedure.

```
    selectQuorumHosts ( heapStruct[], quorumSize ) :
    begin:
          if( length[ heapStruct ]  0 ) then
          start mutex
          for i ← length[ heapStruct ] downto (
length[ heapStruct ]
                      - quorumSize )
                      addToQuorum( heapStruct [i] );
          end mutex
    end;
```

Figure 3.14 The Quorum Host Selection Procedure.

Explanations of the pseudocode, presented in Figures 3.7 through 3.14, is to follow.

Figures 3.7 – 3.9 represent the heap creation, manipulation and sorting procedures necessary to maintain the server side heap data structure. This structure holds a HostData record sent by

registered MHs, and keeps them sorted based on the Priority metric. Figure 3.10 describes the record search procedure, which identifies the position of a MH record, if found, and returns it to the calling procedure. Figure 3.11 describes the procedure that takes care of migrating MH records from servers they were previously registered with. Figure 3.12 describes the record updating procedure, which relies on the pseudocode introduced in Figure 3.10. Last, Figure 3.13 describes how these procedures get called whenever a record is received form an MH and is about to be inserted into the max heap. The pseudocode if Figure 3.13 mimics the three checking steps outlined earlier in this chapter.

The following chapter cover the performance of the presented algorithm, both from a theoretical perspective as well as from a simulated perspective, giving more insight into the advantages and disadvantages of this approach.

Chapter 4- Performance Evaluation and Simulation Results

This chapter presents the results obtained through, both theoretical and simulated measurements, using mathematical tools and the NS2 network simulator [5][22]. As such, this chapter is divided into two parts, each presenting one of the aforementioned aspects of measurements as well as a critical view of the obtained results.

4.1- Theoretical Evaluation and Proofs

This chapter covers the mathematical evaluation of both the algorithm runtime and the expected performance gains, based on equation (1), presented in chapter 2.3.

The algorithm performance is measured in the standard Big O notation. Every essential part of the algorithm will be evaluated individually, and an aggregate performance measure will be derived from the parts. We first identify the following areas of the code that affect the performance of the algorithm, as presented in Figures 3.7, 3.8, 3.9 and 3.10. An aggregation of these results, as well as added complexity will be measured, relating to figures 3.11, 3.12 and 3.13. We also distinguish between the preprocessing stage, which involves building and maintaining the heap structure, and the quorum selection process, which is a bounded function that depends on the size of the expected quorum. The amount of messages passed between the MH agent and the server are measured to insure that the least amount of needed messages are passed and to minimize network congestion problems that adversely affect the overall performance of the

quorum process.

The performance of the *buildHeap* [fig. 3.8] and *sortHeap* [fig. 3.9] are well-known algorithms, whose performance measures have been extensively detailed, so no formal proof will be given. Instead, only the final end result will be presented. The *buildHeap* function calls on *maxHeapify*, and the combined performance of these two procedures is: $O(n.log(n))$, with an identical performance for the *sortHeap* [fig. 3.9] algorithm. The *searchForElement* procedure [fig. 3.10] has linear performance $O(n)$. Although better search performance can be achieved using more efficient algorithms, it is not the focus of this work to tackle this issue, and is left as a future improvement. However, it is worth mentioning that n is bounded by the limited amount of hosts a BS can handle. As such we can represent the search procedure as $O(max(MH))$ for a particular BS.

Taking the above performance measures into account, the following paragraph evaluates the performance of the main procedures that maintain and update the list of MHs currently registered with a particular BS.

Since this particular procedure involves code execution on two sites, a breakdown of the code execution on every site is measured, whereas network delays and other external factors have been ignored. The first step of the migration involves sending a request to a remote server, where a search and delete procedure is executed. The search procedure's performance is $O(n)$, with $O(1)$ performance for deletion once that record is found. When a record is received by the current server, the MH attempts to register using the combined insert and sort operations of the max heap structure. This adds a performance execution overhead of

$O(n^2.\log^2(n))$, for a total combined worst case performance hit of $O(n^3.\log^2(n))$, which executes in polynomial time. Identical performance can be expected of the *insertMaxHeap* procedure in the case of an element update. Generalizing this to m sites, would yield a worst case global performance described in Expression (1).

$$\sum_{k=0}^{m} O(n^3.\log^2(n)) \tag{1}$$

Ordinarily an insert operation on a heap data structure should be of the order $O(\log(n))$. However, due to the choice of the key (the Priority variable) chosen to sort the heap on, finding a host would require linear time instead, based on the MH's identification string. This is one shortcoming that can be remedied in subsequent development of this architecture, and as mentioned prior, it is not the focus of this work to tackle all possible optimization aspects of the algorithm, but rather, to show that it is a viable architecture that can provide improved availability and reliability to epidemic quorum groups.

By selecting the best performing hosts from the max heap data structure to participate in an epidemic quorum, the BS insures that the probability of a process failure on the selected MH is kept to the minimum possible, based on the general state of the network. As such, going back to Eq. (1), we propose to see the effects of minimizing the process failure factor p_f on the overall availability of the epidemic quorum. Earlier, [1] sets p_f as a fixed value and measures availability based on the probabilities for vote repetitions and vote decisions. Whereas, the proposed mathematical model tackled here, measures how a variable p_f affects the overall availability for discrete values of *dec* and *rep*.

The following tables and figures explain the mathematical results obtained, based on the mathematical framework presented in [1] for epidemic quorum availability measurement. The results also indicate that a minimum threshold should be respected when selecting MHs for quorums, below which, availability may suffer. This threshold would allow the MH selection procedure to set a cutoff point below which hosts would not be selected.

Table 4.1 Availability Chart with dec=1 and rep=0

Dec	Rep
1	0
Pf	Sum(Pf)
0	1.0000000000
0.1	0.9999999999
0.2	0.9999998976
0.3	0.9999940951
0.4	0.9998951424
0.5	0.9990234375
0.6	0.9939533824
0.7	0.9717524751
0.8	0.8926258176
0.9	0.6513215599
1	0.0000000000

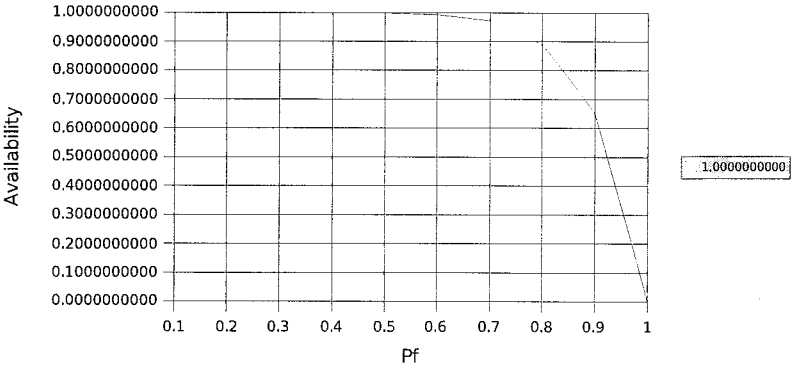


Table 4.2 Availability Chart with dec=0.6 and rep=0

Dec	Rep
0.6	0
Pf	Sum(Pf)
0	0.6000000000
0.1	0.5999999999
0.2	0.5999999386
0.3	0.5999964571
0.4	0.5999370854
0.5	0.5994140625
0.6	0.5963720294
0.7	0.5830514851
0.8	0.5355754906
0.9	0.3907929359
1	0.0000000000

