The Hong Kong Polytechnic University

Department of Computing

Distributed Coordination in Mobile Wireless Environments

by

WU WEIGANG

A thesis submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

November 2006

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signature)

WU WEIGANG (Name of Student)

Abstract

Advances in wireless networking technologies and powerful portable mobile devices have engendered the new paradigm of mobile computing, whereby mobile users carrying portable devices can access the information and services for various tasks regardless of their physical locations or movement behaviors. Mobile computing is a branch of distributed computing, but mobile networks have fundamentally different characteristics from traditional wired networks in aspects of communication, mobility and resource constraints. These characteristics make the development of distributed algorithms much more difficult. In this thesis, we investigate the challenging issues in designing algorithms for solving distributed computing problems in mobile wireless networks. We focus on two distributed coordination problems: the *consensus* problem and the *mutual exclusion* problem.

The consensus problem arises in many distributed computing applications, such as atomic commitment, atomic broadcast, and file replication. So far, little work has been reported on achieving consensus in mobile environments. This thesis makes the following original contributions in this field.

First, we develop a general technique named “Look-Ahead” to speed up the execution of consensus protocols by making use of future messages. Nearly all existing consensus protocols for asynchronous systems are executed in asynchronous rounds and each round is divided into several phases. Due to the asynchrony, some “future” message may be delivered to a receiver that has not yet entered the phase or round of the message. By making use of such future messages, hosts can decide to stop waiting for a message with a long delay or sent late, so as to speed up their executions.

Second, we improve message efficiency and scalability of consensus protocols for mobile ad hoc networks (MANETs) using a hierarchy imposed on the mobile hosts. By clustering the mobile hosts into clusters, a two-layer hierarchy is established. Then, the messages from and to the hosts in the same cluster are merged/unmerged

by the clusterhead in order to reduce the message cost and improve the scalability. Based on different ways for clustering hosts, we propose two hierarchical protocols. The first protocol is based on a static set of clusterheads, in which the function of clustering hosts and the function of achieving consensus are interlaced. In the second hierarchical protocol, clusterheads are dynamically selected, and the functions of clustering hosts and of achieving consensus are separated using a modular approach. With such an approach, the design of hierarchical consensus protocols is simplified, similar as the separation of failure detection and achieving consensus in failure detector based protocols. We define a new oracle, named “eventual clusterer”, which is in charge of the construction and maintenance of the hierarchy in a MANET. Based on the eventual clusterer oracle, we design a hierarchical consensus protocol.

The third contribution to the consensus problem is the design of an eventual leader protocol for “dynamic” infrastructured mobile networks, where the number of participating hosts can change arbitrarily as time passes and an unbounded number of hosts can join or leave the system at any time. The proposed eventual leader protocol can be used to design consensus protocols for dynamic infrastructured mobile networks.

Another coordination problem addressed in this thesis is mutual exclusion (MUTEX), one of typical coordination problems, which is concerned with the coordination of accesses to *critical section* (CS) mutual exclusively. We propose the first permission-based MUTEX algorithm for MANETs. Unlike token-based algorithms, a permission-based algorithm needs neither to maintain any logical topology nor to propagate any message if no host requests the CS. However, a permission-based algorithm has to send messages for each request of hosts, which significantly increases the average message cost. Based on the “look ahead” technique, which enforces the MUTEX only among the hosts that are currently competing for CS, we propose a message efficient MUTEX algorithm for MANETs. The algorithm can also tolerate link and host failures by using timeout-based fault tolerance mechanisms.

Publications

Journal Papers

1. **Weigang Wu**, Jiannong Cao, Jin Yang, Michel Raynal, Design and Performance Evaluation of Efficient Consensus Protocols for Mobile Ad Hoc Networks, *IEEE Transactions on Computers*, Major revision version submitted.
2. **Weigang Wu**, Jiannong Cao, Jin Yang, A Fault Tolerant Mutual Exclusion Algorithm for Mobile Ad Hoc Networks, *Pervasive and Mobile Computing, Elsevier*, Major revision version submitted.
3. **Weigang Wu**, Jiannong Cao, Jin Yang, Michel Raynal, A Fast Consensus Protocol Based on the Unreliable Failure Detector ◊*S*, *Journal of Parallel and Distributed Computing, Elsevier*, under major revision.
4. Jin Yang, Jiannong Cao, **Weigang Wu**, Chengzhong Xu, Efficient Algorithms for Fault Tolerant Mobile Agent Execution, *International Journal of High Performance Computing and Networking (IJHPCN)* (Accepted).
5. Jin Yang, Jiannong Cao, **Weigang Wu**, Chengzhong Xu, Models and Mechanisms for Mobile Agent Transactions, *International Journal of Wireless and Mobile Computing (IJWMC)* (Accepted).

Conference Papers

1. **Weigang Wu**, Jiannong Cao, Jin Yang, A Scalable Mutual Exclusion Algorithm for Mobile Ad Hoc Networks, Proceedings of the *Fourteenth International Conference on Computer Communications and Networks (ICCCN'05)*, October 17-19, 2005, San Diego, California USA
2. **Weigang Wu**, Jiannong Cao, Jin Yang, Michel Raynal, A Hierarchical Consensus Protocol for Mobile Ad Hoc Networks, Proceedings of the *14th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP'06)*, February 15-17, 2006, Montbéliard, France.
3. Jin Yang, Jiannong Cao, **Weigang Wu**, CIC: An Integrating Solution for Checkpointing in MA Systems, *Proceedings of the 2nd International*

Conference on Semantics, Knowledge and Grid (SKG2006), October 31-November 3, 2006, Guilin, China.

4. Jin Yang, Jiannong Cao, **Weigang Wu**, Checkpoint Placement Algorithms for Mobile Agent System, *Proceedings of the 5th International Conference on Grid and Cooperative Computing (GCC'06)*, October 21-23, 2006, Changsha, China.

5. Jiannong Cao, Michel Raynal, Xianbing Wang, **Weigang Wu**, The Power and Limit of Adding Synchronization Messages for Synchronous Agreement, *Proceedings of the 35th International Conference on Parallel Processing (ICPP'06)*, August 14-18, 2006, Columbus, USA.

6. Jin Yang, Jiannong Cao, **Weigang Wu**, Corentin Travers The Notification based Approach for Implementation Failure Detector, Proceedings of the *First International Conference on Scalable Information System (InfoScale'06)*, May 29-June 1, 2006, Hong Kong.

7. Jin Yang, Jiannong Cao, **Weigang Wu**, Chengzhong Xu, Parallel Algorithms for Fault-Tolerant Mobile Agent Execution, Proceedings of the *6th International Conference on Algorithms and Architectures (ICA3PP'05)*, 246-256, October 2-5 , 2005, in Melbourne, Australia.

8. Jin Yang, Jiannong Cao, **Weigang Wu**, Chengzhong Xu, A Framework for Transactional Mobile Agent Execution, Proceedings of the *4th International Conference on Grid and Cooperative Computing (GCC'05)*, November 30-December 3, 2005, Beijing, China.

9. Jiannong Cao, Jin Yang, **Weigang Wu**, Chengzhong Xu, Exception Handling in Distributed Workflow Systems Using Mobile Agents, Proceedings of the *2005 IEEE International Conference on e-Business Engineering (ICEBE'05)*, October 18-20, 2005, Beijing, China.

Acknowledgements

I am deeply grateful to Professor Jiannong Cao for his rigorous supervision of my research. I thank him for his support, patience and encouragement during my Ph.D. study. He unremittingly trained me to be a good researcher. He taught me how to find research issues and how to solve thorny problems. Time after time, he showed me how to express my ideas and write academic papers. His vision, passion, and attitude towards the research deeply affected me. Moreover, his kindness helped me handle problems in my life satisfactorily. What I have learned and experienced during the time I spent in his laboratory will benefit me much in the future.

Another excellent and distinguished person I would like to thank is Professor Michel Raynal from IRISA. I would like to express my heartfelt gratitude to him for his insightful and illuminative suggestions on my research. His acuminous insight and guidance is invaluable to the accomplishment of this thesis.

I thank Corentin Travers from IRISA, and Jin Yang for our fruitful collaborations and discussions. We explored ideas and wrote papers together. My thanks also go to Hui Cheng, Xiaopeng Fan, Yu Huang, Miaomiao Wang, Kirk Wong, Gang Yao, Yuan Zheng, and all other members of Prof. Cao's research group that I cannot enumerate here. I thank them for interesting discussions and suggestions on my work.

Also, I wish to acknowledge my appreciation to Yi Xie, Xiapu Luo and Shuyuan Jin, who shared with me the pain and pleasure of the PhD study at The Hong Kong Polytechnic University.

Last, but not least, I would like to thank my family. Their love, understanding, support and encouragement were in the end what made this dissertation possible.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CS: Critical Section.

DAG: Directed Acyclic Graph.

ET: Execution Time.

FD: Failure Detector.

FIFO: First In First Out.

GST: Global Stabilization Time.

HC: Hierarchical Consensus.

HCD: Hierarchical Consensus based on Delta.

HPCS: number of Hops Per CS entry.

MANET: Mobile Ad hoc NETwork.

MH: Mobile Host.

MPCS: number of Messages Per CS entry.

MUTEX: MUTual EXclusion.

NH: Number of Hops.

NM: Number of Messages.

NR: Number of Rounds.

RT: Response Time.

RZD: Round-Zero-Degradation.

ZD: Zero-Degradation.

Chapter 1. Introduction

A distributed system consists of a collection of computing processes/hosts[1] that are interconnected by a computer network. The hosts communicate and coordinate their actions by only passing messages [44]. The main motivation factor for constructing distributed systems is resource sharing. However, distributed systems potentially provides much more significant advantages, including communication, enhanced performance, improved reliability and availability, good extensibility, and modular expandability [29][147].

In a distributed system, there is no global clock or shared memory. On the other hand, there are concurrent components, which can execute in parallel and fail independently. Therefore, we need to design algorithms and protocols for distributed operating systems or middleware to coordinate the concurrent components.

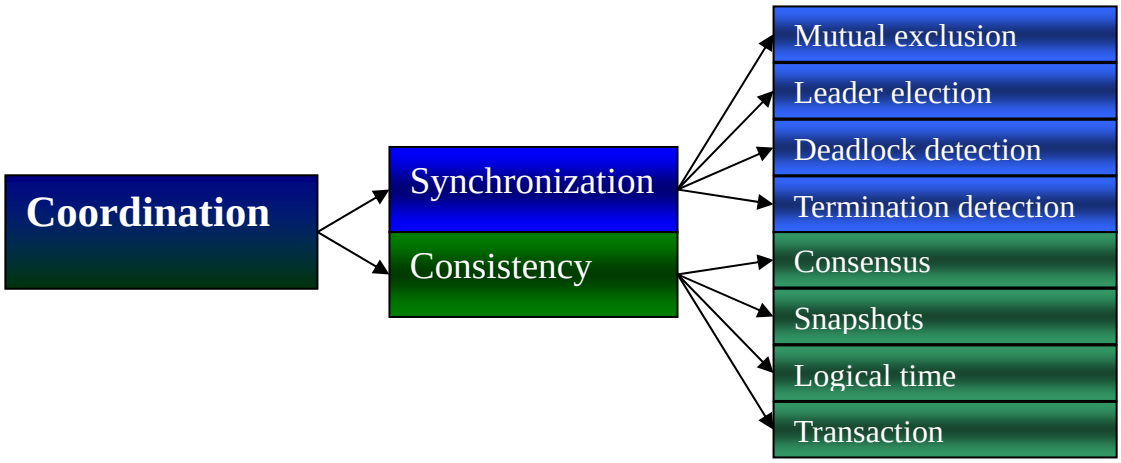

Figure 1-1 Taxonomy of distributed coordination problems

"Coordination" is a general concept which refers to coordinating the operations of distributed hosts or components to cooperatively perform some specific task. As shown in Figure 1-1, according to the problem addressed, coordination can be divided into two broad categories: *synchronization* and *consistency*.

"Synchronization" refers to the control of the sequence of some critical operations performed by the hosts in a distributed system. Synchronization involves many

[1] In this thesis, the terms "process" and "host" are used interchangeably.

different aspects, including leader election, termination detection, deadlock detection and mutual exclusion.

A leader election algorithm is for choosing a unique process to play a particular role in performing a distributed task, e.g. scheduling jobs, regenerating the token [44]. There are two fundamental requirements for leader election. *Safety* requires that there should never be more than one leader, and *liveness* requires that eventually some host is elected.

"Termination detection" refers to the necessity of determining whether a set of distributed processes have entered a "silent" status where all processes are idle and no further computation is possible, taking unpredictable message delays into account [147]. It has applications in diffusion computation and distributed garbage collection. It also serves a part in checking stable states (such as deadlock and token loss) in a distributed system.

A deadlock occurs when processes holding some resources request to access other resources held by other processes in the same set. There are two types of deadlock: *resource deadlock* and *communication deadlock*. A deadlock requires the attention of a process outside those involved in the deadlock for its detection and resolution. A deadlock is resolved by aborting one or more processes involved in the deadlock and granting the released resources to other processes.

In this thesis, our research on synchronization focuses on the mutual exclusion problem, which is introduced in Section 1.2.

"Consistency" refers to achieving agreement by processes on some value or the results of some operation. Consistency problems include logical time, transaction, snapshot and consensus.

To address the lack of physical global clock in a distributed system, people propose and implement the "logical clock" to obtain a "logical" global time [132]. In a system of logical clocks, every process has a logical clock that is advanced using a set of rules. Every event is assigned a timestamp, by which a process can infer the causality relation between events.

The algorithm to record the global state of a distributed system is called "snapshot" algorithm [32][88]. Snapshot algorithms are fundamental for many important distributed applications, such as deadlocks detection [25], termination detection [106] and checkpoint/recovery [2].

The transaction is used to protect shared resources, especially data fields, against inconsistent access by different processes. Generally, a *transaction* is a sequence of code in execution, such that the code will transform a database or server from one consistent state to another consistent state [151]. In *distributed transactions*, a transaction operates on data that are distributed across more than one host.

In this thesis, our research on consistency focuses on the consensus problem, which is introduced in Section 1.1.

1.1. Consensus

Consensus problem is one of the fundamental problems in distributed systems. It arises in many distributed computing applications, e.g. atomic commitment, atomic broadcast, file replication [70][74][75]. Generally, consensus involves getting a set of processes to agree on a value proposed by one or more of the processes [44]. The consensus problem can be defined in terms of two primitives: "propose" and "decide". Initially, each process p_i selects a value v_i from a set of possible values and invokes the primitive "propose" with this value. A process ends its participation by executing "decide", during which some value is decided upon.

The consensus problem is usually studied in environments with host failures [43][60][61][97][152] (without failure, achieving consensus is a trivial task). A process is said to be correct if it behaves according to an agreed specification during the run of a consensus protocol; otherwise, a failure occurs and the process is said to be faulty. A consensus protocol is said to be *t*-resilient if it operates correctly as long as no more than *t* processes fail before or during the execution. A correct consensus protocol should have the following three correctness properties:

Termination: Every correct process eventually decides upon some value;

Agreement: All the values decided upon are the same;

Validity: Any value decided upon should have been proposed some process.

The termination property defines the liveness associated with the consensus protocol, while the agreement property and validity property define the safety. According to the agreement property above, faulty processes may decide differently from correct ones. This is undesirable sometimes because it does not prevent a faulty process to propagate a different decision before crashing. Therefore, a more restrict agreement property is defined as follows:

Uniform-Agreement: No two processes (correct or faulty) decide differently.

The design of consensus protocols is closely related to the underlying system model, such as the synchronous model and partially synchronous model. Among different models, the asynchronous model with oracles has attracted most attention. Our study in this thesis is also based on this model.

In an asynchronous system, there is no bound on the clock drift, process speed or message delivery delay, which make the problem difficult. Fischer et al. [61] have proved that the consensus problem is unsolvable in an asynchronous system even if only one process can crash. To circumvent this impossibility, three types of oracles have been proposed: the *random number generator* [22], the *eventual leader oracle* [31] and the *unreliable failure detector* (FD for short) oracle [31]. An oracle is an abstract tool to provide some kind of information about the state of the system. The detailed description of different system models in presented Chapter 3.

1.2. Mutual Exclusion

Mutual exclusion (MUTEX) is a typical problem in distributed computing, where a group of hosts intermittently require the use of a shared resource or a piece of code called *Critical Section* (CS), which can be accessed by only one host at a time. A solution to the MUTEX problem must satisfy the following correctness properties:

Mutual Exclusion (Safety): At most one host is allowed to enter the CS at any moment;

Deadlock Free (Liveness): If any host is waiting for the CS, then in a finite time some host enters the CS;

Starvation Free (Fairness)[2]: If a host is waiting for the CS, then in a finite time the host enters the CS.

Singhal presented the taxonomy of MUTEX algorithms in [145]. With reference to Singhal's work, here we introduce a 3-dimension classification model, which is illustrated in Figure 1-2.

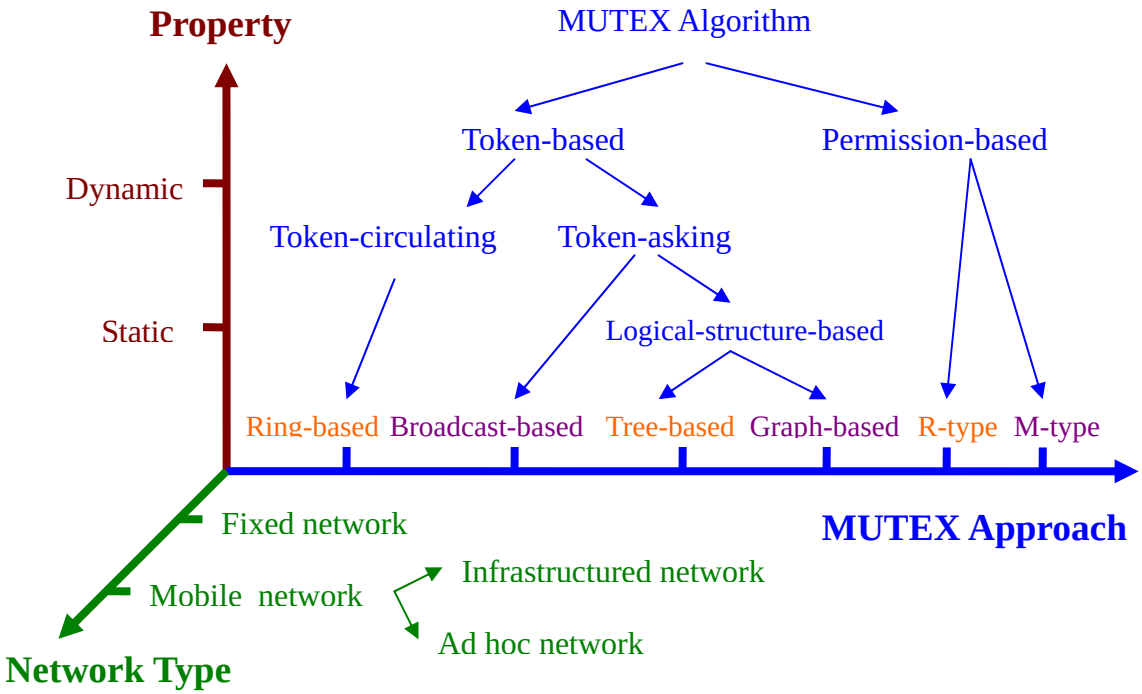

Figure 1-2 A classification of MUTEX algorithms

The first dimension is the approach used to achieve MUTEX. All the MTUEX algorithms can be categorized into two classes: *token-based* algorithms and *permission-based* algorithms.

In token-based algorithms, a unique token[3] is shared among all the hosts. A host is allowed to enter CS only if it possesses the token, thereby the MUTEX is achieved. According to the strategies used to schedule the token, token-based algorithms can be further divided into two types: *token-circulating* and *token-asking*.

[2] Sometimes, the *deadlock free* property is merged into the *starvation free* property.

[3] The token is sometimes named "privilege" of entering the CS.

Token-circulating means that the token is automatically circulated along a logical topology, e.g. a ring. When a host wants to enter the CS, it just waits until the token is passed to it. On exiting the CS, it just passes the token to its successor in the ring.

In token-asking algorithms, a host needs to send request for the token before it can obtain the token. The request can be sent by a broadcast or along some logical structure, e.g. a directed tree [124] or graph [162]. In the tree-based algorithms, the host holding the token is the root of the tree. A request for the token propagates serially along the tree to the root, and the token is passed in the contrary direction. The edges in the tree are redirected with respect to the passing of the token so that the token holder is always the root. In graph-based algorithms, a directed graph rather than a tree is used to organize the hosts. The token holder is the sink node in the graph. The token and requests for token are propagated in the similar way as in tree-based algorithms. The main difference between these two logical structures is that using the graph structure is more reliable, because it can tolerate link and host failures. Of course, additional messages are necessary to prevent cycles.

Different from token-based algorithms, a permission-based MUTEX algorithm does not need a shared token. The requesting host must first get the permissions from other hosts by exchanging messages. Usually, there are two kinds of messages: the request message and the reply message. Sometimes, the release message is also used.

There are two different kinds of permissions: the Ricart-Agrawala permission (“R-permission”) [133] and the Maekawa permission (“M-permission”) [98]. In the R-permission, a host grants the permission to a requesting host immediately if the host itself is not requesting the CS or its own request has a lower priority. Otherwise, it defers granting the permission until the request of itself has been met. The Semantics of the R-permission is “as far as I am concerned, it is OK for you to enter the CS” [145]. Therefore, a host may simultaneously grant permission to more than one host. A host can enter the CS after it gets permission from all hosts.

On the contrary, in the M-permission, a host can grant the permission to at most one host at any time. The semantics of the M-permission is “as far as all the hosts are

concerned in my opinion, it is OK for you to enter the CS" [145]. In M-permission based MUTEX algorithms, a host needs to get permission from a quorum of hosts before it can enter the CS. This is why such algorithms are usually called "quorum-based" algorithms [96].

Each MUTEX approach has its own advantages and disadvantages. The token-based approach has many desirable features, e.g. hosts only need to keep information about their neighbors and few messages are needed to pass the privilege to enter the CS. However, token-based algorithms usually have two critical issues to address. The first is the fairness in scheduling the token pass. The second is how to precisely detect the token loss, caused by link or host failures, and regenerate the token.

Compared with the token-based approach, permission-based algorithms have the following advantages: 1) there is no need to maintain the logical topology to pass the token, and 2) there is no need to propagate any message if no host requests to enter CS. However, the permission-based approach requires a large number of messages for hosts to get permissions.

The second dimension of the classification model is the dynamism property of MUTEX algorithms. If the execution of a MUTEX algorithm depends on the current system state (or history), the algorithm is called a "dynamic" algorithm. Otherwise, it is a static algorithm. The actions of a dynamic MUTEX algorithm are influenced by how the system has evolved.

The third dimension of the classification model is the network environment that a MUTEX algorithm is designed for. Besides traditional fixed networks, there are two types of mobile networks: *infrastructured networks* and *mobile ad hoc networks*. Mobile networks will be introduced in Chapter 2.

1.3. Contributions of the Thesis

The aim of this research is to study and solve coordination problems in mobile environments. Mobile computing is one branch of distributed computing. For traditional distributed systems, much work has been done and many good solutions

have been proposed to solve the coordination problems [97] [152]. However, mobile computing environments introduce many new challenges. Mobile users who carry portable devices can access the information services via various kinds of wireless networks. Mobile networks have fundamentally different properties from traditional wired networks in aspects of communication, mobility and resource constraints. These characteristics make the development of distributed algorithms much more difficult. Traditional distributed algorithms must be adapted or even re-designed to meet the requirements of mobile environments.

The main purposes of our research are: 1) investigating the characteristics of distributed computing in the mobile environments, 2) identifying the key issues in solving distributed coordination problems in mobile environments, and 3) designing new algorithms for distributed coordination for mobile environments.

In this thesis, we focus on the design of algorithms for two coordination problems: consensus and MUTEX.

1.3.1. Contributions in Consensus

Up to now, little work has been done for solving the consensus problem in context of mobile networks. In our research, we consider three issues in solving the consensus problem: time efficiency, message efficiency and dynamism of the system.

1.3.1.1. Speeding up the Execution of Consensus Protocols

We propose a fast consensus protocol based on the failure detector $\Diamond S$, which outperforms existing $\Diamond S$ based protocols in terms of time efficiency. $\Diamond S$ based consensus protocols suffer from the slowdown caused by host failures and mistakes made by the failure detector. Our protocol copes with such slowdown using two novel techniques.

The first technique is a simple and efficient approach to guarantee the Round-Zero-Degradation property (an extension of the Zero-Degradation property) in order to avoid the slowdown caused by a failed coordinator. By dynamically selecting the coordinator of a round, a crashed coordinator can be replaced so as to avoid the

failure of decision making in the round due to a failed coordinator. This technique is effective when the failure detector performs well.

The second technique, named “Look-Ahead”, helps speed up the execution, by making use of the messages delivered before their receivers enter the corresponding phases or rounds of the messages. This technique works well regardless the performance of the underlying failure detector. Look-Ahead is in fact a general technique that can be applied to consensus protocols based on any oracle, e.g. $\Diamond S$.

1.3.1.2. Achieving Message Efficiency in Consensus Protocols

Message efficiency is essential for designing effective and efficient algorithms for mobile environments. Fewer messages consume fewer resources, e.g. bandwidth and energy. To achieve message efficiency, we propose efficient consensus protocols for mobile ad hoc networks. The basic idea is to impose a two-layer hierarchy upon the hosts by clustering the mobile hosts into clusters. Then the messages from and to the hosts in the same cluster are merged/unmerged by the clusterhead so as to reduce the message cost and improve the scalability.

However, adding such a hierarchy is not trivial. First, the messages are not simply forwarded by the clusterhead, and a cluster member needs to synchronize with its clusterhead in the message exchange. Due to mobility or clusterhead failure, a mobile host may need to switch between clusterheads that are executing different steps. Therefore, the switching procedure should be delicately handled in order to maintain the synchronization between a mobile host and its clusterhead. Second, the change of the hierarchy may cause message losses, even if the communication channel is reliable. To cope with such message losses, some redeeming messages should be sent. What and when redeeming messages should be sent depends on the execution state of the mobile host and clusterhead.

Based on different approaches for clustering hosts, we propose two hierarchical protocols. The first protocol is based on a static set of clusterheads. The function of clustering hosts and the function of achieving consensus are interlaced. On the contrary, in the second protocol, clusterheads are dynamically selected. The function

of clustering hosts and the function of achieving consensus are separated using a modular design method. The clustering function, named *eventual clusterer* (denoted by Δ), is proposed to construct and maintain a cluster-based hierarchy over the mobile hosts. Since Δ provides the fault tolerant clustering function transparently, it can be used as an oracle for the design of reliable hierarchical consensus protocols. We then design a hierarchical consensus protocol using the eventual clusterer oracle. We also propose an implementation of Δ, based on the unreliable failure detector $\Diamond S$.

1.3.1.3. Handling Dynamic Mobile Systems

The above two contributions focus on respectively the message efficiency and time efficiency of consensus protocols, so as to cope with the resource constraints of mobile environments. Our third contribution to consensus is on designing protocols for dynamic mobile systems, where the number of participating hosts can change arbitrarily as time passes and an unbounded number of hosts can join or leave the system at any time.

We study how to implement the eventual leader oracle Ω in infrastructured mobile networks. We adopt a time free approach proposed for implementing Ω in fixed networks [109][111]. Each host broadcasts queries and collect responses from others hosts in rounds. With some assumption on the system behaviors, Ω can be implemented. We extend such an approach to infrastructured mobile networks.

The set of mobile support stations is a static system while the MHs constitute a dynamic system. By exchanging queries and responses among the mobile support stations, a correct mobile host is eventually elected as the leader. However, due to be mobility of the hosts, the algorithm in [109][111] does not work in a mobile network. We make some additional assumptions and modify the message exchange procedure to implement an eventual leader oracle for mobile systems.

1.3.2. Contributions in Mutual Exclusion

Another coordination problem studied is mutual exclusion. We propose the first permission-based MUTEX algorithm for mobile ad hoc networks. Compared with

the token-based approach, permission-based algorithms have two main advantages. First, no logical structure is maintained so that no message cost or memory cost is caused by such structures. Second, no message need to be propagated if no host requests the CS. These advantages make the permission-based approach well suitable for MANETs, where the resources are scarce.

One problem of the permission-based approach is the large number of messages to be exchanged to get permissions. To reduce the message cost, we propose a MUTEX algorithm using the "look ahead" technique [146], which enforces mutual exclusion only among the hosts currently competing for the critical section. The constraint of FIFO (First In First Out) channel in the original "look ahead" technique is relaxed. We also propose mechanisms to handle dozes and disconnections. Using timeout, a fault tolerance mechanism is introduced to tolerate link and host failures.

1.3.3. Summary

In summary, the main contributions of this thesis are as follows:

1. The investigation of new challenges in solving distributed coordination problems in mobile environments;

2. The design and application of a general technique named “Look-Ahead” to speed up the execution of consensus protocols;

3. The design of two message efficient consensus protocols for mobile ad hoc networks, which reduce the message cost by using clustering mechanisms;

4. The design of a new oracle, named “eventual clusterer” Δ, which can group mobile hosts into clusters to help design consensus protocols achieving message efficiency;

5. The design of an eventual leader protocol for dynamic infrastructured mobile networks, which can be used for designing consensus protocols in dynamic infrastructured mobile networks;

6. The design of the first permission-based MUTEX algorithm for mobile ad hoc networks, which can reduce message cost and tolerate link and host failures.

1.4. Outline of the Thesis

The rest of the thesis is organized as follows. We discuss the new challenges in mobile computing environments in Chapter 2. Mobile networks are first introduced and then their characteristics are summarized. Based on these characteristics, we investigate the new challenges in the design of distributed coordination algorithms in mobile environments.

Chapter 3 reviews existing work related to the two distributed coordination problems investigated in this thesis – consensus and MUTEX. For the consensus problem, different system models are also introduced.

Chapter 4 deals with the time efficiency in achieving consensus. We first introduce the Look-Ahead technique and then describe our fast consensus protocol with the Look-Ahead technique. The proposed protocol also has the Round-Zero-Degradation property.

Chapter 5 handles the message efficiency of consensus protocols. First, a hierarchical consensus protocol for mobile ad hoc networks is presented, where all MHs are grouped into clusters with predefined clusterheads. Using a dynamically clustering approach, we define the new oracle Δ for consensus, and then propose another hierarchical consensus protocol based on Δ.

Chapter 6 studies the consensus problem in dynamic mobile systems with dynamic join and leave hosts and no pre-defined number of hosts. We present the design of an eventual leader protocol for dynamic infrastructured mobile networks.

Chapter 7 describes a permission-based MUTEX algorithm, which is the first of the same kind for mobile ad hoc networks.

Finally, Chapter 8 concludes the thesis with discussion on directions of our future work.

Chapter 2. Mobile Computing Environment

Mobile computing is a new computing paradigm of distributed systems. Same as traditional wired distributed systems, the distributed coordination problems are still at the core of mobile computing. On the one hand, the applications, e.g. atomic broadcast and file replication, where coordination problems arise from, are still desirable in mobile systems. On the other hand, many new applications and scenarios, e.g. wireless channel allocation [86], emerge in mobile computing. Unfortunately, compared with traditional wired systems, the constitution of mobile systems is more flexible and mobile hosts are more loosely coupled, which make the design of distributed coordination algorithms more challenging.

Mobile computing is distinguished from traditional distributed computing in many ways. The most important ones are the underlying networking infrastructure, the characteristics of the systems, and applications built on the network. In this chapter we first introduce various kinds of mobile networks and their characteristics, and then discuss the new challenges brought to the design of distributed algorithms.

2.1. Mobile Network

"Mobile network" is a network that consists of mobile hosts. Mobile networks often use wireless communication, so in this thesis we use "wireless network" and "mobile network" interchangeably. At present, there are many different mobile networks proposed, such as Personal Communication Systems (PCS), Wireless LAN, Wireless MAN, Paging Network [85][125][157]. The architectures of all the mobile networks can be classed into two classes: infrastructured networks and ad hoc networks [8][114][115][157].

2.1.1. Infrastructured Network

An infrastructured network consists of two distinct sets of entities: a large number of mobile hosts (MHs for short) and relatively fewer but more powerful mobile support stations (MSSs for short). Figure 2-1 gives an example network architecture

for infrastructured mobile networks. MSSs are interconnected using a wired network, while MHs are connected to MSSs using wireless communications.

Each MSS is in charge of a cell. A cell is a logical or geographical coverage area under a MSS. Each MH that has identified itself with a particular MSS is considered to be local to the MSS. A MH can directly communicate with a MSS (and vice versa) only if the MH is physically located within the cell serviced by the MSS. MHs are able to connect to the static segment of the network from different locations at different times. Consequently, the overall network topology changes dynamically as MHs move from one cell to another. However, at any given instant of time, a MH can belong to only one cell and its current cell defines the MH's "location".

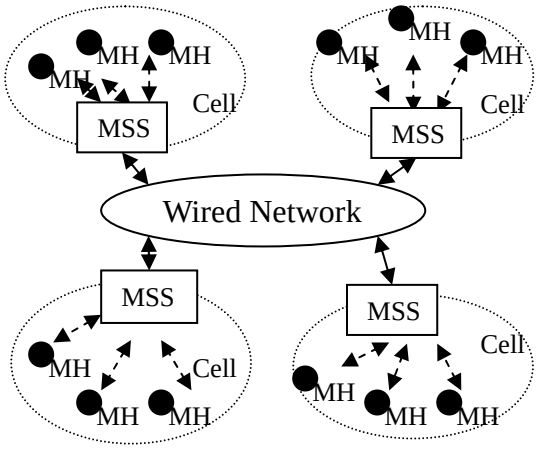

Figure 2-1 The infrastructured network

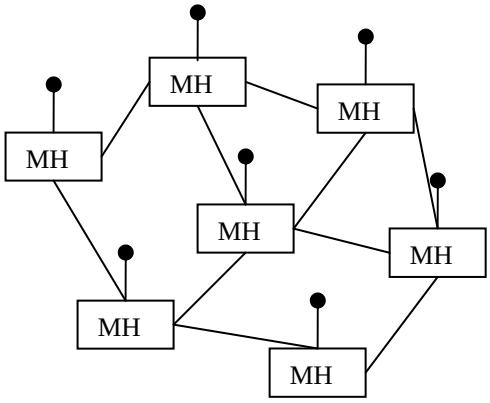

Figure 2-2 The mobile ad hoc network

2.1.2. Ad Hoc Network

As shown in Figure 2-2, a mobile ad hoc network (MANET) consists of a collection of autonomous MHs communicating with each other through wireless channels. Whether two hosts are directly connected is determined by the signal coverage range and the distance between the hosts. Each host is a router and the communication between two hosts can be multiple-hop. Both link and host failures may frequently occur. The topology of a MANET can dynamically change due to the mobility of MHs and link or host failures.

Compared with infrastructured networks, MANET provides greater flexibility at the expense of a larger overhead that stems from the use of broadcasting which is

typically employed to locate MHs. Such overhead also limits the scalability of MANETs. With the help of MSSs, infrastructured networks can reduce the overhead involved in location management and updating. Moreover, MSSs have fewer constraints in resource and processing power, so computation and coordination should be executed on MSSs as much as possible in order to reduce the load of MHs.

2.2. Characteristics of Mobile Network

Mobile networks have fundamentally different properties from wired networks in aspects of communication, mobility and resource constraint [63][84][138][148][164].

2.2.1. Network Communication

Since MHs access information through wireless links, the effect of wireless communication is probably the most prominent in a mobile computing environment. Wireless communication faces more obstacles than wired communications, because the surrounding environment interacts with the wireless signal, which may block the signal path and introduce noise and echoes. Wireless connections can be lost or the bandwidth is degraded due to the movement of users or the dynamical variation of the number of users. The limitation of power is another factor that effects the wireless communications. As a result, the wireless communication is characterized by lower bandwidth, higher error rate, and more frequent spurious disconnection. The challenges of wireless communication are follows:

- **Disconnection and Doze**

In mobile networks, there are two types of disconnections: “accidental disconnections” and “voluntary disconnections”, which have different effects on the design of mobile systems. An accidental disconnection refers to the disconnection caused by network failures. Because wireless communication is so susceptible to network failure, disconnections occur more frequently and unpredictably. On the other hand, a voluntary disconnection is predictable. To reduce the consumption of power and other resources, a MH may voluntarily disconnect from the network. Because such a disconnection does not result from failures, the MH can inform the

system before the disconnection occurs and execute a disconnection protocol if necessary. For example, the detached participant can prefetch necessary data to work in a stand-alone mode, and can offload the data or state information, pertaining to an ongoing algorithm execution, to MSSs or other MHs so that the distributed algorithm can continue.

The doze mode is also a voluntary operation, by which a MH can reduce power consumption. In doze mode, the clock speed is reduced and no user computation is performed. The host simply waits passively to receive messages sent to it. When such a message arrives, the host is waked up and resumes its regular mode. Therefore, a MH in doze mode is reachable form other hosts, which is different from a disconnected host.

- **Low Bandwidth and High Bandwidth Variability**

Bandwidth is the most critical aspect of wireless communications. Compared with wired networks, the bandwidth of a wireless network is far lower because of the physical limitation of transmission devices. Another related problem is bandwidth variation. The bandwidth can be affected by the landform and noise. Some MHs, e.g. laptops, may use both a wired link and a wireless link, so the bandwidth can vary from one to four orders of magnitude. Fluctuant traffic load can also have effect on the bandwidth to a certain extent.

- **Asynchronous Communication**

There are two basic types of communication paradigms in a distributed system: “synchronous communication” and “asynchronous communication”. Synchronous communication is easy to handle. However, because of frequent disconnections and low bandwidth with high variability, synchronous communications between mobile hosts become unpractical. The two peers of communication may not connect at the same time. Therefore, asynchronous communication is often assumed for mobile environments.

- **Heterogeneous Network**

In contrast to most stationary computers, which stay connected to a single network, mobile hosts encounter more heterogeneous network connections with different transmission speeds, protocols or even different networks. For example, when a MH switches from a cellular coverage in a city to satellite coverage in the country, it encounters two different networks.

- **Security Risks**

Creating a connection to a wireless network is much easier than that to a wired network, so the security of wireless communication can be much more easily compromised. Secure communication over insecure channels is accomplished by encryption, which can be done in software or, more quickly, by specialized hardware.

2.2.2. Mobility

Mobility is about moving around. In mobile computing, there are generally two forms of mobility: device or terminal mobility and user mobility [40][153]. Some people extend the concept of mobility to resource constraint [20].

- **Terminal Mobility**

Terminal mobility refers to both the ability of a terminal to access resources and services while in motion, and the capacity of connecting at different points of attachment in new hosting networks. Terminal mobility may be the most directly meaning of “mobile environment”.

Terminal mobility has two aspects effect: physical space and connection space. Movement in physical space means the change of physical location, which affects the quality of connection, e.g. poor connectivity or even disconnection. The network must reconnect the user with respect to new location to adapt the movement. Data and file operations should also be adapted to support disconnection file operations.

Connection space refers to the links that connect between various computer platforms. The movement in this space corresponds to selecting route between links, selecting a specific type platform and other configuration information, such as local name server, time zone.

- **User Mobility**

User mobility refers to the ability of a user to access the mobile system from any terminal at any place with a uniform vision of her/his working environment. To handle user mobility, mobile computing systems must provide mechanism to authenticate user accesses and organize the working environment according to profile information and currently required services. Compared with the research of terminal mobility, the research of user mobility is lagging behind, but is attracting more and more attention.

2.2.3. Resource Constraint

Due to the limited size, MHs, e.g. portable computers, PDAs or SmartPhones, consequently have limited resources.

- **Low Power**

Generally, batteries are the major power source of a MH. Due to the constraint of size, the life of batteries is usually relative short. Therefore saving power is a very important consideration for both hardware and software designers. Though reducing the voltage and clock frequency of chips, power can be saved. Applications can conserve power by reducing their appetite for computation, communication and memory.

- **Limited Processing Power**

The CPU or process of a MH is much slower than that of a desktop computer. Therefore, software designers must reduce the computation and other operations as much as possible to lower the load of CPU.

- **Small Storage Capacity**

Coping with limited storage is not a new problem. Solutions include compressing files automatically, accessing remote storage over the network, sharing code libraries, and compressing virtual memory pages. Reducing the size of code, for example using interpretive script languages instead of executing compiled object codes, is one efficient method to solve this problem.

- **Risk to Data**

Making computers portable increase the risk of physical damage, unauthorized access, loss, and theft. Breaches of privacy or total loss of data become more likely. Minimizing essential data kept on board, encrypting data stored on disks and memory cards, and making backup copies can reduce the risk of data.

- **Small User Interface**

The size constraint on a portable computer results in a small interface, including display and input. Adapted window environments and analog input technologies, e.g. handwriting recognition, voice recognition, would help solve this problem.

2.3. New Challenges for Distributed Coordination

The challenges of mobile computing are caused by the constraints stemmed from communication, mobility and portability as described in the previous section. These constraints necessarily make the distributed software systems for mobile computing different from those for traditional ones. In this section, we discuss the main challenges in designing distributed algorithms.

2.3.1. Reducing Message Cost

To cope with low bandwidth, in both hardware and software scope, many treasures can be taken to improve the bandwidth per user, such as overlapping cells on different wavelengths, reducing transmission ranges, and compressing/buffering the data to be exchanged. However, for a designer of distributed algorithms, the only way can cope with the bandwidth constraint is to reduce messages to exchange.

Besides saving bandwidth, another great benefit of reducing messages is saving power. Every operation needs consume power, so fewer messages means less power consumption. A useful characteristic of mobile communication is that the costs of transmission and reception are asymmetric. The former requires about 10 times as much power as that of reception. Therefore, power can be saved by substituting the transmission operation for a reception one.

To sum up, reducing messages is much more imminence in mobile environments than that in traditional distributed environments.

2.3.2. Handling Topology Change

Many distributed algorithms rely on logical topologies or structures, such as a ring, tree or graph [16]. Such a topology can provide a certain degree of "order" and predictability to the communication among the hosts. Messages exchanged within such topologies follow only selected paths. Now, in a mobile network, the topology is no longer stable due to the frequent movements or disconnections of MHs. Consequently, the corresponding topology information kept by the hosts must be updated to reflect the movements or disconnections, which results in additional message traffic and even search overhead (to locate a MH that moved away).

Therefore, in mobile environments, the cost and benefit of using a topology among MHs must be carefully balanced. In infrastructured systems, a good approach is to impose a logical topology upon MSSs [16][128], which can obtain similar benefit as imposing a logical topology upon MHs, but the cost of maintaining a logical topology is shifted to MSSs. In MANETs, this problem is more difficult to deal with. Constructing the topology dynamically/on-the-fly is a good solution [17].

For MANETs, a more severe effect of the topology change is partitioning. Because of mobility and failures, a MANET may be partitioned into two or more disconnected sub-networks. Partitions may cause a distributed algorithm blocked or even failed. How to handle partitions should be considered seriously.

2.3.3. Reducing Computation

Compare with other constraints, power constraint has much more universal effect on mobile computing. Because any computation and will consume power, reducing computation load of MHs becomes a new consideration in mobile environments. There are two ways to do this: reducing the absolute computation task or shifting the load to the possibly existing MSSs.

2.3.4. Handling Disconnection

As described before, there are two types of disconnections. Accidental disconnections are unpredictable, so an algorithm must have fault tolerance

mechanism to reduce the effect of such disconnections. The more autonomous a MH is, the better it can tolerate such a disconnection. If possible, the computation and communication should be done within the static segment of the network to the extent possible [16].

Compared with accidental disconnections, voluntary disconnections can be handled more easily because a MH can predict such a disconnection. When a MH wants to disconnect from the network, it can offload the data and other information needed by other hosts to the MSS (in an infrastructured network) or some other MHs (in a MANET) and executes a disconnection protocol before the disconnection takes place [146].

Another effect of disconnection is the change of the topology of the mobile system [16][128]. This will be discussed in Section 2.3.3.