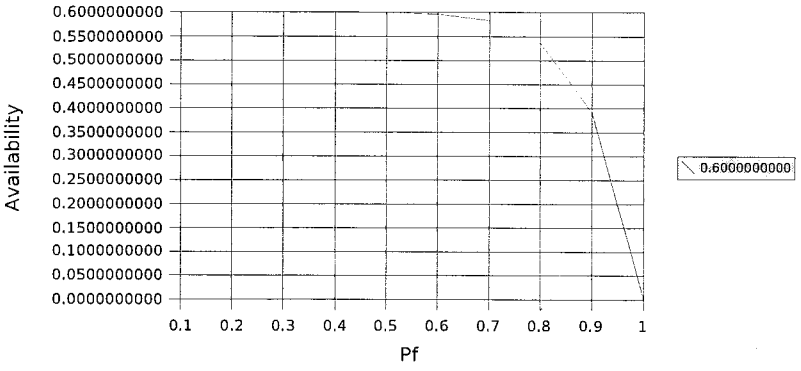
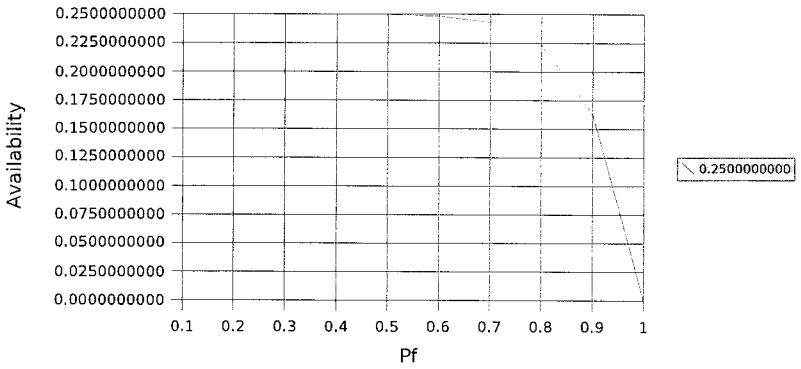


Table 4.3 Availability Chart with dec=0.25 and rep=0

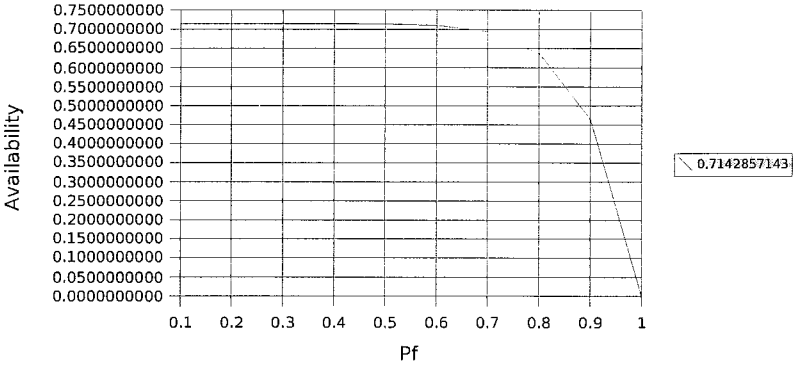
Dec	Rep
0.25	0
Pf	Sum(Pf)
0	0.2500000000
0.1	0.2500000000
0.2	0.2499999744
0.3	0.2499985238
0.4	0.2499737856
0.5	0.2497558594
0.6	0.2484883456
0.7	0.2429381188
0.8	0.2231564544
0.9	0.1628303900
1	0.0000000000



The results clearly show that based on the mathematical model presented in [1], the combination of a variable p_f, decision probability and repetition, although mainly affected by the probability of a quorum reaching a decision, clearly diminishes as the p_f factor increases. Selecting MHs with low probabilities of failures would ultimately lead to a far more stable and available quorum system. The probability for reaching a quorum decision on the other hand, is not directly related to p_f, but an indirect relation with the previous two factors may be inferred. Lesser available systems will most likely delay the outcome of the vote, if systems go off-line, forcing an increase in repetition cycles. Although the model does indicate that high availability is achieved according to the lemmas in chapter 2.3, the convergence time to a consensus will increase with the amount of repetitions, leading to higher delays in up-to-date data acquisition. Table 4.4 shows the variance of p_f with high probability of repetition.

Table 4.4 Availability Chart with dec=0.25 and rep=0.65

Dec	Rep
0.25	0.65
Pf	Sum(Pf)
0	0.7142857143
0.1	0.7142857142
0.2	0.7142856411
0.3	0.7142814965
0.4	0.7142108160
0.5	0.7135881696
0.6	0.7099667017
0.7	0.6941089108
0.8	0.6375898697
0.9	0.4652296856
1	0.0000000000



Trivially, with higher vote repetition, in the worst case scenario, the time t_{vote} it would take to create a quorum and reach a result can be expressed by equation (4):

$$t_{vote} = t_{create} + \sum_{n=0}^{max(rep)} t_{repetition} \tag{4}$$

Where $t_{repetition}$ would vary with each repeat round, depending, among other factors, on network conditions as well. The repetition factor is not only calculated based on process failure, but also assumes the ability of a quorum to reach a decision based on the amount of information available to that quorum. The model in this thesis deals with the process failures due to network outages, rather than quorum failures, and as such, the repetition time and vote time factors expressed in Eq. (4) can be minimized by selecting highly reliable hosts, thus reducing both t_{create}, in the initial creation of the quorum, and t_{repeat} when subsequent quorum votes need to be carried.

Theoretically, the model provides insight into the availability of epidemic quorums, but given the difficulty to model real world

network failures due to its mathematical complexity, only live system tests can verify the viability of such a model. The author, is confident that by applying the presented theoretical model, this will unequivocally have a positive result on the availability of epidemic quorums. Since no live model was made available at the time of writing, a simulation was constructed to portray the availability of an epidemic quorum, by providing MH performance results with a discrete MH failure model. The next chapter explains in more details the simulation environment used, as well as the assumptions made and the results obtained.

4.2- Simulation Results

Below is an account of the simulation environment and the obtained results of a discrete MH failure model. The popular NS2 network simulation package [22] [6] was used to model the network environment, in which these experiments were conducted. The model comprises of one or two fixed node hosts (depending on the measurements taken) representing the DBMS server, a base station bridging the wireless network to the fixed network and mobile nodes moving within a BS's zone of coverage. While transmitting, an MH is made to fail at particular discrete times, modeling MHs with low reliability factors, whereas other MHs are attributed high reliability factors. Bandwidth usage is graphed to show the effect of discrete random MH failure on data transmission and how, reliable MH selection, would help reduce the problem of intermittent or permanent MH process failure in a quorum. Traffic priority modeling is also simulated to view the effects of network congestion on mobile host traffic.

An explanation of the various parameters selected for the

wireless network environment model, will be elaborated upon as needed. The model assumes the use of omni antennas that broadcast in all directions. A simple Two-Ray Ground propagation model is also assumed, which allows signals to, not only propagate in line of sight, but also to be reflected off surfaces depending on the geographical surroundings of a mobile host. The DropTail priority Queue was selected as the interface's queuing protocol, which, assigned to a host's interface, allows the host to drop packets once a maximum traffic threshold is exceeded, simulating a congestion scenario.

The first simulation (See Appendix B) scenario comprises of two mobile nodes, a base station and two virtual fixed hosts that simulate one DBMS server. To measure bandwidth, the simulator requires two distinct interfaces to calculate it for separate hosts. Figure 4.1 depicts the various simulation elements and their interactions.