2.3.5. Handling Dynamism of the System

Here, “dynamism” refers to the change of the constitution of a system. The classical system model for distributed systems is characterized by the following attributes. The system is made up of *n* hosts (processes); *n* is fixed and known by each process; no two processes have the same identity; the whole set of identities is known by each process.

It is worth notice that (since the very early of the eighties) this static model has been questioned by theoreticians interested in the computability power of distributed systems. Their efforts were focused on the following fundamental question [13]: “How much does each process in a network need to know about its own identity, the identities of other processes, and the underlying connection network, in order to be able to carry out useful functions?” This research direction has given rise to notions such as “local knowledge” vs. “global knowledge” [33], anonymous networks, sense of direction [62], etc.

The advent of mobile systems consequently questioned the relevance of the static distributed computing model from a practical point of view. Dynamic systems allow

hosts (or processes) to dynamically enter and leave the system, which is more feasible and reasonable for mobile networks. It follows that no host can know how many hosts currently constitute the system. Roughly speaking, there is no global safe information on the whole system structure that can be used by the hosts.

Chapter 3. Background and Literature Review

In the past decade, a lot of attention has been attracted to solving distributed coordination problems in mobile wireless environments. In this chapter, we review existing work according to the problems addressed. Since our research is focused on the consensus problem and MUTEX problem, we first briefly describe works for other problems and then provide a detailed survey of existing consensus protocols and MUTEX algorithms.

For the leader election problem, several algorithms have been proposed in recent years. Park [121] proposed an algorithm for infrastructure mobile networks based on Garcia Molina's bully algorithm [68]. Like in other algorithms for infrastructured networks, MSSs act as proxies of MHs for reducing the work load of MHs. Other existing leader election algorithms are all designed for MANETs. Based on the well-known diffusion computation based termination detection algorithm by Dijkstra and Scholten [48], Vasudevan et al. [158] designed an election algorithm. Malpani et al. [102] developed an election algorithm based on the TORA routing protocol [122]. Hatzis et al. [78] proposed two leader election algorithms based on the geographical positions of the MHs. Nakano et al. [117] proposed two randomized algorithms for single-hop and two-hop single channel MANETs respectively. In these two algorithms, the leader is elected by the collision state of the channel in which the MHs emit signals randomly and synchronously.

Termination detection algorithms have been proposed for infrastructured mobile networks. Malpani et al. [102] developed an algorithm based on Dijkstra and Scholten's algorithm [48]. In [102], the MSS can help the MHs merge more than one message together so as to reduce the communication cost at MHs. Tseng and Tan [155] proposed another algorithm using a hybrid mechanism: the weight-throwing scheme [81][104][154] on the wired network side, and the diffusing computation

based scheme [48] within each wireless cell. Such a hybrid protocol can better pave the gaps of computation and communication capability between static and mobile hosts, so it is scalable to large distributed systems.

For deadlock detection and logical time, to our knowledge, there is no solution proposed for mobile environments up to now.

For the snapshot problem, we can find two algorithms. Based on the Chandy-Lamport algorithm [32], Sato et al. [137] designed a snapshot algorithm for infrastructured networks. The main mechanism in [137] is the same as that in [32]. The algorithm handles the change of connections based on the information piggybacked on handoff messages, called “IPH” (Information Piggybacked with a Handoff message). Agbaria and Sanders [3] proposed another algorithm based on the Chandy-Lamport algorithm. Considering the FIFO channel is not feasible for channels between MHs, in [3], the authors made use of MSSs to help ensure the message order between a marker message and application messages.

For the transaction problem, much work has been done [95][99][141][142][163], including new models for mobile environments and new protocols.

3.1. Consensus Protocols

Since the design of consensus protocols are closely related to the underlying system model, we first describe the system models for consensus and then introduce existing consensus protocols.

Figure 3-1 shows a classification of distributed system models for solving the consensus problem. Three aspects of a system are involved in developing a consensus protocol: failure model, synchronicity model, and communication channel model.

3.1.1. Failure Model

There are several widely used models for host failures. They can be classified in terms of their “severity” [77][151]. One failure is less severe than another if the faulty behavior allowed by the former is a strict subset of that allowed by the latter.

According to such a classification, the least severe one is *fail-silent* or *crash* failure model where a process fails by crashing when it stops computing sending any message.

A more severe type of failure is the *omission* failure in which the process fails to response. Under this model, there are further two different cases: *receive-omission* failure and *send-omission* failure. In the receive-omission failure, a process may have not gotten the message sent to it. Such a case may happen, e.g. when the connection between two processes has been correctly established but no one listening to the incoming message. Likewise, a send-omission failure happens when the process has done its work but somehow fails in sending a response. Such a failure may happen, e.g. a send buffer overflows.

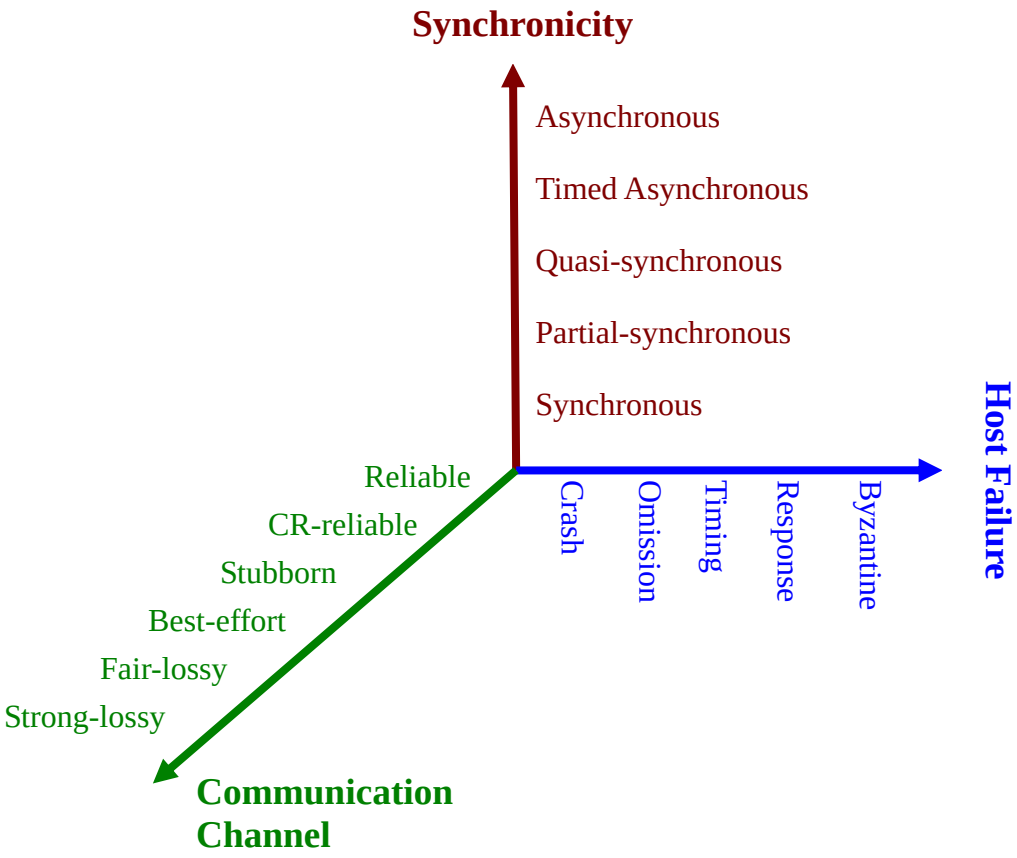

Figure 3-1 The system model for consensus

Timing failures occur when the message arrives too late or too early. A response failure occurs when the response message is incorrect. Two kinds of response failures may happen. In the case of a *value* failure, a wrong reply is provided to a

request. For example, a search engine returns results with no relation to the key words. The other one is known as the *state transition* failure. For example, if a server host receives a message that it cannot recognize and no measure has been defined to handle such a message, a state transition failure happens.

The most severe failure is the *arbitrary* failure (also called Byzantine failure), typical of a process exhibiting an absolutely arbitrary behavior (e.g. sending wrong and conflicting messages to different processes, or arbitrarily changing its state).

3.1.2. Synchronicity Model

The synchronicity dimension defines how closely components in the systems can be synchronized. The two extremes of this property are *synchrony* and *asynchrony* [61][70]. In a synchronous model, three assumptions should hold:

1) There is a known upper bound on the local clock drift;

2) There is a known upper bound on the processing speed;

3) There is a known upper bound on the message delivery delay.

Based on these properties, failure detection is easy to perform by using simple mechanisms like timeouts, and then the consensus problem becomes easy to solve. However in a real distributed system, it is difficult to guarantee the above bounds. Therefore, the asynchronous model attracts more attention because of its practicability. Different asynchronous models related to consensus have been proposed.

3.1.2.1. Asynchronous Model with Oracles

In an asynchronous system, there is no bound on the clock drift, process speed, or message delivery delay, which makes the consensus problem difficult to solve. Fischer et al. [61] proved that the consensus problem is unsolvable in an asynchronous system even if only one process can crash.

To circumvent this impossibility, three types of oracles have been proposed: the *random number generator* [22], the *eventual leader oracle* [31] and the *unreliable failure detector* (FD for short) *oracle* [31]. An oracle is an abstract tool that provides some kind of information about the state of the system.

Unreliable FDs are introduced by Chandra and Toueg [31]. In their work, every pair of processes is connected by a reliable communication channel, and the processes can fail by only crashing. A FD gives (possibly incorrect) hints about which process may have crashed so far. It consists of several modules, each of which is local to a process and periodically consulted by the corresponding process. Each module produces a list of processes suspected to be crashed. The modules are intrinsically unreliable. They can make mistakes, so the lists dynamically change during the computation (and it is possible for two or more lists to be different at the same time).

FDs can be classified according to their accuracy and completeness properties. The accuracy property restricts the mistakes a FD can make, while the completeness represents the capacity of suspecting an actually crashed process. More precisely, two completeness properties and four accuracy properties were defined:

Strong Completeness: eventually every crashed process is permanently suspected by every correct process;

Weak Completeness: eventually every crashed process is permanently suspected by some correct process;

Strong Accuracy: no process is suspected before it crashes;

Weak Accuracy: some correct process is never suspected;

Eventual Strong Accuracy: there is a time after which correct processes are not suspected by any correct process;

Eventual Weak Accuracy: there is a time after which some correct process is never suspected by any correct process.

Table 3-1 Eight classes of FDs

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	Perfect P	Strong S	Eventually Perfect $\Diamond P$	Eventually Strong $\Diamond S$
Weak	Q	Weak W	$\Diamond Q$	Eventually Weak $\Diamond W$

Each pair of accuracy and completeness can define a detector class. Totally there are eight classes as shown in Table 3-1. Interestingly, the eight classes are not

independent of each other. “Reducibility” is introduced to demonstrate the relationships among them [31]. Intuitively, a detector *D*’ is *reducible* to another detector *D* means that *D* can emulate *D*’, so any problem that can be solved using *D*’ can be solved by using *D* instead. Obviously, *D* must provide at least as much information about failures as *D*’ does. Therefore, we can say that *D*’ is *weaker* than *D*.

A concept related to “reducible” is “equivalent”. If *D* is reducible to *D*’ and *D*’ is reducible to *D*, it is said that *D* and *D*’ are equivalent. If *D*’ is reducible to *D* but *D* and *D*’ are not equivalent, *D*’ is said “strictly weaker” than *D*. The relationships among all the classes are shown in Figure 3-2. It is important to notice that each class of detectors with the weak completeness as shown in Table 3-1 is equivalent to the corresponding one with the strong completeness. This enables people to focus on the four classes with strong completeness only.

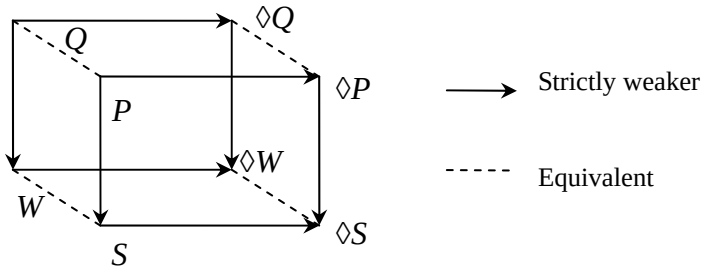

Figure 3-2 Relationships among failure detector classes

Interestingly, Chandra, Hadzilacos and Toueg [30] have proved that to solve the consensus problem, any FD has to provide at least as much information as ◊*W*. Thus, ◊*W* is indeed the weakest FD for consensus in asynchronous systems with a majority of correct processes.

The failure detector approach is particularly attractive. FDs are not defined in terms of any particular implementation (involving network topology, message delays, local clocks, etc.) but in terms of abstract properties (related to the detection of failures) that allow us to solve problems despite process crashes. The FD approach allows a modular decomposition that not only simplifies the consensus protocol design but also provides general solutions [31][82][83][139]. A protocol can be designed and proved correct based on only the properties provided by a FD class.

Therefore, this protocol depends only on a well defined set of abstract properties rather than low-level parameters. The implementation of a FD of the assumed class can be addressed separately.

However, not all the eight classes of FDs are helpful for consensus. It has been reported that the implementation of FDs of with the *strong accuracy* or *weak accuracy* is as difficult as the consensus problem itself [93], which claims that the consensus protocol based on the FDs of class *P*, *Q*, *W* or *S* are unprofitable. Even for the other four classes, based on the impossibility result in [61], it is also impossible to implement them in a purely asynchronous system [93]. Therefore, existing implementations of FDs are all based on models with synchrony in some degree [23][31][59][93].

The eventual leader oracle, usually denoted by Ω, was also proposed by Chandra and Toueg [31]. An eventual leader provides the processes with a leader primitive *leader*() that outputs a process *id* if it is invoked. Ω satisfies the following *eventual leadership* property:

Eventual leadership: Eventually, all invocations of Ω return the same *id*, and that *id* is the identity of a correct process.

Such an oracle is not very powerful, in the sense that there is no knowledge on when a correct leader is elected. Before this occurs, several distinct leaders (possibly conflicting) can co-exist. On the other hand, the leader oracle is powerful because it is possible to solve the consensus problem in asynchronous distributed system equipped with such a "weak" oracle [30][110]. It has also been shown that, as far as failure detection is concerned, Ω and $\Diamond S$ have the same computational power in asynchronous distributed systems prone to process crashes [30][39]. This is why Ω is sometimes viewed as one special type of FD.

A random number generator oracle [22] provides each process p_i with a function *random*() that outputs a binary value randomly chosen. Basically, the primitive *random*() outputs the value with some probability. Since there is no property defined for the output of a random number generator, it cannot be used to solve the

consensus problem deterministically, which is different from the other two types of oracles. Its main advantage lies in the robustness of the resulting consensus protocol: the behavior of such a protocol does not depend on how the system actually behaves.

3.1.2.2. Partial Synchrony Model

A synchronicity model between asynchronous model and synchronous model is the “partial synchrony” model. Analyzing the impossibility result in [61], there appears to be three different types of asynchrony:

Process asynchrony: a process may "go to sleep" for arbitrarily long finite amounts of time while other processes continue to run;

Communication asynchrony: no prior bound exists on message delivery time;

Message order asynchrony: messages can be delivered in a different order from the one in which they have been sent.

Investigating the influence of the different types of asynchrony, Dolev et al. [49] found that it is not necessary to have all the above types of asynchrony to obtain the impossibility result. This observation inspires the definition of the partial synchrony model [53].

Te partial synchrony model is originated relaxing the requisite of synchrony for processes and/or communications. According to this work, calling *D* and *F* the upper bounds on message transmission and on relative clock speeds of processes respectively, a partial synchrony can be caused by two conditions:

– The bounds exist but are not known;

– The bounds are known, but they hold after some unknown time.

In the former case, the system is de facto synchronous, so the impossibility result does not hold. The problem is to manage the messages exchange without the knowledge of the real values of *D* and *F*: using non correct values for these bounds will obviously affect correctness or performance of the protocol. In the latter case, an instant of time, called Global Stabilization Time (GST), is supposed to exist such that the bounds are valid from GST on.

3.1.2.3. Quasi-synchronous Model

Based on some observations on the real-time system, Verissimo and Almeida [9][10][11][12] proposed the quasi-synchronous model. The respect of all the timing constraints is mandatory when life-critical applications are considered. However, there are other real-time applications where, despite the need for dependability, it is acceptable to eventually miss some of the timing constraints (assuming to achieve the most important ones). Using Almeida and Verissimo wording, a system is synchronous if there are:

P_1. Bounded and known processing speeds;

P_2. Bounded and known message delivery delays;

P_3. Bounded and known local clock rate drift;

P_4. Bounded and known load patterns;

P_5. Bounded and known difference among local clocks.

Consequently, a quasi-synchronous system is defined as follows:

D_1. It can be defined by properties P_x;

D_2. There is at least one bound where there is a known probability (>0) that the bound assumption does not hold;

D_3. Any property can be defined in terms of a series of pairs.

A quasi-synchronous system can be viewed as a synchronous system in which the absolute bounds on messages transmission delays, local clock drift rates and process execution times are far away from those observed during the normal operative mode. Therefore, it is more convenient to use different values, even if the coverage of such assumptions is not equal to one [129].

3.1.2.4. Timed Asynchronous Model

The basis of the definition of the timed asynchronous model is the consideration that existing fault-tolerant services for asynchronous distributed systems are usually timed. The specification of the services not only describes the states transitions and the outputs in response to invocations of operations, but also the time interval within which these transitions have to complete [42].

The timed asynchronous system model is characterized by following assumptions:

i) All the services are timed, so it is possible to associate some timeout whose expiration produces a performance failure;

ii) Processes communication with one another using an unreliable datagram service with omission and/or performance failure semantics;

iii) Processes have crash/performance failure semantics [41];

iv) All processes have access to private hardware clocks that run within a linear envelope of real-time;

v) No bound exists on the rate of communication and process failures.

The timed asynchronous model is asynchronous in the sense that it does not require the existence of upper bounds for the message transmission and scheduling delay. However, the access to local hardware clocks and the definition of timeouts enable us to define the performance failure as the failure that occurs when an experienced delay is greater than the associated timeout delay.

3.1.2.5. Comparisons of Synchronicity Models

All the considered models try to overcome the impossibility result in [61] by strengthening the asynchronous system model, i.e. trying to add a sufficient amount of "synchrony" to the system in order to allow the solvability of the consensus problem. Obviously, there are some differences between different models, because the assumptions that they rely upon are different.

The first comparison can be made between the partially synchronous system and the timed asynchronous system. As described before, the former model assumes the existence of bounds on the process speed and message transmission delay with coverage equal to 1. On the contrary, the timed asynchronous model only assumes a bound on clock drift rate (with a coverage equal to 1) and makes no assumption on the load pattern or message transmission delay. The timed asynchronous system model has some points in common with the partially synchronous system model: the Global Stabilization Time of the partially synchronous system model reminds the concept of stability of the timed asynchronous system, but, while the Global

Stabilization requires that the system is not affected by any (timing or crash) failure after a certain instant, the stability of a timed asynchronous system is valid only for bounded time intervals.

It is interesting to compare the partial synchrony model with the notion of an unreliable FD. For every partial synchrony model we considered, it is easy to implement an eventual perfect FD (a FD that satisfies strong completeness and eventual strong accuracy). In fact, one could implement such a FD with an even weaker model of partial synchrony: one in which the bounds on message transmission delay and process speed exist, but they are unknown and hold only after some unknown time.

It is more difficult is to compare the asynchronous model with oracles with the timed asynchronous model. In [58], the impossibility to implement a perfect FD in a timed asynchronous system has been proved. The main difference between the two models is in the philosophy of the design of the system. FDs hide to higher abstraction levels all the aspects related to the time of the fault-tolerant distributed computation. This can constitute a problem if the abstraction levels are more than two. In such a situation all the timeouts are used in each level because a level that depends on another one has to be able to detect its failures [41]. For this reason the meaning of timeouts change in correspondence of the levels they are associated to (usually the higher is the level, the greater is the timeout and the more severe is its violation).

Let us consider now the difference between the partial synchrony model and the quasi-synchronous model. In the latter, it is necessary to define a given bound for the message delay (although the bound does not hold with probability 1), while in the former, this bound is unknown or it holds after an unknown instant.

For the consensus problem, the asynchronous model with oracles has attracted most attention. Our study in this thesis is also based on this model.

3.1.3. Communication Model

The dimension of communication channel can be defined using three properties: reliability, creation, and duplication. All the channels listed in Figure 3-1 have the no-creation property and no-duplication property [71]:

Property 1 (no-creation): *if process q receives a message m, then some process p has sent m to q.*

Property 2 (no-duplication): *every message m sent by any process p is received at most once*.

The difference among different channels lies in the reliability, which is defined as follows [71][97][120][131]:

Property 3 (Reliable): *if process p send a message m to process q, and q does not crash, then q eventually receives m.*

Property 4 (CR-reliable): *if process p sends a message m to process q, and neither p nor q crashes, then q eventually receives m.*

Property 5 (Stubborn): *if process p sends a message m to process q, neither p nor q crashes, and p indefinitely delays sending any further message to q, then q eventually receives m.*

Property 4 (Best-effort): *if process p sends a message m to process q, and neither p suspects q nor q suspects p, then q eventually receives m.*

Property 5 (Fair-lossy): *if process p sends an infinite number of messages to process q, and q does not crash, then q receives an infinite number of these messages.*

Property 6 (Strong-lossy): *if process p sends messages to process q, there is no guarantee on the delivery of any message, i.e. all these messages may be lost.*

The definitions of these channels are self-explanatory except the stubborn channel. Basically, Property 5 says that, if a correct process *p* sends message *m* and afterwards is able to indefinitely delay the sending of any further message then *q* eventually receives *m*. This does not mean that *p* is not allowed to send any new message *m'* to *q* after sending *m*. *m'* can be sent when either *m* has been received or

m may be never received. Intuitively, a stubborn channel guarantees the delivery of the last message transmitted though it.

In general, the degree of reliability of these channels gradually decreases from the reliable channel to the strong-lossy channel. However, for the best-effort channel and fair-lossy channel, it is hard to say which one is more reliable. The reliability of a best-effort channel is directly dependent on the failure detection mechanism. When a false suspicion occurs, no reliability is provided at all.

A channel with higher reliability can be implemented using a channel with lower reliability by message retransmitting mechanism [19][71]. For example, a CR-reliable channel can be implemented using a fair-lossy channel by periodically retransmitting all previous messages. Of course, this may require unbounded buffer space. On the contrary, a stubborn channel requires only finite buffer space if it is implemented using a fair-lossy channel. This is in fact the main motivation of proposing the stubborn channel [71].

In consensus protocols, people usually assume reliable channels in the system [31][51][83][110]. Interestingly, all such protocols are still correct in the environment with CR-reliable channels. Guerraoui et al. [71] and Olivira et al. [120] have proposed some protocols based on the stubborn channel. In fact, the consensus problem is solvable with fair-lossy channels [50].

3.1.4. Protocols for Traditional Fixed Environments

3.1.4.1. FD-based Protocols

Among all the three types of oracles, FD has attracted the most attention. Up to now, all FD-based consensus protocols use a FD of class $\Diamond S$ or S. However, it has been reported that the implementation of S is as difficult as the consensus problem itself [93], which claims that the solutions based on S are unprofitable. Therefore, only the protocols based on $\Diamond S$ are described here.

Chandra et al. [31], Hurfin et al. [83] and Schiper [139] have proposed consensus protocols with the same assumption that a majority of all the *n* processes are correct, which has been proved a necessary condition for solving the consensus problem in

asynchronous systems [31]. All of the protocols adopt the rotating coordinator paradigm and are executed in asynchronous rounds. A round is usually divided into several phases. Each process keeps an estimate of the final decision value.

During each round, a coordinator process, predefined using some deterministic function e.g. (r mod n)+1, tries to impose its own estimate on others by sending proposal messages to all processes. On the reception of the proposal from the coordinator, a process updates its own estimate and sends the echo message to some or all processes. Based on the echo messages received during a round, a process updates its estimate and checks if it can make a decision.

These protocols mainly differ in the message exchange pattern in a round. The protocol presented in [31] adopts a centralized message exchange pattern, while the protocols in [83][139] use the fully distributed pattern. The latter two protocols differ in the way they cope with failures and mistakes made by the FD. More precisely, the protocol in [83] "trusts" the FD, while the protocol in [139] does not.

Hurfin et al. [82] proposed a versatile FD-based protocol (denoted by HMR). HMR has two orthogonal versatility dimensions: the class of the FD (class S or $\Diamond S$) and the message exchange pattern. Since the HMR protocol is the basis of our proposed consensus protocols in this thesis, we introduce it in more details.

There are totally n processes in the system, and f of them can crash, where $f<n/2$. Same as other FD-based protocols, HMR adopts the coordinator rotating paradigm and is executed in asynchronous rounds. Each round of HMR is divided into two phases. In the first phase of a round r, the coordinator p_{cc}, defined by the function *coord(r)*, sends its current estimate est_{cc} to each process (including itself) with the proposal message $PROP(r, est_{cc})$. A process p_i waits for the estimate value from p_{cc} unless p_{cc} is suspected. When a $PROP(r, est_{cc})$ is received, p_i updates its own estimate value est_i and timestamp ts_i and then proceeds to the next phase.

In the second phase, the message exchange pattern is determined by two sets of processes, D and A. The set D, standing for *Decision-makers*, is the set of processes

that need to check the decision status, i.e. whether it can decide in the current round. *D* is defined with the following two requirements:

- *D* is deterministic, i.e. all the processes have the same *D* for the same round;
- *D* contains the current coordinator.

The set *A* stands for *Agreement_keepers*. Since different processes may decide in different rounds, *A* is used to ensure that once a value has been decided upon in a round by some process, no other value can be adopted as the decision value in later rounds. To maintain the agreement property, in the second phase of a round, each process in *A* needs to receive estimate values from other processes and update its estimate to the value with the highest timestamp. Since the coordinator of the "next" round will send proposals to others, it must have a "correct" estimate. Therefore, *A* can be any set of processes that includes the coordinator of the next round.

After entering the second phase of a round *r*, each process p_i sends an echo message *ECHO*(r, est_i, ts_i) to each process p_j in $D \cup A$. p_j waits until it receives echo messages *ECHO*(r, *, *) from no less than $n-f$ processes and then sets its estimate to the value carried by the *ECHO* message with the highest timestamp. A process p_d in set *D* checks the timestamps of the echo messages received. If p_d receives $f+1$ echo messages with a timestamp "*r*", p_d can decide. The number "$f+1$" ensures that at least one correct process knows the decision value. Once a process decides, it broadcasts the *DECISION*(*est*) message using a reliable broadcast mechanism to disseminate its decision value, and then stops participating in the protocol.

All above protocols rely on reliable communication channels between hosts. Protocols that can tolerate message losses are reported in [50][71][120]. The protocol in [50] is based on fair-lossy channels while the protocols in [71][120] adopt stubborn channels. Same as others, these protocols adopt the rotating coordinator paradigm and are executed in asynchronous rounds. However, to tolerate message losses, each host needs to periodically resend the latest message it has sent and, if some message from a higher round is received a host skips to some higher round (the next round or the round of the message received).

The difference between [50] and [71][120] lies in the message resending mechanism. In [50], the resending is undertaken by the consensus protocol itself, while in [71][120] this is delegated to the underlying communication channels. More precisely, the consensus protocol [50] implements stubborn channels implicitly. The protocol in [71] differs from that in [120] in the message exchange pattern and consequently the buffer size for resending messages.

3.1.4.2. Leader-based Protocols

Based on the leader oracle Ω, several consensus protocols have been proposed. The first leader-based consensus protocol is proposed by Lamport [90]. Based on a part-time parliament protocol used by residents of an ancient Greek island Paxos, Lamport devised an algorithm, named Paxos, to implement a highly available deterministic service by replicating it over a set of processes communicating through message passing. A consensus protocol named *Synod* is at the core of Paxos. Because the description in [90] is difficult to understand, efforts have been put to simplify and deconstruct it [24][46][89]. However, the Paxos algorithm is not dedicated for solving consensus so there is no explicit consensus protocol presented.

The *Synod* algorithm can only guarantee the *agreement* property and *validity* property of the consensus problem. To guarantee the *termination* property, two additional issues are involved. First, a leader oracle is assumed for determining the process to initiate the algorithm. Second, there should be a mechanism to determine when to initiate the above algorithm. If the algorithm is invoked by new leaders too frequently, it is possible that each run of the algorithm is ended by a run with a higher number. In [90], it is assumed that it is eventually possible to obtain the precise delay of a message and the time of processing a message, so as to estimate the duration of a run.

The protocols in [51][110] can guarantee all the three correctness properties. Like FD-based protocols, they are executed in asynchronous rounds, but there is no predefined coordinator for a round. Each round of the protocol in [110] consists of three phases. In the first phase, each process broadcasts its estimate value to others

and then waits for the values from the leader process (selected by the leader oracle). If such a value is received, the process updates its estimate value to the value received and broadcasts the new value received from its leader (if any) or the value "⊥" (a value that can never be accepted, which means that the sender failed to get a value from its leader). If a process received the same value v ($v \neq \perp$) from a majority of processes in the second phase, it broadcasts the value v in the third phase. Otherwise, it broadcasts the value "⊥". Finally, based on the messages received in the third phase, a process updates its own estimate and makes a decision if it receives the same value ($v \neq \perp$) from a majority of processes. The protocol in [51] uses a similar procedure but the first two phases are merged into one.

A uniform consensus protocol based on a leader oracle was proposed in [24]. The protocol described in [92] is based on a leader oracle named $\Diamond C$. Different from the commonly used Ω, $\Diamond C$ is defined by a completeness property and an eventual consistent accuracy property. It can be seen as an adaptation of the CT protocol [31].

3.1.4.3. Random Number Generator based Protocols

The first consensus protocol based on the random number generator is proposed in [22]. Same as other consensus protocols, it is executed in asynchronous rounds, but no process acts as a coordinator or leader in a round. A process updates its estimate with the help of a random number generator. Basically, the primitive *random*() outputs the value 0 (respectively 1) with probability 1/2, so [22] solves the binary consensus, i.e. the decision value is 0 or 1.

Ezhilchelvan et al. [55] proposed a multi-valued consensus protocol based on a random number generator. Each process first reliably broadcasts the value v_i proposed by itself and collects the values from others. Then, the processes execute asynchronous consecutive rounds until a decision is made. A round is made up of two communication phases. During the first phase of a round r, the processes exchange their own current estimates by broadcasting. If a process p_i discovers that a majority of estimates have the same value v, it updates est_i to v; otherwise, it updates est_i to ⊥, a value that cannot be decided upon. Then the processes enter the second

phase during which they exchange the new content of their est_i variables. If a process p_i receives the same value v such that $v \neq \perp$ from a majority of processes, it decides on v. Otherwise, it adopts any estimate value not equal to $\perp$ (if such a value is received) or a randomly selected value from the values received at the beginning of the execution (if all the values received in the second phase are $\perp$). Then p_i starts the next round.

3.1.4.4. Hybrid Protocols

Some hybrid consensus protocols [7][113] make use of FD and random number generator at the same time. The first attempt to build a general consensus framework is proposed in [108], which unifies a leader oracle, a random number generator oracle, and a failure detector oracle. Unfortunately, algorithms derived by instantiating that framework with a given oracle are clearly not as efficient as ad hoc algorithms devised directly with that oracle. Efficient consensus protocols with different oracles are presented in [51], where a specific protocol is given for each oracle. An efficient uniform consensus framework is presented in [73].

3.1.4.5. Summary

It is interesting to compare the protocols based on different oracles. A common inconvenience of the FD-based protocols is caused by the rotating coordinator paradigm. A consensus protocol cannot achieve consensus until there is a round coordinated by an unsuspected host. Because the hosts take turns to act as the coordinator of some round in a predefined order, hosts still have to execute a round that coordinated by a crashed and suspected host, which will delay the decision making. Leader-based protocols solve this problem by dynamically selecting rather than statically defining the leader host. One drawback caused by this change is that, in the same round, the leaders selected by different hosts may be different. To cope with this, the leader information must be exchanged in the all-to-all pattern together with the proposal messages [51][110], which is not so flexible as FD-based protocols in terms of the message exchange pattern. In random number generator based protocols, however, no coordinator/leader is needed, so it does not depend on the

behaviors of the system. The price to pay for this advantage is that such protocols cannot guarantee the termination deterministically.

3.1.5. Consensus Protocols for Mobile Environments

So far, very little work has been done on the consensus problem in mobile environments. Several protocols have been proposed for infrastructured networks. Based on the CT protocol [31], Badache et al. [15] proposed a consensus protocol for infrastructured mobile networks, which is denoted by BHM. In BHM, the decision value is a set of values proposed by at least α hosts (α can be determined by the application) instead of a single value. MSSs collect initial values from MHs and achieve consensus on behalf of MHs. The activity of a MSS is divided into three main subtasks:

i) To interact with its local MHs to collect their initial values;

ii) To interact with other MSSs to agree on a subset of proposed values;

iii) To interact with its local MHs to disseminate the final outcome.

Compared with CT, BHM has several differences. First, in CT, a host can change it estimate only when it adopts the value proposed by the coordinator, but in BHM a MSS can also change its proposed value if it gets new values from MHs. Second, once the MSS changes its value set, it sends a new proposal to the coordinator, so a MSS may send more than one proposal message in one round. After a MSS has got enough initial values and sent a positive acknowledgment to a coordinator, it behaves in a way similar to that of a process in CT.

BHM uses a simple handoff mechanism to handle the movements of MHs. The new and old MSSs change their MH lists and the MH sends its initial value to the new MSS if the value is never collected by the new MSS.

The protocol in [140] extends the BHM protocol by considering the dynamism of the set of MSSs. Due to the mobility of MHs, some cells may become empty, i.e. there is no MH in these cells. Using a group membership protocol, the MSSs of such cells are deleted from the set of MSSs executing the consensus protocol. The membership protocol and the consensus protocol are executed concurrently. Since

the group membership problem can also be solved by a consensus protocol [75], there can be two consensus protocols involved.

Both protocols in [15] and [140] rely on the help of MSSs. The principle is to shift the workload from MHs to MSSs. In MANETs, however, there is no MSS and all the work has to be done by MHs themselves.

Chockler et al. [37] developed consensus protocols for a special type of MANETs, single-hop MANETs, where the hosts are located within the communication range of each other. Their work focuses on message losses due to transmission collisions. Collision detectors are designed to monitor the communication media and detect transmission collisions. With the help of collision detectors, in a single-hop MANET can achieve consensus. To achieve consensus in a multi-hop MANET, the network is divided into non-overlapping grids, each of which is a single-hop sub-network. Single-hop consensus is first achieved within each grid and then each host gossips its grid consensus value. Finally, a host can decide after it has received a value for every grid.

Another consensus protocol for MANETs is reported in [160]. The authors developed several fault tolerant broadcast algorithms for MANETs and then apply these broadcast algorithms to the consensus protocol in [55]. As described before, the protocol in [55] is a randomized protocol relying on a random number generator so it can only probabilistically guarantee the termination property.

Both the consensus protocols in [37] and [160] are probabilistic with respect to their approaches of achieving a global consensus in MANETs. Of course, protocols for traditional networks can be used in MANETs, but they are not efficient in terms of the message cost, especially for large scale MANETs [123][166].

3.2. MUTEX Algorithms

Many algorithms have been proposed for the distributed MUTEX problem [21][27][145][159], but few of them are designed for mobile environments. We first

briefly introduce algorithms for traditional fixed environments and then review algorithms for mobile environments.

3.2.1. Algorithms for Traditional Fixed Environments

3.2.1.1. Token-based MUTEX Algorithms

The first token-circulating algorithm is proposed by Le Lann [40], where all hosts are logically organized in a ring and the token circulates following the ring. When the token is received by a host, it enters its CS if it is requesting, and after exiting CS, it sends the token to its successor in the ring.

To reduce the meaningless circulation of the token when no host wants to enter the CS, algorithms combining token-circulating and token-asking are proposed in [18] and [103]. A fault tolerant token-circulating algorithm is proposed in [107] which addressing the token loss detection problem using a time free approach. There are two tokens circulated simultaneously along the same ring while only one of them carries the privilege of entering the CS. These two tokens monitoring each other using a sequence number based mechanism to detect the token loss.

Ricart and Agrawala [134] proposed a broadcast based algorithm that requires at most *n* (the total number of hosts) messages to achieve MUTEX. The requesting host sends the request message to all other *n*-1 hosts and waits for responses. When the token holder needs to send the token, the successor is chosen in a circular manner if there is any pending request; otherwise, it keeps the token idly. Based on the idea in [133][134], Suzuki and Kazami [150] proposed an algorithm in which the queue of requesting hosts is piggybacked on the token. This queue is updated by a local queue of each visited host in an ascending order of *id* so as to ensure the liveness property.

The algorithm in [119] extends the algorithm in [150] by adding timeout-based fault tolerance. The algorithm can tolerate various failures and using timeout based mechanism, a lost token can be regenerated and duplicated tokens are eliminated.

Singhal [144] improved the performance of the Suzuki and Kazami algorithm by sending requests only to the hosts probably holding the token instead of all the hosts.

The knowledge of each host about the requesting hosts is passed together with the token, so a host can guess what hosts are the probable token holders using a heuristically method. The algorithm proposed in [34] is similar to the one in [144] except that different data structures are used to store the request information of hosts.

Tree-based MUTEX algorithms can be found in [35][47][79][116][130]. Different from broadcast based algorithms, such algorithms make use of the logical tree to forward requests for the token and pass the token to requesting hosts. The token holder is always the root of the tree. A parent host sends request on behalf of its children so as to reduce message cost. These algorithms differ in the mechanism of maintaining the tree and the request queue.

Raymond's algorithm [130] requires at most *O(LogN)* messages (*N* is the total number of hosts) to enter the CS. The tree is maintained by the logical pointers distributed over the hosts and directed to the token holder. A request message is routed to the root along the path of pointers from the requestor to the token holder. The token is sent back over the reverse path. The links passed by the token must be reversed so as to always point to the token holder. Based on Raymond's algorithm, Chang et al. [35] developed an algorithm that can tolerate link and host failures by maintaining multiple paths to search the token. The algorithm tries also to avoid cycles when the token returns to the requester along the reversal links. In [47], Dhamdhene and Kulkarni developed an algorithm which aims to eliminate the still remaining cycle in [35].

Different from Raymond's algorithm, in [116], the hosts are arranged in a dynamic logical tree, where the structure of the tree may be changed with respect to the request status of the hosts. No queue of pending requests is maintained by the hosts or the token. Such a queue is implicitly maintained by the state of each host using two variables: *LAST* and *NEXT*. *LAST* indicates the last host from which a request was received and the neighbor host in the path to the root to which this host needs to send a request message for its new request. *NEXT* indicates the host to whom the token will be granted after this host leaves CS.

Helary et al. [79] proposed a general tree-based MUTEX scheme, including an information structure and the associated algorithm. The information structure includes, in particular, a dynamic rooted tree structure logically connecting the hosts, and a behavior attribute *(transit* or *proxy)* dynamically assigned to each host. This general structure not only covers, as particular cases, several known algorithms, but also allows for the design of new ones that are well suitable for various topology requirements.

The graph-based MUTEX algorithms [80][118] are similar to those tree-based algorithms except that a directed acyclic graph (DAG for short) rather than a tree is used to pass the requests and token. In [80], a host requesting the CS sends requests to its neighbors and waits for the token. After completing the execution of the CS, the host finds the oldest request from its own pending queue, updates the time of that request in the token's array with its logical clock, and then sends the token through the return path. The algorithm [118] is similar to the tree-based algorithm [116] except that the hosts are arranged in a DAG rather than a tree.

3.2.1.2. Permission-based MUTEX Algorithms

As introduced before, two different types of permissions have ever been used in existing permission-based algorithms [145]. The R-permission is first proposed in the Ricart-Agrawala algorithm [133]. A host requesting the CS sends request messages to all other hosts. Requests for CS are assigned globally unique priorities, e.g. Lamport-like timestamps [91]. The receiver of a request grants permission to the requester immediately by sending a reply, if it is not requesting the CS or its priority is lower. Otherwise, it grants the permission after its own execution of the CS. A variant of the Ricart-Agrawala algorithm is proposed in [28], by remembering the recent history of the CS execution in order to reduce message cost. Singhal [143] proposed a dynamic R-permission based algorithm by dynamically change the set of the hosts to which a requesting host needs to send request messages.

Another type of permission is M-permission [98], where a requesting host needs to send requests to a quorum of hosts. Many quorum based MUTEX algorithms have

been proposed. Since our study does not adopt the M-permission, we do not introduce them here. A good survey can be found in [27].

3.2.2. MUTEX Algorithms for Mobile Environments

We first briefly describe existing algorithms and then present the comparison and analysis.

3.2.2.1. Mutual Exclusion Algorithms for Infrastructured Networks

The BBAI algorithm [16] focuses on handling the mobility of MHs. The authors proposed a two-tier structure to make full use of MSSs. A guiding principle of this paper is that "the computation and communication demands of an algorithm should be satisfied within the static segment of the system to extent possible." To do so, a logical ring is imposed on all the MSSs and the token visits each MSS in a predefined sequence. A MH that wants to access the CS needs to send a request to its local MSS. When the token visits this MSS, all pending requests at this MSS are serially serviced. Two strategies are adopted to handle the movements of MHs. One is to let a MH proactively inform the MSS about its updated location while the other lets the token holder search a MH before sending the token.

The LPRM algorithm [124] improved Raymond's algorithm [130] by introducing a *d*-level hierarchical logical tree. The main principle of LPRM is similar to that in [16]: letting MSSs act as the proxies of MHs. The tree is imposed on only the MSSs. A host sends a request to the token holder (i.e. the root of the tree) when it wants to enter CS. Because there are much less MSSs than MHs, the message cost is reduced compared with Raymond's algorithm. Although a *d* level tree may help reduce message cost, the overhead to maintain such a multi-level tree may be high.

Singhal [146] presented a MUTEX algorithm (denoted as MSM) based on the "look ahead" technique which can reduce message cost by enforcing the MUTEX only among those concurrently competing for CS in stead of all the hosts. The Ricart-Agrawala approach [133] is directly adopted to achieve MUTEX.

The basic idea of the "look ahead" technique is that if a host S_i informs S_j of its request status, then S_j automatically gets informed whenever S_j requests CS. On each host, there are two sets. The $Info\_Set_i$ includes the *id* of the hosts that S_i needs to inform if it has a request, and the $Status\_Set_i$ includes the *id* of the hosts that will inform S_i when they request for the CS. For each host, the following conditions must be satisfied (S is the set of all hosts):

1) $\forall S_i$: $Info\_Set_i \cup Status\_Set_i = S; \forall S_i$: $Info\_Set_i \cap Status\_Set_i = \emptyset$

2) $\forall S_i \forall S_j$: $S_i \in Info\_Set_j \Rightarrow S_j \in Status\_Set_i$

Through adjusting the two sets according to specific rules, each host can know what other hosts are concurrently requesting CS. Thus, the number of messages exchanged is decreased.

When a host wants to disconnect from the network, it offloads the current values of its data structures to its local MSS which will then act on behalf of the host in the execution of the MUTEX algorithm.

3.2.2.2. Mutual Exclusion Algorithms for MANETs

Several MUTEX algorithms for MANETs have been proposed. A good survey of such algorithms can be found in [21].

Baldoni et al. [17] presented an algorithm (denoted as RBVP) which aims at reducing the meaningless control messages when no host requests to access the CS. In the RBVP algorithm, the structure of the logical ring is computed on-the-fly, and there is a coordinator for each round. A host needs to send a request message to the coordinator when it wants to access the CS. If there is no pending request, the coordinator holds the token idly so as to avoid meaningless circulation of the token.

Walter et al. [161] proposed a token-asking algorithm, denoted by JWSK, which is derived from Raymond's tree-based algorithm [130] with the improvement to handle link failures caused by host mobility. This algorithm defines a DAG of token-oriented pointers, maintaining multiple paths leading to the token holder. Like in [130], requests are forwarded to the token holder along a path in the DAG and the token is delivered along the reverse path to the requesting host. When a host cannot

find a path to the token holder due to failures, it initiates the "update" procedure to find a new path. When a reverse path is broken, the token holder needs to search the requesting host before sending the token. In [162] a variant of the algorithm in [161] is presented to eliminate the overhead introduced by searching the requestor. Instead of that the token holder searches the hosts, a host needs to resend its request when it detects that there is a failure of an outgoing link.

Malpani et al. [101] proposed a parametric token-circulating algorithm with many variants. A dynamic logical ring is imposed on the MHs and the successor of a host is computed on-the-fly. By applying different polices to determine the successor, different variants are derived. Based on the Local-Recency (LR) policy, Chen et al. [36] proposed a MUTEX algorithm (denoted by YCJW) for MANETs. A ring is determined at the beginning of each round, and will not be changed until a new round begins. To guarantee the liveness property, YCJW requires that the topology to be static while the algorithm is converging. To tolerate token losses, one special host acts as a leader. The leader generates the token and sets timeout for it. Each token is marked with an *id*, which is equal to the round number. If a timeout happens, the leader would generate a new token. Through fixing the visit path for each round, the correctness is guaranteed even if there is more than one token in the system.

3.2.2.3. Comparison and Analysis

Now, let us compare different algorithms. There are three classical metrics for distributed MUTEX algorithms [145]:

Number of Messages Per CS Entry (*MPCS*): the average number of messages exchanged among the hosts for each execution of the CS.

Synchronization Delay (*SD*): the number of sequential messages exchanged after a host leaves the CS and before the next host enters the CS.

Response Time (*RT*): the time interval that a host waits to enter the CS after its request for CS arrives.

We also use these metrics to evaluate the performance of MUTEX algorithms, but new measures suitable for mobile environments are adopted. In traditional fixed

networks, the cost of each message is viewed as the same. However, in mobile environments, the communications among MHs become more complicated. There are both wireless links and wired links. The costs of messages sent through different links are also different and should be calculated differently. Similar as in [16], we use different measures for the cost of communications through wired and wireless links. The notations used in the performance analysis are as follows:

n: the number of MHs;

m: the number of MSSs;

C_f: a message between two fixed stations;

D_f: the time delay of C_f;

C_w: a message sent through wireless links. In an infrastructured network, it refers to a message between a MH and the MH's MSS. In a MANET, this refers to a point-to-point message between two MHs;

D_w: the time delay of C_w;

C_l: messages incurred to search a MH. it depends on the searching mechanism.

D_l: the time delay of C_l. Of course, it can be expressed by D_f.

Since the performance of a MUTEX algorithm usually depends on the load level of the system, two special load levels often been involved: *low load level* and *high load level*. Under low load levels, there is seldom more than one request for CS simultaneously in the system. On the contrary, under high load levels, there is always a pending request at each host, i.e. a host sends out a request as soon as it exits CS.

In mobile environments, there is a new important condition – mobility, which has important effect on the performance of MUTEX algorithms. Under high mobility levels, a MH changes its location very frequently, while under low mobility levels, a MH rarely does so.

Table 3-2 shows the performance of the algorithms described above[4]. The performance of BBAI algorithm varies strongly under different load levels. In

[4] Since the *SD* under low load levels depends on the interval of two requests for CS, we only analyze the *SD* under high load levels.

general, under low load levels, the performance is bad, which is caused by meaningless token circulation when there is no request at all. The performance of BBAI is also affected by the mobility level. If the "*inform*" mechanism is used, under high mobility levels, many informing messages are needed. However, if using the *search* mechanism, the performance of BBAI is not affected by the mobility level.

Table 3-2 Performance of MUTEX algorithms for infrastructured networks

		BBAI	LPRM	MSM
MUTEX Mechanism		Ring-based	Tree-based	Permission-based
Property		static	dynamic	dynamic
MPCS	Low load	$3C_w+O(m)C_f+ C_l$ * $3C_w+O(m)C_f+ k*C_f$ **	$3C_w+O(log^m)$ *$2C_f+C_l$	n*$(2C_w+C_f)$
	High load	$3C_w+C_f+ C_l$ $3C_w+C_f+ k*C_f$	$3C_w+2C_f+C_l$	$3n$*$(2C_w+C_f)/2$
SD	High load	$2Cw +C_f +C_l$ $2Cw +C_f$	$2C_w+C_f+C_l$	$2C_w+C_f$
RT	Low load	$3D_w +O(m)*D_f + D_l$ $3D_w +O(m)*D_f$	$3D_w+O(log^m)*2D_f+D_l$	$4D_w+2D_f$
	High load	$O(m)*D_f+O(n)*(2D_w+E+D_l)$ $O(m)*D_f+O(n)*(2D_w +E)$	$O(m)*2D_f+O(n)*(2D_w+E+D_l)$	n*$(2D_w+D_f+E)/2$
Contributions		Reducing message cost Handling topology change	Reducing message cost Handling topology change	Reducing message cost
Drawbacks		Additional mechanism to ensuring the fair access to token.	The overhead to maintain a multi-level tree.	No mechanism for disconnection
Notes		*: this is the *MPCS* value for search mechanism, and the same to *SD, RT* **: this is the *MPCS* value for inform mechanism, where *k* is number of movements after requesting, the same to *SD, RT*		

The performance of LPRM algorithm is similar to BBAI because they are both token based and both use MSSs as proxies of MHs. The wireless message cost of LPRM is the same as that of BBAI. The difference lies in the wired part. BBAI is a token-circulating algorithm while LPRM is a token-asking algorithm. Therefore, under low load levels, the performance of LPRM is better than BBAI with the *search* mechanism. However, under high load levels, the BBAI performs better because it does not need any request message.

MSM algorithm is fundamentally different from the others, because it uses the permission-based MUTEX mechanism. Not making full use of MSSs makes MSM costs more messages, especially under high load levels. The *RT* of MSM is also larger than that of BBAI and LPRM. However, the advantage of MSM is that no logical structure is needed.

Table 3-3 shows the performance of the algorithms for MANETs. The performance of MVW is similar to typical token-circulating algorithms. Profiting from the combination of token-circulating and token-asking, the RBVP algorithm performs well under low load levels. However, the introduction of request messages makes the performance worse under high load levels, compared with typical token-circulating algorithms, e.g. MVW.

Like the LPRM algorithm, JWWV is a logical structure based algorithm, so the load level has similar effect on these two algorithms. However, since the graph is imposed on the MHs rather than MSSs in JWWV, the overhead of maintaining such a structure is high due to the mobility of MHs. Additional messages are needed if the graph is changed due to the link break or establishment. The performance of YCJW is nearly the same as MVW, except that under high load levels, the *MPCS* of YCJW is larger. The difference comes from the relative fixed visit path of YCJW, which lets some requesting hosts send out the token without access the CS

Table 3-3 Performance of MUTEX algorithms for MANETs

		RBVP	JWSK	MVW(LR)	YCJW
MUTEX Mechanism		Token-asking + token-circulating	Graph-based	Token-circulating	Token-circulating
Property		dynamic	dynamic	dynamic	dynamic
MPCS	Low load	$O(n)*C_w$	$O(log^n)*(2+k)C_w$	$O(n)*C_w$	$O(n)*C_w$
	High load	$2C_w$	$(k+1)C_w$	C_w	$>C_w$
SD	High load	C_w	C_w	C_w	C_w
RT	Low load	$O(log^n)D_w$	$O(log^n)*(2+k)D_w$	$O(n)* D_w$	$O(n)* D_w$
	High load	$O(n) * (2D_w+ E)$	$O(n)*((2+k)D_w +E)$	$O(n) * (D_w+ E)$	$O(n) * (D_w+ E)$
Contributions		Reducing message cost	Handling disconnection Handling topology change	Handling topology change	Handling topology change Tolerance of token loss
Drawbacks		Dependence on the route protocol.	Repeated requests.	Additional information for token passing	Large size message(token) Fixed leader host

From the analysis we can see that, the load level has very important effect on the performance of MUTEX algorithms. Nearly all the algorithms have better performance under high load levels. The network type can also affect MUTEX algorithms. In infrastructured networks, the design is easier because MSSs can be used to carry out much work.

The MUTEX mechanism is also an important factor in the design of MUTEX algorithms. Up to now, nearly all the MUTEX algorithms for mobile environments, especially for MANETs, are token-based. Token-circulating algorithms perform better under high load levels, while token-asking algorithms are better if the load level is low.

Token-based mechanisms have many desirable features for mobile networks: the host only needs to keep information about its neighbors, few messages needed to pass the privilege to enter the CS and so on. However, the fatal problem—token loss makes the token-based mechanisms not so robust. What is worse, the mobility and frequent disconnections make the token loss more serious and the maintenance of a logical structure more costly.

Compared with the token-based approach, permission-based algorithms have the following advantages: 1) there is no need to maintain the logical topology, and 2) there is no need to propagate any message if no host requests the CS. These advantages make the permission-based approach well suitable for mobile networks, where all the resources, e.g. the network bandwidth and the battery power, are scarce. A problem of the permission-based approach is the large number of messages exchanged to get permissions. Therefore, to design a permission-based algorithm, the key issue is to reduce the message cost.

Another important problem is fault tolerance. In a mobile environment, especially a MANET, link failures (e.g. signal shielded) and host failures (e.g. battery exhausted) occur very frequently. Link failures can lead to message loss while host failures may result in accidental disconnections. Furthermore, MHs may enter the "doze" mode to save power. Unfortunately, these issues have not been adequately addressed in existing MUTEX algorithms for mobile environments.

Chapter 4. Speeding up the Execution of Consensus Protocols

4.1. Overview

Our study in this chapter focuses on the execution speed of consensus protocols, which is important in both handling topology change and reducing computation load. In general, the earlier a protocol achieves consensus, the less the topology may change and the less resource, e.g. bandwidth and memory, is consumed.

The main objective of this chapter is to cope with the slowdown caused by the mistakes made in the execution of $\Diamond S$ based protocols. All existing oracle based consensus protocols allow the oracle to make mistakes, which is referred as the indulgence property [69][73]. The price to pay for indulgence is the slowdown of the execution.

Usually, there is at least one coordinator or leader for each round of FD or leader based protocols and the coordinator or leader process attempts to impose its own current estimate of the decision value on other processes. Due to the indulgence property, processes have to keep executing consecutive rounds, even if in fact the decision cannot be made due to failures or false suspicions. This slows down the execution speed of a consensus protocol. Although some optimizations, including Zero-Degradation (ZD for short) [45][51][135], One-Step-Decision [51][73] and fast recovery [52] have been developed to speed up the execution of indulgent consensus protocols, they are either unsuitable for $\Diamond S$ based protocols or difficult to be applied.

We propose a fast consensus protocol based on the failure detector $\Diamond S$, which can circumvent slowdowns caused by indulgence, using two novel techniques.

The first technique uses a simple but efficient approach to guarantee the Round-Zero-Degradation (RZD for short) property, which is an extension of ZD. ZD means that, when all the failures are initial crashes and the oracle makes no mistake, two communication steps are sufficient to reach a global decision [51][73]. An initial

crash refers to a crash that occurs before the execution of the protocol starts. ZD is important especially because a consensus protocol is typically invoked from time to time and a process failure occurred during one execution will appear as an initial failure in the subsequent executions [73]. However, ZD only takes into account the first round of an execution. Here, we extend it to "Round-Zero-Degradation". A consensus protocol is said to have the "Round-Zero-Degradation" property if it can achieve the global decision within any round (two communication steps) when the underlying oracle makes no mistake in the round and all failures are "round initial crashes", i.e. all crashes occur before the round starts[5].

Compared with ZD, RZD is more useful. Nearly all implementations of FDs [31][93] rely on the *partial synchrony* system model [53], where the bounds on the message delay and processing speed are unknown and hold only after an unknown stabilization interval. This implies that the FD is likely to fail in the beginning of the execution of a consensus protocol. Thus, "Zero-Degradation" is hard to satisfy.

ZD and RZD are desirable properties. The key point to guarantee ZD or RZD is how to select only correct processes as the coordinator. For leader-based protocols [51][73][110], ZD is guaranteed by nature, because a leader oracle dynamically elects the leader based on the status information. However, most existing FD-based consensus protocols cannot guarantee the ZD or RZD property. FD-based protocols usually adopt rotating coordinator paradigm and the order for processes to be the coordinator is predefined, so they have to execute such rounds that are coordinated by crashed and suspected processes, which defers the decision. Although some solutions [45][51][135] have been proposed to help FD-based protocols achieve ZD, they are either not Round Zero Degrading or too complicated.

In the proposed protocol, a simple and efficient approach is designed to guarantee RZD. The principle is to replace a suspected coordinator when a round starts. However, unlike leader oracle, a FD only provides a process the status information

[5] The start of a round is defined as the time that the first process enters this round.

of other processes but cannot indicate which process should be the coordinator. An arbitrarily selected coordinator may result in the violation of the termination property. Therefore, how to dynamically select coordinators is not trivial.

The first technique, as described above, is effective only in good cases[6]. When the FD performs badly, it will not work. The second technique proposed in this chapter, called “Look-Ahead”, speeds up the execution of the proposed protocol in general cases. The main idea of Look-Ahead is making use of the future messages so as to reduce useless wait time. Due to the asynchrony of the system, some messages of “future” phases or rounds may be delivered to “slow” processes in advance. Based on the information carried by such messages, a process can adapt its execution state to the future so as to speed up its execution. However, not all future messages are beneficial, because some future messages may mislead the receivers so as to postpone potential decisions.

Some papers [50][52] mentioned the use of future messages to handle omission failure or achieve consensus with a lower bound number of rounds after the stabilization interval. On the reception of a future message, a process directly skips to the round of the future message. Such an approach is too extreme in sense of efficiently speeding up the execution because it may destroy potential decisions. The proposed Look-Ahead technique is designed delicately to avoid negative effect of future messages, so Look-Ahead can help speed up execution in general cases.

Look-Ahead is in fact a general technique which can be easily applied to any round based indulgent consensus protocols. However, it is important to notice that, although the Look-Ahead technique is not special for mobile environments, it is especially suitable for mobile environments, where the asynchrony is stronger than that in fixed networks due to the dynamics of mobile hosts.

[6] According to the behaviours of the hosts and oracles, execution cases of a consensus protocol can be informally divided into three categories. A “good” case means that no new crash occur and the oracle performs perfectly (makes no mistake), while a “bad” case means there are new crashes and the oracle performs badly. A “general” case simply refers to any case.

To evaluate the performance of the proposed protocol, we conduct extensive simulations. The proposed protocol is compared with one classical protocol. The results show that the proposed protocol can always achieve the consensus with fewer rounds, regardless the performance of the failure detector and other conditions. In terms of the execution time, the proposed protocol also performs better unless the failure detector makes many mistakes (with an error rate greater than about 35%). Simulations also show that the Look-Ahead technique itself is always effective regardless the performance of the FD and other conditions, which indicates that our objective to avoid destroy of potential decisions is fulfilled.

4.2. A Fast Consensus Protocol with RZD Property

4.2.1. System Model and Data Structures

The system model for the proposed protocol is the same as in [14][31][61][97]. A distributed system consists of a finite set of n processes: $\Pi = \{p_1,p_2,\ldots,p_n\}$, $n > 1$. Processes communicate only by sending and receiving messages. Every pair of processes is connected by a reliable channel that does not create, duplicate, alter, or lose messages. There is no bound on the message delay or the processing speed of a host.

A process can only fail by crashing, i.e. prematurely halting. A process that crashes in a run is faulty in that run, otherwise it is correct. A faulty process executes correctly until it crashes. The maximum number of faulty processes in a run is denoted by f. To guarantee a majority of the processes to be correct, f is bounded by $n/2$, i.e. $f < n/2$. Each process is equipped with a FD module of class $\Diamond S$, which provides unreliable information about the status of other processes.

When executing the protocol, each process p_i needs to maintain necessary information about its state. Such information is stored in the following variables.

r_i: the sequence number of the current round that p_i is participating in.

est_i: the current estimate of the decision value. Initially, it is set to be the value proposed by p_i.

ts_i: the timestamp of est_i. The value is the sequence number of the round in which p_i receives the current value of est_i from the coordinator process.

fld_i: the flag to indicate whether p_i has made the decision.

The following message types are used in the proposed protocol.

PROP(r_i, est_i, *cc*): the proposal message sent in round r_i by process p_i to all other processes, where *cc* is the *id* of the current coordinator selected by p_i. This message serves two purposes: 1) exchanging the coordinator *id* among all processes; 2) delivering the proposal value from the coordinator to all processes.

ECHO(r_i, est_i, ts_i): the echo message sent in round r_i, from process p_i to all processes.

DECISION(*v*): the message sent from a process that has decided to propagate the decision value *v*.

4.2.2. Description of the Protocol

Figure 4-1 shows the pseudocode of the proposed protocol. Like most existing consensus protocols, there are two tasks. Task 1 is for making the decision and constitutes the main body of the protocol. Task 2 is for propagating the value that has been decided upon.

We describe Task 1 first. At the beginning of a round, a process p_i first determines the coordinator (line 3 to line 5). Two functions are involved. The *coord*(r_i) function, as used in existing FD-based protocols, determines the default coordinator of round r_i. It is deterministic and always returns the same value given the round number. This guarantees that each process selects the same default coordinator in the same round. To skip crashed coordinators, we introduce a new function *ncoord*(*cc*). Given the *id* of the current candidate for the coordinator, this function returns the "next" candidate for the current round. Similar to *coord*(), *ncoord*() is deterministic, i.e. all processes get the same result with the same input. A simple implementation of *nccoord*() is *ncoord*(*cc*)= (*cc*+1) *mod n*. *ncoord*() is invoked repeatedly until a "trusted" process is returned.

The remaining actions in a round are divided into two phases. In Phase 1, each process p_i sends its current estimate est_i to all processes, using the proposal message $PROP(r_i, est_i, cc)$. Then p_i waits for the PROP messages from all processes (line 7). p_i stop waiting if any of the following three conditions holds: i) p_i receives the PROP message from p_{cc} and additional n-f-1 processes, ii) p_i receives an ECHO message with a timestamp no less than r_i, i.e. $ECHO(*, *, \geq r_i)$, or iii) p_{cc} is suspected.

```
----------------------------------------Task 1: Consensus----------------------------------------------------------
// The code executed by each process, p_i
BEGIN:
(1)  r_i ←0;  est_i ← v_i;  ts_i ← 0; fld_i ← false;
(2)  while (fld_i=false){
(3)        r_i←r_i+1; cc = coord(r_i);
(4)        while (p_cc ∈ suspected_i){
(5)                cc = ncoord(cc);}
           ---------- Phase 1 Collect Proposal ----------
(6)        send PROP(r_i, est_i, cc) to Π;
(7)        wait until ((received PROP(r_i, *, *) from p_cc and n-f -1 other processes)
                   or (received ECHO(*, *, ≥ r_i ) from some process)
                   or p_cc ∈ suspected_i);
(8)        if(received PROP(r_i, est_cc, cc) from p_cc and
                   PROP(r_i, *, cc) from n-f -1 other processes){
(9)                est_i← est_cc; ts_i ← r_i;}
(10)       else if(received ECHO(r_i, v, r_i) from some process ){
(11)               est_i← v; ts_i ← r_i;}
           ----------Phase 2 Collect Echo - ----------
(12)       send ECHO(r_i, est_i, ts_i) to Π;
(13)       wait until ((received ECHO(r_i, *, *) from n-f processes)
                   or (received  ECHO(*, *, > r_i) from some process));
(14)       if(received ECHO(r_i, *, *) from n-f processes and
                   there are f+1 ECHO(r_i, est, r_i) messages) {
(15)               est_i ← est;
(16)               send DECISION(est_i) to Π \ p_i;
(17)               fld_i←true;}
(18)       else   est_i ← est' carried by the ECHO with the highest timestamp;
     }//endwhile
-----------------------------------Task 2:  Reliable broadcast ----------------------------------------------
// The code executed by each process, p_i
(19) upon reception of DECISION(est) from process p_j: {
(20)       send DECISION(est) to Π \{ p_i, p_j};
(21)       fld_i←true; }
END
```

Figure 4-1 Pseudocode of the fast consensus protocol

After p_i stops waiting at line 7, it checks the messages it has received for this phase. There are three possible cases.

1) p_i has received a $PROP(r_i, est_{cc}, cc)$ message from p_{cc} and a $PROP(r_i, *, *)$ message from additional n-f-1 processes. Then p_i updates its own estimate value est_i

to est_{cc} and ts_i to r_i (line 9).

2) p_i has received an $ECHO(r_i, v, r_i)$ message. p_i will update its own estimate est_i to v and ts_i to r_i.

3) Neither 1) nor 2) is satisfied. p_i will not change its estimate or timestamp.

Phase 2 starts by sending ECHO messages. p_i first sends a $ECHO(r_i, est_i, ts_i)$ message to all processes including itself and then waits for ECHO messages (line 13). p_i stops waiting at line 13 if any of the following two conditions occur : 1) receiving ECHO messages with round number r_i (i.e. $ECHO(r_i,*,*)$) from no less than $n-f$ processes, or 2) receiving an ECHO message with a timestamp greater than r_i, i.e. $ECHO(*,*,>r_i)$). Then p_i checks the ECHO messages received to determine whether it can make the decision. If p_i receives $ECHO(r_i,*,*)$ messages from no less than $n-f$ processes and at least $f+1$ of them contain a timestamp equal to r_i, i.e. the messages received are of the form $ECHO(r_i, est, r_i)$, p_i decides upon the value est and broadcasts est using the $DECISION(v)$ message. Otherwise, p_i updates its estimate value est_i to the value est' that is carried by the ECHO message with the highest timestamp. Then p_i enters the next round.

Task 2 is simple. When a process receives a DECISION message but has not decided, the process forwards the DECISION message to all other processes except the sender, and then makes the decision.

Obviously, the proposed protocol has the RZD property. In any round r (including the first round), the proposed protocol achieves the global decision within two communication steps, if no new crash occurs and the underlying FD makes no mistake during this round (i.e. any crashed process is suspected by all correct processes and any correct process is not suspected by any correct process). This is achieved as follows. In the beginning of round r, each correct process selects the coordinator by executing line 3 to line 5. Because both *coord*() and *ncoord*() are deterministic, all correct processes select the same coordinator p_x. Then all correct processes send out $PROP(r, *, x)$ messages. Because the FD makes no mistake, all correct processes execute line 9 or line 11 and consequently send out $ECHO(r, est_x, r)$

at line 12. At line 13, each process waits for ECHO messages. Obviously, no process proceeds to the next round, so no ECHO message with a timestamp greater than *r* can be received at line 13. Eventually, each process must receive $ECHO(r, est_x, r)$ from at least *n-f* processes and then makes the decision at line 18. The global decision is achieved within two communication steps: the transfer of PROP messages and the transfer of ECHO messages.

4.2.3. The Look-Ahead Technique

The basic idea of the Look-Ahead technique is to speed up the execution of consensus protocols by making use of the future messages delivered in advance. In an asynchronous system, the delay of transmitting messages and the processing speed of different processes may vary significantly. Therefore, when executing a consensus protocol, some processes may proceed faster than others because the processes may be slow in handling messages or some messages take longer time to arrive. Therefore, different processes may be in different phases or rounds at a given moment. Thus, a message may be delivered before the receiver enters the corresponding phase or round. In the view of the receiver, such a message carries information about the "future", which can be used by the process to optimize its operations, e.g. stop waiting for some message that is delayed for a very long time, so as to speed up its execution.

However, not all future messages are beneficial. For example, if some process, which proceeds faster than the others, falsely suspects a correct coordinator or leader, this process sends out negative messages to other processes that are waiting for proposal messages from the correct coordinator or leader. The receivers may be misled by the "future" message from the fast process, i.e. they will stop waiting for the proposal from the correct coordinator or leader. Consequently, the decision that might be made in this round will not be reached. Similar scenarios may appear in the phase of exchanging the echo messages. Our proposed Look-Ahead technique is delicately designed to avoid such negative effect.

The lines 7 and 13, which are boxed, are the embodiment of the Look-Ahead technique. While waiting at line 7, if a process p_i receives an ECHO message carrying an estimate value with a timestamp equal to the sequence number of the current or a "future" round, p_i stops waiting immediately. Reception of such an ECHO message means that some process (may not be the sender of the ECHO message) has received no less than n-f PROP messages for the current or a "future" round with the same value of the coordinator field. Therefore, p_i no longer needs to wait for more PROP messages and can go ahead.

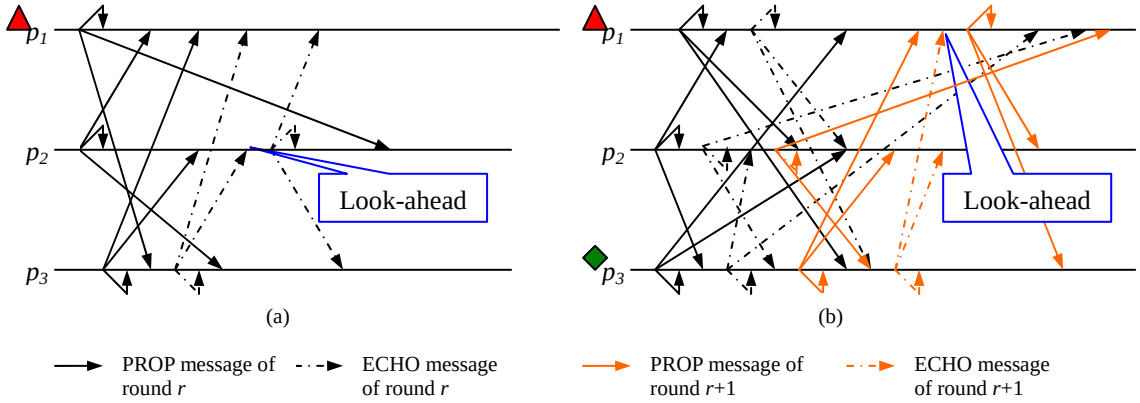

Figure 4-2 Examples of Look-Ahead technique

While waiting at line 13, a process p_i can stop waiting immediately if p_i receives an ECHO message carrying an estimate value with a timestamp equal to the sequence number of a "future" round. Receiving such an ECHO message indicates that at least n-f processes have proceeded to that future round. The probability for p_i to make the decision in the current round is low. Therefore, it would be better for p_i to give up this round and immediately proceed to the next round.

Figure 4-2 illustrates two example scenarios that demonstrate the use of the Look-Ahead technique. Figure 4-2-(a) shows a scenario of looking ahead in Phase 1 of a round r. There are totally three processes in the system and at most one process can crash. At the beginning of round r, all the three processes select p_1 as the coordinator and send PROP messages to one another. Now let us examine the execution of p_2. In Phase 1, p_2 waits for the PROP message from p_1. However, due to the asynchrony, the PROP message from p_1 to p_2 has been delayed much longer than other PROP

messages. After p_3 receives the PROP message from p_1 and itself, it sends out *ECHO*(r, v, r) to all three processes. Consequently, p_2 receives the ECHO message from p_3 before the PROP message from p_1. Then p_2 stops waiting for the "slow" PROP message and updates its estimate and timestamp based on the *ECHO*(r, v, r). Obviously, the execution time of Phase 1 is shortened.

Figure 4-2-(b) shows a scenario of the looking ahead in Phase 2 of a round r. Similar to the case in Figure 4-2-(a), there are three processes in the system and at most one process can crash. At the beginning of round r, all the three processes select p_1 as the coordinator and send PROP messages one another. Each process waits for the PROP from p_1 in Phase 1. However, p_2 and p_3 suspect p_1 before they receive the PROP messages from p_1. Then p_2 and p_3 send out *ECHO*(r, *, $<r$) messages and proceed to round $r+1$. In round r+1, p_2 and p_3 both select p_3 as the coordinator and send out PROP messages. After p_3 receives the *PROP*(r+1, *, 3) from itself and p_2, it sends out *ECHO*(r+1, est_3, r+1). Due to the asynchrony, p_1 receives the *ECHO*(r+1, est_3, r+1) from p_3 before the *ECHO*(r, *, $<r$) from p_2. Then p_2 stops waiting for the "slow" ECHO message and update its estimate based on the *ECHO*(r+1, est_3, r+1). The execution time of Phase 2 is shortened.

Figure 4-2 gives only two possible scenarios. Considering the asynchrony of the underlying system, there are many other situations that can occur in the real execution of a consensus protocol. Therefore, future messages appear frequently and the "Look-Ahead" technique can significantly improve the performance of a consensus protocol.

The performance of the Look-Ahead technique is significantly affected by the degree of asynchrony. The more the message delay or processing speed varies, the more time is saved by one useful future message and the more beneficial future messages appear. Consequently, larger speedup is made by Look-Ahead. Therefore, the Look-Ahead technique is suitable for the system with "high" asynchrony, e.g. mobile networks. In mobile environments, the network topology is changing from time to time, which increases the diversity of the message delay.

4.3. Correctness of the Proposed Protocol

4.3.1. Validity

Lemma 1. *In a round r, if a process sends out an ECHO(r, v, r) message at time tr, then before tr:*

1) *at least n-f processes have sent out PROP(r, *, cc) and p_{cc} has sent out PROP(r, est_{cc}, cc), where cc is the id of the common coordinator;*
2) *v = est_{cc}.*

Proof. Let us represent all the echo messages *ECHO(r, *, r)* as an ordered list $L = [me_0, me_1, me_2, \ldots, me_m]$, where the messages are sorted in the ascendant order of the time that they are sent. The sender of me_i is denoted by p_{mei} and the estimate of me_i is denoted by v_i. Now, let us consider the message $me_0 = ECHO(r, v_0, r)$. An ECHO message can be sent only at line 12. Before p_{me0} sends me_0 at line 12 in round r, p_{me0} must have finished waiting at line 7. Since me_0 is the first ECHO message with the timestamp equal to r in round r, me_0 must have updated its estimate at line 9. Therefore, p_{me0} has received $PROP(r, est_{cc}, cc)$ from p_{cc} and $PROP(r, *, cc)$ from additional $n-f$ -1 processes (line 8). Part 1) of the lemma holds.

The proof of part 2) is by induction on the sequence number i of the ECHO messages in the list L.

Base case: $i = 0$. As proved above, p_{me0} has received $PROP(r, est_{cc}, cc)$ from p_{cc} and additional $n-f$ -1 processes before it sends out me_0. Obviously, p_{me0} updated its estimate to est_{cc} at line 9 and sent out $ECHO(r, est_{cc}, r)$ at line 12. Therefore, $v_0 = est_{cc}$. The lemma holds.

Induction hypothesis: $0 \le i \le k$. Let us assume $v_i = est_{cc}$, $0 \le i \le k$. Now we prove that $v_{(k+1)} = est_{cc}$. Before sending $me_{(k+1)}$, $p_{me(k+1)}$ must have updated its estimate at line 9 or line 11, so it had received: either $PROP(r, est_{cc}, cc)$ from p_{cc} and $PROP(r, *, cc)$ from other $n-f$ -1 processes, or an $ECHO(r, v_j, r)$, denoted by me_j, at line 7. For case 1), obviously $v_{(k+1)} = est_{cc}$. For case 2), $p_{me(k+1)}$ updated its estimate to v_j at line

11. From the definition of list *L*, we have $0 \leq j < k+1$. From the induction hypothesis, we have $v_j = est_{cc}$, so $v_{(k+1)} = est_{cc}$. The lemma holds.□

Theorem 1 (Validity). *If a process decides upon some value v, then some process has proposed v.*

Proof. If a process decides upon a value at line 21, then this value must have been decided upon by another process at line 17. Therefore, we only consider the values decided upon at line 17.

If a process decides upon *v* at line 17 in round *r*, then *v* must come from an ECHO message (line 14). Furthermore, the value carried by the ECHO message must come from a PROP message at line 6 (by Lemma 1). Therefore, the value *v* comes from the estimate stored by a process at the beginning of round *r*. Of course, each estimate kept by a process at the beginning of a round *k* is the same as the estimate kept by the process at the end of the previous round *k*-1. By simple induction, we can conclude that *v* is the estimate value proposed by some process in the beginning of the execution. The theorem holds.□

4.3.2. Termination

Lemma 2. *If no process decides in any round r'≤ r, then all correct processes start round r+1.*

Proof. If some correct process blocks forever before round *r*+1, then there must be a earliest round, say *rs (rs < r+1)*, during which some correct process is blocked forever. Now, we only need to prove that "no correct process can be blocked in round *rs* forever."

The proof is by contradiction. Assume that some correct process p_i is blocked forever in round *rs*. Then, p_i must be blocked at some "wait" statement on line 7 or line 13, in round *rs*.

First, let us examine the case where p_i is blocked at line 7. Since *rs* is the earliest round where a correct process is blocked forever, all correct processes eventually proceed to round *rs* and send out PROP messages. Consequently, p_i can receive at

least *n-f* PROP messages. If the coordinator p_{cc} selected by p_i, is a correct process, p_i eventually receives the PROP message from p_{cc} and is unblocked. Otherwise, p_{cc} is a faulty process and will eventually crash. Then p_i eventually suspects p_{cc} and unblocks. Therefore, p_i cannot be blocked forever at line 7.

Now, let us consider the case where process p_i is blocked at line 13. As proved above, no correct process can be blocked forever at line 7 in round *rs*, so each correct process sends *ECHO*(r_i, *, *) message at line 12. Since at most *f* process(es) can crash, each correct process can receive at least *n-f ECHO* messages from correct processes. Therefore, p_i cannot be blocked forever at line 13.

Therefore, no correct process can be blocked forever in round *rs*, which contradicts the assumption. The lemma holds.□

Lemma 3. *For any round r, if the process p_i is the first process that finishes the round r, then p_i cannot receive any ECHO(x, *, y) with x>r or y>r before it finishes the round r.*

Proof. The proof is by contradiction. Assume that p_i receives an *ECHO(x, *, y)* with $x>r$ from some process p_j during round *r*. Obviously, p_j must have finished round *r* before it sends out the *ECHO(x, *, y)* to p_i, which contradicts the definition of p_i ("the first process that finishes round *r*"). Then we have $x \leq r$. Trivially, we have $y \leq x$, so $y \leq x \leq r$. The lemma holds.□

Theorem 2 (Termination). *If a process is correct, it decides eventually.*

Proof. If one (correct or faulty) process decides, all correct processes eventually decide due to the reliable broadcast in Task 2. Therefore, we just need to prove that "at least one process decides."

The proof is by contradiction. Assume that no process decides. According to the accuracy of $\Diamond S$, there is a time *t* after which: 1) no new crash occurs, i.e. all faulty processes crashed before *t*, and 2) there is a correct process p_x that is no longer suspected by any correct process. Without loss of generality, let *rx* be the first round that starts after time *t* and $x = coord(rx)$. By assumption, no process decides, so all correct processes start round *rx* eventually (by Lemma 2). As p_x is no longer

suspected by any correct process after *t*, all correct processes select p_x as the coordinator at line 3 and skip line 5. Then each correct process sends *PROP*(*rx*, *, *x*) to all processes and wait for PROP messages at line 7. Now let us consider the first process, say p_f, which finishes round *rx*.

First, we prove that p_f must update its estimate to est_x and its timestamp to *rx* at line 9 or line 11. Since no correct process suspects p_x, the only possible conditions for p_f to end the waiting at line 7 are: 1) receiving *PROP*(*rx*, est_x, *x*) from p_{cc} and *PROP*(*rx*, *, *x*) from other *n-f*-1 processes, or 2) receiving an ECHO(*x*, *, *y*) with $y \geq rx$ from some process p_j. If condition 1) holds, p_f must update its estimate to est_x and timestamp to *rx* at line 9. Otherwise, condition 2) holds. Since p_f is the first process that finishes round *rx*, by Lemma 3, we have $y \leq x \leq rx$. Combining $y \geq rx$ and $y \leq x \leq rx$, we have $x=y=rx$, so *ECHO*(*x*, *, *y*) must be an *ECHO*(*rx*, *, *rx*). By Lemma 2, p_f updates its estimate to est_x and timestamp to *rx* at line 11.

After p_f updates its estimate to est_x and its timestamp to *rx* at line 9 or line 11, p_f sends out *ECHO*(*rx*, est_x, *rx*) at line 12 and waits for ECHO messages from all processes at line 13. By Lemma 1 and Lemma 2, all correct processes send out *ECHO*(*rx*, est_x, *rx*) at line 12. Since p_f is the first process that finishes round *rx*, p_f must receive *ECHO*(*rx*, est_x, *rx*) from *n-f* processes at line 13 (by Lemma 3). Consequently, p_f decides at line 17. The theorem holds.□

4.3.3. Agreement

Lemma 4. *If r is the smallest round in which some process decides upon some value v at line 17, then every process that completes round r has an estimate equal to v at the end of the round r.*

Proof. Without loss of generality, let p_i be the process that decides at line 17 in round *r*. Then p_i must have sent *DECISION*(*v*) messages at line 16. Obviously, p_i has received *ECHO*(r_i, *, *) from *n-f* processes and at least *f*+1 of them are with the same estimate value *v*, i.e. they received at least *f*+1 *ECHO*(*r*, *v*, *r*) messages. By Lemma 2, at least *n-f* processes selected the same coordinator p_{cc} and $est_{cc} = v$. Let us

denote all the processes that complete round r by a list P= [pr_0, pr_1, pr_2, …, pr_m], where the processes are sorted in the ascendant order of the time when they finish round r.

The proof of the lemma is by induction on the sequence number i of the processes in the list P.

Base case: i = 0. Obviously, pr_0 is the first process that finishes round r. By Lemma 3, pr_0 must have received *ECHO*(r, *, *) messages from at least n-f processes at line 13 in round r. Since (n-f)+(f+1)>n, pr_0 must have received an ECHO message from some process p_k which had sent *ECHO*(r, v, r) to p_i. By Lemma 3, r must be the highest timestamp of all the ECHO messages received by pr_0. Therefore, pr_0 must update its estimate to v at line 15 or 18. The lemma holds.

Induction hypothesis: $0 \le i \le k$. Assume $est_i = v$, $0 \le i \le k$ at the end of the round r. Now we prove that $est_{(k+1)} = v$ at the end of round r. Let us consider the behaviours of process $pr_{(k+1)}$ when it ends the waiting for ECHO messages at line 13. There are two possible conditions for $pr_{(k+1)}$ to stop waiting at line 13: 1) receiving *ECHO*(r, *, *) from n-f processes, or 2) receiving an *ECHO*(x, vy, y) with y>r, from some process p_j. For condition 1), we can prove $est_{(k+1)} = v$ at the end of round r in the same way as in the base case. Then let us consider condition 2). Since $x \ge y$, p_j must send out the *ECHO*(y, vy, y) at line 12 in round y. By Lemma 1, the value vy comes from the estimate of the coordinator p_t, which was selected by at least n-f processes in round y and p_t had send out *PROP*(y, vy, t) at line 6 in round y. Since an estimate value can be adopted by a process only after the value is proposed by some process at line 6, the estimate carried by an ECHO message must be equal to the estimate of the coordinator at the end of the previously round. By a simple induction, we know that vy equals to the estimate of some process p_z at the end of round r. Obviously, p_z has finished round r before $pr_{(k+1)}$ does so. Therefore, we have $p_z = pr_s$ where $0 \le s \le k$. By the induction hypothesis, we have $vy = v$. The lemma holds.□

Theorem 3 (Agreement). *No two processes decide differently.*

Proof. If a process decides upon some value at line 21, then this value must have

been decided upon by another process at line 17. Therefore, we only consider the values decided upon at line 17.

Let r be the earliest round in which some process p_i decides upon the value v at line 17. Assume that another process p_j decides upon another value y in round k. By the definition of r, we have $k \geq r$. If $k=r$, by Lemma 1, we have $y = v$. If $k>r$, every process that executes round k must have finished round r. By Lemma 4, every process that finishes round k has estimate value v at the end of round r. Therefore, no other value can be decided upon in subsequent rounds, so we have $y = v$. The theorem holds.□

4.4. Performance Evaluation

We have carried out simulations to evaluate the performance of the proposed protocol. We first describe the simulation setup and then report the results of performance evaluation.

4.4.1. Simulation Setup

The simulated system consists of three main parts: the network, the failure detector and the consensus protocol. The main parameters of the simulations are listed in Table 4-1. Faulty processes are selected randomly and the life time of a faulty process satisfies the exponential distribution. For message routing, we used the well-known "least hops" policy, which is adopted in many existing routing protocols. As in existing implementations of $\Diamond S$, the underlying network is set to be partial synchronous [8]: the timing attributes are bounded, but the bounds are unknown and hold only after an unknown stabilization interval. The

Table 4-1 Simulation settings for the Look-Ahead consensus protocol

No. of processes, n	20
Maximum of the crashed processes, f	9
Mean life of crashed processes, λ_h	25 ms
Stabilization interval, GST	500 ms
Mean link delay, λ_l	From 1 ms to 45 ms
Max link delay for synchronous period	1000 ms
Error rate of failure detector, err_{fd}	From 0% to 80%
Interval of Heartbeat messages	10 ms
Routing Protocol/Policy	Least hops

message delay is also assumed to satisfy an exponential distribution. Since the error rate of FD and the mean link delay can affect the performance significantly, we varied these two parameters to observe their effects.

To compare with existing work, we also simulated the HMR protocol [82]. Although HMR is simple, it is versatile and can derive different protocols. To clearly show the benefit of our two proposed techniques, we also simulated a variant of the proposed protocol without using Look-Ahead. For convenience, the proposed protocol and its variant are named “ZD-LA” protocol and “ZD” protocol, respectively.

4.4.2. Performance Metrics

In literature, “number of communication steps (or rounds)” is usually used to evaluate the performance of a consensus protocol [83][139]. This metric is closely related to both time cost and message cost because, roughly speaking, more rounds or communication steps mean more time and more messages. However, this metric cannot precisely reflect either of the two costs. Consensus protocols are executed in asynchronous rounds and the communications are asynchronous. The number of rounds only indicates how many rounds are needed, but the rounds may overlap each other due to the asynchrony and the duration of one round in different protocols or scenarios may be different. Similarly, besides the number of rounds, the number of messages exchanged in one communication step or round also affects the message cost significantly. To evaluate the performance more precisely, we adopt the following three metrics in the simulations:

NR (*Number of Rounds*): the average number of rounds executed by the processes to achieve the global decision.

NM (*Number of Messages*): the total number of messages exchanged to achieve the global decision.

ET (*Execution Time*): the “real” time needed by a consensus protocol to achieve the global decision.

The simulation is run 300 times and the average values of metrics are reported. In each run of the simulation, the consensus protocols are invoked 30 times.

4.4.3. Simulation Results

We discuss the simulation results according to the three metrics. In the following figures, if no indication is made, err_{fd}= 5% and λ_l = 5ms.

4.4.3.1. Number of Rounds, NR

Figure 4-3 shows the results of NR with varied err_{fd}. First, we can see that, with the ZD property, when err_{fd} is 0 or very small (less than 5%), both ZD protocol and ZD-LA protocol terminate within nearly one round. The little deviation from 1 is due to the stabilization interval of the system. Since it is hard to examine the RZD property by simulations, only the ZD property is discussed here.

Now let us examine the effect of err_{fd}. With the increase of err_{fd}, NR of all three protocols also increases. This can be explained by the effect of err_{fd} on the probability of termination of a round. The more mistakes are made by the FD, the higher possibility the current coordinator is falsely suspected. Consequently, it is more likely that this round will fail to make decision, so more rounds are needed to achieve the consensus.

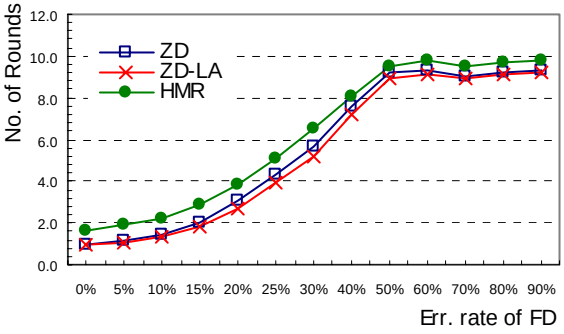

Figure 4-3 NR vs. error rate of FD

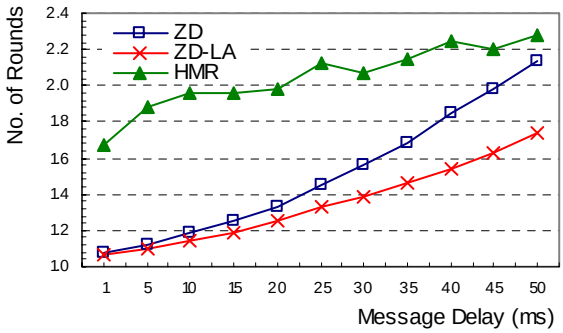

Figure 4-4 NR vs. mean link delay

However NR does not increase linearly with the increase of err_{fd}. When err_{fd} reaches about 50%, the increase of NR slows down. This indicates that the effect of err_{fd} becomes smaller when its value becomes large. The protocols are executed in repeated form, so most of the runs are executed in the stable state, i.e. no new crash happens and at least one process p_g is trusted by all correct processes (by the

property of $\Diamond S$). When err_{fd} is large enough, it becomes almost impossible to make a decision in a round not coordinated by p_g, i.e. the protocol can terminate only when p_g is the default coordinator. Since each process has the same probability to become the default coordinator, the average number of rounds is approximately $n/2$, i.e. 10, in the simulations, as shown in Figure 4-3.