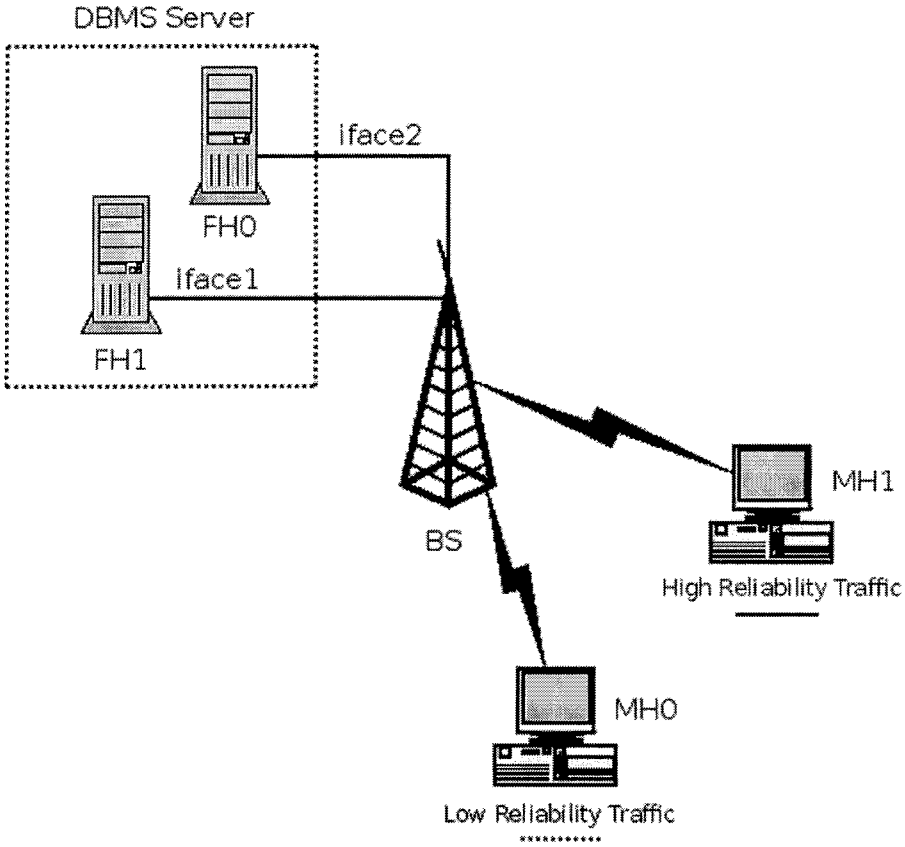


Figure 4.1 Simulation Scenario 1

The purpose of this simple setup, is to show a comparison between the traffic pattern generated by a reliable host, versus traffic patterns generated by a lesser reliable host likely to exhibit discrete process failure. A graph generated by xgraph, with labels *out.0* and *out.1* referring respectively to, traffic from MH0 to virtual FH0 and from MH1 to FH1. This graph is presented in Figure 4.2 and shows how reliability can be a crucial factor in sustaining communication with the least amount of lag time and/or downtime. Both nodes are configured identically to generate traffic of size 500 bytes with 2 second burst rates, less than 1 second idle times and sustained rates of 600k.

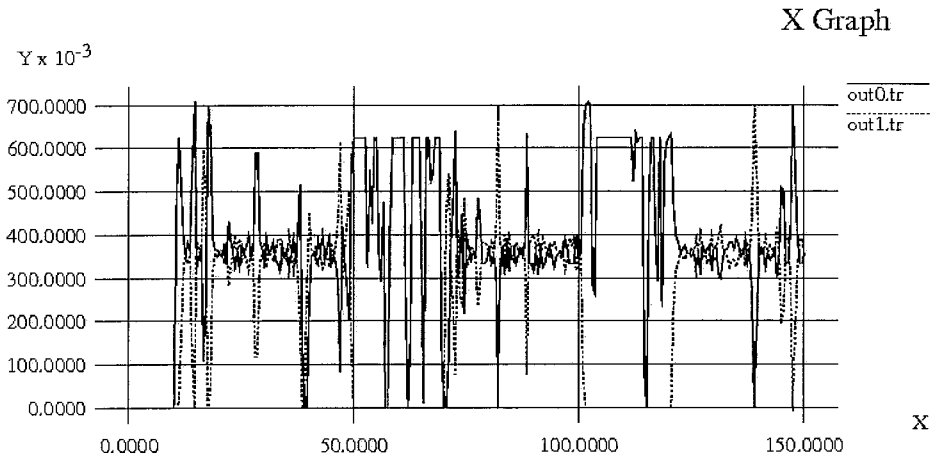


Figure 4.2 Traffic Pattern Graph from the 2 Nodes Simulation

Host MH1, represented by the dotted line *out1.tr*, exhibits the discrete communication failure with FH1 at times 50 and 100, and recovery at times 70 and 120. Whereas, MH0 represented by the solid line *out0.tr*, shows a sustained communication round with FH0. Such discrete failures by an MH participating in an epidemic quorum, may lead to longer quorum formation times, higher rates of quorum vote repetition or, in the worst case, to quorum consensus failure pending a subsequent voting round. In all three cases, the selection of highly reliable hosts (as opposed to random selection) would invariably lead to a more healthy quorum and increase the chances of consensus and, by it, data propagation and data availability.

The second simulation, uses the same basic environment parameters mentioned prior with identical fixed host and base station setups, but involves up to 7 mobile nodes associated with BS0. The setup serves to study the effects of network congestion on quorum traffic, as well as the effects of increased traffic priority for quorum members. In this case as well, the model assumes one

physical server with two virtual interfaces. The layout of this scenario is depicted in Figure 4.3.

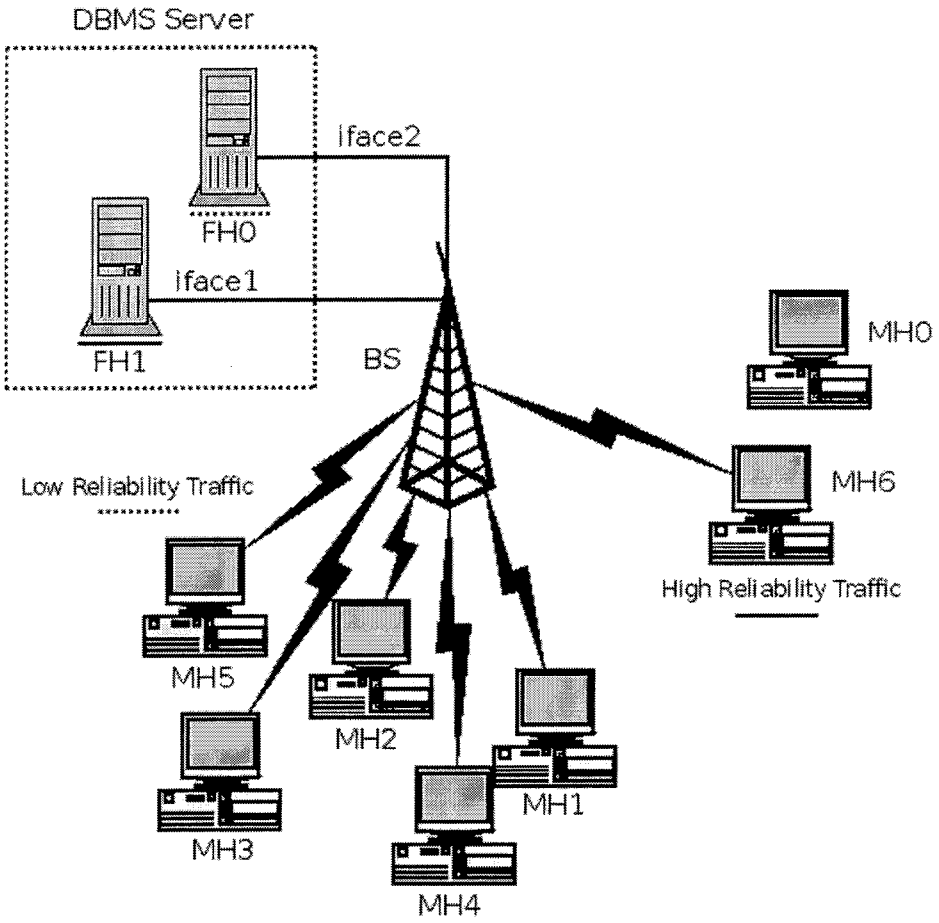


Figure 4.3 Simulation Scenario 2

Out of the 7 MHs, MH0 and MH6 were chosen to communicate with FH0 at speeds of 16Mbps (combined data rate of 32Mbps) with minimal idle time and large data payloads, whereas the rest of the nodes communicate with FH1 at a combined rate of 50Mbps and variable idle times. The graph in Figure 4.4 depicts traffic patterns between MH0 and MH6 with FH0 on one hand, marked as *out1.tr*, and the rest of nodes with FH1 marked as *out0.tr*.

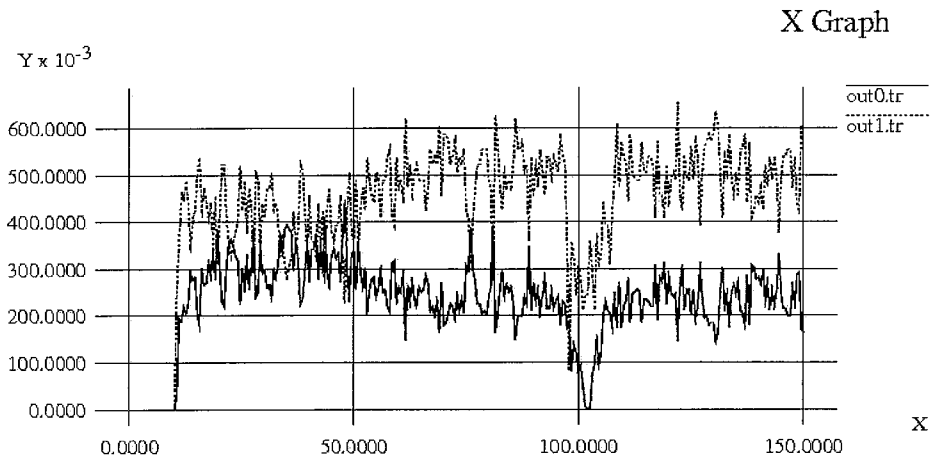


Figure 4.4 Traffic Pattern Graph from the 7 Nodes Simulation

This scenario shows that, by selecting reliable hosts for quorum priority traffic, the sustained rate of data transfer between client and server has a higher bandwidth throughput, leading to increased quorum reliability and availability.

Both simulation scenarios show that MHs with more reliable traffic patterns and less random disconnections, sustain better traffic and higher bandwidth than MHs with less reliable traffic. This observation supports the architecture, presented earlier, categorizing MHs for quorum selection, based on pertinent metrics. From the graphs in figures 4.2 and 4.4 we can clearly infer how MHs with high failure probabilities (p_f factor) affect the overall performance and availability of a epidemic quorum, by analyzing the generated traffic patterns. The results obtained also corroborate the theoretical model in chapter 4.1, as to the effects of p_f on the general availability of an epidemic quorum.

Chapter 5- Conclusion

This work has presented an overview of the various mobile database models currently available, focusing mainly on the epidemic model, and more precisely on epidemic quorums. A novel approach to epidemic quorum selection based on an effort to minimize network disconnections, often experienced by wireless mobile hosts, was presented. The purpose of which, is to provide a more reliable quorum, with higher data availability. The classification of hosts according to measured and derived metrics, allows the model to be extended and incorporate other metrics which may be deemed important in later revisions, such as: database connection counts and performed transactions counts, to further refine the classification of mobile hosts. Although this primary study of the architecture was deemed satisfactory by its author, more work is needed to further refine the mathematical model and incorporate more complex network failure models that may further indicate the usefulness of this model. Work done in [6] and [7] can also be incorporated in the model to provide better quorum placement, minimizing network congestion and delays, instead of relying solely on traffic priority settings.

One of the main points to focus on in future studies on this topic, would include, building historical track record of mobile hosts based on more elaborate regression models (such as a Bayesian regression model), rather than the simplistic linear model used herein. This would require that real performance data be made available to the system in order to allow the Bayes engine's learning process to evaluate its current state, based on measurements that reflect the reality of the system. The author also recognizes that much improvement should also be done on the

algorithm itself, allowing for much better performance, especially in the area of host lookup, where a hash lookup table may be constructed in order to minimize lookup times. All in all the studies and results provided herein show how important a role the network environment plays in the performance of quorums in general, and epidemic quorums in particular. Although this is still a very recent area of interest and study in mobility, it is however receiving just attention, especially in the wake of the wireless revolution, where reliable connectivity is considered whimsical in comparison to its wired counterparts.

REFERENCES

[1] Baretto, J., & Ferrero, P., (February 2007).The Availability and performance of epidemic quorum algorithms. *INESC-ID Technical Report 10/2007.*

[2] Bernard, G., Ben Othman, J., Bouganim, L., et al. (June 2004).Mobile databases: a selection of open issues and research directions. *ACM SIGMOD record,* 33(2) 78-83.

[3] Cormen,T., Leiserson,C.E., Rivest,R.,L., & Stein, C., (2001).Introduction to Algorithms Second Edition. Massachusets: MIT Press.

[4] Golovin, D., Gupta, A., Maggs, B.M., Oprea, & F., Reiter, M.K., (July 2005). Quorum Placement in Networks: Minimizing Network Congestion. *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing (PODC'05), Denver, Colorado, USA,* 16–25.

[5] Gray, J., Helland, P., O'Neil, P., & Shasha,D., (June 1996).The dangers of replication and a solution. *ACM SIGMOD Conference*, Montreal, Canada.

[6] Greis, M ns Tutorial. Retrieved from Marc Greis' Tutorial for the UCB/LBNL/VINT Network Simulator "ns" Web site: http://www.isi.edu/nsnam/ns/tutorial/index.html

[7] Gupta, A., Maggs, B.M., Oprea, F., & Reiter, M.K., (July 2005). Quorum Placement in Networks to Minimize Access Delays. *Proceedings of the twenty-fourth annual ACM*

symposium on Principles of distributed computing (PODC'05), Las Vegas, NV, USA, 87–96.

[8] Holliday, J., Agrawal, D., & El Abbadi, A., (July 2002).Disconnection Modes for Mobile Databases. *Wireless Networks,* 8(4) 391-402.

[9] Holliday, J., Steinke, R., Agrawal, D., & El Abbadi, A., (September 2003).Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering,* 15(5), 1218-1238.

[10] Kafri, N., & Janeček, J., (2002).Dynamic Behaviour of the Distributed Tree Quorum Algorithm. *Proceedings of the twenty-second international conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria,* 517-524.

[11] Kumar, V., (2006). Mobile Database Systems. New Jersey: J. Wiley & Sons Inc.

[12] Kuruppilai, R., Dontamsetti, M., & J. Cosentino, F. (1997). Wireless PCS. New York: McGraw-Hill.

[13] Lee, M., & Helal, S., (2002).HiCoMo: High Commit Mobile Transactions. *Kluwer Academic Publishers Distributed and Parallel Databases (DADPD),* 11, 1.

[14] Madria, S.K., & Bhargava, B., (2001).A transaction model for improving data availability in mobile computing. *Kluwer Academic Publishers Distributed and Parallel Databases (DAPD),* 10, 2.

[15] Mouly, M., & Pautet M.-B. (1992) The GSM System for Mobile Communications. France: Cell & Sys Publications.

[16] Peleg, D., & Wool, A., (1995).The availability of quorum systems. *Information and Computation,* 123(2), 210-223.

[17] Peysakhov, M., Dugan, C., Modi, P.J., & Regli, W., (May 2006). Quorum Sensing on Mobile Ad-Hoc Networks. *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems (AAMAS'06), Hakodate, Japan,* 1104-1106.

[18] Pitoura, E., & Bhargava, B., (1995).Maintaining Consistency of Data in Mobile Distributed Environments. *Proceedings of 15th International Conference on Distributed Computing Systems.*

[19] Pitoura, E., & Bhargava, B., (1999).Data consistency in intermittently connected distributed systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE),* 11, 6.

[20] Ross, K., A., & Wright, C., R., B., (1999). Discrete Mathematics Fourth Edition. New Jersey: Prentice Hall.

[21] Serrano-Alvarado, P., et al., (2004).A Survey of Mobile Transactions. *Distributed and Parallel Databases.* 16, 193-230.

[22] VINT GROUP, from the Manual (formerly Notes and Documentation) Web site: http://www.isi.edu/nsnam/ns/doc/everything.html

[23] Walborn, G.D., & Chrysanthis, P.K., (Sept. 1995).Supporting semantics-based transaction processing in mobile database applications. *Symposium on Reliable Distributed Computing Systems (SRDS)*, Bad Neuenahr, Germany.

[24] Walborn, G.D., & Chrysanthis, P.K., (March 1997).PRO-MOTION: Management of mobile transactions. *ACM Symp. on Applied Computing*, San Jose, USA.

[25] Walborn, G.D., & Chrysanthis, P.K., (Feb. 1999).Transaction processing in PRO-MOTION. *ACM Symp. on Applied Computing*, San Jose, USA.