Now we compare the three protocols. As shown in Figure 4-3, HMR needs the largest number of rounds among the three protocols, whereas ZD-LA needs the fewest. The difference between HMR protocol and the ZD protocol is due to the ZD property. With the increase of err_{fd}, the advantage of ZD protocol decreases. This indicates that, even if the FD indeed makes some mistakes, the ZD mechanism can still help to some extent, and the better the FD performs the better the ZD performs. ZD is more useful than expected because it is effective not only in good cases.

We see that ZD-LA always performs better than ZD. The performance gain by ZD-LA comes from the Look-Ahead technique. Since the Look-Ahead technique can shorten the duration of a round, fewer crashes and false suspicions will happen during a round and the probability of making the decision during the round is increased. Consequently, fewer rounds are needed.

The results of NR under varied λ_l are shown in Figure 4-4. With the increase of λ_l, NR of all the three protocols increases. When λ_l increases, more time is needed to deliver a message. Therefore, the duration of a round is increased. Consequently, more crashes and false suspicions may happen during the round. The probability of making the decision during this round decreases and more rounds are needed to achieve the consensus. The difference among the protocols is similar to that in Figure 4-3 except that the difference between ZD and ZD-LA becomes greater as λ_l increases. The larger difference is attributed to the feature of the exponential distribution of link delay. When λ_l increases, the variance of link delay, and thus message transmission delays, also increases. Therefore, the advantage of using the Look-Ahead technique becomes more obvious.

4.4.3.2. Number of Messages, NM

The results of NM with varied err_{fd} and λ_l is shown in Figures 5 and 6 respectively. Each curve in Figure 4-5 (or Figure 4-6) has similar trend as in Figure 4-3 (or Figure 4-4). NM is determined by two factors: NR and the number of messages exchanged in one round (NMR for short). Since NMR is stable for each protocol, for the same protocol, NR dominates the change of NM.

For different protocols, however, their performance also depends on NMR, as shown by the curves in Figure 4-5 or Figure 4-6. For HMR, it has the largest NR but the smallest NM, largely due to the effect of NMR. HMR has the smallest NMR: it needs $n+n^2$ messages in each round whereas ZD and ZD-LA needs $2n^2$ messages. On the contrary, ZD and ZD-LA have the same NMR, so NR dominates the difference between the performance of the ZD protocol and ZD-LA protocol.

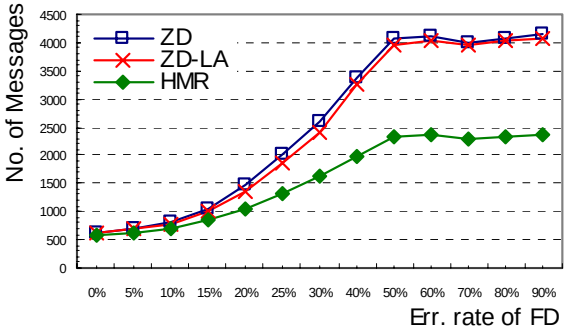

Figure 4-5 NM vs. error rate of FD

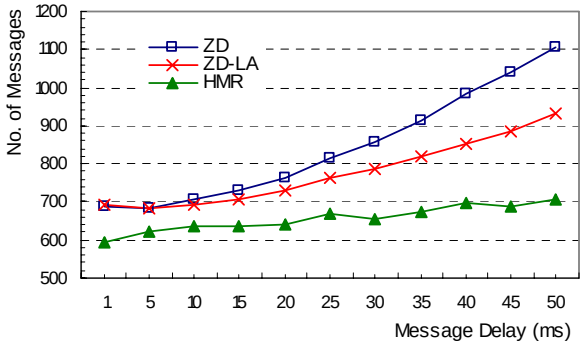

Figure 4-6 NM vs. mean link delay

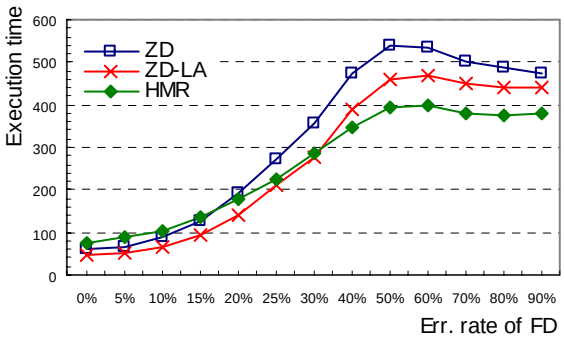

Figure 4-7 ET vs. error rate of FD

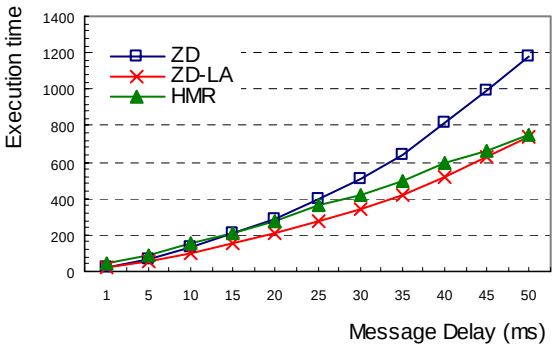

Figure 4-8 ET vs. mean link delay

4.4.3.3. Execution Time, ET

The results of ET with varied err_{fd} and λ_l are shown in Figures 7 and 8 respectively. ET is significantly affected by NR and NM. Because of the joint effect

of NR and NM as shown in Figure 4-3 to Figure 4-6, HMR achieves consensus faster than ZD only when λ_l or err_{fd} is large. Benefiting from the Look-Ahead technique, ZD-LA can always terminate earlier than the other two protocols. It is important to notice that ZD-LA can always achieve consensus with fewer rounds and shorter time than ZD, which demonstrates that the Look-Ahead technique can avoid the negative effect of future messages.

```
----------------------------------------------------Task 1: Consensus----------------------------------------------------
// The code executed by each process, p_i
initialization;
while (not decide yet){ //a new round starts;

    determine p_cc, the coordinator or leader for the new round;
    ---------------------------------- Phase 1 Collect Proposal ----------------------------------------
    { p_cc, each process } sends PROPOSAL message to all processes;

    wait until (PROPOSAL is received from { p_cc, a quorum of processes } )or (p_cc ∈ suspected)
        or [(ECHO with an estimate value updated in the current or some future round is received) ;]

    update estimate if possible;
    ----------------------------------Phase 2 Collect Echo ----------------------------------------
    send ECHO message to Π;
    { some specific processes, each process } wait until (ECHO is received from a quorum of processes)
        or [(ECHO with an estimate value updated in some future round is received) ;]

     make decision and broadcast the decision value if possible;
     update estimate if possible;
}
-----------------------------------------Task 2: Reliable broadcast ----------------------------------------
upon reception of DECISION(est) from process p_j: {
      make decision and send DECISION(est) to Π \ p_i \p_j; }
```

Figure 4-9 The Look-Ahead technique

4.5. Applying Look-Ahead to Other Protocols

In this section, we describe how to apply the Look-Ahead technique to existing oracle based consensus protocols. We first propose a general scheme of applying the technique with a consensus protocol and then illustrate the use of the scheme with examples.

4.5.1. A Scheme of Using Look-Ahead

Figure 4-9 shows the general description of the proposed technique. Consensus protocols, including FD-based and leader-based protocols, are abstracted into a general form. Statements in brackets represent different operations for FD-based protocols and leader-based protocols. The upper line is for FD-based protocols whereas the lower line is for leader-based protocols. Although existing protocols [31] [110] may be proposed in a different way, they can be easily converted into the form of Figure 4-9. The two in boxes constitute the Look-Ahead technique.

In each round, there are two phases to make the decision. In Phase 1, the coordinator or leader tries to impose its own estimate value on other processes. With the Look-Ahead technique, a process stops waiting for the proposal message if it receives an echo message with an estimate value updated in the same or some future round. Obviously, such an estimate value has been adopted by others. Because the value decided upon is unique, such a skip does not prevent any potential decision to be made.

In Phase 2, decision makers (some or all processes) wait for ECHO messages and try to make the decision based on these messages. With the Look-Ahead technique, a process stops waiting for ECHO messages if it receives an ECHO message with an estimate value updated in some future round. This will not affect any potential decision. Decision is made based on the ECHO messages from some quorum of processes, e.g. a "majority" [31][110] or "*n-f*" [82]. If a quorum of processes has changed to some future phase or round, either correctly or falsely, the remaining processes can follow them. If a process receives an ECHO message with an estimate updated in a future round, the quorum must have finished the current round, because the coordinator or the leader of the "future" round has received ECHO messages from a quorum processes in the "current" round.

The Look-Ahead technique is said to be "general" because of two important features. First, Look-Ahead can help speed up the execution of an indulgent consensus protocol in general cases (see footnote at page 6), because it does not

impose any assumption on the behavior of the oracle or processes. Second, Look-Ahead can be easily applied to nearly all round based indulgent consensus protocols in asynchronous systems, including both the leader-based and FD-based protocols.

4.5.2. Application of Look-Ahead Technique

In this section, two example protocols are given to show how to apply the Look-Ahead technique. The first example is HMR protocol [82]. The second example is a leader-based protocol proposed in [51]. We chose these two protocols because they are representative and simple. Here we present only the revised pseudocode.

```
----------Task 1: Consensus----------
// The code executed by each process, p_i
BEGIN:
(1)  r_i ←0;  est_i ← v_i;  ts_i ← 0; fld_i ← false;
(2)  while (fld_i=false){
(3)       r_i←r_i+1; cc = coord(r_i);
          ---------- Phase 1 Collect Proposal ----------
(4)       if(i=cc) send PROP(r_i, est_i) to Π;
(5)       wait until ((received PROP(r_i, *) from p_cc )
                  or (received ECHO(*, *, ≥ r_i ) from some process)
                  or p_cc ∈ suspected_i);
(6)       if(received PROP(r_i, est_cc) from p_cc){
(7)            est_i← est_cc; ts_i ← r_i; }
          else if (received an ECHO(r_i, v, r_i) ){
(8)            est_i← v; ts_i ← r_i;}
          ----------Phase 2 Collect Echo - ----------
          // D is any set that { p_cc }⊆ D⊆ Π;
          // A is any set that { p_nc }⊆ A⊆ Π, where nc = coord(r_i+1);
(9)       send ECHO(r_i, est_i, ts_i) to D∪A;
(10)      if(pi∈ D∪A) {
(11)           wait until ((received ECHO(r_i, *, *) from n-f processes)
                       or (received ECHO(*, *, > r_i) from some process));
(12)           if(received ECHO(r_i, *, *) from n-f processes and
                       there are f+1 ECHO(r_i, est, r_i) messages) {
(13)                   est_i ← est;
(14)                   send DECISION(est_i) to Π \ p_i;
(15)                   fld_i←true;}
(16)           else   est_i ← est' carried by the ECHO with the highest timestamp;
          }endif
    }//endwhile
----------Task 2:  Reliable broadcast ----------
// The code executed by each process, p_i
(17) upon reception of DECISION(est) from process p_j: {
(18)      send DECISION(est) to Π \{ p_i, p_j};
(19)      fld_i←true; }
END
```

Figure 4-10 HMR protocol with Look-Ahead technique

The revised HMR protocol with Look-Ahead technique is shown in Figure 4-10. Although HMR presents a unifying approach based on two different classes of FDs, for simplicity, only the protocol based on $\Diamond S$ is reported in Figure 4-10.

The revised leader-based protocol in [51] with Look-Ahead technique is shown in Figure 4-11. To present the protocol in the form similar to as in Figure 4-9, we change some variable and message names.

The detailed correctness proof of the revised protocols can be easily derived based on original papers.

```
----------Task 1: Consensus----------
// The code executed by each process, p_i
BEGIN:
(1)  r_i ←0;  est_i ← v_i; newest_i ← ⊥; leader_i ← ⊥;  fld_i ← false;
(2)  while (fld_i=false){
(3)       leader_i ←Ω.trusted; newest_i ← ⊥;
          ---------- Phase 1 Collect Proposal ----------
(4)       send PROP (r_i, est_i, leader_i) to Π;
(5)       wait until ((received PROP (r_i, *, *) from leader_i and
                    ⌈(n+1)/2⌉ -1other processes)
                  or (received ECHO (≥ r_i, newv) where newv≠ ⊥ from some process)
                  or leader_i ≠ Ω.trusted);
(6)       if(received PROP (r_i, ets_l, leader_i) from leader_i and
                  PORP (r_i, *, leader_i) from ⌈(n+1)/2⌉ -1 other processes){
(7)               newest_i← ets_l;}
(8)       else if(received ECHO(r_i, v) from some process ){
(9)               newest_i← v;}
          ----------Phase 2 Collect Echo - ----------
(10)      send ECHO(r_i, newest_i) to Π;
(11)      wait until ((received ECHO(r_i, *) from ⌈(n+1)/2⌉ processes)
                  or (received ECHO(>r_i, newest) where newest≠ ⊥ from some process)) ;
(12)      if(received ECHO (r_i, v) with v ≠ ⊥ from ⌈(n+1)/2⌉ processes){
(13)              est_i ← v;
(14)              send DECISION(est_i) to Π \ p_i;
(15)              fld_i←true;}
(16)      else   est_i ← v', where v' ≠ ⊥ and carried by ECHO with the highest round number;
(17)      r_i←r_i+1;
      }//endwhile
----------Task 2:  Reliable broadcast ----------
// The code executed by each process, p_i
(18) upon reception of DECISION(est) from process p_j: {
(19)      send DECISION(est) to Π \{ p_i, p_j};
(20)      fld_i←true; }
END
```

Figure 4-11 A leader-based protocol with Look-Ahead technique

4.6. Summary

This chapter is concerned with time efficiency of consensus protocols. By using two novel techniques, we proposed a $\Diamond S$ based fast consensus protocol that can circumvent the slowdowns caused by failures and false suspicions. The first technique is an efficient approach to guarantee the Round-Zero-Degradation property, which can speed up the execution of the proposed protocol when the underlying failure detector performs well. The coordinator of a round is dynamically selected based on the process status information provided by the failure detector, so as to eliminate the slowdown caused by an already crashed coordinator.

The second technique is Look-Ahead, which speeds up the execution of the proposed protocol regardless the performance of the underlying failure detector. Due to the asynchrony, some messages may be delivered to such receivers that have not yet entered the corresponding phase or round. By making use of the information carried by future messages, a “slow” process can skip some messages it is waiting for so as to speed up its execution. Besides speeding up the execution of the consensus protocol in general cases, Look-Ahead can also be easily applied to other indulgent consensus protocols for asynchronous systems. To facilitate the application of Look-Ahead technique, an abstraction of the Look-Ahead technique is presented with two examples.

Extensive simulations are conducted to evaluate the performance of the proposed protocol. The results show that, compared with existing consensus protocols, the proposed protocol can achieve consensus with fewer rounds under various conditions and shorter time when the failure detector performs well (with an error rate less than about 35%).

Chapter 5. Improving Message Efficiency of Consensus Protocols

5.1. Overview

The goal of this chapter is to improve the message efficiency of achieving consensus in MANETs. Here, the message cost is in terms of the number of hops, rather than the number of end-to-end messages as in existing works. The former can reflect the message cost of a distributed algorithm/protocol more precisely, so it is important in mobile environments, where resource constraints are serious. This is discussed in more detail in the performance evaluation part of this chapter.

We adopt the clustering approach, which has been widely used in MANETs to reduce message cost of achieving consensus. By clustering the mobile hosts into clusters, a two-layer hierarchy is established. The messages sent by the hosts in the same cluster can be merged by the clusterhead before they are forwarded to other hosts. Similarly, when a message needs to be sent to the hosts in the same cluster, it can be sent to the corresponding clusterhead, which will unmerge these messages and deliver them. In this way, the message cost can be significantly reduced. Based on different ways for clustering hosts, we propose two hierarchical protocols.

The first protocol, named “HC” (“Hierarchical Consensus”), uses a predefined set of clusterheads. The HC protocol follows the architecture of the HMR protocol [82], extending it to a hierarchical approach. Using a predefined set, some hosts are selected to act as clusterheads, and each MH is associated with one clusterhead.

However, adding the hierarchy is not trivial. First, the messages are not simply forwarded by the clusterhead, and a cluster member needs to synchronize with its clusterhead in the message exchange step. Due to the mobility and clusterhead failure, a MH may need to switch between clusterheads that are executing different steps. Therefore, the switch procedure should be delicately handled in order to maintain the synchronization between a MH and its clusterhead. Second, nearly all

consensus protocols, including the CT protocol [31], HMR protocol [82] and BHM protocol [15], requires that no message can be lost. However, the change of the hierarchy in a MANET may cause message losses, even if the communication channel is reliable. To cope with such message losses, some “redeeming” messages should be sent. What and when such messages should be sent depends on the execution state of the MH and its clusterhead. In HC, we develop efficient mechanisms to send and handle redeeming messages.

Unfortunately, the HC protocol has three problems. First, the function of achieving consensus is tightly coupled with the function of clustering. When a MH switches to a new cluster, its execution has to be changed with respect to the status of the new clusterhead. Such a design makes the protocol complicated. Second, the set of clusterheads is predefined, so it cannot adapt to the crashes that occur during the execution, which delays the decision making. Finally, the protocol requires a failure detector of class $\Diamond P$, which is stronger than the weakest and most commonly used failure detector $\Diamond S$ [31].

To address these problems, we therefore propose the second hierarchical protocol, named HCD (“Hierarchical Consensus based on Delta”). The functions of clustering hosts and achieving consensus are separated using a modular approach. The clustering function, named *eventual clusterer* (denoted by Δ), is proposed to construct and maintain the cluster-based hierarchy over MHs. Since Δ provides the fault tolerant clustering function transparently, it can be used as a new oracle for the design of hierarchical consensus protocols.

Base on Δ, we design the HCD protocol. With the help of Δ, the problems of HC are easily solved. However, how to handle the change of the clusterhead of a MH must be seriously considered. When a host switches from one cluster to another, the consensus protocol must change its state to adapt to the new clusterhead. Since the clustering procedure is transparently carried out by Δ, the consensus protocol cannot participant in the switch procedure. In HCD, we adopt a variant of the Look-Ahead

technique to solve this problem. The basic idea is that once a mobile host finds that its new clusterhead is in a higher round than its own, it skips to that round.

In the rest of this chapter, we first describe the HC protocol. Then, the clusterer oracle Δ is defined and implemented based on $\Diamond S$. Finally, the HCD protocol is presented.

5.2. The HC Protocol

5.2.1. System Model

We consider in a MANET that consists of a set of n (n>1) MHs, $M = \{m_1, m_2,\ldots, m_n\}$. All MHs are distributed into clusters. Some of the MHs are selected as clusterheads, and each is in charge of one cluster. The number of clusterheads is denoted by k. A MH can only fail by crashing, i.e. prematurely halting, but it acts correctly until it possibly crashes. A MH that crashes in a run is faulty in that run, otherwise it is correct. The maximum number of faulty MHs in a run, denoted as f, is bounded by k and $n/2$, i.e. $f < minimum(k, n/2)$.

MHs communicate by sending and receiving messages. Every pair of MHs is connected by a reliable channel that does not create, duplicate, alter, or lose message. It is important to notice that the assumption on reliable channels can be reduced to one on lossy channels, which are more feasible for MANETs, but require a much more complicated design. This is discussed in Section 5.2.6. For simplicity, we assume the channels are reliable in the description of the HC protocol.

The system is equipped with an unreliable FD of class $\Diamond P$. $\Diamond P$ is defined using the following properties:

Strong Completeness: eventually each crashed process is permanently suspected by each correct process.

Eventual Strong Accuracy: there is a time after which every correct process is not suspected by any correct process.

Among all the eight classes of FDs proposed by Chandra and Toueg, $\Diamond S$ is the weakest but strong enough to solve the consensus problem [30][31]. $\Diamond P$ has stronger

accuracy property than $\Diamond S$, but it has been proved that $\Diamond P$ and $\Diamond S$ are equivalent in the power of solving the consensus problem [64]. Although $\Diamond P$ is stronger than $\Diamond S$, existing implementations of $\Diamond P$ [31][93] are not more complex than those of $\Diamond S$. Of course, $\Diamond P$ may take more time to reach a stable state.

Our protocol uses $\Diamond P$ instead of $\Diamond S$ because the eventually strong accuracy property is necessary to guarantee the termination. There are two necessary conditions to guarantee the termination of our HC protocol. First, there is at least one correct host to act as a clusterhead eventually. This can be satisfied by including more than f MHs in the set of clusterheads. Second, after some time, some correct clusterhead must be no longer suspected by any correct MH. A FD of class $\Diamond S$ can only guarantee that at least one correct host is never suspected after some time. However, such a host may not be a clusterhead. Therefore, $\Diamond P$ is necessary to satisfy the second condition (see the proof in Section 5.2.4 for more details).

5.2.2. Data Structures and Message Types

When executing the protocol, each host needs to maintain necessary information about its state. Such information is stored in the following variables.

fl_i: the flag indicating whether m_i has made the decision. The initial value is *false*.

r_i: the sequence number of the current round that m_i is participating in.

ph_i: the phase number of the current phase that m_i is participating in.

est_i: the current estimate of the decision value. Initially, it is set to the value proposed by m_i.

ts_i: the timestamp of est_i. The value is the round number of the round in which m_i receives the est_i proposed by the coordinator host. The update of ts_i is entailed by the reception of estimate from a coordinator.

During the execution of the protocol, the mobile hosts need to communicate with each other by exchanging messages. The message types involved in the proposed protocol are as follows.

PROP(r, est_{cc}): the proposal message sent from the coordinator to clusterheads or from a clusterhead to the hosts in its cluster. est_{cc} is the current estimate kept by the coordinator. In each round, the coordinator tries to impose est_{cc} on other hosts by sending proposal messages.

ECHOL(r, est_i, ts_i): the echo message from m_i to its clusterhead in the round r.

ECHOG(r, v, ts_v, x, y): the echo message from a clusterhead to other clusterheads in round r. *ECHOG*(r, v, ts_v, x, y) is constructed by merging the *ECHOL* messages in the same cluster. v is the estimate carried by the *ECHOL* message with the highest timestamp and ts_v is the timestamp of v. x is the set of MHs that send the *ECHOL* message with ts_v whereas y is the set of MHs that send other *ECHOL* messages.

LEAVE(r, sn): the message sent by a MH to its clusterhead to inform the clusterhead that the MH wants to disassociate itself from the current cluster. sn is the sequence number to distinguish different *LEAVE* messages from the same host.

JOIN(r_i, sn): the message sent by a MH to the clusterhead of a new cluster that the MH wants to join. sn is a sequence number to distinguish *JOIN* messages from the same host.

DECISION(est): the message sent by a MH to broadcast the decision value est.

PROPH(r, est_{cc}): same as a *PROP* message except that this is for a MH that newly joins.

5.2.3. Operations of HC Protocol

A two-layer hierarchy is imposed on the network of MHs. The *Clusterhead layer* consists of a predefined set H of MHs which act as clusterheads to merge/unmerge and forward messages for the MHs. The *Host layer* consists of a set M of all MHs, including those in set H.

Only the hosts in set H can act as coordinators, *decision_makers*, or *agreement_keepers*. To guarantee the termination of the protocol, at least one correct host should be included in H, i.e. $|H| = k \geq f+1$. Each host chooses the nearest[7]

[7] A threshold of the distance difference between the distance to the old and new clusterheads can be set.

unsuspected clusterhead in *H* as its clusterhead. The distance between two hosts is defined as the path length in hops. Such distance information can be obtained through the underlying routing protocol, which is in charge of the establishment and maintenance of the path between any two hosts[8]. Obviously, a clusterhead host always chooses itself. The hosts that choose the same clusterhead constitute a cluster. A host associated with a clusterhead is called a "*local host*" of the clusterhead and correspondingly, the clusterhead is called "*local clusterhead*" of its local hosts.

To balance the workload and energy consumption, MHs can take turns (e.g. according to some deterministic function) to serve as clusterhead for different runs of the protocols. Since the only requirement for forming the *H* set is that at least one clusterhead is correct, it does not matter that a crashed host appeared in the *H* set, and MHs can always find and join a correct clusterhead in *H*. Although *H* remains unchanged for each run of the protocol, it can be periodically re-formed, and a MH can switch to be a clusterhead and vice versa.

The proposed protocol consists of four tasks. Like most existing consensus protocols, Task 1 is the main body of the protocol for making decision and Task 2 is a simple broadcast algorithm for propagating the value decided upon. Other two additional tasks are designed in our protocol. Task 3 is used to handle late *ECHOL* messages arrived at a clusterhead and Task 4 is used to switch the cluster of a host. The pseudocode of Task 1 and Task 2 is shown in Figure 5-1 and the pseudocode of Task 3 and Task 4 is shown in Figure 5-2. In the following, we describe the tasks in more detail.

Task 1: This task consists of two phases. In the beginning of round *r*, the current coordinator m_{cc} sends $PROP(r, est_{cc})$ to the hosts in set *H*. Upon receiving the $PROP(r, est_{cc})$ message from m_{cc}, a clusterhead forwards the *PROP* message to all its local hosts. If a clusterhead suspects m_{cc} before receiving $PROP(r, est_{cc})$, it sends a $PROP(r, \perp)$ message to its local hosts, where "$\perp$" is a value that can never be

[8] For geographical routing protocols, the "distance" can be defined as the geographical distance between two hosts. The distance information can still be obtained through the underlying routing protocol.

proposed or adopted. A host m_i waits until a *PROP*(r, -) message is received from its local clusterhead, the local clusterhead is suspected, or its local clusterhead is no longer the nearest one. The symbol "–" in the message means any possible value. If a *PROP*(r, v) message with $v \neq \perp$ is received, m_i updates its estimate value to v and timestamp to r. If the local clusterhead is suspected or its local clusterhead is no longer the nearest one, m_i invokes Task 4, the "switch" procedure, to associate with another clusterhead, which will be presented later. Then Phase 1 is finished.

In Phase 2, the message exchange pattern is determined by a set *DA*, the set of *decision_makers* and *agreement_keepers*. Same as in HMR, *decision_makers* (in set *D*) are the hosts that have to check the decision predicate that allows them to know if they can decider during the current round; *agreement_keepers* (in set *A*) are the hosts that should keep the updated estimate of the final decision. Different from HMR, we use a single set to store both the *decision_makers* and *agreement_keepers*. The roles of a *decision_maker* and an *agreement_keeper* are the same in terms of message exchange. Combining the sets *D* and *A* can help increase the probability of making decision in a round without any additional overhead caused. Therefore, in HC, each host in *DA* simultaneously plays two roles: *decision_maker* and *agreement_keeper*. *DA* is defined by the function *dec_agr*(r), which has to satisfy the following three constraints:

i) *dec_agr*(r) is deterministic so that all hosts have the same *DA* in the same round.

ii) *dec_agr*(r) contains only clusterheads, i.e. $DA \subseteq H$.

iii) *dec_agr*(r) contains the coordinator of the rounds r and r+1.

Phase 2 is started by sending *ECHOL* messages. Each host first sends an echo message *ECHOL*(r_i, est_i, ts_i) to its local clusterhead. If the host itself is not a clusterhead, it enters the next round r+1. Each clusterhead waits for an echo message *ECHOL*(r, -, -) from each local host that is not suspected. Then each clusterhead constructs an echo message *ECHOG*(r, v, ts_v, x, y) by merging the *ECHOL*(r, -, -) messages collected. v is the estimate value carried by the *ECHOL*(r, -, -) message with the highest timestamp and ts_v is that timestamp. x is the set of the hosts that

send the *ECHOL*(*r*, -, -) messages with ts_v whereas *y* is the set of the hosts that send *ECHOL*(*r*, -, -) messages with other timestamps.

```
----------Task 1: Consensus----------
// The code executed by each host, m_i
COBEGIN:
(1) r_i ←0; est_i ← v_i; ts_i ← 0; fl_i ← false;
    while (fl_i≠true){
(2)     r_i←r_i+1; ph_i←1; cc = coord(r_i);
---------- Phase 1: from m_cc to clusterheads ----------
       //p denotes the local clusterhead of m_i
(3)    if(i=cc) send PROP(r_i, est_i) to H;
       if(m_i∈H) {
(4)       wait until (PROP(r_i, est_cc) is received or
                                   m_cc ∈ suspected_i);
(5)       if(PROP(r_i, est_cc) message received from p_cc)
                broadcast (PROP(r_i, est_cc) locally;
(6)       else broadcast (PROP(r_i,⊥) locally; }//endif
(7)    wait until PROP(r_i, v) from p is received or
                p is suspected or p is not the nearest one ;
(8)    if(PROP((r_i, v) is received and v ≠ ⊥){
          est_i← v; ts_i←r_i;}
(9)    if(p is suspected or p is not the nearest one)
          invoke Task 4;
----------Phase 2: from all to H ----------
       ph_i←2;
(10)   send message ECHOL(r_i, est_i, ts_i) to  p;
       if (m_i∈ H) {
(11)      wait until an ECHOL(r_i, -, -) is received from
                each local host m_j or m_j∈ suspected_i;
(12)      merge the ECHOL messages{
             ts_v←the highest timestamp;
             v← the estimate of the ECHOL with ts_v;
             x← the hosts that send ECHOL with ts_v;
             y← the hosts that send other ECHOL;}
(13)      send ECHOG(r_i, v, ts_v, x, y) to DA;
          if(m_i∈ DA){
(14)         wait until ((∪ x∪ y)of ECHOG(r_i, -, -, x, y)
                received includes at least n-f hosts) or
                (ECHOG(-,-,>r_i,-,-) received);
(15)         if(i≠cc) est_i←the est  with the highest ts;
(16)         if(ECHOG with (ts= r=r_i) represent
                at least (f+1) hosts){
                fl_i←true;
(17)            ∀j ≠ i: send DECISION(est_i) to m_j;}
          }
       }
}
----------Task 2:  Reliable broadcast ----------
(18) upon reception of DECISION(est) from host m_k:
        fl_i←true; ∀j ≠ i, k: send DECISION(est) to m_j;
COEND
```

Figure 5-1 HC protocol - Task 1and Task 2

```
----------Task 3: Handling Late ECHOL----------
// The code executed by each clusterhead;
while (fl_i≠true){
(19)  upon reception of ECHOL (r,v,ts) with (r<r_i) or (r=r_i
and an ECHOG(r_i, *, *, *,*) has been sent);
(20)      construct an new ECHOG and send it to DA;}
----------Task 4: Clusterhead switch ----------
--------Task 4.1: code executed by host m_i ----------
(21) while(fl_i≠true and (p∈ suspected_i or
                  p is not the nearest one)) {
(22)    sn←sn+1; q ←the nearest unsuspected clusterhead;
(23)    send a LEAVE(r_i, sn) to p;
(24)    send a JOIN(r_i, sn) to q;
(25)    wait until PROPH(r_q, v)received or q∈ suspected_i;
        if (PROPH(r_q, v) received){
           if(r_i< r_q){
(26)         r_i ← r_q;
(27)         for(ts_i ≤ rr<r_i) send ECHOL(rr, est_i, ts_i) to q;
(28)         if(v ≠ ⊥){ est_i← v; ts_i←r_i;}
(29)         GOTO (10);
           }else if (r_i = r_q){
              if(ph_i =1){
(30)            for(ts_i≤ rr< r_i) send ECHOL(rr, est_i, ts_i) to q;
(31)            if(v ≠ ⊥){est_i← v; ts_i←r_i;}
(32)            GOTO (10);
             }else if (ph_i =2){
(33)            for(ts_i≤ rr≤ r_i) send ECHOL(rr, est_i, ts_i) to q;
(34)            r_i ← r_i+1; GOTO (4);}
           }else if(r_i > r_q){
              if(ph_i =1){
(35)            for(ts_i≤ rr< r_i) send ECHOL(rr, est_i, ts_i) to q;
(36)            GOTO (4); }
             else if (ph_i =2){
(37)            for(ts_i≤ rr≤ r_i) send ECHOL(rr, est_i, ts_i) to q;
(38)            r_i ← r_i+1; GOTO (4);}}
        } else GOTO (22);
    }
----------Task 4.2: code executed by clusterhead g------
    while(fl_i≠true){
       upon reception of LEAVE(r_i, sn) from host m_i{
(39)       delete m_i from local host list;}
       upon reception of JOIN(r_i, sn) from host m_i {
          add m_i to local host list;
(40)      if(ph_g=2){
             if(PROP(r_g, est_cc) received from m_cc)
                send PROPH(r_g, est_cc) to m_i;
(41)         else send PROPH(r_g, ⊥) to m_i;}}
    }//endwhile;
```

Figure 5-2 HC protocol - Task 3 and Task 4

The clusterhead then sends the *ECHOG*(r, v, ts_v, x, y) message to the hosts in the sets *DA*. Each clusterhead in *DA* waits for *ECHOG* messages until: 1) the *ECHOG*(r,-,-,-,-) messages received represent no less than (n-f) hosts, or 2) an *ECHOG*(-,-,ts_v,-,-) with $ts_v > r$ is received. Here, "represent" means the host is included in the set x or y of the *ECHOG* message. A clusterhead in *DA* updates its estimate to the value carried by the *ECHOG* message with the highest timestamp, but keeps the timestamp unchanged. Finally, a clusterhead in *DA* checks whether it can decide in the current round. If there are f+1 or more hosts in x sets of the *ECHOG*(r, v, ts_v, x, y) messages with $ts_v = r$, it decides upon the value v and broadcasts the final value.

Task 2: Task 2 simply broadcasts the decision value. When a host receives a *DECSION* message, it decides upon the same value and forwards the DECSION message to all other hosts except the sender.

Task 3: Task 4 handles the late *ECHOL* messages. An *ECHOL* message is "late" if it arrives at a clusterhead after the clusterhead has sent out an *ECHOG* message for the corresponding round. This happens when a clusterhead p suspects a correct local host or a host m_i joins a new cluster where the clusterhead is in a round greater than the ts_i. The hosts in set H may be blocked forever if a late *ECHOL* message is ignored. To avoid this, when a clusterhead p receives an *ECHOL*(r_i, est_i, ts_i) with ($r_i < r_p$) or ($r_i = r_p$ but p has sent out an *ECHOG* for the round r_i), p constructs a redeeming *ECHOG* for m_i and sends it to all clusterheads.

Task 4: This task is for a mobile host to switch its clusterhead. It is invoked when a host m_i suspects its current clusterhead p or p is no longer the nearest clusterhead. m_i needs to choose a new clusterhead q, which is the nearest among the unsuspected clusterheads. First, m_i sends a message *LEAVE*(r_i, sn) to p and a message *JOIN*(r_i, sn) to q. Upon reception of the leave message, p deletes m_i from its local host list. Upon reception of the join message, q adds m_i to its local host list. Then if q is in Phase two it sends *PROPH*(r_q, est_{cc}) or *PROPH*(r_q, $\perp$) to m_i as it has sent to other local hosts in

Phase one. Upon reception of the *PROPH*(r_q, w) message from q, the behaviors of host m_i can be classified into 3 cases.

Case 1: ($r_i < r_q$) or ($r_i = r_q$, $ph_i = 1$): m_i updates its round number to r_q and sends *ECHOL*(rr, est_i, ts_i) messages to q where $ts_i \leq rr < r_q$. If $w \neq \perp$, m_i sets its estimate to w and timestamp to r_q. m_i then resumes the normal execution by entering Phase 2 of round r_q.

Case 2: ($r_i > r_q$, $ph_i = 1$): m_i sends *ECHOL*(rr, est_i, ts_i) messages to q where $ts_i \leq rr < r_i$ and then resumes the normal execution by continue the Phase 1 of round r_i.

Case 3: ($r_i = r_q$, $ph_i = 2$) or ($r_i > r_q$, $ph_i = 2$): m_i sends *ECHOL*(rr, est_i, ts_i) messages to q where $ts_i \leq rr \leq r_i$ and then resumes the normal execution by entering the next round r_i+1.

5.2.4. Correctness of the HC Protocol

Since the validity property of the proposed protocol is obvious, in this section, we only present proofs for the termination property and agreement property. The term "*indirect suspicion*" used here refers to the situation that a host itself does not suspect the current coordinator but it receives a *PROP*(r, $\perp$) from its clusterhead.

5.2.4.1. Termination

Lemma 1. *If no host decides in a round $r' \leq r$, then all correct hosts eventually start round $r+1$.*

Proof. If some correct host blocks forever before round $r+1$, then there must be a smallest round, say *rs* ($rs < r+1$), during which some correct host is blocked forever. Therefore, we only need to prove that "no correct host can be blocked in the round *rs* forever." The proof is by contradiction.

Assume that some correct host m_i is blocked forever in round *rs*. Then m_i must be blocked in a wait statement, i.e. lines 5, 8, 25, 11 or 14, in the round *rs*. Let us analyze these cases one by one.

Case 1: m_i is blocked at line 4. Obviously, m_i is a clusterhead. If $i = cc$, m_i cannot be blocked (it receives the proposal message sent by itself). Then, $i \neq cc$. If the

coordinator m_{cc} is a correct host, m_i eventually receives the proposal message from m_{cc}. If m_{cc} is a faulty host, m_i eventually suspects m_{cc} after m_{cc} crashes. Therefore, m_i cannot be blocked forever at line 4.

Case 2: m_i is blocked at line 7. If m_i is a clusterhead, it is the local clusterhead of itself. Since m_i cannot be blocked forever at line 4, it eventually receives the *PROP*(r,-) message sent by itself at line 5 or 6. If m_i is not a clusterhead, on the other hand, there are two possible situations. Let p be the local clusterhead of m_i. If p is a correct host and keeps to be the nearest to m_i, it eventually sends out a *PROP*(r, -) message (the clusterhead cannot be blocked at line 4 forever) and m_i eventually receives it. Otherwise, after p crashes or turns to be no longer the nearest to m_i, m_i eventually suspects it and invokes the clusterhead switch procedure. Therefore, m_i cannot be blocked forever at line 7.

Case 3: m_i is blocked at line 25. Obviously, the clusterhead switch procedure has been invoked. There are two possible cases. If the new clusterhead selected is a faulty host, m_i eventually suspects it after it crashes and invokes the clusterhead switch procedure again. Since at least one clusterhead is correct ($k \geq f+1$), m_i eventually finds a correct clusterhead (by the eventually strong accuracy of $\Diamond P$). This case turns to be the second one. For the second case, i.e. the new clusterhead selected is a correct host, it eventually sends a *PROPH*(r,-) message to m_i (no host is blocked at line 4 forever) and m_i eventually receives the message. Therefore, m_i cannot be blocked forever at line 25.

Case 4: m_i is blocked at line 11. Obviously m_i is a clusterhead and is waiting for *ECHOL* messages from its local MHs. All hosts in the local host list of m_i can be categorized into three classes: a) faulty hosts, b) correct hosts that have left m_i (but m_i has not received their *LEAVE* messages) and c) the other hosts. For hosts in class a), m_i eventually suspects them after they crash. For hosts in class b), each of them must have sent a *LEAVE* message to m_i before it leaves m_i (line 23). m_i eventually receives the *LEAVE* messages and deletes them from local host list. For class c), m_i

eventually receives an *ECHOL* from each of them because they cannot be blocked at line 4, line 7 or line 25. Therefore, m_i cannot be blocked forever at line 11.

Case 5: m_i is blocked at line 14. Obviously, m_i is a *decision_maker* and *agreement_keeper*. There are two possible conditions to unblock m_i: 1) m_i receives *ECHOG* messages that can represent no less than *n-f* hosts or 2) m_i receives an *ECHOG* message with timestamp *ts*>*rs*. We now prove that at least one of the two conditions is satisfied eventually. Since at most *f* hosts can crash, there are at least *n-f* correct hosts. By assumption, *rs* is the smallest round in which a correct host is blocked forever, so all these *n-f* correct hosts eventually proceed to the round *rs* and execute line 10. Then we categorize all the correct hosts into two classes:

i) the hosts with correct clusterheads when they execute line 10, and

ii) the hosts with faulty clusterheads when they execute line 10.

For a host m_j in class i), the local clusterhead of m_j eventually receives m_j's *ECHOL* message and includes m_j in an *ECHOG* message to m_i.

For a host m_j in class ii), after its clusterhead crashes, m_j eventually invokes the cluster switch procedure, finds a correct clusterhead host *q*, after one or more cluster switches, and receives a *PROPH*(r_p,-) message from *q*. Then we consider different situations according to ts_j:

ii. a) if $ts_j \leq rs$, an *ECHOL*(*r*, est_j, ts_j) is sent to *q* at line 27, 30, 33, 35 or 37;

ii. b) if $ts_j > rs$, an *ECHOL*(-,-,>*rs*) is sent to *q* at line 27, 30, 33, 35 or 37.

Considering *q* is a correct host, it eventually includes m_j in an *ECHOG* to m_i. Let examine the ECHOG messages received by m_i in round *rs*. If some host belongs to class ii)-b), m_i eventually receives an *ECHOG*(-,-,>*rs*,-,-) and consequently condition 2) is satisfied; otherwise all *n-f* correct hosts belong to class i) or ii)-a), and m_i eventually receives enough *ECHOG*(*r*,-,-,-,-), i.e. the condition 1) is satisfied. Therefore, m_i cannot be blocked forever at line 14. □

Lemma 2. *For any round r, if the coordinator c_r sends out a PROP(r, v) at time tr and less than n-f hosts suspect c_r directly or indirectly in Phase 1 of r, then no PROP(r', v) with r'>r can be sent out before tr.*

Proof. The proof is by contradiction. Assume that at least one *PROP*(*r'*, *v*) message with *r'*>*r* has been sent out by the time *tr*. Let *rm* be the greatest round number of all the *PROP*(*r'*, *v*) messages that have been sent out by time *tr*, then *rm*>*r* and *rm*-1≥ *r*. Obviously the coordinator of the round *rm*, i.e. the host c_{rm}, must have finished line 14 in round *rm*-1 (since it has sent out *PROP*(*rm*, *v*) before *tr*). Since the timestamp of the estimate at any host can only be changed at line 8, 28 or 31, and *rm* is the greatest round number in *PROP*(*r'*, *v*) messages by time *tr*, c_{rm} must have not received a *ECHOG* with *ts*>*rm*-1 in round *rm*-1. Therefore, c_{rm} must have received *ECHOG* messages representing at least *n-f* hosts at line 14 of round *rm*-1. This means that at least *n-f* hosts sent out *ECHOL*(*rm*-1,-,-) messages in round *rm*-1 before time *tr*, so at least *n-f* hosts finished *Phase* 1 of round *rm*-1 before time *tr*. Since *rm*-1≥ *r*, at least *n-f* hosts finished *Phase* 1 of the round *r* before c_r sends out *PROP(r, v)* in the round *r*. Therefore at least *n-f* hosts suspected c_r directly or indirectly in *Phase* 1 of round *r*, which contradicts the assumption in the lemma.□

Corollary 1. *In any round r, if the coordinator of r+1, c_{r+1}, receives an ECHOG message with ts>r, then at least n-f hosts suspect c_{r+1} directly or indirectly in Phase 1 of round r+1.*

Proof. By the assumption in the lemma, at least one *PROP* message with round number *s*>*r* has been sent out. Since c_{r+1} has not yet finished round *r*, no *PROP* message with round number *r*+1 can be sent out. Therefore, *s*>*r*+1. By Lemma 2, at least *n-f* hosts suspect c_{r+1} directly or indirectly in Phase 1 of round *r*+1.□

Theorem 1. *If a host is correct, it eventually decides.*

Proof. If one host decides, all correct hosts eventually decide due to the reliable broadcast mechanism (lines 17 and 18). Therefore, we only prove that at least one host decides. The proof is by contradiction.