APPENDIX A: Source Code

Class Name: *HostData*

File Name: *hostdata.h*

```
1.  #ifndef HOSTDATA_H
2.  #define HOSTDATA_H
3.
4.  #include <string>
5.  #include <vector>
6.
7.  using std::string;
8.  using std::vector;
9.
10. class HostData {
11.
12. public:
13.     HostData();
14.     HostData(int, double, string, float);
15.     string replaceSpace(string&);
16.     string restoreSpace(string&);
17.     const char * toCString(const string&);
18.     string toString(HostData&);
19.     HostData toData(const string&);
20.     void printHostData() const;
21.     const float getTrend();
22.     const float getPriority();
23.     const string getHostname();
24.     double calculateTrend(vector<float>&,
    vector<double>&);
25.     void setPrioritySignal(float, int);
26.     void setTrend();
27.     float weightedLinearDist(int, float);
28.
29. private:
30.     int sigStrength;
31.     double trend;
```

```
32.      string hostname;
33.      float priority;
34.      vector<float> priosig_vec;
35.      vector<double> time_vec;
36.  };
37.
38.  #endif
```

File Name: *hostdata.cpp*

```
1.   #include <iostream>
2.   #include <string>
3.   #include <iomanip>
4.   #include <sstream>
5.   #include <vector>
6.   #include <sys/time.h>
7.
8.   using std::cout;
9.   using std::endl;
10.  using std::string;
11.  using std::ostringstream;
12.  using std::istringstream;
13.  using std::setw;
14.  using std::vector;
15.  using std::time;
16.
17.  #include "hostdata.h"
18.  #include "linear.h"
19.
20.  HostData::HostData() {
21.      sigStrength = 0;
22.      trend = 0.0;
23.      hostname = "";
24.      priority = 0.0;
25.  }
26.
27.  double HostData::calculateTrend(vector<float>&
     ipriosig_vec, vector<double>& itime_vec) {
```

```
28.      if(ipriosig_vec.size() >= 2) {
29.            Maths::Regression::Linear
     A(itime_vec.size(), itime_vec, ipriosig_vec);
30.            return A.getSlope();
31.      }
32.      else {
33.            return 0;
34.      }
35.  }
36.
37.  float HostData::weightedLinearDist(int i_sigStrength,
     float i_priority) {
38.      float signalWeight = 10.00;
39.      float priorityWeight = 3.00;
40.      return (i_sigStrength * signalWeight) +
     (i_priority * priorityWeight);
41.  }
42.
43.  HostData::HostData(int i_sigStrength, double i_trend,
     string i_hostname, float i_priority) {
44.      sigStrength = i_sigStrength;
45.      hostname = i_hostname;
46.      priority = i_priority;
47.      time_vec.push_back((double) time(0));
48.
     priosig_vec.push_back(weightedLinearDist(i_sigStrengt
     h, i_priority));
49.      trend = calculateTrend(priosig_vec, time_vec);
50.  }
51.
52.  string HostData::replaceSpace(string& str) {
53.      int x = str.find(" ");
54.      while(x < string::npos) {
55.            str.replace(x, 1, "_");
56.            x = str.find(" ", x+1);
57.      }
58.      return str;
59.  }
60.
61.  string HostData::restoreSpace(string& str) {
```

```
62.          int x = str.find("_");
63.          while(x < string::npos) {
64.                  str.replace(x, 1, " ");
65.                  x = str.find("_", x+1);
66.          }
67.          return str;
68.  }
69.
70.  const char * HostData::toCString(const string& str) {
71.      return str.c_str();
72.  }
73.
74.  string HostData::toString(HostData &record) {
75.      ostringstream outputStr;
76.      outputStr << record.sigStrength << " " <<
     record.trend << " " << replaceSpace(record.hostname)
     << " " << record.priority << endl;
77.      return outputStr.str();
78.  }
79.
80.  HostData HostData::toData(const string &str) {
81.      istringstream inputStr(str);
82.      HostData tmpRec;
83.      inputStr >> tmpRec.sigStrength >> tmpRec.trend >>
     tmpRec.hostname >> tmpRec.priority;
84.      tmpRec.hostname = restoreSpace(tmpRec.hostname);
85.      return tmpRec;
86.  }
87.
88.  void HostData::printHostData() const {
89.
     cout<<sigStrength<<setw(16)<<trend<<setw(16)<<hostnam
     e<<setw(16)<<priority<<endl;
90.  }
91.
92.  const float HostData::getTrend() {
93.      return trend;
94.  }
95.
96.  const float HostData::getPriority() {
```

```
97.       return priority;
98.   }
99.
100. const string HostData::getHostname() {
101.       return hostname;
102. }
103.
104. void HostData::setPrioritySignal(float i_priority,
     int i_sigStrength) {
105.       priority = i_priority;
106.       sigStrength= i_sigStrength;
107.
     priosig_vec.push_back(weightedLinearDist(sigStrength,
     priority));
108.       time_vec.push_back((double) time(0));
109. }
110.
111. void HostData::setTrend() {
112.       trend = calculateTrend(priosig_vec, time_vec);
113.}
```

Class Name: VecHeap

File Name: *vecheap.h*

```
1. #ifndef VECHEAP_H
2. #define VECHEAP_H
3.
4. #include <vector>
5. #include "hostdata.h"
6.
7. using std::vector;
8.
9. class VecHeap : public HostData {
10.
11.public:
12.       VecHeap(vector<HostData>&);
13.       void displayVector(vector<HostData>&, const long
   int&);
14.       long int getParent(long int);
15.       long int getLeft(long int);
```

```
16.       long int getRight(long int);
17.       void insertHeap(vector<HostData>&, HostData&);
18.       void maxHeapify(vector<HostData>& , long int ,
   const long int&);
19.       void displayArray(vector<HostData>& , const long
   int&);
20.       void buildHeap(vector<HostData>&);
21.       void heapSort(vector<HostData>& , long int);
22.
23.private:
24.       long int heapSize;
25.
26.};
27.
28.#endif
```

File Name: *vecheap.cpp*

```
1. #include <iostream>
2. #include <iomanip>
3. #include <math.h>
4. #include <vector>
5. #include "vecheap.h"
6. #include "hostdata.h"
7.
8. using std::cout;
9. using std::endl;
10.using std::setw;
11.using std::setprecision;
12.using std::vector;
13.
14.VecHeap::VecHeap(vector<HostData>& vec_arr) {
15.       vec_arr.clear();
16.}
17.
18.long int VecHeap::getParent(long int index) {
19.       return (long int) ceil(index/2)-1;
20.}
21.
```

```
22.long int VecHeap::getLeft(long int index) {
23.       return 2*index + 1;
24.}
25.
26.long int VecHeap::getRight(long int index) {
27.       return 2*index + 2;
28.}
29.
30.void VecHeap::maxHeapify(vector<HostData>& arr_heap,
   long int index, const long int &hsize) {
31.       long int largest = -1;
32.       HostData tmp_ex;
33.       long int left = getLeft(index);
34.       long int right = getRight(index);
35.
36.       //displayArray(arr_heap);
37.       if((left < hsize) &&
   (arr_heap[left].getPriority() >
   arr_heap[index].getPriority()))
38.             largest = left;
39.       else
40.             largest = index;
41.       if((right < hsize) &&
   (arr_heap[right].getPriority() >
   arr_heap[largest].getPriority()))
42.             largest = right;
43.       if(largest != index) {
44.             tmp_ex = arr_heap[index];
45.             arr_heap[index] = arr_heap[largest];
46.             arr_heap[largest] = tmp_ex;
47.             maxHeapify(arr_heap, largest, hsize);
48.       }
49.}
50.
51.void VecHeap::displayVector(vector<HostData>& heap_arr,
   const long int &size) {
52.       for(long int i=0; i<size; i++)
53.             cout<< setw(10) << setprecision(4) <<
   heap_arr[i].getPriority();
54.       cout<<"\n\n\n";
```

```
55.}
56.
57.void VecHeap::buildHeap(vector<HostData>& heap_arr) {
58.       for(long int i=(long int)
   floor(heap_arr.size()/2); i>=0; i--) {
59.             maxHeapify(heap_arr, i, heap_arr.size());
60.       }
61.}
62.
63.void VecHeap::heapSort(vector<HostData>& heap_arr, long
   int size) {
64.       HostData tmp_ex;
65.       for(long int i=size-1; i>=1; i--) {
66.             tmp_ex = heap_arr[0];
67.             heap_arr[0] = heap_arr[i];
68.             heap_arr[i] = tmp_ex;
69.             --size;
70.             maxHeapify(heap_arr, 0, size);
71.       }
72.}
73.
74.void VecHeap::insertHeap(vector<HostData>& heap_arr,
   HostData& element) {
75.       heap_arr.push_back(element);
76.}
```