Assume that no host decides. According to the accuracy and completeness of ◊*P*, there is a time *t* after which all correct hosts are never suspected by any correct host and all faulty hosts are permanently suspected by every correct host after they crash. Since there is at least one correct host m_x, in set *H* after time *t* (*k*≥ *f*+1), every correct

host eventually associate itself with a correct clusterhead. Let r be the first round coordinated by m_x and started after t. By the assumption (no process decides) and Lemma 1, all correct hosts eventually enter round r. Since no new suspicion occurs after time t and at most f hosts can crash, there are at least n-f correct hosts that execute round r. By Corollary 1, m_x cannot receive an *ECHOG* with $ts>r$ at line 14, so m_x eventually decides in round r, which contradicts the assumption "there is no host decides."□

5.2.4.2. Agreement

Lemma 3. *Let r be the first round in which f+1 hosts send ECHOL(r, v, r) and r' be any round that r' ≥ r. Then:*

1) No host decides before r;

2) If the coordinator of r' sends a PROP message, this message carries the estimate value v.

Proof. Proof for 1): The proof is by contradiction. If no host decides at line 16, no host can decide at line 18, we therefore only consider the decision at line 16. Assume that some host m_j decided at line 16 in some round s before r, i.e. $s < r$ and the decision value is u. m_j must have received at least one *ECHOG* message carrying a timestamp equal to s and the union set of the x sets in those *ECHOG* messages includes at least f+1 hosts. Since all *ECHOG* messages are constructed based on *ECHOL* messages, at least f+1 *ECHOL*(s, u, s) must have been sent out. From the definition of r ("…first round in which…"), we have $r \leq s$, which contradicts the assumption $s<r$. Part 1) holds.

Proof for 2): In any round r, the timestamp ts of the estimate at any host can only be changed to r at line 9, 28 or 31. By the assumption in the lemma, a *PROP*(r, v) has been sent out by c_r, the coordinator of round r and at least f+1 hosts have received the *PROP*(r, v) in *Phase* 1 of round r. Let tp be the moment that c_r sent out the *PROP*(r, v) message. Since n-(f+1)<n-f, by Lemma 2, all *PROP*(r', -) messages with $r'>r$ must be sent out after time tp. Let R be the list of the round numbers of all *PROP*(r', -) messages with $r'>r$. Without loss of generality, we assume R= (r_0=r, r_1,

r_2, $r_{3,...}$,$r_{i,...}$), where the round numbers are sorted in the ascending order of the moments when the corresponding *PROP* messages are sent out.

Now, we prove that for each round r_i in R, the proposal value carried by *PROP*(r_i, u) is equal to v, i.e. u=v. The proof is by induction on the sequence number i in R.

Base case: i=0. According to the HC protocol, a host sends an *ECHOL(r, v, r)* only if it has received a *PROP*(r, v) or *PROPH*(r, v). Therefore, the local clusterhead of this host must have received a *PROP*(r, v). The lemma holds.

Induction hypothesis: i>0. Assume that the lemma holds for any round r_i such that $0 \leq i \leq k$, we show that the lemma holds for round r_{k+1}. Now, we define two sets of hosts.

- The set G includes all the hosts that have received a *PROP*(r_i, w) or *PROPH*(r_i, w) message with $0 \leq i \leq k$. By the induction hypothesis, $\forall m_j \in G$: $est_j = w = v$ and $ts_j = r_i$. Since at least f+1 hosts send *ECHOL*(r, v, r), $|G| \geq f$+1.
- The set B includes the hosts that have not received a *PROP*(r_i, w) message with $0 \leq i \leq k$. Obviously, $\forall m_j \in B$: $ts_j < r$. Therefore, all timestamps of the hosts in set B are less than those of the hosts in set G.

Now Let us consider the behaviors of host c_{rk+1} in Phase 2 of the round (r_{k+1})-1. By the definition of *DA*, $c_{rk+1} \in DA$ during the round (r_{k+1})-1, so c_{rk+1} waits for the *ECHOG* messages at line 14 of the round (r_{k+1})-1. There are two conditions to stop the wait at line 14.

1) c_{rk+1} receives an *ECHOG*(-, u, tsm, -, -) with tsm >(r_{k+1})-1. Then c_{rk+1} updates its estimate to the value u at line 15. In fact the value u must come from an *ECHOL*(-, u, tsm), so the sender of this *ECHOL* must have received a *PROP*(tsm, u) or *PROPH*(tsm, u). This means that the local clusterhead of the sender of this *ECHOL* message must have received a *PROP*(r, u). By the definition of R and the induction hypothesis, $tsm \in \{r_0, .., r_k\}$, so u=v.

2) c_{rk+1} receives *ECHOG*((r_{k+1})-1,-, -, -, -) messages that can represent at least n-f hosts, which means that at least n-f message *ECHOL*((r_{k+1})-1, -, -) are merged. Let X denote the set of the hosts that sent these *ECHOL* messages. Obviously, $|X| \geq n$-f. At

line 15, c_{rk+1} updates its estimate to the value u carried by the echo message $ECHOL((r_{k+1})\text{-}1, u, tsm)$, where tsm is the highest timestamp. Since $|G| \geq f+1$, $G \cap X \neq \emptyset$. Therefore, the $ECHOL((r_{k+1})\text{-}1, u, tsm)$ message must be sent by a host in G. By the definition of G, $u=v$.

Then for both cases 1) and 2), the estimate value of c_{rk+1} is updated to v in round (r_{k+1})-1 and consequently in round r_{k+1}, c_{rk+1} sends out a $PROP(r_{k+1}, v)$. The lemma holds.□

Theorem 2. *No two hosts decide upon different values.*

Proof. If a host decides upon a value at line 18, then this value must have been decided upon by another host at line 16. Therefore, we only consider values decided upon at line 16.

Let m_i be a host that decides upon a value v_i in round r_i. Since m_i decides in round r_i, it has received at least one $ECHOG(r_i, v_i, r_i, -, -)$ message. Therefore, the coordinator of round r_i had sent out a $PROP(r_i, v_i)$. Similarly, if another host m_j decided upon another value v_j in round r_j, the coordinator of round r_j must have sent out $PROP(r_j, v_j)$. Let r be the round characterized in Lemma 3 (the first round in which f+1 hosts send $ECHOL(r, v, r)$). By Lemma 3, $r \leq r_i$ and $r \leq r_j$, so $v=v_i=v_j$.□

5.2.5. Performance Evaluation

In this section, we evaluate and compare the performance of the HC protocol, the HMR protocol and the BHM protocol by simulations in a MANET environment.

5.2.5.1. Performance Metrics

Besides the three metrics used in Chapter 4, a new metric "*NH*" is used here. In a MANET, the concepts of "message" and "hop" must be distinguished. In traditional distributed systems, the performance is computed in terms of the number of messages, where one "message" means one "end-to-end" message. However, one message may take one or more hops to reach the destination in the underlying network. One "hop" means one network layer message, i.e. a point-to-point message. In traditional systems, messages that cost different number of hops are regarded as

messages with the same cost. However, in a MANET, the resource constraint is serious. We propose the new metric, the number of hops to measure the message cost more precisely. Therefore, here, we use the following four metrics:

NR (*Number of Rounds*): the average number of rounds executed by the hosts to achieve consensus.

ET (*Execution Time*): the "real" time needed by a consensus protocol to achieve the global decision.

NM (*Number of Messages*): the total number of messages exchanged to achieve the global decision. For the HC protocol, the messages for cluster switch are also included.

NH (*Number of Hops*): the total number of hops of the messages exchanged to achieve the global decision.

5.2.5.2. Simulation Setup

The simulation system consists of three modules: mobile network, FD and consensus protocol. The main parameters of the simulations are shown in Table 5-1.

All hosts are randomly scattered in a rectangular territory. To evaluate the scalability of the protocols, we varied the number of hosts (i.e. the system scale) and accordingly the territory scale in proportion, so that the performances under different number of hosts are comparable. To simulate the movements of hosts, the well-known random waypoint mobility model [26] is adopted. The mobility level, defined as the percentage of the time that

Table 5-1 Simulation settings for the hierarchical consensus protocol

No. of the Hosts	10 to 100
Territory (m)	200 to 630
f/*n*	10% to 50%
Mean life of crashed hosts	30 ms
Stabilization interval	600 ms
Transmission radius	100 m
Mean link delay	5 ms
Max link delay (after stabilization interval)	100 ms
Error rate of failure detector	10%
Interval of Heartbeat messages	10 ms
Routing Protocol/Policy	Least hops
Threshold of clusterhead switch	2 hop
Min Speed	10 m/s
Max Speed	30 m/s
Mobility model	Random Waypoint
Mobility Level	50%

a host does move over the total life time of the host, is fixed to 50%. The speed of the movements satisfies the uniform distribution between 10m/s and 30m/s.

To guarantee the properties of $\Diamond P$, the network is set to be partially synchronous [31]: the bounds on the message delay and processing speed are unknown and hold only after an unknown stabilization interval. Each host can crash, but the total number of crashes is bounded by f. We varied f by changing the value of f/n from 10% to 50%. The life time of a faulty host satisfies the exponential distribution.

For message routing, we implemented a simple protocol based on the "least hops" policy, which is adopted in many classical routing protocols in MANETs, such as AODV [127], DSDV [126] and DSR [87]. A routing table is maintained at each host proactively. The message delay is also assumed to satisfy an exponential distribution. The threshold of clusterhead switch in the HC protocol is 2 hops.

FD is simulated using a heartbeat mechanism which is the adopted in nearly all implementations of unreliable FDs [93]. Each host is augmented with a FD module. FD modules make mistakes randomly with an average error rate of 10%. However, to guarantee the properties of $\Diamond P$, no mistake is made after the stabilization interval.

All the three protocols are implemented as separate modules at each host. For HMR, a variant with a single set *DA* of *decision_makers* and *agreement_keepers*, as in our HC protocol, is simulated. As to the BHM protocol, since it relies on MSSs, it cannot be implemented in a MANET directly. Because there is no MSS in MANETs, we simulated a variant of the BHM by selecting $2f+1$ MHs as the privileged hosts. The privileged hosts execute the HMR protocol with $|DA|=2$ and the rest of hosts only passively wait for the decision value (because each MH has its own initial value, it does not need to collect initial values from others). To get stable results, each execution was repeated 100 times and the average values are reported.

5.2.5.3. Simulations Results

Since HMR is the basis of the other two protocols, we first examine the performance of HMR. Then the performance of our HC protocol is studied. Finally all the three protocols are compared and discussed.

1) Performance of HMR

The performance of HMR under different numbers of faulty hosts (*f*/*n*) is shown in Figure 5-3 to Figure 5-6. We varied the size of *DA*, the major parameter that can significantly affect the performance of HMR. The two extreme sizes of *DA* are 2 (the current and next coordinator) and *n* (all the hosts). Besides 2 and *n*, we also simulated the HMR with a middle size *DA*, *n*/2. The curves with different sizes of *DA* are labeled "*SmallSetDA*", "*MiddleSetDA*" and "*FullSetDA*" respectively.

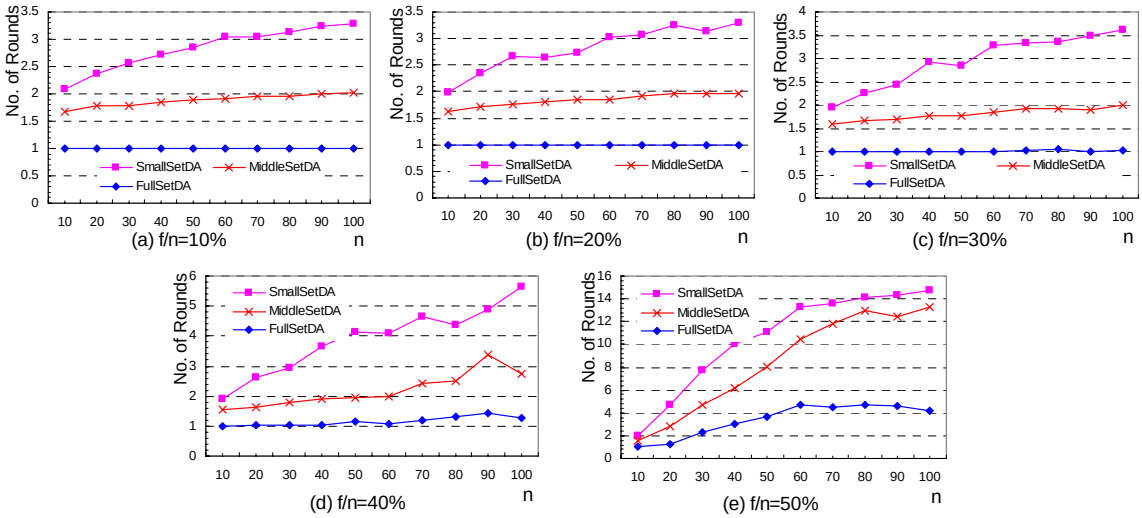

Figure 5-3 The *NR* of HMR

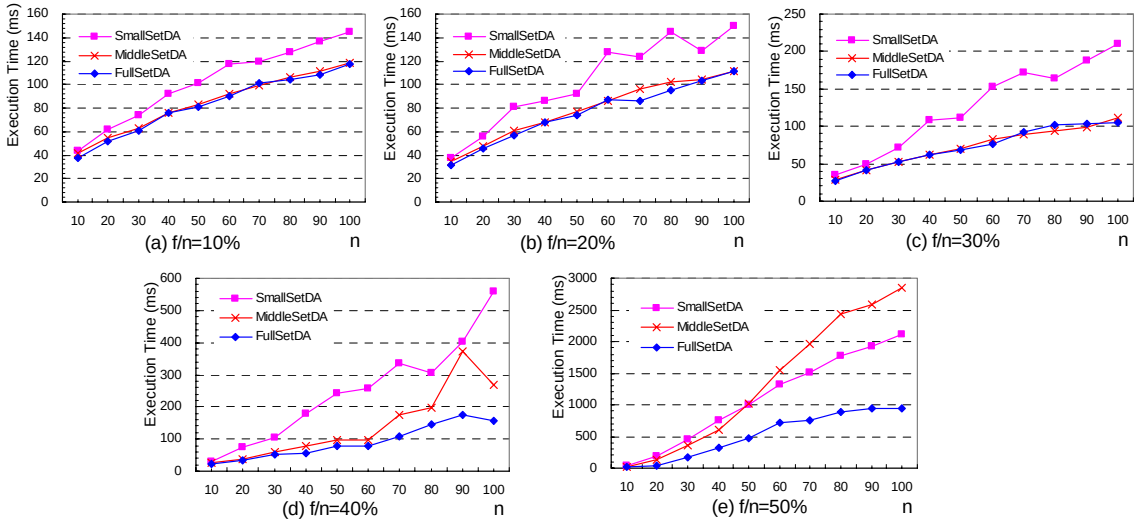

Figure 5-4 The *ET* of HMR

The effect of system scale is simple. The larger the system is, the more messages are needed in a round. Consequently, a round lasts longer and more failures may occur. Therefore, *NR*, *ET*, *NM* and *NH* all increase with the increase of system scale.

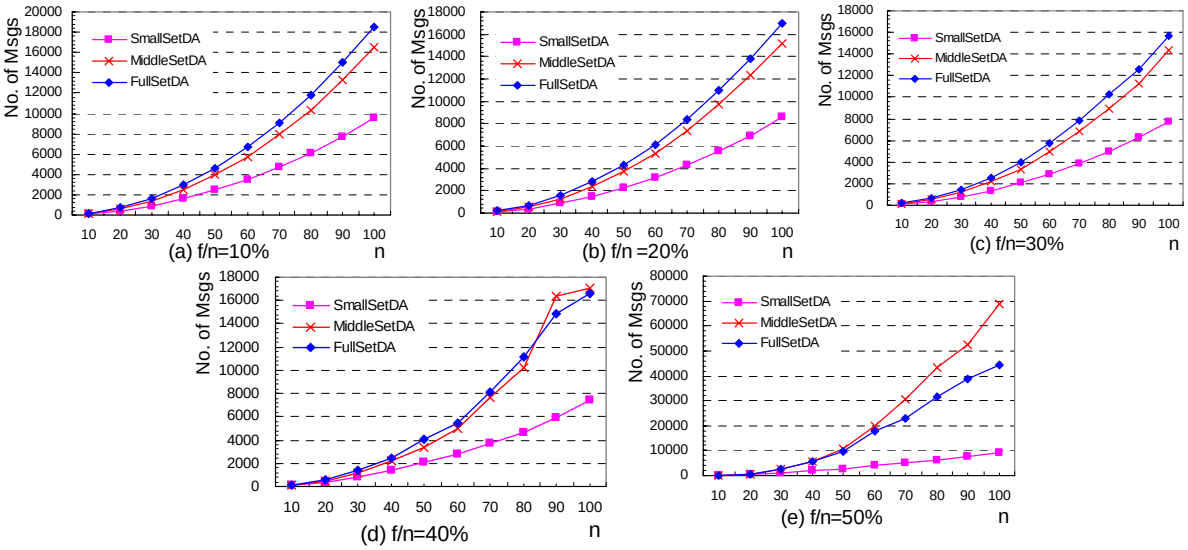

Figure 5-5 The *NM* of HMR

Now, let us see the effect of the size of *DA*. When |*DA*|=*n* (|*DA*|=2), HMR needs the fewest (most) rounds, and when |*DA*| = *n*/2, *NR* is in the middle. This is because that a smaller *DA* results in a smaller probability of making decision in a round. Consequently, more rounds are needed to achieve the consensus. The effect of |*DA*| on *ET* and *NM*/*NH* is more complex. When |*DA*|=*n* (|*DA*|=2), HMR needs the shortest (longest) time but the most (fewest) messages/hops. The *ET* or *NM*/*NH* is the accumulation of two values: the number of rounds and the time/message cost per round. A smaller *DA* results in fewer messages and shorter time per round but more rounds. However, the value of *NR* is much smaller than the number of messages per round ($O(10)$ vs. $O(n^2)$), the number of messages per round dominates the change of *NM*/*NH* when |*DA*| changes, as shown in Figure 5-5 and Figure 5-6. However, Figure 5-4 shows that the change of *ET* with different sizes |*DA*| is dominated by *NR*, which indicates that the change of time cost per round with different |*DA*| sizes is small.

In general, there is a tradeoff between the message cost and time cost of HMR, when |*DA*| changes. A smaller *DA* results in a smaller message cost but larger time cost. However, the effect on *ET* is not so significant as that on *NM*/*NH*, so we fix |*DA*| to 2 in the rest of simulations[9].

[9] |*DA*|=2 may not be the optimal value, but adopting this value does not affect the fairness of the following comparisons of the three protocols, because |*DA*| is also a parameter for both the HC protocol and BHM protocol.

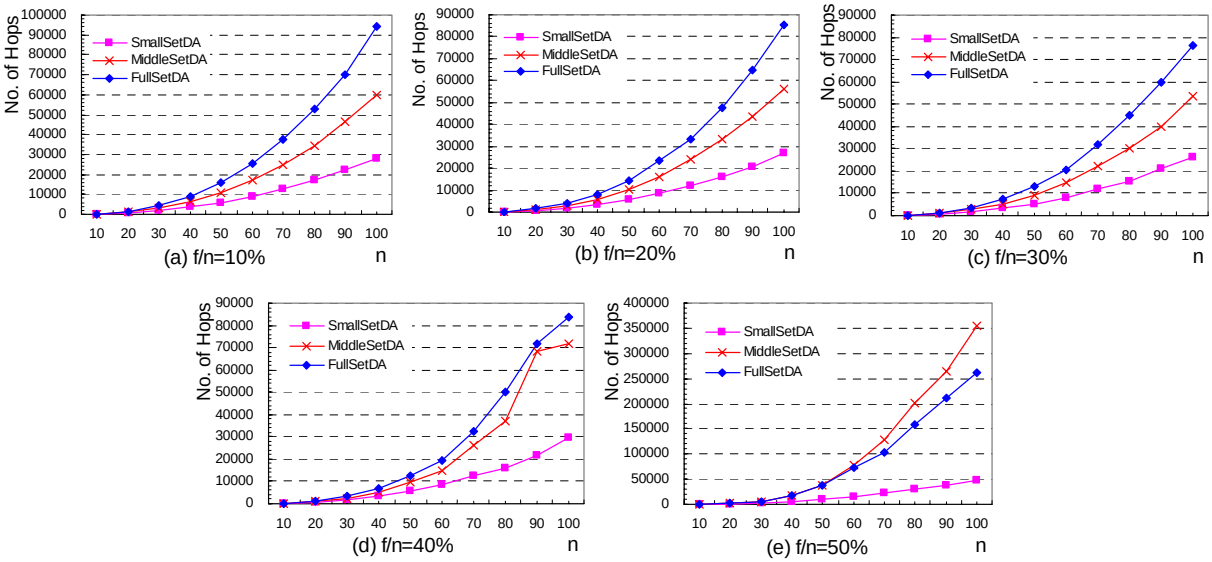

Figure 5-6 The *NH* of HMR

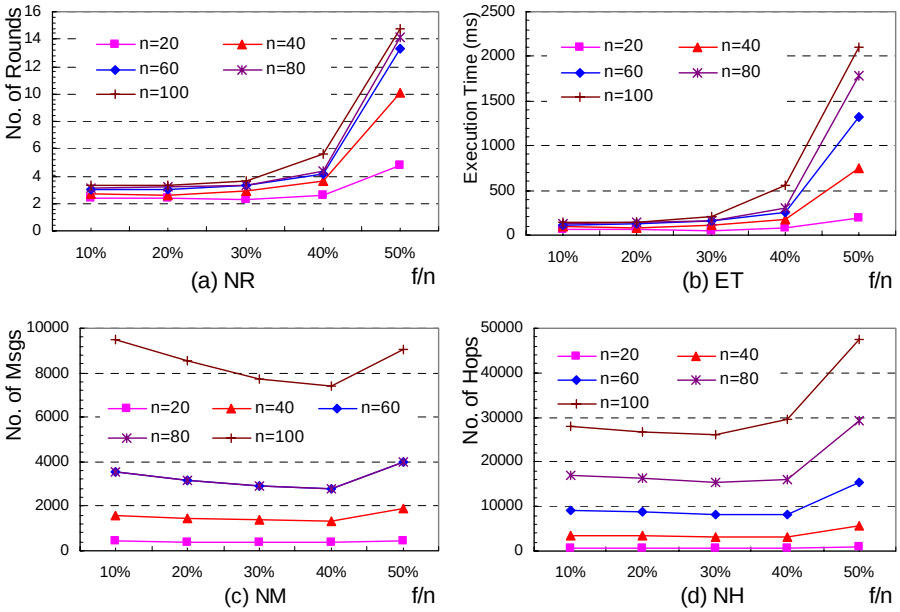

Figure 5-7 Performance of HMR vs. *f*/*n*, with |*DA*| =2

Figure 5-7 shows the effect of *f*/*n* clearly. When *f*/*n* increases, more rounds are needed. This is because that a large *f*/*n* means a large probability that the round is coordinated by a crashed host, which is prone to fail to make a decision. Consequently, more rounds are executed to achieve the consensus. The sharp increase when *f*/*n* increases from 40% to 50% may be caused by the sharp increase of false suspicion of coordinators. The *ET* changes similarly to *NR* when *f*/*n* changes, but *NM*/*NH* is affected differently. With the *f*/*n* increasing from 10% to 50%, *NM*/*NH* decreases first and then increases again. On the one hand, more faulty hosts cause

more rounds. On the other hand, more faulty hosts result in fewer hosts really participating in the execution. As a joint result, the fewest messages/hops are needed when *f*/*n* reaches about 40%.

2) Performance of HC

Besides the system scale, which affect the performance of HC similarly as it does in HMR, the size of *DA* and the size of *H*, i.e. the parameter *k*, are other parameters that significantly affect the performance of the HC protocol. Since DA is inherited from HMR, based on the simulations results of HMR, we fixed |*DA*| to 2 in the simulation of HC. To examine the effect of *k*, the performance of HC against *k*/*n* is plotted in Figure 5-8. The value of *f*/*n* is fixed to 10%, because under a large *f*/*n*, due to the constraint of *f*<*k*, *k*/*n* cannot be varied with a large scope[10].

From Figure 5-8 we can see that, with the increase of *k*/*n*, the *NR* increases slowly while the *ET* decreases. This can be explained by the operation at line 14 in the HC protocol. The wait at line 14 may be ended earlier due to the reception of an ECHOG message with a high timestamp (Figure 5-1). Such an operation can shorten the average wait time of a host at line 14 but may destroy a potential decision. A larger *k*/*n* means more ECHOG messages exchanged in a round, and due to the asynchrony of the network, more hosts end the wait at line 14 earlier. Consequently, more rounds but shorter time is needed to achieve the consensus.

The *NM* decreases very slowly when *k*/*n* increases. The effect of *k*/*n* on *NM* is two-edged. The increase of *k*/*n* causes the increase of global messages (i.e. messages between *DA* and *H*) but the decrease of the messages for cluster switches, including LEAVE, JOIN, late ECHOL (in Task 4.1) and PROPH messages (Figure 5-8-e). As an accumulative result, the *NM* changes very little when *k*/*n* increases. The change in Figure 5-8-e can be explained as follows. Since a clusterhead host always selects itself as its local clusterhead, the more hosts act as clusterheads, the fewer hosts need to switch their clusters and the fewer messages are cost by cluster switches.

[10] In fact, due to the constraint of *f*<*k*, *f* is set to (*n**10%)-1, but for convenience, we still use "10%" to refer to the value of *f*/*n*. To guarantee the fairness of comparisons, the f of HMR and BHM is set in the same way.

Same as *NM*, the *NH* also decreases very slowly with the increase of *k*/*n*, except that there is thorough in the middle (especially when the system scale is large). *NH* is affected by *NM* and the number of hops per message. When *k*/*n* becomes large, the average distance between a host and its clusterhead in hops is reduced and consequently the average number of hops per message is reduced. As an accumulative result of the *NM* and number of hops per message, *NH* becomes the least when *k*/*n* is about 30%.

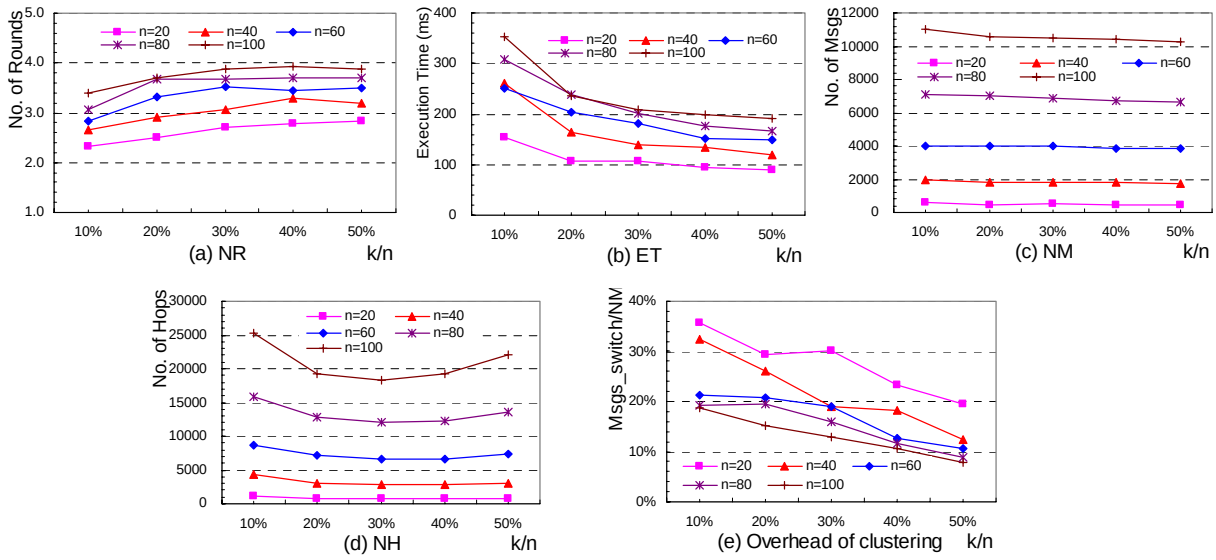

Figure 5-8 Performance of HC vs. *k*/*n*, with |*DA*| =2 and *f*/*n* = 10%

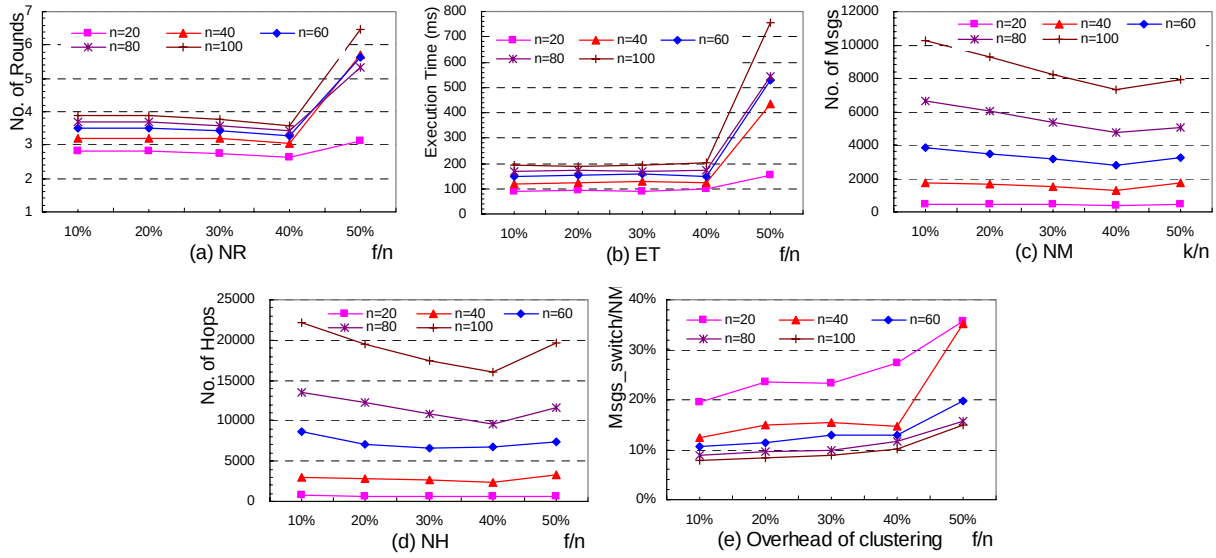

Figure 5-9 Performance of HC vs. *f*/*n*, with |*DA*| =2 and *k*/*n* = 50%

Figure 5-9 shows the performance of HC against *f*/*n* with *k*/*n* =50%. Comparing Figure 5-7 and Figure 5-9, we can find that the effect of *f*/*n* on HC is nearly the same

as that on HMR, so Figure 5-9 can be explained similarly as Figure 5-7. It is worth notice the results in Figure 5-9-e, which shows that the additional message cost introduced by the two-layer hierarchy is small (in most cases it is less than 15%), especially when the system scale is large.

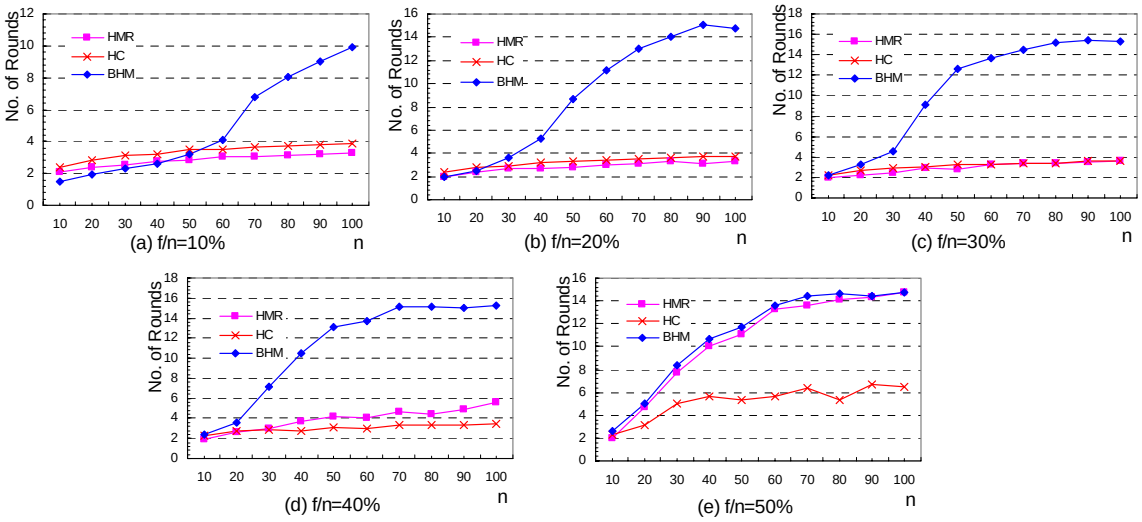

Figure 5-10 Performance comparison of HMR, BHM and HC – *NR*

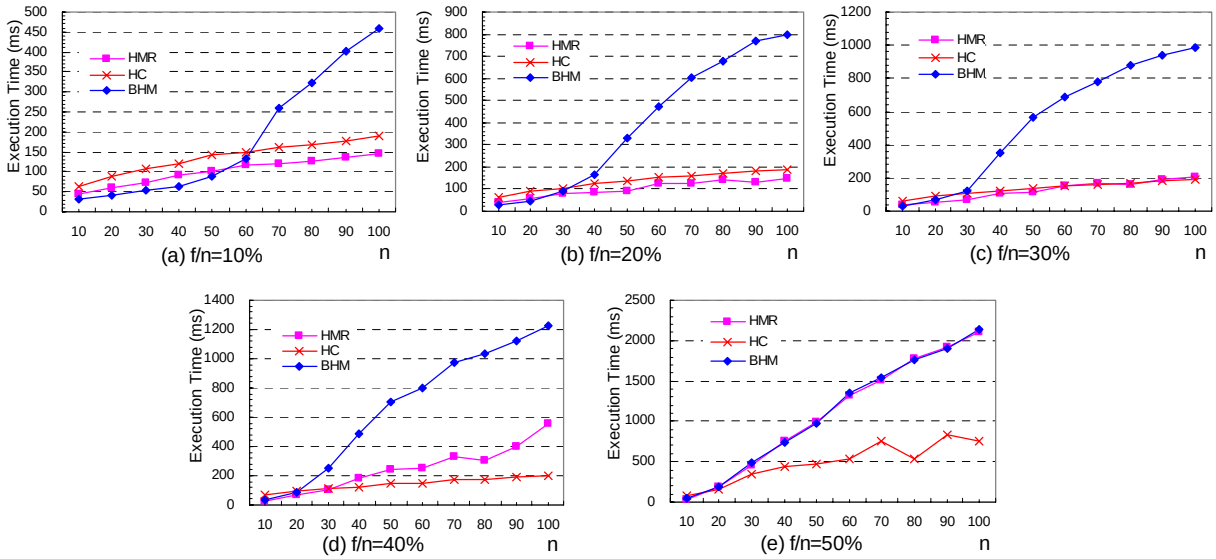

Figure 5-11 Performance comparison of HMR, BHM and HC – *ET*

3) Performance of BHM and Comparisons

The performance of the BHM protocol is shown in Figure 5-10 to Figure 5-13. Similar as the other two protocols, both the message cost and time cost increase with the increase of the system scale and percentage of faulty hosts. Now, let us compare

the three protocols. Without loss of generality, the k/n of HC is fixed to 50% in the comparisons.

i) Comparisons in *NR* and *ET*

Figure 5-10 and Figure 5-11 show the *NR* and *ET* of all the protocols under varied f/n values respectively. When the percentage of faulty hosts and system scale are small, the BHM can achieve consensus with the fewest rounds and shortest time. This because that, under a small f/n, the BHM protocol involves much fewer hosts in the procedure of achieving consensus, i.e. it can be viewed as a HMR protocol running in a much smaller system. Therefore, BHM can achieve consensus with fewer rounds and shorter time. However, with the f/n increasing from 10% to 50%, the number of hosts really executing the protocol in BHM gradually turns to be the same as in HMR and consequently the performance of BHM becomes the same as that of HMR.

The other factor affecting the difference between BHM and the other two protocols is the system scale. When the system scale becomes large, the *NR* and *ET* of BHM increase sharply and become the worst among the three protocols. As discussed above, BHM can be viewed as a HMR protocol running in a system of $2f+1$ hosts. The real percentage of faulty hosts in such a "smaller system" may be between f/n and $f/(2f+1)$, though the average of this percentage is equal to f/n. Since a large percentage of faulty hosts results in the sharp increase of *NR* in HMR (as shown in Figure 5-7), *NR* and *ET* of BHM under a large system scale becomes the largest.

The difference in *NR* and *ET* between HMR and HC is also affected by the value of f/n and n. Basically, due to the message forwarding mechanism by clusterheads, HC needs two more communication steps than HMR. Therefore, each round of HC lasts longer than that of HMR and more failures may happen during one round. Consequently, HMR achieves the consensus earlier and faster.

However, when the f/n is large, HC performs better in terms of *NR* and *ET*. This is also caused by the two-layer hierarchy. In HMR protocol, when an ordinary host (i.e.

a host does not belong to *DA*) suspects the coordinator, it proceeds to the next round after sending echo messages. However, in HC protocol, a host outside the set *H* has to wait its clusterhead to forward the proposal from the coordinator, so the proceeding of HC in the first phase is determined by only clusterheads. Therefore, the effect of false suspicions made by ordinary hosts is avoided and consequently fewer rounds are needed to make the decision. With the increase of *f*/*n*, the probability of such a false suspicion increases. Therefore, the difference between HC and HMR is reversed under a large *f*/*n*.

With the increase of the system scale, the advantage of HC in *NR* and *ET* also increases. This can also be explained based on the discussion above. The proceeding of the first phase is determined by only clusterheads, so the effect of system scale on *NR* in the HC protocol stems from the change of the number of clusterheads, i.e. |*H*|. Since |*H*|=*n*/2, |*H*| changes more slowly than *n* changes and consequently the *NR* and *ET* of HC increase more slowly than those of HMR.

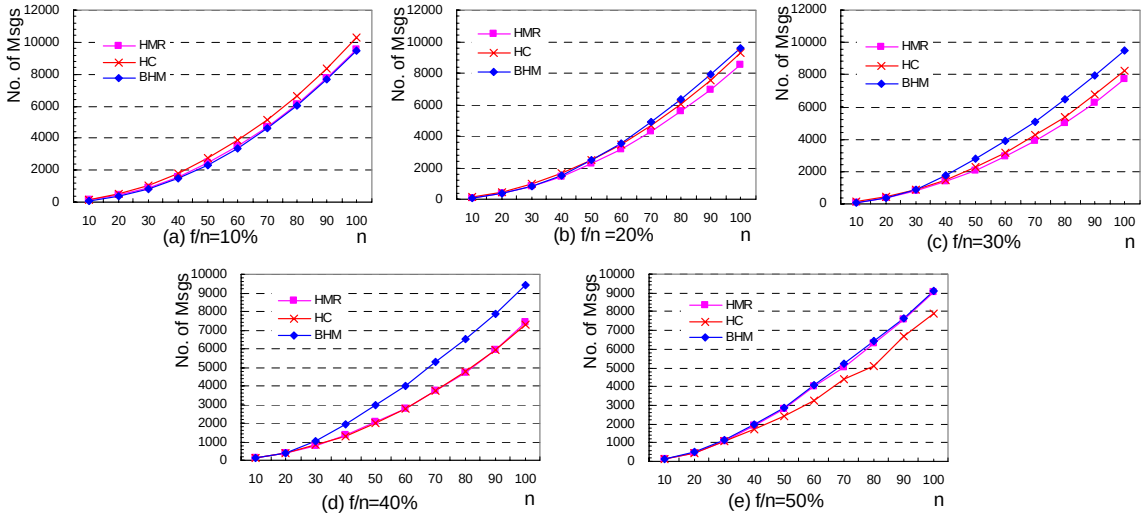

Figure 5-12 Performance comparison of HMR, BHM and HC – *NM*

ii) Comparisons in *NM* and *NH*

Figure 5-12 and Figure 5-13 show the performance in *NM* and *NH* respectively. The HC protocol performs badly only when very few hosts crash and the system scale is very small. With the increase of *f*/*n* and *n*, HC performs better and better. When *f*/*n* = 50% and *n* =100, HC achieves the consensus with only less than half of

the hops cost by BHM or HMR. As discussed before, both *NM* and *NH* are determined by two aspects: *NR* and the message cost per round. Comparing Figure 5-12 with Figure 5-10 we can see that, the relationships among the three protocols in *NM* are nearly the same as in *NR*. Therefore, *NR* dominates the difference in *NM*.

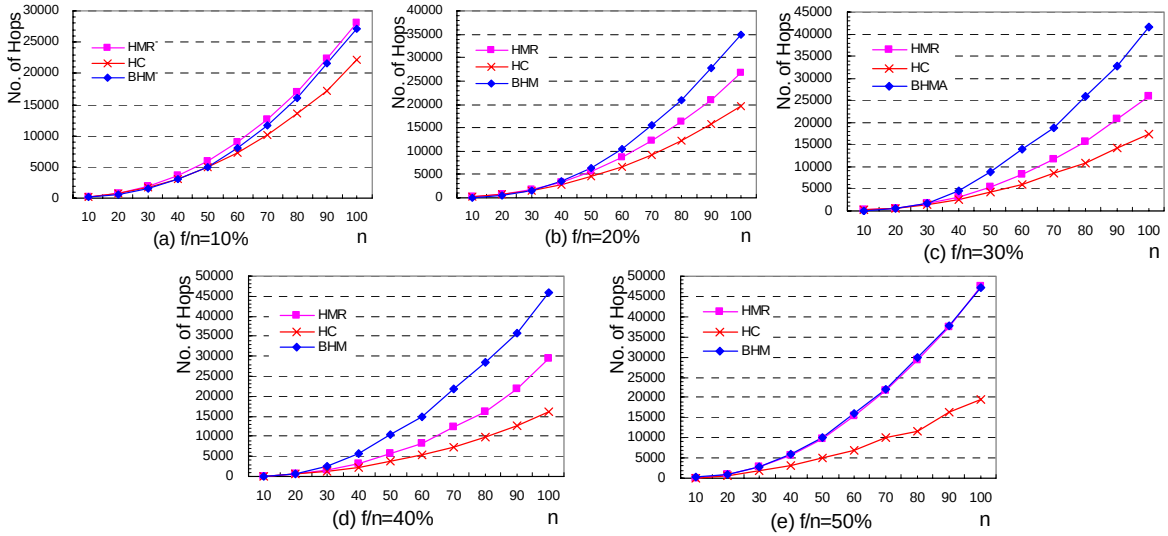

Figure 5-13 Performance comparison of HMR, BHM and HC – *NH*

The difference in *NH* between BHM and HMR is also determined by *NR*. However, the difference between HC and the other two is not dominated only by *NR*. When the system scale is not very small, HC can achieve consensus with the fewest hops even if its *NR* is not the smallest. Such an advantage comes from the two-layer hierarchy, which reduces the message cost per round in hop by merging messages with the same type. The larger the system is, the more messages are merged and consequently the more cost is saved. This indicates that our objective to reduce message cost using the two-layer hierarchy is fulfilled.

5.2.6. Tolerance of Message Loss

Same as most consensus protocols, our HC protocol assumes reliable communication channels between hosts. However, compared with wired networks, MANETs are more prone to message losses due to the characteristics of wireless communications. To cope with this, one direction is to design reliable communication protocols for MANETs, Efforts have been made to improve the reliability of end-to-end communications in MANETs [54][149][165]. However,

how to provide reliable end-to-end channels is still a challenging topic in MANETs. Here, we take an alternative approach by enhancing the HC protocol to handle message losses by itself. We divide the channel failures into two types: permanent failures and transient failures, and design solutions for handling the two types of channel failures respectively.