Class Name: *(No Class)*

File Name: ***ioclient.cpp***

```
1. #include <socket++/sockinet.h>
2. #include <iostream>
3. #include "hostdata.h"
4. #include <math.h>
5. #include <sstream>
6. #include <ctime>
7.
8. using std::cout;
9. using std::endl;
```

```
10.using std::flush;
11.using std::rand;
12.using std::stringstream;
13.
14.int main ()
15.{
16.      time_t curr_time;
17.      srand((unsigned)time(0));
18.      string temp_hostname = "Client";
19.      string rand_str="";
20.      stringstream ss;
21.      ss << rand()%1000;
22.
23.      string temp_str = "";
24.      HostData data(10, 30.5,
   temp_hostname.append(ss.str()) ,fmodf(rand(),99.99));
25.      time(&curr_time);
26.      cout<<ctime(&curr_time)<<" Sending the following
   info. to server: "<<endl;
27.      data.printHostData();
28.      //char  buf[1024];
29.      iosockinet io (sockbuf::sock_stream);
30.      io->connect ("ronin", 4500);
31.      temp_str = data.toString(data);
32.      io << data.replaceSpace(temp_str) << endl;
33.
34.      sleep(2);
35.      iosockinet io2 (sockbuf::sock_stream);
36.      io2->connect ("ronin", 4500);
37.      data.setPrioritySignal(20.17, 10);
38.      data.setTrend();
39.      time(&curr_time);
40.      cout<<ctime(&curr_time)<<" Sending the following
   info. to server: "<<endl;
41.      data.printHostData();
42.      temp_str = data.toString(data);
43.      io2 << data.replaceSpace(temp_str) << endl;
44.
45.      sleep(2);
```

```
46.       iosockinet io3 (sockbuf::sock_stream);
47.       io3->connect ("ronin", 4500);
48.       data.setPrioritySignal(20.17, 10);
49.       data.setTrend();
50.       time(&curr_time);
51.       cout<<ctime(&curr_time)<<" Sending the following
   info. to server: "<<endl;
52.       data.printHostData();
53.       temp_str = data.toString(data);
54.       io3 << data.replaceSpace(temp_str) << endl;
55.
56.       return 0;
57.}
```

Class Name: *(No Class)*

File Name: ***ioserver.cpp***

```
1. #include <socket++/sockinet.h>
2. #include <iostream>
3. #include <string>
4. #include <socket++/fork.h>
5. #include "hostdata.h"
6. #include "vecheap.h"
7. #include <vector>
8.
9. using std::cout;
10.using std::endl;
11.using std::flush;
12.using std::string;
13.using std::getline;
14.using std::vector;
15.
16.void incrementCount(int * count) {
17.       *count = *count + 1;
18.}
19.
20.string process_socket(iosockinet *sock) {
21.       string  buf;
```

```
22.       getline (*sock, buf); cout << buf << endl;
23.       sock->flush();
24.       delete sock;
25.       return buf;
26.}
27.
28.int main ()
29.{
30.       int count=0;
31.       vector<HostData> heap_arr;
32.       VecHeap heap_ops(heap_arr);
33.       HostData temp_data;
34.       string tmp_str = "";
35.
36.       Fork *pF;
37.       sockinetbuf sin (sockbuf::sock_stream);
38.       sin.bind (sin.localhost(), 4500);
39.       sin.listen();
40.
41.     while(1) {
42.       iosockinet *io;
43.       io = new iosockinet (sin.accept ());
44.             incrementCount(&count);
45.//                 pF = new Fork (0, 1);
46.//                 if (pF->is_child ()) {
47.                         io->rdbuf()->keepalive (1);
48.                   tmp_str = process_socket(io);
49.                   tmp_str =
   temp_data.restoreSpace(tmp_str);
50.                   temp_data =
   temp_data.toData(tmp_str);
51.                   cout<<"Received info from:
   "<<temp_data.getHostname()<<endl;
52.                   temp_data.printHostData();
53.                   heap_ops.insertHeap(heap_arr,
   temp_data);
54.                   heap_ops.buildHeap(heap_arr);
55.                   heap_ops.heapSort(heap_arr,
   heap_arr.size());
56.                   heap_ops.displayVector(heap_arr,
```

```
   heap_arr.size());
57.                   cout<<"Processed "<<count<<" Requests
   so far"<<endl;
58.  //                       return (0);
59.//          }
60.
61.    }
62.
63.    return 0;
64.}
```

APPENDIX B: NS2 Simulation Model (Tcl Source Code)

2 Node Simulation Model

```
1. sset opt(chan)           Channel/WirelessChannel    ;#
   channel type
2. set opt(prop)            Propagation/TwoRayGround   ;#
   radio-propagation model
3. set opt(netif)           Phy/WirelessPhy            ;#
   network interface type
4. set opt(mac)             Mac/802_11                 ;#
   MAC type
5. set opt(ifq)             Queue/DropTail/PriQueue    ;#
   interface queue type
6. set opt(ll)              LL                         ;#
   link layer type
7. set opt(ant)             Antenna/OmniAntenna        ;#
   antenna model
8. set opt(ifqlen)          50                         ;#
   max packet in ifq
9. set opt(nn)              2                         ;#
   number of mobilenodes
10.set opt(adhocRouting)    DSDV                       ;#
   routing protocol
11.
12.set opt(cp)              ""                         ;#
   connection pattern file
13.set opt(sc)      "~/ns-allinone-2.31/ns-
   2.31/tcl/mobility/scene/scen-3-test"    ;# node
   movement file.
14.
15.set opt(x)       170                              ;# x
   coordinate of topology
16.set opt(y)       170                              ;# y
   coordinate of topology
17.set opt(seed)    0.0                              ;# seed
   for random number gen.
18.set opt(stop)    250                              ;# time
   to stop simulation
19.
20.set opt(ftp1-start)      160.0
21.set opt(ftp2-start)      170.0
```

```
22.
23.set num_wired_nodes      2
24.set num_bs_nodes         1
25.
26.
27.set ns_     [new Simulator]
28.
29.$ns_ node-config -addressType hierarchical
30.AddrParams set domain_num_ 2           ;# number of
   domains i.e. x1 Wired - x1 Wireless
31.lappend cluster_num 2 1                ;# number of
   clusters in each domain i.e. x2 Wired subdomains - x1
   Wireless subdomain
32.AddrParams set cluster_num_ $cluster_num
33.lappend eilastlevel 1 1 8              ;# number of
   nodes in each cluster
34.AddrParams set nodes_num_ $eilastlevel ;# for each
   domain
35.
36.set tracefd [open wireless.tr w]
37.$ns_ trace-all $tracefd
38.
39.set nf [open out.nam w]
40.$ns_ namtrace-all-wireless $nf $opt(x) $opt(y)
41.
42.proc finish {} {
43.        global ns_ nf tracefd f0 f1
44.        $ns_ flush-trace
45.        close $nf
46.      close $tracefd
47.        close $f0
48.      close $f1
49.        exec nam out.nam &
50.      exec xgraph out0.tr out1.tr -geometry 800x400 &
51.        exit 0
52.}
53.
54.# create wired nodes
55.set temp {0.0.0 0.1.0}           ;# hierarchical
   addresses to be used
```

```
56.for {set i 0} {$i < $num_wired_nodes} {incr i} {
57.    set W($i) [$ns_ node [lindex $temp $i]]
58.}
59.
60.
61.set topo    [new Topography]
62.$topo load_flatgrid $opt(x) $opt(y)
63.
64.create-god [expr $opt(nn) + $num_bs_nodes]
65.
66.# configure for base-station node
67.$ns_ node-config -adhocRouting $opt(adhocRouting) \
68.                 -llType $opt(ll) \
69.                 -macType $opt(mac) \
70.                 -ifqType $opt(ifq) \
71.                 -ifqLen $opt(ifqlen) \
72.                 -antType $opt(ant) \
73.                 -propType $opt(prop) \
74.                 -phyType $opt(netif) \
75.                 -channelType $opt(chan) \
76.             -topoInstance $topo \
77.                 -wiredRouting ON \
78.             -agentTrace ON \
79.                 -routerTrace OFF \
80.                 -macTrace OFF
81.
82.
83.#create base-station node
84.set temp {1.0.0 1.0.1 1.0.2}   ;# hier address to be
   used for
85.                                        ;# wireless domain
86.set BS(0) [ $ns_ node [lindex $temp 0]]
87.$BS(0) random-motion 0                  ;# disable random
   motion
88.
89.#provide some co-ordinates (fixed) to base station node
90.$BS(0) set X_ 1.0
91.$BS(0) set Y_ 2.0
92.$BS(0) set Z_ 0.0
```

```
93.
94.# create mobilenodes in the same domain as BS(0)
95.# note the position and movement of mobilenodes is as
   defined
96.# in $opt(sc)
97.# Note there has been a change of the earlier
   AddrParams
98.# function 'set-hieraddr' to 'addr2id'.
99.
100.#configure for mobilenodes
101.$ns_ node-config -wiredRouting OFF
102.
103.# now create mobilenodes
104.for {set j 0} {$j < $opt(nn)} {incr j} {
105.    set node_($j) [ $ns_ node [lindex $temp \
106.            [expr $j+1]] ]
107.    $node_($j) base-station [AddrParams addr2id \
108.            [$BS(0) node-addr]]   ;# provide each
   mobilenode with
109.                                  ;# hier address of
   its base-station
110.}
111.
112.#create links between wired and BS nodes
113.$ns_ duplex-link $W(0) $W(1) 5Mb 2ms DropTail
114.$ns_ duplex-link $W(1) $BS(0) 5Mb 2ms DropTail
115.
116.$ns_ duplex-link-op $W(1) $W(0) queuePos 0.5
117.
118.$ns_ duplex-link-op $W(0) $W(1) orient down
119.$ns_ duplex-link-op $W(1) $BS(0) orient left-down
120.
121.$node_(0) set X_ 20.0
122.$node_(0) set Y_ 60.0
123.$node_(0) set Z_ 0.0
124.
125.$node_(1) set X_ 45.0
126.$node_(1) set Y_ 30.0
127.$node_(1) set Z_ 0.0
```

```
128.
129.$ns_ at 0.0 "$node_(0) setdest 60.0 20.0 1.0"
130.$ns_ at 0.0 "$node_(1) setdest 90.0 20.0 1.0"
131.
132.$ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"
133.$ns_ at 50.0 "$node_(1) setdest 40.0 40.0 15.0"
134.
135.$ns_ at 100.0 "$node_(0) setdest 100.0 18.0 1.0"
136.$ns_ at 50.0 "$node_(1) setdest 140.0 40.0 15.0"
137.
138.proc attach-expoo-traffic { node sink size burst idle
   rate id color priority} {
139.     set ns_ [Simulator instance]
140.
141.     set source [new Agent/UDP]
142.     $ns_ attach-agent $node $source
143.     $source set class_ $id
144.     $source set prio_ $priority
145.     $ns_ color $id $color
146.
147.     set traffic [new Application/Traffic/Exponential]
148.     $traffic set packetSize_ $size
149.     $traffic set burst_time_ $burst
150.     $traffic set idle_time_ $idle
151.     $traffic set rate_ $rate
152.
153.     $traffic attach-agent $source
154.
155.     $ns_ connect $source $sink
156.     return $traffic
157.}
158.
159.set sink0 [new Agent/LossMonitor]
160.set sink1 [new Agent/LossMonitor]
161.
162.
163.
164.
```

```
165.$ns_ attach-agent $W(0) $sink0
166.$ns_ attach-agent $W(1) $sink1
167.
168.set source0 [attach-expoo-traffic $node_(0) $sink0 500
   2s 0.5s 200k 1 Red 10]
169.set source1 [attach-expoo-traffic $node_(1) $sink1 500
   2s 0.7s 200k 2 Blue 1]
170.
171.set f0 [open out0.tr w]
172.set f1 [open out1.tr w]
173.
174.proc record {} {
175.     global sink0 sink1 f0 f1
176.
177.     set ns_ [Simulator instance]
178.     set time 0.5
179.     set bw0 [$sink0 set bytes_]
180.     set bw1 [$sink1 set bytes_]
181.     set now [$ns_ now]
182.
183.     puts $f0 "$now [expr $bw0/$time*8/1000000]"
184.     puts $f1 "$now [expr $bw1/$time*8/1000000]"
185.
186.     $sink0 set bytes_ 0
187.     $sink1 set bytes_ 0
188.
189.     $ns_ at [expr $now+$time] "record"
190.}
191.
192.$ns_ at 0.0 "record"
193.$ns_ at 10.0 "$source0 start"
194.$ns_ at 10.0 "$source1 start"
195.
196.$ns_ at 50.0 "$source1 stop"
197.$ns_ at 70.0 "$source1 start"
198.
199.$ns_ at 100.0 "$source1 stop"
200.$ns_ at 120.0 "$source1 start"
201.
```

```
202.$ns_ at 150.0 "$source0 stop"
203.$ns_ at 150.0 "$source1 stop"
204.
205.
206.for {set i 0} {$i < $opt(nn) } {incr i} {
207.    $ns_ at 150.0 "$node_($i) reset";
208.}
209.$ns_ at 150.0001 "finish"
210.$ns_ at 150.0002 "puts \"NS EXITING...\" ; $ns_ halt"
211.
212.puts "Starting Simulation..."
213.$ns_ run
```