5.2.6.1. Handling Message Losses as Host Failures

If a channel fails by crashing, i.e. permanently losing all the messages transmitted through it, a permanent failure occurs. To handle message losses caused by a crashed channel, a possible solution is to treat the message loss caused by communication channels as host crashes. If a channel between a pair of hosts loses some message, then the sender, instead of the channel, is said to be faulty. In this way, the system has only host failures and all channels can be thought of as reliable. Although the correctness of the protocol will not be affected, the resilience, i.e. the capability of tolerating faults, of the HC protocol is degraded. There can be at most t ($t < minimum(k, n/2)$) host failures during a run of the protocol, including those caused by the lossy channels.

5.2.6.2. Reducing Reliable Channels to Fair-lossy Channels

A channel with a transient failure only loses messages for some finite time and then recovers to be a correct channel again. Such a failure may recur for the same channel. More precisely, such a channel is defined as a fair-lossy channel [50][120]:

If a host m_i sends an infinite number of messages to host m_j, then the channel attempts to deliver an infinite number of messages to m_j.

Following the approach in [120][131], our HC protocol can be extended for use in a system with fair-lossy channels. To tolerate message losses caused by channels with transient errors, the following three rules are added:

i) When a clusterhead host p enters a new round r that is not coordinated by p, it sends a $NEW(r)$ message to all other clusterheads;

ii) Each host periodically re-sends the latest message it ever sends out. For example, during the wait at line 7 of round r_i, a clusterhead periodically re-

sends the *PROP*(r_i, *) message to its local hosts, i.e. it periodically repeats the execution of line 5 to line 6;

iii) When a host m_i, either an ordinary host or a clusterhead, receives a message with a higher round number r with $r>r_i$, m_i aborts its current round and enters the round r.

With these rules, a correct host is eventually unblocked if it is blocked due to a message loss of the channel. Therefore, the termination property is not violated due to message losses. Since the other two correctness properties, validity and agreement are not affected by messages losses, the enhanced HC protocol still satisfies the correctness requirements of a consensus protocol.

5.3. The Clusterer Oracle Δ

5.3.1. System Model

We consider an asynchronous MANET system consisting of a set of n (n>1) MHs, $M = \{m_1, m_2,\ldots, m_n\}$. A MH can only fail by crashing, i.e. prematurely halting, but it acts correctly until it possibly crashes. There is at least one correct host in the system. MHs communicate by sending and receiving messages. Every pair of MHs is connected by a reliable channel that does not create, duplicate, alter or lose messages.

5.3.2. Definition of Δ

Like unreliable failure detectors or other oracles, the eventual clusterer oracle Δ is also a tool that provides some kind of information about the system. Δ groups the hosts in a MANET into clusters, each which is dominated by a clusterhead, so as to establish a two-layer hierarchy in the system. Each host is associated with an eventual clusterer oracle module. On the query from a host m_i, the clusterer oracle module returns three outputs:

i) Δ.*CH*: a set of MHs that currently act as clusterheads;

ii) Δ.*trusted*: a set of hosts that are currently trusted by Δ (correct hosts);

iii) Δ.*clusterhead*: the clusterhead host that m_i currently associates with, i.e. the local clusterhead of m_i.

Similar to the definition of unreliable failure detectors [31], we define the eventual clusterer oracle Δ using abstract properties. Though we also define the *completeness* property and *accuracy* property as in [31], the names of the properties are reversed, i.e. the completeness (accuracy) in [31] is named accuracy (completeness) here. This is because that the properties in [31] are defined for the set of suspected hosts but here they are defined for the set of trusted hosts.

Completeness: There is a time after which some correct host is permanently included in the clusterhead set Δ.*CH* and trust set Δ.*trusted* at each correct host.

Accuracy: Eventually every host that crashes is permanently excluded from the clusterhead set Δ.*CH* and trust set Δ.*trusted* at each correct host.

Uniformity: Eventually, all correct hosts permanently keep the same clusterhead set Δ.*CH*.

Stability: There is a time after which each correct host is associated with some correct clusterhead permanently.

From the definition we can see that, like other oracles, there is a *Global Stabilization Time* (*GST*) for Δ to reach a stable state. Before *GST*, different hosts may have different clusterhead set Δ.*CH* and a host may switch to a new cluster again and again. However, after *GST*, all correct hosts have the same Δ.*CH* and each correct host associates with a correct host in Δ.*CH*. We call such a clusterhead set *CH* a "stable *CH*". It is important to notice that a stable *CH* includes only correct hosts (but maybe not all hosts).

Same as other oracles, Δ facilitates the design of consensus protocols by separating the function of detecting the status of the system and the function of achieving consensus. But Δ is more powerful in the sense that it can help the consensus protocols built on top of it improve their message efficiency and scalability, which is especially important for large scale MANETs. The messages from and to the hosts in the same cluster are merged by the clusterhead so as to reduce the message cost and improve the scalability.

5.3.3. An Implementation of Δ

To implement a Δ, there are two main issues to be addressed: a) failure detection, i.e. the construction of Δ.*trusted* and, b) the construction of clusters. Since unreliable failure detectors have been proposed in [31], we adopt the unreliable failure detectors to detect failures. The properties of the failure detector ◊*S* are as follows:

Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.

Eventually Weak Accuracy: There is a time after which some correct process is never suspected by any correct process.

Comparing the properties of Δ and ◊*S*, we know that the completeness and accuracy of Δ.*trusted* are the same as the accuracy and completeness of ◊*S* respectively. Therefore we adopt the unreliable failure detector ◊*S* to detect failures.

The second issue can be further divided into two problems: i) the selection of clusterheads, i.e. the construction of Δ.*CH*, and ii) the construction and maintenance of clusters. Analyzing the properties of Δ, we know that the only difference between Δ.*trusted* and Δ.*CH* is the uniformity. We adopt the flush algorithm in [39], which reduces a leader oracle to the ◊*W* failure detector[11], to establish Δ.*CH* based on the Δ.*trusted*. The corresponding pseudocode is shown as Task c1 in Figure 5-14. The code is simple and self-explanatory. Since Figure 5-14 shows the implementation of Δ, we use "*CH*" and "*clusterhead*" rather than "Δ.*CH*" and "Δ.*clusterhead*" to refer to the clusterhead set and local clusterhead respectively.

The pseudocode for the construction and maintenance of clusters is shown as Task c2 in Figure 5-14. First, we show how to construct clusters based on the clusterhead set Δ.*CH*. The clustering procedure is cluster member initiated. Each host in Δ.*CH* acts as a clusterhead and dominates the corresponding cluster. A host m_i selects the nearest host, say m_n, in Δ.*CH* using the function *NEAR*(Δ.*CH*) and sends a JOIN message to m_n to join the corresponding cluster. On the reception of the JOIN

[11] ◊*W* is equivalent to ◊*S*.

message, m_n sends a positive or negative ACK message to m_i. The type of the ACK depends on the state of m_n. If m_n is the clusterhead of itself (a self-aware clusterhead), it accepts the request of m_i and sends a positive ACK message; otherwise, it rejects the request of m_i and sends a negative ACK message. Then, if a positive ACK is received, m_i ends the switch procedure. Since the $\Delta.CH$ is set to M in the beginning, each MH selects itself as the clusterhead and gets positive ACK when the algorithm starts to execute.

```
COBEGIN// The code executed by a host, m_i
----------Task c1: Construction of Clusterhead Set CH ----------
 (c01) CH ← M; seq_i← 0; //M is the set of all MHs, seq_i is a sequence number;
----------Task c1.1: Send CH ----------
(c02) while(true){
(c03)     CH ← CH ∩◊S.trusted;
(c04)     send (CH, seq_i) to M;}
----------Task c1.2: Receive CH ----------
(c05) upon reception of (CH', seq_q) from host m_q:
(c06)    if (seq_q= seq_i) CH ← CH ∩ CH';
(c07)    if (seq_q > seq_i) { CH ← CH'; seq_i ←seq_q;}
(c08)    if (CH = φ) { CH ← M; seq_i ←seq_i +1;
(c09)              send (CH, seq_i) to M;}
----------Task c2: Clustering Host ----------
(c10) clusterhead←i; rejected ←φ; sn ← 0;
----------Task c2.1: Action of Cluster Member ----------
(c11) while(true){
(c12)    if(clusterhead∉ CH or  (a RELEASE(sn') from m_k with clusterhead = k)){
(c13)        if(clusterhead = i)
                 send RELEASE(sn) to all local hosts except m_i;
 (c14)       else
                  send LEAVE(sn) to clusterhead;
(c15)        sn←sn+1; rejected ←φ;
(c16)        if (CH \rejected)= φ){ rejected ←φ; sn←sn +1;}
(c17)        clusterhead←NEAR(CH \rejected);
(c18)        send JOIN(sn) to clusterhead;
(c19)        wait until ACK(type, sn) received from clusterhead;
(c20)        if(ACK.type = false){
(c21)             rejected ← rejected ∪ {clusterhead};
(c22)             GOTO (c15);}
          }
      }
----------Task c2.2: Action of Clusterhead ----------
(c23) while(true){
(c24)  upon reception of a JOIN(sn) message from a host m_k:
(c25)      if(clusterhead≠i) send ACK(false, sn) to m_k;
(c26)       else {add m_k to local host list;
(c27)             send ACK(true, sn) to m_k;}
(c28)  upon reception of a LEAVE(sn) message from a host m_k:
(c29)      if(clusterhead=i) delete m_k from local host list;
       }
COEND
```

Figure 5-14 The implementation of Δ

Then let us consider the maintenance of clusters. Due to failures or false suspicions, a clusterhead may be removed from the set $\Delta.CH$. Then the members of the corresponding cluster have to switch to a new cluster. When a host m_i needs to switch to another cluster (finding the current clusterhead is removed from$\Delta.CH$ or receiving a RELEASE message from its current clusterhead), it firstly sends a LEAVE message to its current clusterhead if it is not the clusterhead of itself or sends a RELEASE message to inform the cluster members if it is the current clusterhead of itself. Then m_i selects the nearest host in $\Delta.CH$ as the candidate of the new clusterhead and sends a JOIN message to the candidate. If a positive ACK is received from the candidate, the switch successes; otherwise, m_i selects a new candidate and sends a JOIN message again. This can be repeated again and again until m_i is accepted by a clusterhead host.

Since the communication channels are not FIFO, a sequence number *sn* is attached to each JOIN, LEAVE, ACK or RELEASE message to avoid the effect of disorder. When a host receives one of such messages, it needs to check the *sn* of the message to guarantee that only updated messages are handled.

Based on the properties of $\Diamond S$ and the correctness of the flush algorithm in [39], it is easy to prove that the proposed implementation of Δ satisfies the definition of Δ, i.e. the algorithm in Figure 5-14 is correct.

5.4. The HCD Protocol

The HC protocol is based on the same system model as the HC protocol, except that in HCD, each host is equipped with a clusterer oracle in stead of a $\Diamond P$.

5.4.1. Data Structures and Message Types

When executing the proposed consensus protocol, each host, say m_i, needs to maintain necessary information about its state. Such information is stored in the following variables.

fl_i: the flag indicating whether m_i has made the decision. The initial value is *false*.

r_i: the sequence number of the current round that m_i is participating in.

est_i: the current estimate of the decision value. Initially, it is the value proposed by m_i.

ts_i: the timestamp of est_i. The value is the sequence number of the round in which m_i receives the value of est_i, which is proposed by a coordinator host. The update of ts_i is entailed by the reception of estimate from a coordinator.

lr_i: the sequence number of the last round that m_i has completed normally, i.e. m_i has collected an echo message from its clusterhead, in Phase 2 of round lr_i.

During execution of the protocol, MHs need to communicate with each other by exchanging messages. The message types involved in a round r_i are as follows.

PROPL(r_i, est_i, lr_i, *cc*): the proposal message sent by a host m_i to its clusterhead. *cc* is the ID of the coordinator host selected by m_i.

PROPG(r_i, *X*, *L*, *C*, *Y*): the proposal message sent by a clusterhead to all clusterheads. PROPG is constructed by merging the PROPL messages received from the hosts in the cluster. *Y* is the set of the IDs of the hosts that send the PROPL messages; *X* is the set of estimate values corresponding to *Y*; *L* is the set of *lr* values corresponding to *Y*; *C* is the set of *cc* values corresponding to *Y*.

PROP(r_i, *est*): the proposal message sent by a clusterhead to the hosts in its cluster. PROP is constructed based on the PROPG messages received. *est* is equal to est_{cc} (the estimate of the coordinator selected by the clusterhead) or "⊥"(a value that can not be decided upon).

ECHOL(r_i, est_i, ts_i): the echo message from a host m_i to its clusterhead.

ECHOG(r_i, *v*, *tsv*, *W*, *Z*): the echo message from a clusterhead to all clusterheads. ECHOG is constructed by merging the ECHOL messages from the hosts in the cluster. *v* is the estimate carried by the ECHOL with the highest timestamp and *tsv* is the timestamp of *v*. *W* is the set of MHs that send the ECHOL with *tsv* whereas *Z* is the set of MHs that send other ECHOL messages.

ECHO(r_i, *vm*, *tsvm*, *A*, *B*): the echo message from a clusterhead to the hosts in the cluster. ECHO is constructed based on the ECHOG messages received. *vm* is the estimate carried by the ECHOG with the highest timestamp and *tsvm* is the

timestamp of *vm*. *A* is the set of MHs that send the ECHOL (not ECHOG) with *tsvm* whereas *B* is the set of MHs that send other ECHOL (not ECHOG) messages.

DECISION(*est*): the message sent by a MH to propagate the decision value *est*.

NEWR(*r*): the message sent by a clusterhead to ask its local MHs to enter a new round *r*.

```
COBEGIN // The code executed by each host, m_i
----------Task 1: Consensus ----------
(101) r_i ←0; est_i ← v_i; ts_i ← 0; lr_i←0; fl_i ← false;
(102) while (fl_i≠true){
(103)    r_i←r_i+1;
(104)    cc = MIN(Δ.CH); ch = Δ.clusterhead;
---------- Phase 1: Collect Proposal ----------
(105)    send PROPL(r_i, est_i, lr_i, cc) to m_ch;
         if(i=ch){// m_i is a clusterhead;
(106)       wait until (received a PROPL(r_i, *,*, *) from each local host m_j∈ Δ.trusted);
(107)       merge the PROPL messages received into a PROPG(r_i, X, L, C,Y) and send it to Δ.CH;
                    //X: set of estimates; L: set of lr values; C: set of cc values; Y: set of hosts IDs;
(108)       wait until ((received PROPG(r_i, X, L,C,Y) messages where |∪Y |≥ n-f  and m_cc∈ ∪Y) or
                       m_cc ∉ Δ.trusted or m_i∉ Δ.CH);
(109)       if(m_cc∈ ∪Y , lr_cc=r_i-1and m_cc appears at least n-f times in ∪C)
                  send (PROP(r_i, est_cc) to all local hosts;
(110)       else send (PROP(r_i, ⊥) to all local hosts;}
(111)    wait until ((received PROP(r_i, v) from m_ch) or ch ≠ Δ.clusterhead);
(112)    if(PROP((r_i, v) is received and v ≠ ⊥){est_i← v; ts_i=r_i;}
----------Phase 2: Collect Echo ----------
(113)    send message ECHOL(r_i, est_i, ts_i) to m_ch;
         if (i=ch) {
(114)       wait until(received ECHOL(r_i,*,*) from each local host m_j∈ Δ.trusted);
(115)       merge the ECHOL messages received into an ECHOG(r_i, estm, tsm, W, Z);
               // tsm: the greatest ts; estm: the estimate value with timestamp tsm;
               //W: the hosts that sent ECHOL with tsm; Z: the hosts that sent other ECHOL;
(116)       send ECHOG(r_i, estm, tsm, W, Z) to Δ.CH;
(117)       wait until ((received ECHOG(r_i, *, *, W, Z) messages with |∪W∪Z| ≥ n-f ) or m_i∉ Δ.CH);
(118)       merge ECHOG messages into ECHO (r_i, vm, tsvm,A,B) and send it to all local hosts;}
(119)    wait until ((received ECHO(r_i, vm, tsvm,A,B) from m_ch) or ch ≠ Δ.clusterhead);
(120)    if ((received ECHO(r_i, vm, tsvm,A,B) from m_ch){
(121)       est_i← estm; lr_i ←r_i;
(122)       if(tsm =r_i and |A|≥ f+1){
(123)          ∀j ≠ i: send DECISION(est_i) to m_j;
(124)          fl_i←true;}}
      }endwhile
----------Task 2:  Reliable Broadcast ----------
      upon reception of DECISION(est) from host m_k:
(201)    if(fl_i≠true) {
(202)       ∀j ≠ i, k: send DECISION(est) to m_j;
(203)       fl_i←true;}
COEND
```

Figure 5-15 The consensus protocol based on Δ – Task 1 and Task 2

5.4.2. Description of the Protocol

The proposed consensus protocol consists of four tasks. Task 1 is the main body of the protocol for making decision and Task 2 is a simple broadcast algorithm for propagating the value decided upon. The pseudocode of Task 1 and Task 2 is shown in Figure 5-15. Task 3 is used to handle late PROPL and ECHOL messages at a clusterhead while Task 4 handles futures messages. The pseudocode of Task 3 and Task 4 is presented in Figure 5-16 and Figure 5-17 respectively. In the following paragraphs, we describe the tasks in detail.

Task 1: Like most consensus protocols, Task 1 is executed in asynchronous rounds, each of which is divided into two phases. In each round, each MH selects one coordinator.

At the beginning of a round r_i, a host m_i first queries the clusterer oracle Δ to get the ID of the coordinator and clusterhead, i.e. *cc* and *ch*. The remaining actions during the round are divided into two phases. In Phase 1, m_i first sends *PROPL*(r, est_i, lr_i, *cc*) to m_{ch}, where *cc* is the ID of the coordinator selected by m_i. If m_i itself is not the clusterhead, it waits until it receives a PROP message from m_{ch} unless m_{ch} is removed from $\Delta.CH$.

Each clusterhead m_{ch} needs to collect one PROPL message from each correct host in its cluster, including m_{ch} itself. Following this, m_{ch} merges the PROPL messages received into one *PROPG*(r_i, X, L, C, Y) message (where Y is the set of the IDs of the hosts that send the PROPL messages; X is the set of estimate values corresponding to Y; L is the set of lr values corresponding to Y; C is the set of *cc* values corresponding to Y) and sends this PROPG to all hosts in the clusterhead set $\Delta.CH$. m_{ch} then waits for the PROPG messages from others clusterheads until: i) m_{cc} and additional n-f-1 hosts are included in the PROPG messages received, or ii) m_{cc} is removed from $\Delta.CH$.

After m_{ch} finishes the wait at line 108, it checks the PROPG messages received. If condition i) is satisfied and m_{cc} has finished last round normally, i.e. $lr_{cc} = r_i$-1, m_{ch} sends *PROP*(r_i, est_{cc}) to the hosts in its cluster, including itself; otherwise it sends

PROP(r_i, $\perp$) to the hosts in its cluster, including itself. Upon the reception of *PROP*(r_i, v) with $v \neq \perp$ from the clusterhead, each host m_i (maybe a clusterhead) updates its estimate est_i to v and timestamp ts_i to r_i. Phase 1 ends.

Phase 2 is started by sending ECHOL messages. In Phase 2, each host m_i first sends an *ECHOL*(r_i, est_i, ts_i) to m_{ch}. If m_i itself is not the clusterhead, it waits until it receives an ECHO message from m_{ch}, unless m_{ch} is removed from $\Delta.CH$.

Each clusterhead m_{ch} needs to collect one ECHOL message from each correct host in its cluster, including m_{ch} itself. Then m_{ch} merges the ECHOL messages received into one *ECHOG*(r_i, *estm*, *tsm*, *W*, *Z*) message (where *tsm* is the greatest *ts* in the ECHOL messages; *estm* is the estimate value with timestamp *tsm*; *W* is the IDs of the hosts that have sent the ECHOL messages with timestamp *tsm*; *Z* is the IDs of the hosts that have sent other ECHOL messages) and sends this ECHOG to all hosts in the clusterhead set $\Delta.CH$. m_{ch} then waits for the ECHOG messages from others until *n-f* hosts are included in the ECHOG messages received, i.e. $|\cup W \cup Z| \geq n\text{-}f$. After getting enough ECHOG messages, m_{ch} merges the ECHOG messages received into an *ECHO*(r_i, *vm*, *tsvm*, *A*, *B*) message and sends it to the hosts in m_{ch}'s cluster (*vm* is the estimate carried by the ECHOG with the highest timestamp and *tsvm* is the timestamp of *vm*. *A* is the set of MHs included in the "*W*" sets of ECHOG messages with timestamp *tsvm* whereas *B* is the set of other MHs included in the ECHOG messages, i.e. $B = (\cup W \cup Z) \backslash A$.).

When a host m_i (maybe a clusterhead) receives the *ECHO*(r_i, *vm*, *tsvm*, *A*, *B*) message from its clusterhead, it updates its estimate est_i to *vm* and lr_i to r_i. Then, if $tsm = r_i$ and $|A| \geq f+1$, m_i makes the decision upon *vm* and sends the value to all other hosts using the message *DECISION*(*vm*).

Task 2: This task simply broadcasts the decision value. When a host receives a DECISION message, it decides upon the same value and forwards the DECISION message to all other hosts except the sender.

Task 3: This task handles late PROPL and ECHOL messages. A PROPL/ECHOL message is "late" if it arrives at a clusterhead after the clusterhead has sent out a

PROPG/ECHOG message for the corresponding round. This happens when a clusterhead m_{ch} suspects a correct host in the cluster or a host newly joins the cluster. Clusterheads may be blocked forever if a late PROPL or ECHOL message is ignored. To avoid this, when a clusterhead m_{ch} receives a PROPL (or ECHOL) message from a MH m_i with a round r_i where ($r_i<r_{ch}$) or ($r_i=r_{ch}$ but m_{ch} has sent out a PROPG (or ECHOG) for round r_{ch}), m_{ch} constructs a redeeming PROPG (or ECHOG) sends it to all clusterheads. Moreover, for a late PROPL message, if m_i is a local host of m_{ch} and m_{ch} has sent out a *PROP*(r_{ch}, *lv*) message, m_{ch} sends a redeeming *PROP*(r_{ch}, *lv*) to m_i.

```
----------Task 3: Handling Late Messages----------
// The code executed by each clusterhead m_ch;

 (301) upon reception of PROPL(r_i, *, *, *) from m_i, with ((r_i<r_ch) or
            (r_i=r_ch but m_ch has sent a PROPG(r_ch, *, *, *,*) )){
(302)        construct a PROPG for the PROPL and send it to Δ.CH;
(303)        if(m_i is a local host and a PROP(r_ch, lv) has been sent )
                 send PROP(r_ch, lv) to m_i;
      }
(304) upon reception of ECHOL(r_i, *, *) from m_i, with ((r_i<r_ch) or
            (r_i=r_ch but m_ch has sent an ECHOG(r_i, *, *, *,*) )){
(305)        construct an ECHOG for the ECHOL and send it to Δ.CH;
      }
```

Figure 5-16 The consensus protocol based on Δ – Task 3

```
----------Task 4: Handling Future Messages----------
// The code executed by each MH m_i;

(401) upon reception of a message msg with r>r_i:
     {
(402)       if(i=ch and msg is not a NEWR message) // for a clusterhead
                send NEWR(r_i) to local hosts;
(403)       r_i← r; GOTO (104);
                                    }
```

Figure 5-17 The consensus protocol based on Δ – Task 4

Task 4: This task handles future messages. A message *msg* is a future message if it arrives at a MH before the MH enters the corresponding round of *msg*. When a MH m_i receives a future message with the round number $r>r_i$, it stops waiting for messages for round r_i (at the wait statements in Task 1) and skips to the round r. If m_i is a clusterhead host, it sends a *NEWR*(r) to its local hosts before it skips to r, so that the local hosts can also skip to round r.

5.4.3. Correctness of the HCD Protocol

In this section, we prove the correctness of the proposed consensus protocol. Since the validity property is obvious, here, we only present proofs for the termination property and agreement property.

Theorem 1 (Termination). *If a host is correct, it eventually decides.*

Proof. If one host decides, all correct hosts eventually decide due to the reliable broadcast mechanism (lines 124 and 202). Therefore, we only prove that at least one host decides. The proof is by contradiction.

We assume that no host decides. By the properties of Δ, after the time *GST* all correct hosts get the same stable *CH* set denoted as *GT*. Let $p=MIN(\Delta.CH)$ and *rm* be the highest round number held by correct hosts at time *GST*. Since all correct hosts never crash, they eventually receive a message from some correct host in round *rm* and move to round *rm*.

By the assumption, no host decides in round *rm*. This could only happen if in round *rm*: 1) no more than *n-f*-1 hosts select the same correct coordinator, 2) no less than *n-f* hosts select the same correct coordinator m_x, but m_x did not execute line 121 in round *rm*-1, i.e. $lr_x \neq rm\text{-}1$, or 3) no less than *n-f* hosts select the same correct coordinator m_x, but one or more hosts suspect m_x, move to round *rm*+1 and cause *f*+1 or more hosts move to round *rm*+1 before they send out an *ECHOL*(*rm*, *) message. Let examine these three cases as follows.

Cases 1) and 2): All possible PROP messages sent by clusterheads are *PROP*(*rm*, $\perp$). By the definition of stable *CH*, eventually only hosts in *GT* act as clusterheads and each correct host joins a cluster with a clusterhead in *GT*. Therefore, in round *rm*, all possible PROP messages sent are *PROP*(*rm*, $\perp$) and all possible ECHO messages sent at line 118 are *ECHO*(*rm*, *, <*rm*, *, *). Therefore, all correct hosts eventually enter a round greater than *rm* after they finish the wait at line 119 or receive a message with a round number greater than *rm*.

Case 3): Obviously, at least one correct host enters round *rm*+1. Then, all correct hosts eventually enter round *rm*+1 after they receive messages from that correct host

already in round *rm*+1.

From the analysis of the three cases above we know that, if no host decides in round *rm*, all correct hosts eventually enter a round greater than *rm*. Such a scenario, i.e. no host decides in a round $rn \geq rm$ and all correct hosts eventually enter a round greater than *rn*, may be repeated again and again until the first round started after time *GST*[12]. Let *rx* denote this round.

Now, let us consider the execution of round *rx*. By the definition of the stable *CH*, in round *rx*, all correct hosts selected the same coordinator m_p and never suspect it. Then each clusterhead eventually collects one PROPL message (maybe a late PROPL) from each correct local host and exchanges *PROPG* message with others. Because at most *f* hosts can crash, each clusterhead eventually collects *PROPG* messages which is constructed based on at least *n-f* PROPL messages, including the one from m_p, i.e. $PROPL(rm, v, lr_p, p)$ with $v \neq \perp$.

The following execution depends on the value of lr_p. There are two possible cases. First, $lr_p = rx\text{-}1$. Each clusterhead sends *PROP* (*rm*, *v*) to the hosts in its cluster and each correct host eventually receives the *PROP*(*rm*, *v*). Then each correct host updates its *est* to *v* and *ts* to *rm*, and sends an *ECHOL*(*rm*, *v*, *rm*) message to its clusterhead. Eventually each clusterhead collects one ECHOL message (maybe a late ECHOL) from each correct local host. Following this, all clusterheads exchange *ECHOG* messages and each of them eventually collects *ECHOG* messages which are constructed by merging at least *n-f* ECHOL messages. Since all *ECHOL* messages are with the same values, i.e. *ECHOL*(*rm*, *v*, *rm*), each clusterhead sends an *ECHO*(*rm*, *v*, *rm*, *W*, φ) message with $|W| \geq f+1$. Therefore, each correct host eventually decides after it receives the ECHO message from its clusterhead, which contradicts the assumption “no host decides.” The theorem holds.

Second, $lr_p \neq rx\text{-}1$. All possible PROP messages sent by clusterheads are *PROP*(*rx*, $\perp$). Eventually all correct hosts enter round *rx*+1 after they update their *lr*

[12] The start of a round means that some host firstly enters the round.

values to *rx*. The execution of round *rx*+1 is the same as the execution of round *rx* except that the *lr* of each host equals to *rx*. Therefore, each correct host eventually decides in round *rx*+1, which contradicts the assumption "no host decides." The theorem holds.□

Lemma 1. *Let r be the smallest round in which some host receives an ECHO(r, vm, tsvm, A, B) messages with tsvm = r and |A| ≥ f+1.Then, in any round r'>r, if a host receives a PROP(r', v') with v'≠ ⊥ at line 111, v' = v.*

Proof. Without loss of generality, let m_i be the host that receives the *ECHO(r, vm, tsvm, A, B)* message with *tsvm = r* and *|A| ≥ f+1* in round *r*. Then, *ECHOL*(*r, v, r*) messages have been sent in round *r* by at least *f*+1 hosts and merged into ECHOG messages by one or more clusterheads. Assume that *rs* is the first round after round *r*, in which some host, say m_h, sends a PROP message with a estimate value not equal to *v* or ⊥ , i.e. *PROP*(*rs, vx*) with *vx* ≠ ⊥ and *vx* ≠ *v*,. Let m_{cx} denote the coordinator selected by m_h in round *rs*. Therefore, m_h must have received *PROPG*(*rs,X,L,C,Y*) messages with $m_{cx} \in \cup Y$, $lr_{cx} = rs\text{-}1$ and m_{cx} appears at least *n-f* times in the set $\cup C$. This means that m_{cx} has executed line 121 of round *rs*-1 and updated its estimate to *vx*. Therefore, m_{cx} must have received *ECHO(rs-1, vx, tsx, A, B*) at line 124 in round *rs*-1. Obviously, *tsx*≤ *rs*-1. Since *rs*>*r* and at least *f*+1 hosts have sent out *ECHOL*(*r, v, r*) messages in round *r* (by assumption in the lemma), we have *tsx*≥ *r* (the highest timestamp at any host, which executes line 121 in round *rs*-1, must be no less than *r*). Then we have *r* ≤ *tsx*≤ *rs*-1. Since a host updates its timestamp *ts* only when it receives PROP message with estimate not equal to ⊥, there must be some host that have sent a *PROP(tsx, vx)* with a estimate value not equal to *v* or ⊥ before round *rs* but after round *r*. This contradicts the assumption "*rs* is the first round in which some host sends a PROP message with a estimate value not equal to *v* or ⊥." The lemma holds.□

Theorem 3 (Agreement). *No two processes decide differently.*

Proof. If a process decides upon a value at line 203, then this value must have been decided upon by another process at line 124. Therefore, we only consider the values

decided upon at line 124. Let r be the smallest round in which some host m_i decides upon a value x at line 124.

Assume that another process m_j decides upon another value y in round k. By the definition of r, we have $k \geq r$. If $k=r$, obviously, $y = x$. If $k>r$, y must come from a $PROP(k, y)$. By lemma 1, $y = x$. The theorem holds.□

5.5. Summary

This chapter is concerned with the message efficiency of achieving consensus in MANETs. A cluster based two-layer hierarchy is imposed on the system by clustering MHs into clusters. Each cluster is dominated by a MH acting as the clusterhead. With the hierarchy, a coordinator sends proposal messages only to clusterheads and a clusterhead unmerges and forwards the proposal to its local hosts. On the other hand, the echo messages from the hosts in the same cluster are merged into one message before they are sent to decision makers. In such a way, the message cost can be significantly reduced.

Based on different mechanisms of clustering the MHs, We develop two different hierarchical protocols. In the first protocol, the set of clusterheads is predefined and the functions of clustering hosts and achieving consensus are closely coupled. The second protocol adopts a modular approach to select clusterheads dynamically. The function of achieving consensus and the function of clustering MHs are separated by defining the eventual clusterer oracle Δ. Δ is in charge of constructing and maintaining the cluster-based two-layer hierarchy over MHs.

Since Δ provides the fault tolerant clustering function transparently, it can be used as a new oracle for the design of reliable hierarchical consensus protocols. Δ can be implemented using $\Diamond S$, so it is equivalent to $\Diamond S$ in terms of the power of failure detection. However, Δ is more powerful in the sense that it can help the consensus protocols built on top of it improve their message efficiency and scalability, which is especially important for large scale MANETs.

Chapter 6. Handling Dynamic Mobile Systems

6.1. Overview

In recent years, a major advance in distributed computing is due to the development of dynamic systems, e.g. peer-to-peer systems, mobile systems, where an unbounded set of processes can join or leave the system at any time and the number of participating processes can change arbitrarily as time passes.

The inherent dynamic nature of processes introduces a new kind of uncertainty, constitution uncertainty: the global constitution of the network is unknown to the processes. This additional difficulty makes dynamic system more challenging than traditional ones, in designing coordination protocols, e.g. consensus protocols. Efforts have been made to implement the eventual leader protocol [109][111] in dynamic wired systems and then consensus can be achieved in dynamic systems [5][6][38][67][72]. However, to our knowledge, the consensus problem in dynamic mobile systems is not considered.

In this chapter, we investigate the implementation of the eventual leader oracle Ω in dynamic infrastructured mobile networks. MSSs and the network connecting them form a static asynchronous system. The number of MSSs is known. We adopt a time free approach proposed in [109][111] and extends it to the context of mobile environments by combining it with the notion of process accessibility [100]. This time free approach explicitly uses the values of n, the number of processes in the system, and t, the upper bound on the number of processes that can crash. It relies on an assumption on the behavior of the flow of messages exchanged. More precisely, processes can broadcast query and then wait for responses from other processes. The first (n - t) responses received are *winning* (the other responses, if any, are called *losing* responses; they can be slow or never sent because their sender has crashed). It is shown in [111] that Ω can be built if the following behavioral property is satisfied: “there is a correct process p and a set Q of t+1 processes such that eventually the

responses of p to each query issued by any $q \in Q$ is always a winning response." Intuitively, this means that for any q, the link connecting q to p is not among the t slowest links of q. We can also think of process p as being eventually accessible by processes in set Q.

To extend this idea in a dynamic system, we assume that each MSS is equipped with a device that provides it with partial information about the mobile processes that are present in the system. More precisely, each MSS b_i is provided with a set $local\_trust_i$ of mobile process identities that represents b_i's current view of the mobile processes that are currently present in the systems. We consider the following additional assumption MP_{dyn}: "There is a stable[13] mobile process m and a time τ such that, $\tau' > \tau$, there is a set $Q^{\tau'}$ of at least $2t + 1$ MSSs such that $b_i \in Q^{\tau'}$, either bi locally trusts m at time τ' or b_i has crashed by time τ'." It is important to notice that the set of "*witnesses*" MSSs Q can vary over time.

Let us say that a mobile process m is x-accessible at time τ if it appears in the local trusted list $local\_trust_i$ of x MSSs at timeτ. Moreover, we define $local\_trust_i$ at time τ for a MSS bi that has crashed before that time as the whole set of mobile processes. MP_{dyn} can thus be interpreted as follows: "there is a stable mobile process m that eventually becomes permanently $(2t+1)$-accessible." This study investigates MP_{dyn} and shows that, if a majority of the MSSs are correct, the oracle Ω can be built on top of asynchronous mobile environment that satisfy the assumption MP_{dyn}. Interestingly, no additional assumption is required on MSSs. In that sense, the proposed protocol is time-free. Nearly all the computation and communication are carried out by MSSs, so that the workload of a MH is minimized.

6.2. Computational Model

The mobile environment is a distributed system consisting of two distinct sets of entities: a set of mobile hosts (MHs) and a set of fixed hosts usually named mobile support station (MSSs). The set of MSSs and the communications channels between

[13] a stable mobile process is a process that after entering the system, does not crash nor be disconnected.

them form a static distributed system. On the other hand, the mobile processes can be viewed as a dynamic system. The mobile hosts move in a geographical area, which is partitioned into cells. Each MSS covers one cell and a MH can only communicate (and vice versa) with the MSS responsible the cell in which it is located. A MH is connected to the system if and only if it is up and running and located in a cell covered by an alive MSS.

For the ease of the exposition, we assume the existence of a global discrete clock. This clock is a fictional device which is not known by the processes; it is only used to state specifications or prove protocol properties. The range Γ of clock values is the set of natural integers. We denote $B = \{b_1,..., b_n\}$ the set of MSSs processes (for B stands for base stations) and $M= \{m_1,m_2,...\}$, the set of MHs.

6.2.1. Mobile Support Stations: a Static System

The set of MSSs and its underlying communication network is modeled as a static asynchronous system. The set of MSSs and its underlying communication network is modeled as a static asynchronous system. The wired network of MSSs is made of a finite set of $n \geq 2$ fixed processes, namely, $B = \{b_1,..., b_n\}$. A MSS can fail by *crashing*, i.e., prematurely halting. It behaves correctly (i.e., according to its specification) until it possibly crashes. A process b_i is *correct* in a run if it does not crash in that run, otherwise it is *faulty*. We assume that a majority of MSSs are correct. We use the following notations concerning the B of MSSs:

t: the maximum number of faulty MSSs in a run, $1 \leq t \leq n/2$.

C: the set of MSSs that are correct in a run, $C \subseteq B$.

MSSs communicate by sending and receiving messages through reliable yet asynchronous channels. Each pair of MSSs $\{b_i, b_j\}$ is connected by a wired channel. Channels are reliable in the following sense: they do not alter, create or lose messages. However, channels are asynchronous: the time to transfer a message from b_i to b_j is finite but unbounded, i.e. if b_i sends a message to b_j, then b_j eventually

receives the message unless it fails (let us observe that channels are not required to be FIFO). There is no assumption about the relative speed of processes.

Query-response mechanism. We consider that each process is provided with a query-response mechanism. Such a query-response mechanism can be easily implemented in a time-free manner on top of an asynchronous distributed system. More precisely, any MSS b_i can broadcast (to other MSSs) a QUERY message and then wait for corresponding RESPONSE from n-t MSSs (these are the *winning* responses for that query). The other RESPONSE messages associated with the query, if any, are systemically discarded (they are the *losing* responses for that query).

Both QUERY messages and RESPONSE messages can be used to piggyback data. This allows the querying process to disseminate data to all the processes and obtain data from other processes.

A query issued by b_i is *terminated* if b_i has received n-t corresponding responses. We assume that a process issues a new query only when the previous one has terminated. Without loss of generality, the response from a process to its own query is assumed to always arrive among the first n-t responses. Moreover, QUERY and RESPONSE are assumed to be implicitly tagged in order not to confuse RESPONSE messages corresponding to different QUERY messages. It is assumed that a MSS b_i issues forever sequential queries until it possibly crashes.

6.2.2. Mobile Hosts: a Dynamic System

The system has infinitely many mobile hosts $M = \{m_1, m_2, \ldots\}$ but each run of the leader election protocol has only finitely many. This means that there is no bound on the number of MHs for all runs: whatever be the integer value k, there are runs with more than k MHs. There is a bound on the number of MHs in each run, but a protocol does not know that information because it varies from run to run. This is the *finite arrival model* described and investigated in [4][105].

Each mobile process has a unique identity. A process knows its own identity but does not necessarily know the identities of other MHs. Moreover, identities are

comparable. In the following, we consider that identities are taken from the set of positive integers. m_i denotes the MH whose identity is i. Like MSSs, MHs are asynchronous and can suffer from crash failures. After a MH has crashed, it is seen as permanently disconnected from the system.

A MH is connected only if it is located in a cell covered by a MSS. A MH can only directly communicate with the MSS located in its current cell. Messages between two MHs must be forwarded by corresponding MSSs. When a MH moves from one cell to another, it executes a JOIN operation to inform the MSS of the new cell. A *hand-off* procedure is then executed between the MSSs of the old and new cell.

We assume the existence of an underlying routing layer that allows messages being forwarded in the static system from the source MSS to the local MSS of the destination MH. If the local MSS crashes, the link between the MH and the rest of the system is lost. However, the MH can reconnect by moving into a new cell covered by an alive MSS.

We use the following notation concerning the set M of MHs:

$up(\tau)$: the set of mobile hosts that are connected to the system at time τ, $up(\tau) \subseteq M$. Obviously, these MHs joined the system before time τ and, never crash or disconnects before time τ.

6.3. Problem Definition and Additional Assumptions

Like other leader oracle protocol, some assumptions are needed to cope with the impossibility of implementing a leader oracle in an asynchronous system [111].

6.3.1. Stability Condition

The set of MHs is inherently dynamic: due to mobility, crash failures or energy saving, MHs can join or leave the system at any time. However, if each MH periodically join and then leave the system, being connected only for a short period (i.e., the system is unstable), it is impossible to elect any MH. To allow electing a

leader, the system should exhibit stable periods that last long enough. The following set definition captures this notion of stability [65][112]:

STABLE = $\{m_i \mid \exists \tau, \forall \tau' \geq \tau: m_i \in up(\tau')\}$. *STABLE* is the set of MHs that, once have entered the system, do not crash or disconnect. To guarantee the termination of the proposed protocol, we assume that $STABLE \neq \emptyset$.

6.3.2. Problem Definition

A *leader* oracle is a distributed facility that provides the mobile processes with a function *leader*() that returns the identity of a mobile process each time it is invoked. A unique mobile process is eventually elected but there is no knowledge of when the leader is elected. Moreover, to be useful the eventual unique leader m_l must be a stable mobile process (i.e., $m_l \in$STABLE). More precisely, the leader oracle Ω satisfies the following property:

Eventual Leadership: There is a time τ and a mobile process $m_l \in STABLE$, such that after τ, any invocation of *leader*() by any process m_i returns l.

6.3.3. Local Failure Detection

We suppose that each MSS b_i is able to gather partial knowledge about the mobile processes that are present in the system. We define the information available to MSSs in the failure detector framework [30][31].

We assume that each MSS b_i is equipped with a local failure detector that provides a set $local\_trust_i \subseteq M$, which means that the MHs in the set currently up and connected. More precisely, at each MSS b_i, the set $local\_trust_i$ satisfies the following properties ($local\ trust_i^{\tau}$ denotes the value of $local\_trust_i$ at process b_i at time τ):

Eventual Accuracy: $\exists \tau, \forall \tau' \geq \tau: STABLE \subseteq \cup_{i \in C}\ local\_trust_i^{\tau'}$

Completeness: If mobile m_j never join the system, crashes or permanently leaves the system then $\exists \tau, \forall \tau' \geq \tau: m_j \notin \cup_{i \in C}\ local\ trust_i^{\tau'}$

The completeness part requires that a MH that crashes or permanently leaves the system is eventually no longer trusted by any MSS. The accuracy part requires that

eventually, at least one stable mobile process *m* is continuously trusted by MSSs. However, it is not necessary that the same MSS eventually permanently trust this stable mobile process *m*. On the contrary, we only require that after some time, *m* is always trusted by some MSS and at different time it may be trusted by different MSSs.

From a practical point of view, this failure detection capability can be implemented as follows. Each MSS monitors the mobile processes located in its cell by periodically broadcasting queries. When a mobile process receives such a query, it sends back its identity. To ensure that stable mobile processes are continuously trusted, some delay is necessary before a MSS removes a MH from its $local\_trust_i$, after the MH is perceived to be disconnected from the cell.

A different approach along the line of gossip-based failure detection implementation [66][76] can also be used. An alive MH periodically broadcasts *ALIVE* messages through the wireless medium. When a process (mobile or fixed station) receives such a message, forwards it. When a MSS b_i receives such a message, it adds the initiator of the message to its $local\_trust_i$. If the *ALIVE* message has not been received from a MH *m* for Δ units of time, *m* is removed from the $local\_trust_i$. The value of Δ depends on the behavior of the underlying network. Investigating implementations of such a local failure detector service is out of the scope of this thesis.

Let us observe that this failure detector does not provide much information on MHs that are present in the system. It only guarantees that eventually, at least one stable mobile process *m* is trusted by some MSS at each time instant, but is possible that the *local_trust* sets permanently disagree. Since communications between MSSs are asynchronous and a MSS may have a different *local_trust* set at each time instant, MSSs cannot agree on the stable mobile processes they trust.

6.3.4. An Assumption on the Movement of MHs

We consider the following additional assumption, called MP_{dyn} (a shortcut for dynamic mobility pattern):

There are a stable mobile process m and a time τ (m and τ are not known in advance) such that at any time instant $\tau' \geq \tau$, there exists a set $Q^{\tau'} \subseteq B$ that satisfies the following property:

$\forall \tau' \geq \tau$: $|Q^{\tau'}| \geq 2t+1$;

$\forall b \in Q^{\tau'}$: if b has not crashed by time τ, $m \in local\_trust_b^{\tau'}$.

The intuition that underlies this property is the following. Even if nothing is known about the movement of mobile processes, it is possible that their behavior exhibits some regularity that can be exploited in order to implement a leadership facility. More precisely, the assumption MP_{dyn} states that, eventually, there is a set of $2t+1$ MSSs that trust the same mobile process. Moreover, this set can continuously change over the time.

The property MP_{dyn} can be interpreted as follows: among the mobile processes, there is at least one stable mobile process m that eventually moves "fast enough" across a "large enough" (namely, $2t + 1$) number of cells. Due to the latency between the instants at which a mobile process leaves the cell of a MSS b_i and the instant at which it is removed from the set $local\_trust_i$, it follows that if the mobile process m permanently traverses sufficiently fast at least $2t + 1$ distinct cells then at any point of time, the stations responsible for those $2t + 1$ cells locally trust m.

Another possible solution to guarantee the MP_{dyn} assumption is to deploy a multiple coverage mobile network. Each point in the territory is covered by at least $2t+1$ MSSs rather than one MSS. Then each MH keeps contact with at least $2t+1$ MSSs simultaneously. In such a way, the MP_{dyn} assumption can be guaranteed deterministically. Although in existing infrastructured mobile systems, e.g. GSM and CDMA networks, one point is typically covered by one MSS, some area is covered by more than one MSS to smooth the handoff procedure or deal with the shadow fading problem [1][56][57][94].

With the advance in hardware technologies, in the future, it should be more feasible to deploy a fully (2*t*+1)-coverage cellular system to provide better performance and service with little additional cost.

init: $sn_i \leftarrow 0$; $trust_i \leftarrow \top$;

Task 1

(01) Repeat foreach $j \in B$ do send *PH1_QUERY*() to b_i endfor;

(02) wait until corresponding PH1_RESPONSE() has been received form $\geq n\text{-}t$ MSSs;

(03) $PH1\_rec_i$ = {*j*: a *PH1_RESPONSE* received from b_j at line 03};

(04) foreach $j \in B$ do send *PH2_QUERY*(sn_i, $trust_i$) to b_j endfor;

(05) wait until corresponding *PH2_RESPONSE*(*L_TURST*) received from $\geq n\text{-}t$ MSSs;

(06) $PH1\_rec_i$ = {*j*: a *PH2_RESPONSE* received from b_j at line 06};

(07) let $REC\_FROM_i = \cup_{i \in PH1\_rec_i \cap PH2\_rec_i} L\_TRUST_j$;

(08) $trust_i \leftarrow trust_i \cap REC\_FROM_i$;

(09) endrepeat

Task 2

(10) upon reception of PH1_QUERY() from b_j:

(11) $query\_start_i[j] \leftarrow current\_time()$; send *PH1_RESPONSE*() to b_j;

(12) upon reception of PH2_QUERY() from b_j:

(13) if $sn_i = sn_j$ then $trust_i \leftarrow trust_i \cap trust_j$ endif;

(14) if $sn_i < sn_j$ then $trust_i \leftarrow trust_j$, $sn_i \leftarrow sn_j$ endif;

(15) if $trust_i = \emptyset$ then $trust_i \leftarrow \top$, $sn_i \leftarrow sn_i + 1$ endif;

(16) $LOCAL\_TRUST_i \leftarrow \cup_{query\_start_i[j] \leq \tau \leq current\_time()} local\_trust_i^{\tau}$;

(17) send PH2_RESPONSE($LOCAL\_TRUST_i$) to b_j;

(18) upon reception of *LEADER_QUERY*(*m*) from a mobile host *m*:

(19) if $trust_i = \emptyset \vee trust_i = \top$ then $l_i = m$;

(20) else $l_i = min(trust_i)$;

(21) endif

(22) send LEADER(l_i) to *m*;

Figure 6-1 Eventual leadership protocol: code for MSSs

6.4. Description of the Protocol

The protocol is described in Figure 6-1 and Figure 6-2. It extends ideas that previously appear in [39][111][112]. The MSSs act as servers to provide an eventual

leadership service to the mobile processes. To attain this goal, each MSS *bi* maintains a set $trust_i$ of identities of mobile processes. The aim of the protocol is for the MSSs to eventually have the same set of identities. When a mobile process m invokes the primitive *leader*(), it sends a LEADER REQUEST message to its local MSS (Figure 6-2). When a MSS receives such a message, it deterministically chooses an identity among the mobile processes it currently trusts and sends back this identity (line 18 to line 22).

The protocol consists in two tasks run in parallel at each MSS. Task 1 is the core task in which each process initiates sequential queries and wait for corresponding responses. Task 2 is triggered by reception of messages. It implements the response mechanism associated with the queries: when a process b_i receives a query, it sends back a response carrying values that depend on the type of query received (line 11 and line 17).

Each process b_i associates with its $trust_i$ set a sequence number sn_i. sn_i is a logical date defining the "age" of set $trust_i$. The period, during which the sn_i keeps the same value, is called an "epoch". When b_i receives a pair $< sn_j , trust_j >$ (line 12), it updates $trust_i$ according to the respective values of sn_i and sn_j. If they are equal, it considers $trust_i \cap trust_j$ as the new value of the set of mobiles processes it trusts (line 13).

```
When Leader() is invoked:
(23) send LEADER_QUERY(m) to the current local MSS;
(24) wait until LEADER(l) is received;
(25) return (l)
```

Figure 6-2 Eventual leadership protocol: code for MHs

If its current knowledge is too old, it adopts the set received (line 14). Otherwise, it discards the message received. If bi then discovers that its set $trust_i$ is empty, bi starts a new "epoch" by increasing its sequence number sn_i and resets $trust_i$ to its initial value (line 15). Let us notice that during an "*epoch*", a set $trust_i$ can only decrease or remain constant. The proof shows that it exists an epoch with a finite age

after which the sn_i values no longer increase and the sets $trust_i$ are (and remain) non-empty and equal. They actually converge towards to a subset of the *STABLE* set. The mobile process in these trusti sets with the smallest identity is then elected as the unique leader.

In order to benefit from the MP_{dyn} assumption, each process bi collects local trust sets of other processes by sequentially issuing two-phase query-responses cycles. In the first phase, b_i broadcasts a *PH1_QUERY*. When a process b_j receives such a query, it sends back a *PH1_RESPONSE* and starts recording the identities of processes it locally trusts until it receives a *PH2_QUERY* from b_i. The *PH2_ RESPONSE* message sent back by b_i carries the identities of the mobile processes that have been locally trusted by b_j since it has received the matching *PH1_QUERY* of b_i (lines 16-17). To see why this two-phase query-response cycle is necessary, let us assume that when a process b_j received a query at a time τ, its sends back a response message that contains the value of local $trust_j$ at time τ. b_i collects *n-t* local trust sets, but these sets may have been "seen" at distinct times. Since the set *Q* of "witness" processes defined in property MP_{dyn} can change over time, it is possible that the local trust sets collected by b_i does not satisfied any global 13 property, even if the property MP_{dyn} is established. On the contrary, we will show in the proof that this two-phase query-response mechanism guarantees that the sets REC_FROM_i (i.e., the union of *local_trust* collected, lines 07) eventually satisfy a global property. More precisely, Lemma 1 states that there exists a stable mobile process that eventually is always contained in any *REC_FROM* sets.

$\top$ is a special symbol that represents the whole universe of the mobile processes. Moreover, $\top \cap A = A$ (where *A* is any set of mobile processes).

6.5. Correctness Proof

In the following x_i^{τ} denotes the value of the local variable *x* of process p_i (MH or MSS) at time τ. Given an execution, *C* is the set of MSSs that are correct in that

execution. *STABLE* is the set of mobile processes that, after having entered the system, do not crash or disconnect.

Lemma 1 *There is a time* τ *and a stable mobile process m (i.e.* $m \in$STABLE*) such that every REC_FROM set computed (at line 07) after* τ *is such that* $m \in REC\_FROM$.

Proof. Given an execution that satisfies the *MPdyn* assumption, there is a time τ_0 and a mobile process $m \in STABLE$ such that $\in \tau' \geq \tau_0$, there exists a set $Q^\tau \subseteq B$ where:

(1) $|Q^\tau| \geq 2t + 1$ and

(2) $\forall b_i \in Q^\tau$: $m \in local\_trust_i$.

Let us consider a MSS b_i that starts a query (at line 02) after τ_0. Let b_j be a MSS such that $j \in PH1\_rec_i \cap PH2\_rec_i$. This means that, for each phase of the query issued by b_i, the responses messages sent by b_j arrived among the first $n- t$ ones at process b_i. Let τ_start_j be the time instant at which the *PH1_QUERY* from b_i is delivered to b_j and $\tau\_ end_j$ be the time at which b_j sends back the *PH2_RESPONSE* message. Let us observe that the *PH2_RESPONSE* message sent by b_j carries the identities of all the mobile processes that has been trusted at least once by b_j during the time interval $[\tau\_start_j, \tau\_ end_j]$ (lines 16-17). The rest of the proof relies on the two following observations: O_1: $|PH1\_rec_i \cap PH2\_rec_i| \geq n\text{-}2t$; O_2: $\exists \tau, \forall j \in PH1\_rec_i \cap PH2\_rec_i$: $\tau \in [\tau\_start_j, \tau\_ end_j]$.

Let us consider the set REC_FROM_i computed by b_i after completing its query. This set is the union of the mobile processes that has been trusted by the MSSs b_j, $j \in PH1\_rec_i \cap PH2\_rec_i$ at some time instant between the beginning and the end of the two-phase query of b_i. Let τ_1 be the time instant introduced in observation O_2. In particular, $\cup_{j \in PH1\_rec_i \cap PH2\_rec_i} local\_trust_j^{\tau 1} \subseteq REC\_FROM_i$. As $\tau_1 \geq \tau_0$, it follows from the assumption MP_{dyn} that at time $\tau 1$ there exist a set $Q^{\tau 1}$ of at least $\alpha \geq 2t + 1$ MSSs that either have crashed or trust m at time τ_1 . Since $|PH1\_rec_i \cap PH2\_rec_i| \geq n\text{-}2t$ (O_1), it follows that $Q^{\tau 1} \cap (\cup_{j \in PH1\_rec_i \cap PH2\_rec_i} local\_trust_j^{\tau 1})$, from which we conclude that $m \in REC\_FROM_i$.

We have shown that there exists a time τ_i after which any REC_FROM_i set computed by b_i contains the identity of the stable mobile m. Taking $\tau_{max} = max\{\tau_i: i \in B\}$ completes the proof.

Observation O_1: *For any process b_i that initiates and completes a two phases query,* $|PH1\_rec_i \cap PH2\_rec_i| \geq n\text{-}2t \geq 1$.

Proof of O_1: Since in each phase of the query, b_i waits for $n\text{-}t$ winning responses, we have $|PH1\_rec_i| \geq n\text{-}t$ and $|PH2\_rec_i| \geq n\text{-}t$. Consequently, sets $PH1\_rec_i$ and $PH2\_rec_i$ differ in at most t identities. It follows that $|PH1\_rec_i \cap PH2\_rec_i| \geq n\text{-}2t$. As $t < n/2$, $PH1\_rec_i \cap PH2\_rec_i \neq \emptyset$. *End of proof O_1*

Observation O_2: *Let b_i be a process that initiates and completes a two-phase query:*

$$\cap_{j \in PH1\_rec_i \cap PH2\_rec_i} [\tau\_start_j, \tau\_end_j] \neq \emptyset.$$

Proof of O_2: Let $j \in PH1\_rec_i$. Let us recall that τ_start_j is the time at which *PH1_QUERY* is delivered at process b_j and τ_end_j is the time at which b_j sends the *PH2_RESPONSE* message to b_i. Let τ be the time at which the second phase of the query is initiated by b_i (i.e., the time at which b_i broadcasts a *PH2_QUERY* message, line 04). We show that $\tau \in [\tau\_start_j, \tau\_end_j]$. When b_i starts the second phase of the query, it has received a *PH1_RESPONSE*() from b_j. As b_j sends such a message when it has delivered a *PH1_QUERY* from b_i, we have $\tau\_start_j < \tau$. Similarly, b_j sends a *PH2_RESPONSE* to b_i when it has received the corresponding *PH2_QUERY* from b_i, which implies that $\tau < \tau\_end_j$. Consequently, we obtain that $\tau \in [\tau\_start_j, \tau\_end_j]$.***End of proof O_2*** □

Lemma 2 $\exists SN, \exists \tau, \forall i \in B, \forall \tau' \geq \tau: i \in C \Rightarrow sn_i(\tau') = SN$.

Proof. Let τ_0 be a time such that: 1) all faulty MSSs have crashed, and 2) all messages sent by faulty MSSs have been delivered. Let τ_1 be the time defined in Lemma 1 and let $\tau_{clean} = max(\tau_0, \tau_1)$. The idea is that after time τ_{clean} the system exhibits a "*clean*" behavior.

Let $SN^{\tau clean}$ be the maximal sequence number sn_i among the correct MSSs b_i at time τ_{clean}. Moreover, if there is a correct MSS b_j such that $trust_j = trusted$ and $sn_j =$

sn, we say that "the set *trusted* is associated with the sequence number *sn*" (please notice that several sets can be associated with the same sequence number).

Claim C₁. *If Ø is associated with $SN^{\tau clean}$, then: (1) a process b_j that executes the reset statement at line 15, after which we have $(trust_j, sn_j) = (\top, SN^{\tau clean} + 1)$, and (2) $(\top, SN^{\tau clean} + 1)$ is sent to all the processes.*

Proof of C₁. Let us first observe that (Observation O_3) a set $trust_i$ can only decrease while sn_i remains equal to $SN^{\tau clean}$, (Observation O_4) there is no gap in sequence numbers (which means that if a sequence number variable is equal to *SN*, then there are sequence number variables that had previously the values 0, 1, . . . , *SN*-1), and (Observation O_5) the update by a process b_j of its sn_j variable to the value *SN*+1 (at line 13 or 14) is always due to the fact that some process b_k (which is possibly b_j itself) has executed $sn_k \leftarrow sn_k+1$ at line 15 (where *SN* is the value of sn_k before the update; notice that b_k also sets $trust_k$ to $\top$).

Let b_i be a process that associates Ø with $SN^{\tau clean}$. If the pair $(trust_i, sn_i)$ remains equal to $(Ø, SN^t)$ until b_i receives a *PH2_QUERY*, it executes line 15 and consequently resets $(trust_i, sn_i)$ to $(\top, SN^{\tau clean} + 1)$.

The only other possibility for that pair to be modified is at line 14, but in that case p_i received a sequence number > $SN^{\tau clean}$, and it follows from the observations O_4 and O_5 that some process b_j has executed line 14, updating the pair $(trust_i, sn_i)$ to $(\top, SN^{\tau clean} + 1)$. This proves the first part of the claim.

The proof of the second part of the claim is by contradiction. Let us assume that no process sends a *PH2_QUERY* with the pair $(\top, SN^{\tau clean} + 1)$. This means that the pairs that are sent have the form $(X, SN^{\tau clean} + 1)$ with $X \neq \top$. Let b_{i1} be a process such that at some time τ_{i1} we have $(trust_{i1}, sn_{i1}) = (\top, SN^{\tau clean} + 1)$. As it sends at some time $\tau_{i1'} > \tau_{i1}$ the pair $(X_{i1}, SN^{\tau clean} + 1)$ with $X_{i1} \neq \top$ (by assumption), we conclude that, between τ_{i1} and $\tau_{i1'}$, b_{i1} received a query from some process b_{i2} carrying $(X_{i2}, SN^{\tau clean} + 1)$ with $X_{i2} \neq \top$. The same reasoning can be applied to b_{i2}, from which we conclude that there is a process b_{i3}, etc. It follows that we can construct an infinite chain of distinct processes, which is clearly impossible as there

is a finite number of mobile support stations. It follows that there is a process that sends a *PH2_QUERY* carrying the pair ($\top$, $SN^{\tau clean} + 1$). *End of the proof of Claim* C_1.

We show that $SN = SN^{\tau clean}$ or $SN = SN^{\tau clean} + 1$. According to the definitions of τ_{clean} and $SN^{\tau clean}$, there exists a correct process b_i such that $sn_i = SN^{\tau clean}$. Due to the gossiping mechanism, after some time we will have $sn_j > SN^{\tau clean}$ for each $j \in C$. We consider two cases:

Case 1: Ø is never associated with $SN^{\tau clean}$. In that case, no correct process b_i will ever execute the reset statement at line 15. It follows that no process b_i will increase its sn_i variable, and the lemma follows.

Case 2: Ø is associated with $SN^{\tau clean}$. From the claim C_2, there is inevitably a process b_j that executes the reset statement at line 15, after which we have ($trust_j$, sn_j) = ($\top$, $SN^{\tau clean} + 1$), and this pair is sent to all the correct processes. This means that after some time, each process b_i will be such that $sn_i \geq SN^{\tau clean} + 1$. As this occurs after time τ_1, the time defined in Lemma 1, it follows that from now on, any set $trust_i$ permanently contains the stable mobile process m defined in Lemma 1. This is because each time b_i updates its set of trusted mobile processes (line 08), it intersects $trust_i$ which has been reset to $\top$ (the whole universe of mobile processes) with REC_FROM_i that always contains m. Consequently, no *PH2_QUERY*(Ø, $SN^{\tau clean} + 1$) are sent. Hence, no process can execute the reset statement at line 15, from which we conclude that no sequence number $SN^{\tau clean} + 1$ can be generated and the lemma holds.□

Theorem 1 *In any execution that satisfies the* MP_{dyn} *assumption, the protocol described in Figure 6- 1 implements a leader facility in a mobile environment.*

Proof. Given a run that satisfies MP_{dyn}, let $PL = \cap \{trust_i : i \in C \wedge trust_i$ is associated with $SN\}$, where SN is defined in Lemma 2.

We first show that $PL \neq \emptyset$. Due to lemma 2, no sequence number greater than SN can be generated. This implies that Ø cannot be associated with SN. Moreover, it follows from Lemma 1 and updates of $trust_i$ (line 08) that any $trust_i$ associated with SN contains the stable mobile process m introduced in Lemma 1.

We now show that $PL \subseteq STABLE$. This a consequence of the completeness property satisfied by the local $trust_i$. More precisely, the completeness property states that a mobile process that crashes or gets disconnected from the system is eventually no longer locally trusted by each MSS *b*. Consequently, there is a time after which every *REC_FROM* does not contain crashed or disconnected mobile processes.

Therefore, there is a time after which the REC_FROM_i contains only stable processes. Moreover, as the $trust_i$ sets are never reset to $\top$, it follows that, after that time, these $trust_i$ sets can contain only stable *m*obile processes.

Finally, there is a time τ after which we have $\forall i \in C$: $trust_i = PL$. This is a consequence of the finite arrival model (after some time, no more mobile processes join the system) and the gossiping mechanism (lines 04 and reception of *PH2_QUERY* in Task 2). Let us consider an invocation made after τ of *leader*() that returns m_l. We have $m_l = min\ (trust_i)$ where *i* is the identity of some correct MSS. Since $trust_i = PL \subseteq STABLE$, it follows that any of these invocations returns the same stable mobile process.□

6.6. Summary

Leader primitive is at the core of many coordination problems, e.g. consensus. We investigated the implementation of the eventual leader oracle Ω in the context of dynamic infrastructured mobile environments, where the number of participating processes can change arbitrarily as time passes and processes can join or leave the system at any time.

The set of MSSs is viewed as a static system, while the MHs constitute a dynamic system. It has shown that as soon as there is a stable mobile process continuously seen by a large enough set of MSSs, Ω can be implemented. Moreover, this set can be different at different time. Interestingly, no additional assumption is made on the network of MSSs. This fact, combining with the dynamic nature of assumption MP_{dyn}, makes attractive to investigate this approach in the context of unstructured mobile networks, e.g., MANETs.

Chapter 7. A Permission-based MUTEX Algorithm for MANETs

7.1. Overview

As discussed before, all existing MUTEX algorithms for MANETs use token-based approaches. In this chapter, we propose the first permission-based MUTEX algorithm for MANETs. The proposed algorithm is based on the “look ahead” technique [146] proposed for infrastructured mobile networks. To apply the “look ahead” technique in MANETs, the following issues have to be considered.

First and most importantly, there is no MSS in a MANET, so some additional steps should be taken to ensure that the algorithm can continue its execution after one or more hosts disconnect or doze. Second, the algorithm in [146] does not provide methods for some important functions. There is no method for initializing the two key data structures – *Info_set* and *Status_set*. Third, the assumption about the FIFO channel becomes infeasible in MANETs. Because the route between two MHs changes from time to time due to the movements of the MHs, implementing the FIFO channel in MANETs is very costly. Finally, there is no fault tolerance mechanism for handling host failures or link failures.

The problems above are all addressed in our proposed algorithm. We propose a simple and efficient method to initialize *Info_set* and *Status_set*. The disconnections and dozes of hosts are also handled. When a host wants to disconnect from the network or enter the doze mode, it informs other hosts by sending messages. Both the sender and receiver modify the information maintained accordingly. When a host wakes up from the "doze" mode, it resumes the execution immediately without performing any special action. When a host reconnects to the network, it informs other hosts and resumes the execution.

To relax the constraint of requiring FIFO channels, we add a new variable Q_{req}. In [146], if a channel is not FIFO, a REPLY message from a host S_i (with lower priority)

to a host S_j (with higher priority) may be delivered following the REQUEST message from S_i to S_j. On the reception of the REPLY message, S_j moves S_i to the $Status\_set_j$ and consequently S_i can never get a REPLY from S_j. With the help of Q_{req}, such a request is recorded and the host with the lower priority would not be put in *Status_set* after the reception of the REPLY. FIFO is no longer necessary.

Using timeout, a fault tolerance mechanism is developed to tolerate both link and host failures. A timeout value is set for each request message sent out. Intermittent and recoverable link and host failures are handled by resending request messages when the timeout expires.

7.2. System Model and Assumptions

A MANET consists of a collection of n autonomous MHs, $S = \{S_1, S_2, \ldots, S_n\}$, communicating with each other through wireless channels. Whether two hosts are directly connected is determined by the signal coverage range and the distance between the hosts. Each host is a router and the communication between two hosts can be multiple hops. Both link and host failures can occur. The topology of the interconnection network can change dynamically due to mobility of hosts and failures of links and hosts.

At any moment, each MH is in one of three different states: *normal*, *doze*, and *disconnection*. For the disconnection mode, two different cases are considered: *voluntary disconnection* and *accidental disconnection*. An "accidental disconnection" refers to disconnection aroused by failures of links or hosts. Such disconnections occur more frequently and unpredictably than that in wired networks due to the unstable links and characteristics of mobile hosts. A MH may also voluntarily disconnect from the network to save the battery power [17][115]. Since the MH knows such a disconnection in advance, it can execute predefined operations for the distributed algorithm that it currently participates in.

A distributed system built on a MANET is an asynchronous system, so there is no bound on the processing speed of hosts or message delay. To provide support for

tolerating recoverable link and host failures, we use the timeout and message retransmission mechanisms. We assume that the link failure or host failure is eventually recovered within the retrying period.

7.3. Data Structures and Message Types

Following the definitions in [136][146], each host S_i maintains two sets, which are defined below:

$Info\_set_i$: an array of the IDs of the hosts to which S_i needs to send request messages when it wants to enter CS.

$Status\_set_i$: an array of the IDs of the hosts which, upon requesting to access CS, would send the request messages to S_i.

To ensure the correctness of the algorithm, the following conditions must be satisfied:

1) $\forall S_i$: $Info\_Set_i \cup Status\_Set_i = S; \forall S_i$: $Info\_Set_i \cap Status\_Set_i = \emptyset$

2) $\forall S_i \forall S_j$: $S_i \in Info\_Set_j \Rightarrow S_j \in Status\_Set_i$

Obviously, Condition 1) guarantees that host S_i knows the request status of all the other hosts and there is no redundancy information. Condition 2) guarantees the consistency among the sets of all MHs.

In addition, each host maintains the following data structures.

ts_{req}: the timestamp for the request of S_i. It is used as the priority of the request of S_i. If S_i is not requesting for CS, it is set to NULL.

Q_{req}: the array of the IDs of the hosts which have sent requests to S_i but S_i has not sent back a reply yet.

TO_{req}: the array of timers each associated with a REQUEST message sent out.

T_{rec}: the timestamp of the last reconnection. It is set to “0” initially.

The messages used in the algorithm are classified into the following types.

REQUEST: the message sent from a host requesting CS to other hosts for getting their permissions. The message contains the priority of the request (e.g. a unique timestamp).

REPLY: the message sent by a receiver of a REQUEST to grant the permission of accessing CS.

DOZE: the message to inform others that the sender is entering the doze mode.

DISCONNECT: the message to inform others that the sender is disconnecting voluntarily.

RECONNECT: the message to inform others that the sender has reconnected to the network after a voluntary or accidental disconnection. The message contains the priority (e.g. a unique timestamp) of the reconnection.

7.4. Description of the Algorithm

7.4.1. Initialization of *Info_set* and *Status_set*

The algorithm for initializing the *Info_set* and *Status_set* is shown in Figure 7-1. We use an $n \times n$ matrix M, where n is the number of hosts in the network, to represent the relationships among the hosts. The value of each element of M, m_{ij}, represents the relationship between the pair of hosts S_i and S_j. If $m_{ij} = 0$, S_j is in the *Info_set* of S_i. If $m_{ij} = 1$, S_j is in the *Status_set* of S_i. To ensure that the sets of all the hosts satisfy the conditions 1) and 2) specified in Section 7.3, an arbitrary host, say S_0, is selected to act as the initiator of the algorithm. The initial value of M is determined by the initiator.

--Executed by the initiator: --	**--Executed by all the hosts: -----**

//Generate the Upper Triangular Matrix M_u
for{ i = 0 to |*S*|-1}
  for{j= i+1 to |*S*|-1}
  m_{ij} = random(0,1);
  endfor
endfor
broadcast M_u to all other hosts;
	//Step 1: Generate the Lower Triangular Matrix M_l
	for{ i = 0 to |*S*|-1}
	  for{j= 0 to i-1} m_{ij} =1-m_{ji}; **endfor**
	endfor
	//Step 2: Initialize the *Info_set* and *Status_set*
	// for host S_i//
	for { j= 0 to |*S*|-1and j !=i}
	  if {m_{ij}==0} put S_j into $Info\_set_i$;
	  else put S_j into $Status\_set_i$;
	endfor

Figure 7-1 Algorithm for initialization of *Info_set* and *Status_set*

S_0 generates the upper triangular matrix M_u randomly and broadcasts M_u to all other hosts. Then, all the hosts, including S_0, set the value of each element of the lower triangular matrix M_l to be the 2's complement of the corresponding element in

the upper triangle. Finally, according to the corresponding row in *M*, each host initializes its *Info_set* and *Status_set*. It is easy to verify that our initialization algorithm meets the specified conditions with only a few messages.

In the initialization algorithm, the failures of hosts or links are not considered. To tolerate failures, each host sets a timeout for the message of M_u. If the timeout expires, the host sends a query message to the initiator. This may be repeated until M_u is received from the initiator. Since a host eventually recovers from a failure, each host eventually receives the M_u and initializes its sets.

```
CoBegin
//Send Request//
if(host S_i wants to enter CS
{   set ts_req to the current time;
   for(S_j∈Info_set_i ) do begin
      Send REQUEST to S_j;
      Set timeout in TO_req for S_j;
  endfor
  goto "Enter CS";}
//Enter CS//
if(TO_req ==Φ) Enter CS;
//Exit CS//
   Set ts_req to NULL;
   for (S_j∈Info_set_i) do begin
      Send REPLY to S_j;
      Remove S_j from Q_req;
   Endfor
//Enter Doze Mode//
Broadcast "DOZE";
Set Status_set= Φ;
Set Info_set= S;
//Exit Doze Mode//
Set Status_set= Φ;
Set Info_set= S;
//Disconnect voluntarily//
Broadcast "DISCONNECT";
Set Status_set= Φ;
Set Info_set= S;
//Reconnect//
Broadcast "RECONNECT";
Sets Status_set= Φ;
Info_set= S;
//Handling timeout//
if(timeout happens for host S_j )
{    Resend REQUEST o S_j;
     Set timeout for S_j in TO_req;}
```

```
//Handling messages//
Upon S_i receives a message from S_j:
if(REQUEST)
{   Put S_j into Q_req;
    if (S_j ∈Status_set_i)
    {   Move S_j into Info_set_i;
        if (S_i is with lower priority)
        {    Send REQUEST to S_j;
             Set timeout in TO_req for S_j;}
        }
        if (S_i is not requesting or
            priority of S_i is lower)
        {     Send REPLY to S_j;
              Remove the S_j from Q_req;}
    }
}
if(REPLY)
{  if(S_j∉ Q_req)
   {   Move S_j to Status_set_i,
       Remove S_j from TO_req ;
       goto "Enter CS";}
}
if(DISCONNECT or DOZE)
{    Remove S_j from TO_req and Q_req;
     if(S_j∈ Info_set_i)
        Move S_j into Status_set_i;
}
if(RECONNECT)
   if(T_rec<timestamp of RECONNECT)
   {    Remove S_j from TO_req and Q_req;
        Move S_j into Status_set_i;
    }
CoEnd
```

Figure 7-2 Body of the permission-based MUTEX algorithm

7.4.2. Normal Execution (without Disconnection or Doze)

The pseudocode of the proposed MUTEX algorithm is shown in Figure 7-2. All the hosts execute the same code.

When a host wants to enter the CS, it first sets ts_{req} to the current time and sends the REQUEST messages to all the hosts in its *Info_set*. To tolerate link and host failures, a timeout is set in TO_{req} for each request message sent. The host then waits for a REPLY message corresponding to each REQUEST message sent out. If the *Info_set* is empty, it enters CS immediately.

When a host S_i receives a REQUEST message from another host S_j, it moves S_j to $Info\_set_i$ and records the request in Q_{req}. If S_i itself is not requesting for CS or its priority is lower, it sends a REPLY message to S_j and removes the record for S_j in Q_{req}. If S_j is in $Status\_set_i$ before S_i receives the REQUEST form S_j and S_i is requesting for CS with a lower priority, S_i sends a REQUEST to S_j.

Upon receiving of a REPLY message from host S_j, S_i removes the timeout (in TO_{req}) associated with S_j. If S_i finds no request from S_j in its Q_{req}, S_j is moved to $Status\_set_i$.

When the timeout for a REQUEST message expires, the requesting host sends a REQUEST again. When all the replies for REQUEST messages have been received, the requesting host enters CS.

On exiting CS, a host sends REPLY messages to all hosts in its *Info_set*.

It is worth notice that when two hosts compete for the CS simultaneously, if we do not recorder the REQUEST separately in Q_{req}, it is possible that the host with the lower priority never gets a REPLY from the other host. This is caused by the non-FIFO property of communication channels. The following example execution shown in Figure 7-3 illustrates the usage of Q_{req} clearly.

The example execution shows the scenario with only two hosts. At the beginning, the host S_i is in the $Status\_set_j$ while S_j is in the $Info\_set_i$ (Figure 7-3-(a)). Then S_i and S_j each generate a request while S_i has a higher priority. Since S_j is in $Info\_set_i$, S_i has to send a REQUEST message to S_j (Figure 7-3-(b)). Upon receiving the REQUEST

message from S_i, S_j moves S_i from $Status\_set_j$ to $Info\_set_j$ and records the request in its Q_{req}. Because the priority of S_j is lower, S_j sends a REPLY and a REQUEST to S_i. However, due to the non-FIFO channel, the REQUEST arrives at S_i first, and consequently S_i records this request in Q_{req} (Figure 7-3-(c)). In Figure 7-3-(d), the REPLY arrives at S_i, but S_i does not move S_j into $Status\_set_i$ because there is a request from S_j recorded in Q_{req}. After receiving replies for all REQUEST messages sent out, S_i enters the CS. Upon exiting from the CS, S_i sends a REPLY message to S_j (Figure 7-3-(e)) (Without Q_{req}, S_j should be in $Status\_set_i$ and cannot get the REPLY). S_j moves S_i to $Status\_set_j$ and enters CS (Figure 7-3-(f)).

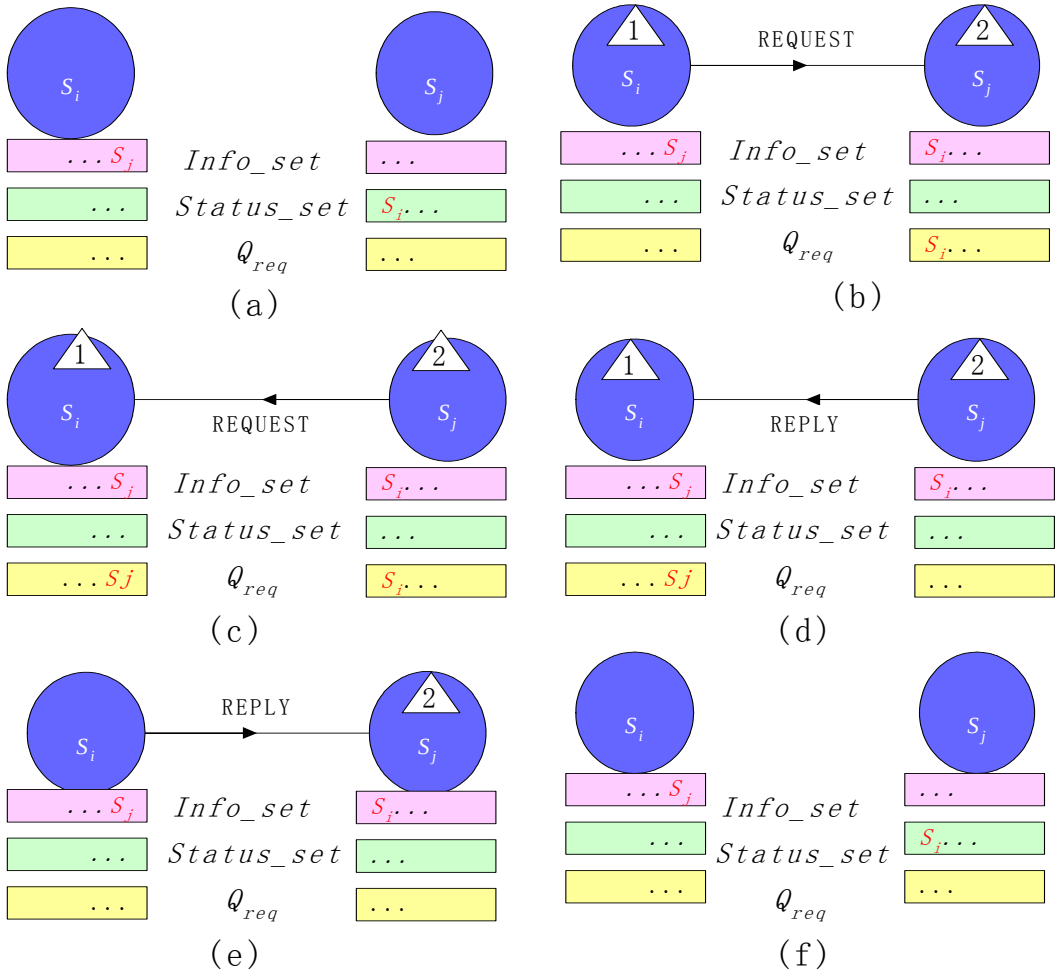

Figure 7-3 An example execution of the permission-based algorithm

7.4.3. Handling Doze and Disconnection

When a host S_i wants to enter the "doze" mode, it broadcasts a DOZE message to all other hosts and moves all hosts in its *Status_set* to its *Info_set*. All other hosts

move S_i into their *Status_set* after they receive the DOZE message from S_i. This ensures that the dozing host would not be disturbed. When a dozing host wakes up, it resumes the execution without any special operation.

If a host S_i wants to disconnect voluntarily, the same steps would be taken except that S_i will broadcast a DISCONNECT message, rather than a DOZE message. When a host S_i reconnects to the network after a disconnection, either a voluntary one or an accidental one, S_i needs to broadcast a RECONNECT message to inform other hosts and move all the hosts in its $Status\_set_i$ to its $Info\_set_i$.

When a host S_j receives a RECONNECT message from S_i, it compares its T_{rec} with the timestamp of the received RECONNECT message. If T_{rec} is less, moves S_i to $Status\_set_i$ and if it is waiting for a REPLY message from S_i, S_j removes the corresponding timeout in TO_{req}. The comparison is necessary because several hosts may send out a RECONNECT message concurrently. For example, if S_i and S_j send out RECONNECT concurrently, they will move each other to *Status_set* if they do not compare the time of reconnection. This violates the conditions for *Status_set* and *Info_set* specified before.

7.5. Correctness of the Proposed Algorithm

In this section we prove the correctness of the proposed algorithm by showing that the three correctness requirements for distributed MUTEX algorithms are satisfied.

Lemma 1*: *Based on the assumptions, the effect of recoverable link or host failures can be eliminated.

Argument. Without loss of generality, we assume that the link between S_i and S_j fails and some message is lost. If neither of the two hosts is waiting for reply from another, the link failure has no effect on their executions. So, we assume that S_i is waiting for the reply of S_j. Eventually, the timeout for S_j would expire and the request is resent. Since we assume that a link failure can be recovered within the retrying period, S_j eventually will receive the request after the request is resent one

or more times.

Similarly, when a host e.g. S_i, fails, only the hosts waiting for the reply from S_i are affected. Since S_i can recover within the specified time period for retrying, it can eventually receive the request after it reconnects to the network.□

Lemma 2*: *If a host S_i wants to enter CS, it eventually learns about all the hosts concurrently requesting CS.

Argument. For a host S_j in $Status\_set_i$ of host S_i, if S_j wants to enter CS, it will send a REQUEST message to S_i. S_i will receive this request even there are failures (Lemma 1). For a host S_k in $Info\_set_i$ of host S_i, S_i sends a REQUEST message to S_k. S_k will eventually receive the request (Lemma 1). If S_i receives the reply from S_k, it knows that S_k is not requesting CS. Otherwise, S_i is blocked until S_k sends a reply.□

Theorem 1*: *At most one host can be in the CS at any time (safety).

Argument. We prove the theorem by contradiction. Assume that two hosts S_i and S_j are executing the CS simultaneously. From Lemma 2, each of them has learned the status of the other, which implies that they had sent reply to each other before they entered the CS. However, this is impossible because no two hosts have the same priority. This is a contradiction.□

Theorem 2*: *The algorithm is deadlock free (liveness).

Argument. A deadlock occurs when there is a circular wait and there is no REPLY in transit. This means that each host in the cycle is waiting for a REPLY from its successor host in the cycle. Since each request has a distinct priority, there is a host, e.g. S_h whose priority is the highest. Assume that the successor of S_h as S_j. We claim that S_h eventually receives one REPLY from host S_j. In the cases with no failure or disconnection, the safety property can be proved in a way similar to that in [146]. Here we only consider the cases with failures or disconnections.

Case 1: S_j fails. If S_j failed before it sent out reply to S_h, it will send the reply after it recovers (Lemma 1). Eventually S_h can receive the reply from S_j.

Case 2: S_j runs normally. In this case, S_j would receive the request from S_h and handle it. This can be further divided into two cases: 1) S_j has no request for CS or

its request has a lower priority (because S_h has highest priority). Then S_j would send reply to S_h immediately. 2) S_j is in the CS. Q_{req} is set for S_h, and S_h should be moved to $Info\_set_j$ if it is in $Status\_set_j$ before. After S_j exits CS, it must send a reply to S_h.

In all cases, the circular wait is broken eventually. The theorem holds.□

Theorem 3*: *The algorithm is starvation free (fairness).

Argument. In the cases with no failure or disconnection, the safety property can be proved in a way similar to that in [146]. Therefore we only need to consider the situations with failures and disconnections. If there are link failures or host failures, the effect would be eventually eliminated by the timeout mechanism (Lemma 1). If there are disconnected hosts, all their current requests would be deleted after they reconnection or wake up. So, failures and disconnections do not affect the fairness of the algorithm. The fairness is guaranteed.□

7.6. Performance Evaluation

7.6.1. Analytic Evaluation

In this section, we analyze the performance of the proposed algorithm. As discussed in Chapter 3, the performance is computed under two special load levels: *low load level* and *high load level*. Under low load levels, there is seldom more than one request for CS simultaneously in the system; while under high load levels, there is always a pending request for CS at a host. We use three commonly used measures [145] in the analysis:

Number of Messages Per CS Entry (MPCS): the average number of messages exchanged among the hosts for each execution of the CS.

Synchronization Delay (SD): the number of sequential messages exchanged after a host leaves the CS and before the next host enters the CS.

Response Time (RT): the time interval that a host waits to enter the CS after its request for CS arrives.

Since it is hard to quantitatively study the performance under failures, in the following analysis, we consider only the normal executions without failure or

disconnection. The performance of the proposed algorithm under failures is evaluated using simulations and the results are reported in the next section.

7.6.1.1. Number of Messages per CS Entry

The MPCS of the proposed algorithm is in fact determined by the average size of the *Info_set*. In general, all the hosts are equally active, so a host is put in an *Info_set* and a *Status_set* with same probability. Therefore, the average size of the *Info_set* is $n/2$. Under low load levels, there is usually only one or no request in the system at any moment of time. When a host requesting for the CS sends REQUEST messages to the hosts in its *Info_set*, those hosts send back REPLY messages immediately. Therefore, MPCS under low load levels is:

$$MPCS_{low}=2*n/2=n$$

Under high load levels, each host always has a pending request. For an arbitrary host S_i, on average, $n/2$ hosts will issue their current requests for CS earlier than S_i does. In general, half of these $n/2$ hosts are in the *Status_set* of S_i, and they will send REQUEST messages to S_i. On the reception of REQUEST messages from these $n/4$ hosts, S_i sends REQUEST messages back to the senders (S_i has the lower priority). For each REQUEST message, one REPLY message must be sent. Therefore, the average number of messages for one CS entry at a host under high load levels is:

$$MPCS_{hig}=2*(n/2+n/4)=3*n/2$$

In the above analysis, the load at each host is assumed to be the same. However, as discussed in [146], an interesting feature of *Info_set* is that its average size is affected by the activeness of the hosts, i.e. the *Info_set* of the host requesting for CS is smaller if the arrival of CS requests is localized at few hosts. In such conditions, the MPCS under low load levels and high load levels are $|\Phi|$ and $3*|\Phi|/2$ respectively, where Φ is the set of active hosts. This results in a substantial reduction in message cost. The effect of this feature is validated in the simulations.

It is important to notice that, the condition that “the arrival of CS requests is localized at a few hosts” does not mean that only these few hosts have requests for

CS during the execution. The set Φ may vary from time to time. As long as it can keep stable for a "quite" long time, the proposed algorithm can benefit.

7.6.1.2. Synchronization Delay

The synchronization delay is meaningless under low load levels, because it measures the interval between the arrivals of two requests. Under high load levels, when a host S_i exits the CS, it will send REPLY messages to all the hosts in its *Info_set*, i.e. the hosts that have pending requests. Then, the host issuing request at the earliest time will enter the CS immediately after it receives the REPLY from S_i. Therefore, the synchronization delay under high load levels is:

SD_{hig}= 1, i.e. the time of transferring one message.

7.6.1.3. Response Time

Under low load levels, most of the time, no more than one host competes for the CS. When a host wants to enter CS, it sends REQUEST messages to the hosts in its *Info_set* and then all these hosts send REPLY immediately after they receive the REQUEST. Therefore the response time under the low load levels is:

RT_{low} = 2, i.e. twice of the time of transferring one message.

Under high load levels, there is always a pending request at each host. The hosts are in the waiting chain with respect to the timestamps of their requests, i.e. the time when they issue requests for CS. A host in the chain can enter CS after its predecessor exits, so each host needs to wait for the hosts whose requests are earlier. On average, each host has to wait for $n/2$ such hosts. Assuming the average time of an execution of CS is E, the response time under high load levels is:

RT_{hig}=(E+ SD_{hig})*n/2= (E+1)*n/2

7.6.2. Simulation Study

Simulations have been carried out to evaluate the performance of the proposed algorithm. In the simulation, we adopted Glomosim [156] as the platform which has been widely used for simulating algorithm in MANETs. The proposed algorithm is implemented as an application level protocol.