7 Node Simulation Model

```
1.set opt(chan)           Channel/WirelessChannel    ;#
  channel type
2.set opt(prop)           Propagation/TwoRayGround   ;#
  radio-propagation model
3.set opt(netif)          Phy/WirelessPhy            ;#
  network interface type
4.set opt(mac)            Mac/802_11                 ;#
  MAC type
5.set opt(ifq)            Queue/DropTail/PriQueue    ;#
  interface queue type
6.set opt(ll)             LL                         ;#
  link layer type
7.set opt(ant)            Antenna/OmniAntenna        ;#
  antenna model
8.set opt(ifqlen)         50                         ;#
  max packet in ifq
9.set opt(nn)             7                         ;#
  number of mobilenodes
10.set opt(adhocRouting)   DSDV                        ;#
  routing protocol
11.
12.set opt(cp)             ""                          ;#
  connection pattern file
13.set opt(sc)     "~/ns-allinone-2.31/ns-
  2.31/tcl/mobility/scene/scen-3-test"    ;# node movement
  file.
14.
```

```
15.set opt(x)      170                              ;# x
  coordinate of topology
16.set opt(y)      170                              ;# y
  coordinate of topology
17.set opt(seed)   0.0                              ;# seed
  for random number gen.
18.set opt(stop)   250                              ;# time
  to stop simulation
19.
20.set opt(ftp1-start)      160.0
21.set opt(ftp2-start)      170.0
22.
23.set num_wired_nodes      2
24.set num_bs_nodes         1
25.
26.
27.set ns_  [new Simulator]
28.
29.$ns_ node-config -addressType hierarchical
30.AddrParams set domain_num_ 2           ;# number of
  domains i.e. x1 Wired - x1 Wireless
31.lappend cluster_num 2 1                ;# number of
  clusters in each domain i.e. x2 Wired subdomains - x1
  Wireless subdomain
32.AddrParams set cluster_num_ $cluster_num
33.lappend eilastlevel 1 1 8              ;# number of
  nodes in each cluster
34.AddrParams set nodes_num_ $eilastlevel ;# for each
  domain
35.
36.set tracefd [open wireless.tr w]
37.$ns_ trace-all $tracefd
38.
39.set nf [open out.nam w]
40.$ns_ namtrace-all-wireless $nf $opt(x) $opt(y)
41.
42.proc finish {} {
43.        global ns_ nf tracefd f0 f1
44.        $ns_ flush-trace
45.        close $nf
46.   close $tracefd
```

```
47.         close $f0
48.    close $f1
49.         exec nam out.nam &
50.    exec xgraph out0.tr out1.tr -geometry 800x400 &
51.         exit 0
52.}
53.
54.# create wired nodes
55.set temp {0.0.0 0.1.0}            ;# hierarchical
  addresses to be used
56.for {set i 0} {$i < $num_wired_nodes} {incr i} {
57.     set W($i) [$ns_ node [lindex $temp $i]]
58.}
59.
60.
61.set topo [new Topography]
62.$topo load_flatgrid $opt(x) $opt(y)
63.
64.create-god [expr $opt(nn) + $num_bs_nodes]
65.
66.# configure for base-station node
67.$ns_ node-config -adhocRouting $opt(adhocRouting) \
68.                  -llType $opt(ll) \
69.                  -macType $opt(mac) \
70.                  -ifqType $opt(ifq) \
71.                  -ifqLen $opt(ifqlen) \
72.                  -antType $opt(ant) \
73.                  -propType $opt(prop) \
74.                  -phyType $opt(netif) \
75.                  -channelType $opt(chan) \
76.          -topoInstance $topo \
77.                  -wiredRouting ON \
78.          -agentTrace ON \
79.                  -routerTrace OFF \
80.                  -macTrace OFF
81.
82.
83.#create base-station node
```

```
84.set temp {1.0.0 1.0.1 1.0.2 1.0.3 1.0.4 1.0.5 1.0.6
  1.0.7}   ;# hier address to be used for
85.                                        ;# wireless domain
86.set BS(0) [ $ns_ node [lindex $temp 0]]
87.$BS(0) random-motion 0                  ;# disable random
  motion
88.
89.#provide some co-ordinates (fixed) to base station node
90.$BS(0) set X_ 1.0
91.$BS(0) set Y_ 2.0
92.$BS(0) set Z_ 0.0
93.
94.# create mobilenodes in the same domain as BS(0)
95.# note the position and movement of mobilenodes is as
  defined
96.# in $opt(sc)
97.# Note there has been a change of the earlier
  AddrParams
98.# function 'set-hieraddr' to 'addr2id'.
99.
100.#configure for mobilenodes
101.$ns_ node-config -wiredRouting OFF
102.
103.# now create mobilenodes
104.for {set j 0} {$j < $opt(nn)} {incr j} {
105.    set node_($j) [ $ns_ node [lindex $temp \
106.            [expr $j+1]] ]
107.    $node_($j) base-station [AddrParams addr2id \
108.            [$BS(0) node-addr]]   ;# provide each
  mobilenode with
109.                                       ;# hier address of
  its base-station
110.}
111.
112.#create links between wired and BS nodes
113.$ns_ duplex-link $W(0) $W(1) 1Mb 2ms DropTail
114.$ns_ duplex-link $W(1) $BS(0) 1Mb 2ms DropTail
115.
116.$ns_ duplex-link-op $W(0) $W(1) orient down
117.$ns_ duplex-link-op $W(1) $BS(0) orient left-down
```

```
118.
119.$node_(0) set X_ 20.0
120.$node_(0) set Y_ 60.0
121.$node_(0) set Z_ 0.0
122.
123.$node_(1) set X_ 45.0
124.$node_(1) set Y_ 30.0
125.$node_(1) set Z_ 0.0
126.
127.$node_(2) set X_ 90.0
128.$node_(2) set Y_ 20.0
129.$node_(2) set Z_ 0.0
130.
131.$node_(3) set X_ 15.0
132.$node_(3) set Y_ 10.0
133.$node_(3) set Z_ 0.0
134.
135.$node_(4) set X_ 65.0
136.$node_(4) set Y_ 60.0
137.$node_(4) set Z_ 0.0
138.
139.$node_(5) set X_ 75.0
140.$node_(5) set Y_ 40.0
141.$node_(5) set Z_ 0.0
142.
143.$node_(6) set X_ 85.0
144.$node_(6) set Y_ 90.0
145.$node_(6) set Z_ 0.0
146.
147.$ns_ at 0.0 "$node_(0) setdest 60.0 20.0 1.0"
148.$ns_ at 0.0 "$node_(1) setdest 90.0 20.0 1.0"
149.$ns_ at 0.0 "$node_(2) setdest 90.0 20.0 1.0"
150.$ns_ at 0.0 "$node_(3) setdest 15.0 10.0 1.0"
151.$ns_ at 0.0 "$node_(4) setdest 65.0 60.0 1.0"
152.$ns_ at 0.0 "$node_(5) setdest 75.0 40.0 1.0"
153.$ns_ at 0.0 "$node_(6) setdest 85.0 90.0 1.0"
154.
155.$ns_ at 10.0 "$node_(0) setdest 20.0 18.0 1.0"
```

```
156.$ns_ at 50.0 "$node_(1) setdest 40.0 40.0 15.0"
157.
158.$ns_ at 100.0 "$node_(0) setdest 100.0 18.0 1.0"
159.$ns_ at 50.0 "$node_(1) setdest 140.0 40.0 15.0"
160.
161.proc attach-expoo-traffic { node sink size burst idle
  rate id color priority} {
162.  set ns_ [Simulator instance]
163.
164.  set source [new Agent/UDP]
165.  $ns_ attach-agent $node $source
166.  $source set class_ $id
167.  $source set prio_ $priority
168.  $ns_ color $id $color
169.
170.  set traffic [new Application/Traffic/Exponential]
171.  $traffic set packetSize_ $size
172.  $traffic set burst_time_ $burst
173.  $traffic set idle_time_ $idle
174.  $traffic set rate_ $rate
175.
176.  $traffic attach-agent $source
177.
178.  $ns_ connect $source $sink
179.  return $traffic
180.}
181.
182.set sink0 [new Agent/LossMonitor]
183.set sink1 [new Agent/LossMonitor]
184.
185.
186.
187.
188.$ns_ attach-agent $W(0) $sink0
189.$ns_ attach-agent $W(1) $sink1
190.
191.set source0 [attach-expoo-traffic $node_(0) $sink1
  15500 2s 0.1s 16000k 1 Red 10]
192.set source1 [attach-expoo-traffic $node_(1) $sink0 300
```

```
  12s 0.1s 200k 2 Blue 1]
193.set source2 [attach-expoo-traffic $node_(2) $sink0 300
  12s 0.1s 200k 3 Green 7]
194.set source3 [attach-expoo-traffic $node_(3) $sink0 300
  12s 0.1s 200k 4 Magenta 6]
195.set source4 [attach-expoo-traffic $node_(4) $sink0 300
  12s 0.1s 200k 5 Cyan 3]
196.set source5 [attach-expoo-traffic $node_(5) $sink0 300
  12s 0.1s 200k 6 White 9]
197.set source6 [attach-expoo-traffic $node_(6) $sink1
  15500 2s 0.1s 16000k 7 Black 10]
198.
199.set f0 [open out0.tr w]
200.set f1 [open out1.tr w]
201.
202.proc record {} {
203.  global sink0 sink1 f0 f1
204.
205.  set ns_ [Simulator instance]
206.  set time 0.5
207.  set bw0 [$sink0 set bytes_]
208.  set bw1 [$sink1 set bytes_]
209.  set now [$ns_ now]
210.
211.  puts $f0 "$now [expr $bw0/$time*8/1000000]"
212.  puts $f1 "$now [expr $bw1/$time*8/1000000]"
213.
214.  $sink0 set bytes_ 0
215.  $sink1 set bytes_ 0
216.
217.  $ns_ at [expr $now+$time] "record"
218.}
219.
220.$ns_ at 0.0 "record"
221.$ns_ at 10.0 "$source0 start"
222.$ns_ at 10.0 "$source1 start"
223.$ns_ at 10.0 "$source2 start"
224.$ns_ at 10.0 "$source3 start"
225.$ns_ at 10.0 "$source4 start"
226.$ns_ at 10.0 "$source5 start"
```

```
227.$ns_ at 10.0 "$source6 start"
228.
229.$ns_ at 50.0 "$source1 stop"
230.$ns_ at 55.0 "$source4 stop"
231.
232.$ns_ at 150.0 "$source0 stop"
233.$ns_ at 150.0 "$source1 stop"
234.
235.
236.for {set i 0} {$i < $opt(nn) } {incr i} {
237.    $ns_ at 150.0 "$node_($i) reset";
238.}
239.$ns_ at 150.0001 "finish"
240.$ns_ at 150.0002 "puts \"NS EXITING...\" ; $ns_ halt"
241.
242.puts "Starting Simulation..."
243.$ns_ run
```