7.6.2.1. Simulation Parameters and Setup

In the simulations, we set the parameters of the MANET with the same values as those used in [17]. Since the network partition problem is not considered, we adopted such a territory scale that can minimize the probability of network partitions while maximizing the number of hops for each application message [17].

All the hosts are scattered into a rectangular territory. To evaluate the scalability of the algorithm, we varied the number of hosts, and accordingly the territory scale in proportion, so that the performances under different numbers of hosts are comparable. The number of hosts with corresponding territory scale and other main parameters are shown in Table 7-1.

Table 7-1 Simulation settings for the MUTEX algorithm

Number of Hosts	4, 8, 12, 16, 20
Territory Scale	313m, 443m,543m,626m, 700m
Average speed of movement	20m/sec
Mobility model	Random-waypoint
Transmission radius	200m
Routing-protocol	BELLMANFORD
Link bandwidth	2M bits/Sec
Simulation Time	300 Hours

We used UDP as the transport layer protocol at first, but the high percentage of packet loss (more than 50%) always made the simulation blocked. Therefore we finally used TCP. However, even using TCP, there are still some packets lost (near 2%), which may be caused by the movements of the hosts.

The arrival of the requests at a host is assumed to satisfy a Poisson distribution with mean λ, which represents the number of requests generated by a single host per second. Simulations were carried out under three different load levels, i.e. high (λ=1.00E-2), middle (λ=1.00E-3) and low (λ=1.00E-4).

The simulations can be divided into three parts. First, there are only link failures in the network. Just as mentioned above, packet loss is already there, so we did not simulate link failures by ourselves in this part. Second, we introduced host failures. The arrival of host failures at a host is also assumed to satisfy a Poisson distribution and the duration of host failures satisfies the exponential distribution. To simplify the simulation, we fixed the percentage of host failure to 10%, a value that is quite high.

In the two parts described above, the load levels of all the hosts are the same, i.e. the arrival rates of requests at the hosts are uniform. However, as discussed in Section 7.6.1, one feature of the algorithm is that the performance is better if the arrival of CS requests is localized at few hosts. This feature makes the algorithm scalable to large system. Therefore, we also conducted simulations under the condition that different hosts have different load levels.

All simulation were carried out under three different mobility levels set by adjusting the pause time so that the time a host does move accounts for 100%, 50% and 10% of the total simulation time respectively.

7.6.2.2. Simulation Results and Discussion

In the simulations, we measured the message cost using two metrics. Besides the MPCS introduced before, the number of hops per CS entry (HPCS) is also adopted. In this thesis, a "message" means an application layer message, i.e. the end-to-end message; while a "hop" means a network layer message, i.e. the point-to-point message. Obviously, the latter can reflect the message cost of an algorithm more precisely. Because of the resource constraints, HPCS is important for MANETs. As to the cost in time, we measured the RT of our algorithm. However, the following discussions focus on the MPCS and HPCS, which is at the core of the paper. The simulation results are described and discussed according to the main factors that affect the performance.

To understand the simulation results well, we first need to find out the relationship between MPCS and HPCS. The difference between the two metrics depends on the number of hops per message, which is affected by the topology of a MANET. In a MANET, the topology is dynamic due to the movements and failures of hosts. Therefore, the number of hops per message is significantly affected by the mobility of the hosts, which has been validated in [17]. Figure 7-4 depicts the average number of hops needed for each application level message with 20 hosts in our simulation environment. When the mobility increases, the number of hops decreases. In fact, the number of hops is determined by the distance between the

source and destination host. In a MANET, the distance between any two hosts is changed from time to time. However, the higher the mobility is, the higher the probability for the distance between any two hosts to be short. Therefore, under high mobility, the number of hops is small. Of course, the limitation of the number of hops is one.

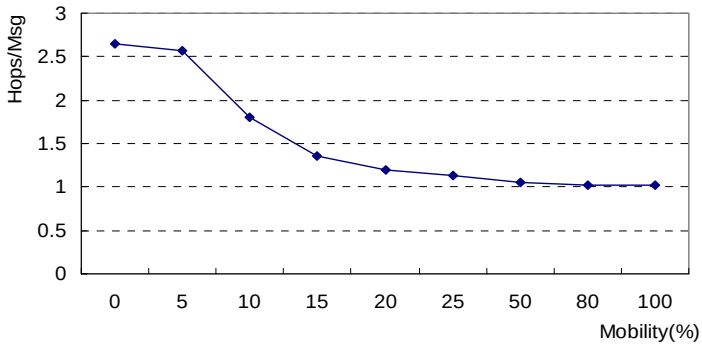

Figure 7-4 No. of hops per application message

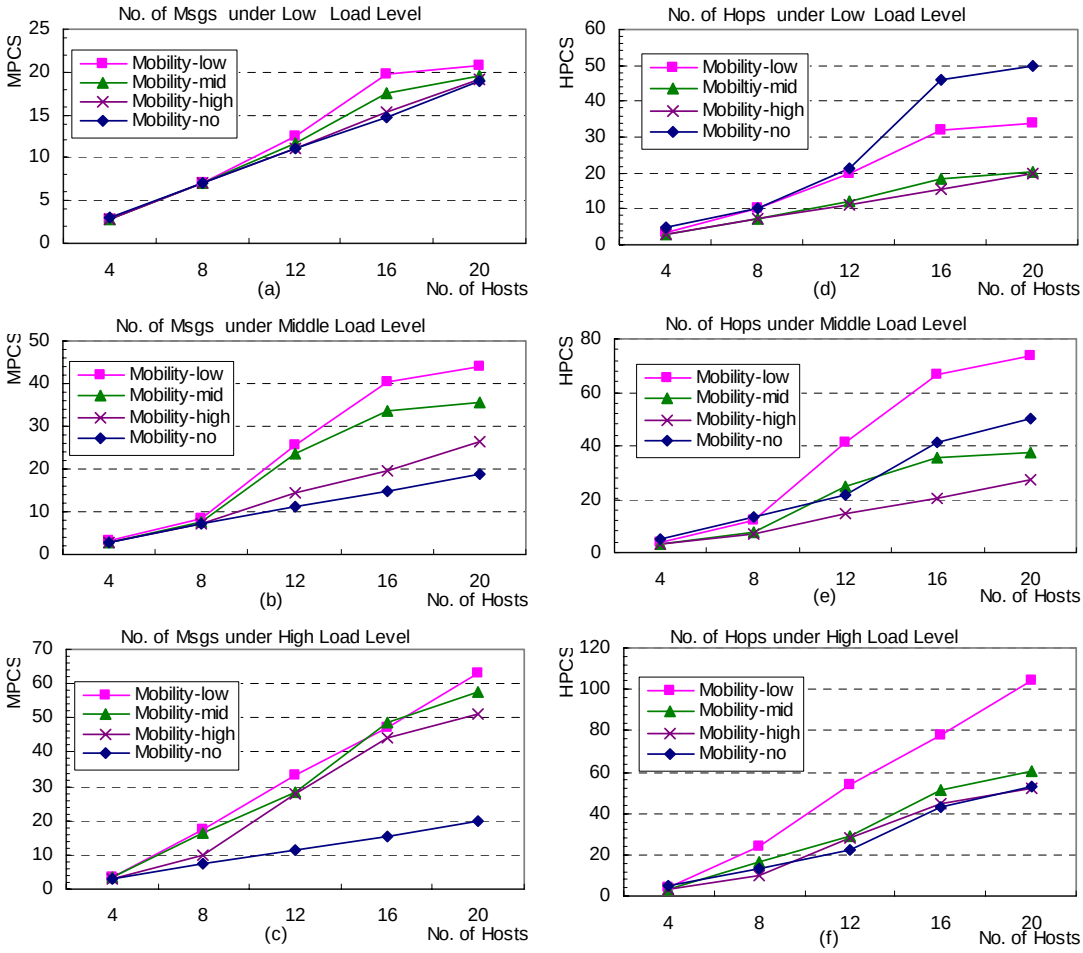

Figure 7-5 MPCS/HPCS vs. No. of hosts - effect of mobility

1) Effect of System Scale

Both Figure 7-5 and Figure 7-7 show that MPCS/HPCS increases linearly while the number of hosts increases. Using the "look ahead" technique, a host only needs to send requests to those hosts in the *Info_set* or competing for the CS concurrently. Since all the hosts are equally active, the average size of the *Info_set* is in proportion with the system scale, as analyzed in previous section. Therefore, the message cost increases nearly linearly when the system scale increases, and the algorithm is scalable to the system scale. When the activeness of hosts is not uniform, the scalability of our algorithm is much better. See "5) Effect of uniformity of load level" for more discussions.

The effect of system scale on RT is shown in Figure 7-6 and Figure 7-8. Same as HPCS/MPCS, the response time increases with the increase of the system scale. A large number of hosts in the system lead to more messages exchanged and higher competitions for CS, so a host needs to wait longer before it can enter the CS.

2) Effect of Mobility

Figure 7-5 and Figure 7-6 show the effect of mobility on MPCS/HPCS and RT respectively. Under different load levels, the MPCS without host movement is always the best. This is easy to understand. If the hosts do not move, the TCP connections can be established easily and keep stable. So, all the requests are handled quickly and no message re-sending is needed. When hosts move during the execution, it becomes difficult to establish and maintain a connection due to the package loss caused by host movements. As a result, the number of messages increases. However, MPCS under low mobility is higher than that under high mobility. This can be explained using the effect of mobility on the distance between two hosts. Just as discussed before, the higher the mobility is, the shorter the average distance between any two hosts is and the higher probability for the connection to be established. Therefore, the response time would be shorter (as shown in Fig 6) and fewer hosts compete for the CS concurrently, leading to less request messages.

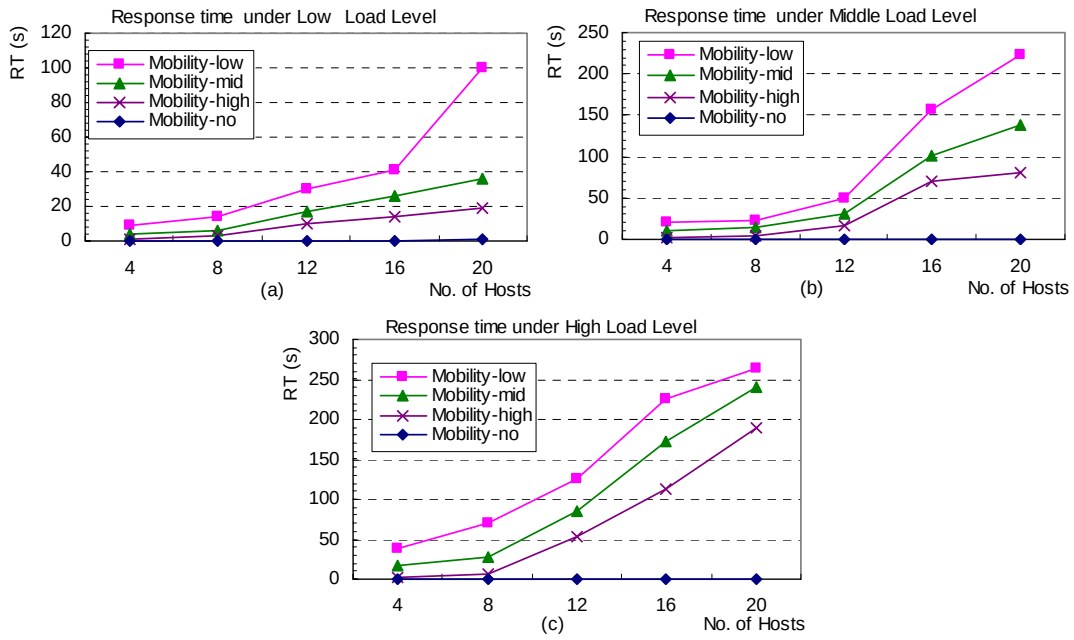

Figure 7-6 RT vs. No. of hosts - effect of mobility

The effect of mobility on the HPCS, shown in Figure 7-5-(d), (e) and (f), is similar to that on the MPCS, except that the performance under no movement is no longer the best. This is because that, besides its effect on the establishment of a connection, the mobility level also affects the average number of hops per message as discussed in Figure 7-4. If there is no movement, the number of hops per message becomes large, which makes the performance bad.

As shown in Figure 7-6, the mobility affects the RT in a very similar way as it does on MPCS. More messages mean longer delay and more competitions, so RT varies with the same trend of MPCS. It is important to notice that when the MHs do not move, the RT is much less than that with movements. This is because that the mobility of MHs affects not only the establishment of connections among hosts but also the transmission of a message.

3) Effect of Load Level

Figure 7-7 shows the MPCS against the load level. In general, MPCS increases with the increase of the load level. From the curves in Figure 7-7 we can see, under low load levels, a host needs to send request messages to nearly half of all hosts. Under high load levels however, there are pending requests at any time, so a host

that wants to enter CS needs to send request messages to not only all the hosts in its *Info_set*, but also those requesting for the CS. Therefore, the MPCS increases.

Figure 7-8 shows the effect of the load level on RT, similar to the effect on MPCS. The higher the load level is, the higher the RT.

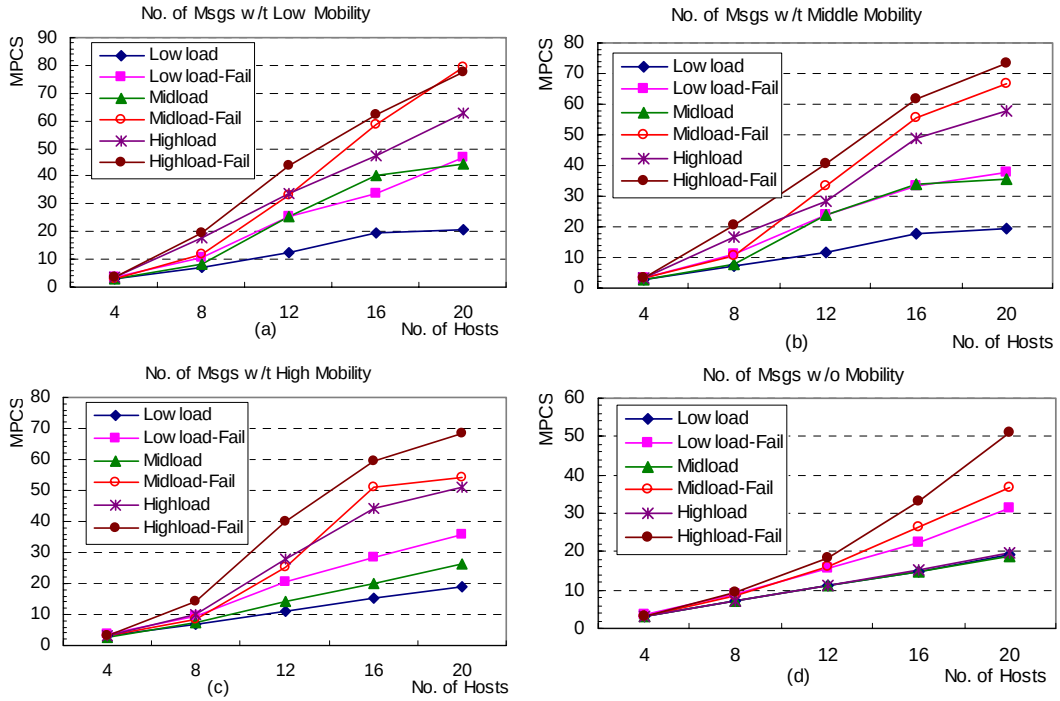

Figure 7-7 MPCS vs. No. of hosts - effect of load level and host failures

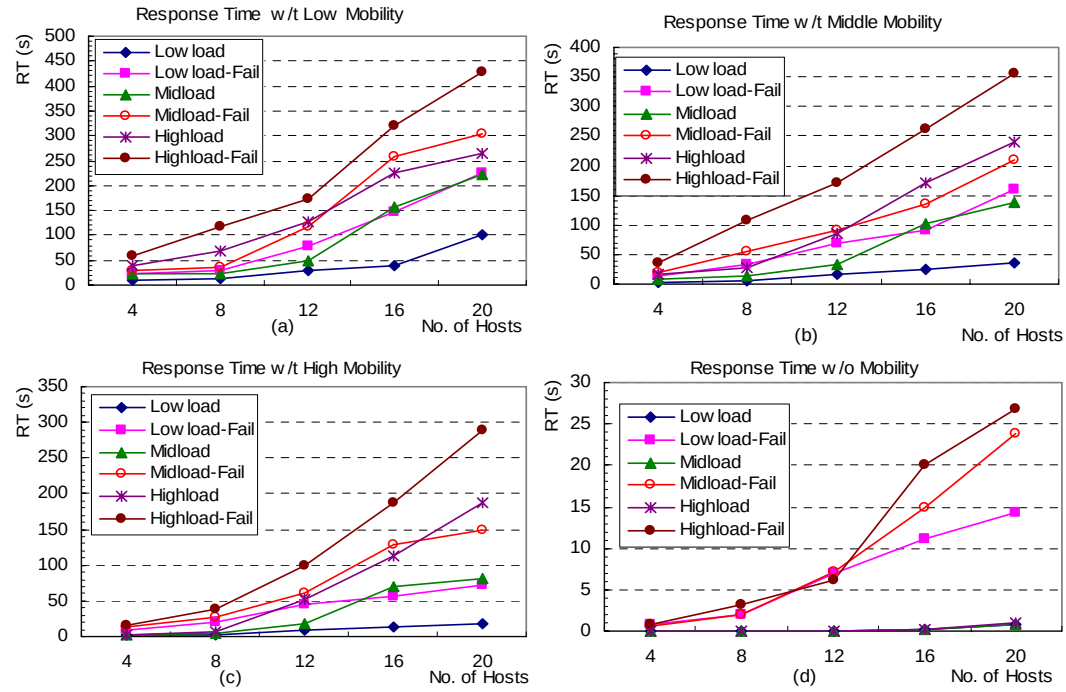

Figure 7-8 RT vs. No. of hosts - effect of load level and host failures

4) Effect of Host Failure

From Figure 7-7 and Figure 7-8, we can see that more time and messages are needed when there are host failures. The curves with host failures are named with a suffix "-Fail". Under different load levels and system scales, the increase in MPCS caused by host failures varies strongly. In some cases, the number of messages is doubled, but for most cases, about 40% more messages are needed. Considering the high failure rate (10%), this is acceptable.

5) Effect of Uniformity of Load Level

As discussed in Section 7.6.1, the performance of the propose protocol is affected by the activeness of the hosts. To evaluate this feature, we let some hosts generate requests more actively than others. In this part of simulations, we fix the number of hosts to 20 and mobility to 50%. The requests for CS of four hosts out of all the 20 hosts constitute 80% of all requests, while the rest of the hosts generate only 20% requests. Under a non-uniform load level, some hosts have a higher load level than the others, but the total load level of all the hosts is the same as that under a uniform load level.

To measure the performance precisely, more load levels were examined. Figure 7-9 shows the HPCS against the load level. Obviously, if some hosts are more active than others, the message cost is significantly reduced, which agrees to the analysis in Section 7.6.1. This feature is especially important to the scalability, because it makes the increase of message cost slower than the increase of system scale.

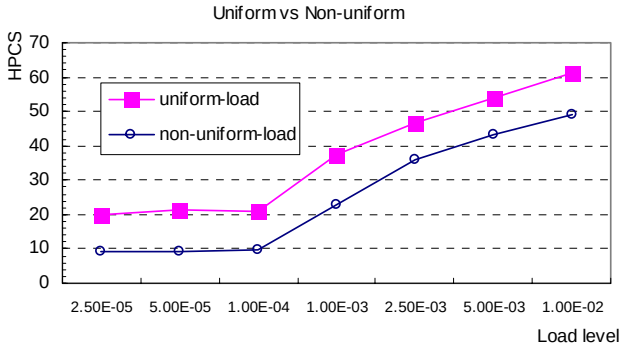

Figure 7-9 HPCS w/t non-uniform load level

7.7. Making the Algorithm More Robust

In the algorithm described in Section 7.4, permanent link or host failures are not considered. In an asynchronous system, there is no solution to precisely detect such failures. However, the algorithm can be extended to handle permanent failures, with the assumption that such failures can be suspected using some approach, e.g. a timeout based approach. Once a permanent failure or network partition is perceived to occur, we enhance our algorithm in the following way to handle it.

7.7.1. Permanent Host Failures

First, let us consider the permanent failures during the initialization. As described in Section 7.4, a timeout is set for the initialization message from the initiator. To handle the permanent failure of the initiator, a host S_i sends query messages to all other hosts rather than only the initiator, when the timeout expires. On the reception of the query, the receivers send back replies with the possibly received M_u. If no host receives the initialization M_u, a new initiator can be selected using some predefined order, e.g. *ID_new_initiator* = (*ID_old_initiator* +1) mod *n*. Then the initialization is tried again.

Now, we discuss how to handle permanent failures in the mutual exclusion. To handle permanent host failures, a straightforward method is to set a threshold for resending a REQUEST message. When the number of the times of resending a REQUEST reaches the threshold, the destination host is perceived to be crashed and it is moved to the *Status_set*. However, the accuracy of detecting permanent host failures using such a method is significantly affected by the load levels. To avoid this, another method is to detect such failures using a separate heartbeat like mechanism, which is commonly used in distributed systems. After the crash of a host is detected, the host is move to *Status_set*.

7.7.2. Network Partition

The execution of a host may be blocked when the network is partitioned (because of link failures, host failures, or host movements) until the partitions are merged. If

the partitions keep unconnected for very long time, the delay caused may be intolerable. Whether such a long time delay can be avoided depends on the property of the CS. If the CS cannot be accessed by two or more hosts at any moment of time, even if they are in unconnected partitions, no action can be taken.

Otherwise, if the hosts in different partitions can access the CS simultaneously, it is possible to reduce the wait time caused by partitions. In fact, the main difficulty to handle the network partitioning is how to detect it, which is out of scope of this paper. Here, we assume there is such a partition detection module available. With such a module, handling the partitioning is simple. Once a partition is detected, a host moves all the hosts in other partitions to its *Status_set*. However, it is more complex to handle the merging of partitions. The relationship between the hosts in different partitions must be considered. For a pair of hosts in two different partitions, it is possible that both hosts put each other in the *Status_set*, which violates the conditions specified in Section 7.3, when the partitions are merged. To deal with this, at least one host has to change its *Status_set*. A simple but efficient method is that the host with the small ID moves the other host from *Status_set* to *Info_set*.

7.8. Summary

In this chapter, we propose an efficient and reliable permission-based MUTEX algorithm for MANETs. This algorithm does not depend on any logical topology so as to eliminate the cost of maintaining such a topology. To reduce the number of message exchanged, the “look ahead” technique is used. We designed a fault tolerance mechanism using timeout to tolerate intermittent and recoverable link/host failures, which very frequently occur in MANETs. The algorithm can also handle dozes and disconnections of hosts. The simulation results show that the algorithm performs better under low load levels and high mobility level. One important feature of the algorithm is the scalability to large system scale, especially when some hosts are more active than the others.

Chapter 8. Conclusions and Future Directions

8.1. Conclusions

In this thesis, we first investigated the characteristics of mobile networks, in aspects of communication, mobility and resource constraints, and identified the new challenges caused by these characteristics in the design of distributed algorithms. Based on the investigations, we study how to design distributed coordination algorithms in mobile wireless environments. We focus on solutions to two distributed coordination problems: *consensus* and *mutual exclusion*.

Our work on consensus consists of three parts. The first part is concerned with how to increase the execution speed of consensus protocols. Usually, the execution of a consensus protocol is slowed down by host failures and mistakes made by the underlying oracle, e.g. the failure detector or leader oracle. To avoid such slowdowns, we developed the Look-Ahead technique, which can speed up the execution of consensus protocols by making use of “future” messages. Due to the asynchrony of the system, some messages may be delivered to their destination hosts that have not entered the corresponding phase or round. On the reception of such future messages, the receiver host can extract the knowledge about the future of its execution and then adapt its state to the future situation so as to reduce the waiting time for some slow messages. Look-Ahead is a general technique that can be easily applied to existing consensus protocols.

The second part is concerned with improving the message efficiency of consensus protocols in MANETs. A two-layer hierarchy is imposed on the MHs by grouping MHs into clusters. The clusterheads help merge/unmerge the messages with destinations in the same cluster so as to reduce the message cost and improve the scalability. Using different clustering approaches, we developed two hierarchical consensus protocols. In the first protocol, the set of clusterheads is defined in advance and the MHs find and associate themselves with the clusterheads. This

protocol requires a failure detector $\Diamond P$, which is stronger than the commonly used $\Diamond S$. Moreover, the functions of achieving consensus and clustering hosts are interlaced, which makes the protocol complicated. To address these problems, we designed another hierarchical protocol, in which the function of clustering hosts is undertaken by a separate module named clusterer oracle Δ. Δ is equivalent to $\Diamond S$ in the power of tolerate failures in solving consensus but it is more powerful in the sense that it can help the consensus protocols built on top of it improve their message efficiency and scalability, which is especially important for large scale MANETs.

The last part of our research on consensus is about achieving consensus in dynamic mobile systems, where the number of participating processes can change arbitrarily as time passes and processes can join or leave the system at any time. We proposed an eventual leader protocol for dynamic infrastructured mobile networks. The network of MSSs is a static system, while the MHs constitute a dynamic system. By exchanging queries and responses among the MSSs using a time free approach, a correct MH is eventually elected as the leader.

For the mutual exclusion problem, we developed a permission-based algorithm, which adopts the "look ahead" technique (proposed by Singhal et al. [146]) to achieve message efficiency. A host needs to get permissions from a subset instead of all MHs before it can access the CS. Timeout-based fault tolerance mechanisms are designed to tolerate both link and host failures.

Extensive evaluations, including analysis and simulations, have been conducted to examine the performance of our proposed algorithms. The results show that our objectives are well fulfilled.

8.2. Future Directions

Our research in this thesis mainly focuses on the efficiency and fault tolerance in MANETs. It remains as our future work to improve the proposed algorithms and to investigate related research directions.

One issue that deserves further study is to improve the implementation of the clusterer oracle Δ. In our current implementation, the set of clusterhead is too large at the beginning (all the hosts are included); while it may be too small at the end (there may be only one host in the set). Such extreme sizes significantly affect the efficiency of the two-layer hierarchy. For the former case, the size can be simply reduced using a uniform initial value, e.g. half the number of MHs. The latter case, however, it is more difficult to handle. The underlying reason for this problem is the weak accuracy of $\Diamond S$. One possible solution is to replace $\Diamond S$ with a stronger FD, $\Diamond P$, but this violates one of the objectives to define Δ: clustering MHs with the weakest FD of $\Diamond S$.

Another interesting topic is combining the time efficiency approach and message efficiency approach so as to save both time cost and message cost in consensus simultaneously. In this thesis, the time efficiency and message efficiency are studied separately. However, combining the approaches for these two aspects is not trivial. There is some connection between our proposed techniques/protocols for time efficiency and message efficiency. In the hierarchical protocols, to handle the message losses caused by the dynamics of the two-layer hierarchy, “future” messages are also involved. Therefore, how to integrate the mechanisms the use of “future” messages in Look-Ahead technique and hierarchical protocols should be investigated carefully.

The dynamic system is one promising research direction in distributed computing. In this thesis, an eventual leader protocol for dynamic infrastructured mobile networks has been proposed, but many other topics, such as how to elect an eventual leader in dynamic MANETs and how to achieve consensus based on these eventual leader oracles, still need further study.

In infrastructured networks, the system is modeled as two sub-systems: the static system of MSSs and the dynamic system of MHs. With such a model, existing eventual leader protocols for wired dynamic system can be adapted to the sub-system of MSSs, which will elect the eventual leader on behalf of MHs. In MANETs,

however, there is no MSS and all the work must be done by MHs themselves. Intuitively, in a dynamic MANET, the implementation of eventual leader still needs some “stable” part as the sub-system of MSSs in an infrastructured network. Therefore, how to define and make use of such a stable sub-system is at the core of an eventual leader protocol in dynamic MANETs.

Besides the implementation of oracles, the design of consensus protocols in dynamic systems is also worth investigation. In infrastructured networks, the principle is still letting MSSs do as much work as possible. With the help of MSSs, which can be modeled as a static system, achieving consensus should be much easier than in MANETs. Similar as the implementation of the eventual leader in dynamic MANETs, how to define and make use of a stable sub-system is the key issue.

Finally, we would like to investigate whether it is possible to achieve consensus with certain specified probability. In this thesis, only deterministic consensus is considered. However, due to the characteristics of mobile wireless networks, probabilistic consensus may be more suitable and efficient for some applications in MANETs. Although some efforts have been made to develop randomized consensus protocols using random number generator, how to guarantee a specified probability of achieving consensus is a challenging task due to the asynchrony of the mobile system.

References

[1] B. J. Abbari, E. Dinan, W. and Fuhrmann, Performance Analysis of a Multilink Packet Access for Next Generation Wireless Cellular Systems, *Proc. of the 9th IEEE Int'l Symp. on Personal, Indoor, and Mobile Radio Communications (PIMRC'98)*, pp. 131-135, 1998.

[2] A. Agbaria, H. Attiya, R. Friedman, and R. Vitenberg, Quantifying Rollback Propagation in Distributed Checkpointing, *Proc. of the 20th Symposium on Reliable Distributed Systems (SRDS'01)*, pp. 36-45, 2001.

[3] A. Agbaria, and W. H. Sanders, Distributed Snapshots for Mobile Computing Systems, *Proc. of the 2nd Int'l Conference on Pervasive Computing and Communications (PerCom04)*, pp. 177-186, 2004.

[4] M.K. Aguilera, A Pleasant Stroll Through the Land of Infinitely Many Creatures, *ACM SIGACT News, Distributed Computing Column*, vol. 35 no. 2, pp. 36-59, 2004.

[5] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, Communication-Efficient Leader Election and Consensus with Limited Link Synchrony, *Proc. of the 23rd ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 328-337, 2004.

[6] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, On Implementing Omega with Weak Reliability and Synchrony Assumptions, *Proc. of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'03)*, pp. 306-314, 2003.

[7] M.K. Aguilera, and S. Toueg, Failure Detection and Randomization: A Hybrid Approach to Solve Consensus, *SIAM J. Computing*, vol. 28 no. 3, pp. 890-903, 1998.

[8] S. A. Ahson, and I. Mahgoub, Research Issues in Mobile Computing, *Proc. of IEEE Int'l Performance, Computing, and Communications Conference (IPCCC'98)*, pp. 209-215, 1998.

[9] C. Almeida, and P. Verissimo, An Adaptive Real-Time Group Communication Protocol, *Proc. of the 1st IEEE Workshop on Factory Communication Systems*, pp. 63-71, 1995.

[10] C. Almeida, and P. Verissimo, The Quasi-Synchronous Approach to Distributed Real-Time Databases, *INESC tech rep RT/02-96*, 1996.

[11] C. Almeida, and P. Verissimo, Timing Failure Detection and Real-time Group Communication in Quasi-Synchronous Systems, *Proc. of the 8th Euromicro Workshop on Real-Time Systems*, pp. 230-235, 1996.

[12] C. Almeida, and P. Verissimo, Using Light-Weight Groups to Handle Timing Failures in Quasi-Synchronous systems, *Proc. of the 19th IEEE Real-Time System Symposium*, pp.

430-439, 1998.

[13] D. Angluin, Local and Global Properties in Networks of Processes. *Proc. 12th ACM Symp. on Theory of Computing (STOC'80)*, ACM Press, pp. 82-93, 1980.

[14] H. Attiya, and J. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, *McGraw-Hill*, 1998.

[15] N. Badache, M. Hurfin. and R. Macedo, Solving the Consensus Problem in a Mobile Environment, *Proc. of the 18th IEEE Int'l Performance Computing and Communications Conference (IPCCC'99)*, pp. 29-35, 1999

[16] B. Badrinath, A. Acharya, and T. Imielinski, Designing Distributed Algorithms for Mobile Computing Networks, *Computer Communications*, vol. 19 no. 4, pp. 309-320, 1996.

[17] R. Baldoni, A. Virgillito, and R. Petrassi, A Distributed Mutual Exclusion Algorithm for Mobile Ad-Hoc Networks, *Proc. of the* 7th *IEEE Symp. on Computers and Communications (ISCC'02)*, pp. 539-544, 2002.

[18] S. Banerjee, and P. K. Chrysanthis, A New Token Passing Distributed Mutual Exclusion Algorithm, *Proc. of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, pp. 717 -724, 1996.

[19] A. Basu, B. Charron-Bost, and S. Toueg, Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes, *Proc.of the 10th International Workshop on Distributed Algorithms (WDAG'96)*, LNCS 1151, pp. 105-122, 1996.

[20] P. Bellavista, A. Corradi, and C. Stefanelli, A Mobile Agent Infrastructure for Terminal, User, and Resource Mobility, *Proc. of the 7th IEEE/IFIP Network Operations and Management Symposium (NOMS'00)*, pp. 877-890, 2000.

[21] M. Benchaïba, A. Bouabdallah, N. Badache, and M. Ahmed-Nacer, Distributed Mutual Exclusion Algorithms in Mobile Ad Hoc Networks: an Overview, *ACM SIGOPS Operating Systems Review,* vol. 38 no.1, pp. 74-89, 2004.

[22] M. Ben-Or, Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols, *Proc. of the 2nd ACM Symposium on Principles of Distributed Computing (PODC '83)*, pp. 27-30, 1983.

[23] M. Bertier, O. Marin, and P. Sens, Implementation and Performance Evaluation of an Adaptable Failure Detector, *Proc. of International Conference on Dependable Systems and Networks (DSN '02)*, pp. 354-363, 2002.

[24] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui, Deconstructing Paxos, *SIGACT News, Distributed Computing Column*, vol.34 no.1, pp. 47-67, 2003.

[25] G. Bracha, and S. Toueg, A Distributed Algorithm for Generalized Deadlock Detection, *Distributed Computing*, vol. 2, pp. 127-138, 1987.

[26] T. Camp, J. Boleng, and V. Davies, A Survey of Mobility Models for Ad Hoc Network

Research, *Wireless Communications & Mobile Computing (WCMC)*, vol. 2 no. 5, 2002

[27] G. Cao and M. Singhal, A Delay-Optimal Quorum-based Mutual Exclusion Algorithm for Distributed Systems, *IEEE Trans. on Parallel and Distributed Systems*, vol. 12 no. 12, pp. 1256-1268, 2001

[28] O. S. F. Carvalho and G. Roucairol, On Mutual Exclusion in Computer Networks, technical correspondence, *Communications of the ACM*, vol. 26 no. 2, pp. 146-149, 1983.

[29] T. L. Casavant, and M. Singhal, Readings in Distributed Computing Systems, *IEEE computer society press*, 1994.

[30] T. Chandra, V. Hadzilacos, and S. Toueg, The Weakest Failure Detector for Solving Consensus, *Journal of the ACM*, vol. 43 no. 4, pp. 685-722, 1996.

[31] T. Chandra, and S. Toueg, Unreliable Failure Detectors for reliable distributed Systems, *Journal of the ACM*, vol.43 no. 2, pp. 225-267, 1996.

[32] K. M. Chandy, and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, vol. 3 no.1, pp. 63-75, 1985.

[33] K.M. Chandy, and J. Misra, How Processes Learn. *Distributed Computing*, vol. 1 no. 1, pp. 40-52, 1986.

[34] Y. Chang, M. Singhal, and M. Liu, A Dynamic Token-based Distributed Mutual Exclusion Algorithm, *Proc. of the 10th Annual International Phoenix Conference on of Computers and Communications*, pp. 240-246, 1991

[35] Y. Chang, M. Singhal, and M. Liu, A Fault Tolerant Algorithm for Distributed Mutual Exclusion, *Proc. of the 9th IEEE Symp. on Reliable Dist. Systems (SRDS'90)*, pp. 146-154, 1990.

[36] Y. Chen and J. Welch, Self-stabilizing Mutual Exclusion Using Tokens in Mobile Ad Hoc Networks, *Proc. of the* 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (*Dial-M'02)*, pp. 34-42, 2002.

[37] G. Chockler, M. Demirbas, S. Gilbert, C. C. Newport, and T. Nolte, Consensus and Collision Detectors in Wireless Ad Hoc Networks, *Proc. of the 24th ACM Symp. on Principles of Distributed Computing* (*PODC'05)*, pp. 197-206, 2005.

[38] G. Chockler and D. Malkhi, Active Disk Paxos with Infinitely Many Processes, *Proc. of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, pp. 78-87, 2002.

[39] F. Chu , Reducing Ω to ◊*W*, *Information Processing Letters*, vol. 76 no. 6, pp.293-298, 1998.

[40] L. Cininiera, P. Maggi, and R. Sisto, SCARAB: Innovative Services Supporting User and Terminal Mobility, *Proc. of Int'l Conf. on Distributed Computing Systems*

Workshop (ICDCSW), pp. 487-493, 2001.

[41] F. Cristian, Understanding Fault-tolerant Distributed System, *Communications of the ACM*, vol.34 no. 22, pp. 56-78, 1991.

[42] F. Cristian, and C. Fetzer, The Timed Asynchronous Distributed System Model, *IEEE Transactions on Parallel and Distributed Systems*, vol.10 no. 6, pp.642-657, 1999

[43] A. Coccoli, A. Bondavalli, and L. Simoncini, Consensus in Asynchronous Distributed Systems, *Proc. of the 5th Int. Conf. on Integrated Design and Process Technology (IDPT'00)*,. 2000.

[44] G. Coulouris, J. Dollimore, and T. Kindberg, Distributed Systems: Concepts and Design (3rd edition), *Addison-Wesley*, 2001.

[45] X. Defago, and A. Schiper, Semi-passive replication and Lazy Consensus, *Journal on Parallel and Distributed Computing*, vol. 64 no. 12, pp. 1380-1398, 2004.

[46] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos Algorithm, *Theoretical Computer Science*, vol. 243 no. 1-2, pp. 35-91, 2000.

[47] D. Dhamdhere, and S. Kulkami, A Token based *k*-resilient Mutual Exclusion Algorithm for Distributed Systems, *Information Processing Letters*, vol. 50, pp. 151-157, 1994.

[48] E. Dijkstra and C. Scholten, Termination Detection for Diffusing Computations, *Information Processing Letters*, vol. 11 no. 1, pp. 1-4, 1980.

[49] D. Dolev, C. Dwork, and L. Stockmeyer, On the Minimal Synchronism Needed for Distributed Consensus, *Journal of the ACM*, vol. 34, no. 1, pp. 77-97, 1987.

[50] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi, Failure Detectors in Omission Failure Environments, *Technical Report: TR96-1608*, Department of Computer Science, Cornell University, Sep. 1996.

[51] P. Dutta, and R. Guerraoui, Fast Indulgent Consensus with Zero Degradation, *Proc. of the 4th European Dependable Computing Conference* (*EDCC'02*), France, *LNCS 2485*, pp. 191-208, 2002.

[52] P. Dutta, R. Guerraoui, and I. Keidar, The Overhead of Consensus Failure Recovery, *IC Technical Report 200456*, EPFL, Jun. 2004.

[53] C. Dwork, N. Lynch, and L. Stockmeyer, Consensus in the Presence of Partial Synchrony, *Journal of the ACM*, vol. 35 no. 2, pp. 288-323, 1988.

[54] H. Elaarag, Improving TCP Performance over Mobile Networks, *ACM Computing Surveys (CSUR)*, vol. 34 no. 3, pp. 357-374, 2002.

[55] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, Randomized Multivalued Consensus, *Proc. of the 4th IEEE Int'l Symp. on Object-Oriented Real-Time Computing*, pp. 195-200, 2001.

[56] J. Fernandes, and J. Garcia, Cellular Coverage for Efficient Transmission Performance in MBS, *Proc. of the 52nd IEEE Vehicular Technology Conference (VTC'00)*, pp. 2225-

2232, 2000.

[57] J. Fernandes , and J. Garcia, Multiple coverage for MBS environments, *Proc. of the 11th IEEE Int'l Symp. on Personal, Indoor, and Mobile Radio Communications (PIMRC'00)*, pp. 644-649, 2000.

[58] C. Fetzer, and F. Cristian, Fail-aware Failure Detectors, *Proc. of the 15th Symposium on Reliable Distributed Systems (SRDS'96)*, pp. 200-209, 1996.

[59] C. Fetzer, M. Raynal, and F. Tronel, An Adaptive Failure Detection Protocol, *Proc. of the 8th Pacific Rim International Symposium on Dependable Computing (PRDC'01)*, pp. 146-153, 2001

[60] M. Fischer, The Consensus Problem in Unreliable Distributed Systems (A Brief Survey), *Research Report YALE/DCS/RR-273, Yale Univ*., 1983.

[61] M. Fischer, N. Lynch, and M. Paterson, Impossibility of Distributed Consensus with One Faulty Process, *Journal of the ACM*, vol. 32 no. 2, pp. 374-382, 1985.

[62] P. Flocchini, B. Mans, and N. Santoro, Sense of Direction in Distributed Computing, *Proc. 12th Int'l Symp. on Distributed Computing (DISC'98)*, Springer-Verlag LNCS #1499, pp. 1-16, 1998.

[63] G. Forman, and J. Zahorjan, The Challenges of Mobile Computing, *IEEE Computer*, vol. 27 no. 4, pp. 38-47, 1994.

[64] R. Friedman, A. Mostefaoui, M. Raynal, On the Respective Power of $\Diamond P$ and $\Diamond S$ to Solve One-Shot Agreement Problems, *Technical Report of IRISA*, No. 1547, Jul. 2003.

[65] R. Friedman, M. Raynal, and C. Travers, Two Abstractions for Implementing Atomic Objects in Dynamic Systems, *Proc. of the 9th Int'l Conf. on Principles of Distributed Systems (OPODIS'05)*, pp. 73-87, 2005.

[66] R. Friedman, and G. Tcharny, Evaluating Failure Detection in Mobile Ad-Hoc Networks, *Tech Report #CS-2003-06*, Computer Science Departement, Technion (Israel), 2003.

[67] E. Gafni, and L. Lamport, Disk Paxos, *Proc. of the 14th Int'l Symposium on Distributed Computing (DISC'00)*, LNCS #1914, pp. 330-344, 2000.

[68] H. Garcia-Molian, Elections in a Distributed Computing System, *IEEE Transactions on Computers*, vol. C-31 no. 1, pp. 48-59, 1982.

[69] R. Guerraoui, Indulgent Algorithms, *Proc. of the 19th ACM Symp. on Principles of Distributed Computing, (PODC'00)*, ACM Press, pp. 289-298, 2000.

[70] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper, Consensus in Asynchronous Distributed Systems: A Concise Guided Tour, *Advances in Distributed Systems, LNCS 1752*, pp. 33-47, 2000.

[71] R. Guerraoui, R. Oliveira, and A. Schiper, Stubborn communication channels, *Technical report*, LSE, Ecole Polytechnique Federale de Lausanne, Switzerland, 1996.

[72] R. Guerraoui, and M. Raynal, The Alpha and Omega of Asynchronous Consensus, *Tech*

Report #1676, IRISA, University of Rennes 1 (France), 2005.

[73] R. Guerraoui, and M. Raynal, The Information Structure of Indulgent Consensus, *IEEE Transactions on Computers*, vol. 53 no. 4, pp. 453-466, 2004.

[74] R. Guerraoui, and A. Schiper, Consensus: the Big Misunderstanding, *Proc. of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems*, pp. 183-188, 1997.

[75] R. Guerraoui, and A. Schiper, The Generic Consensus Service, *IEEE Transactions on Software Engineering*, vol. 27 no. 1, pp. 29-41, 2001.

[76] I. Gupta, T. D. Chandra, and G. S. Goldszmidt, On Scalable and Efficient Distributed Failure Detectors, *Proc. of the 20th ACM Symp. on Principles of Distributed Computing (PODC '01)*, pp. 170-179, 2001.

[77] V. Hadzilacos, and S. Toueg, Fault-tolerant Broadcasts and Related Problems, *Distributed Systems, Addison-Wesley*, 1993.

[78] K. P. Hatzis, G. P. Pentaris, Paul G. Spirakis, Vasilis T. Tampakas, and Richard B. Tan, Fundamental Control Algorithms in Mobile Networks, *Proc. 11th Annual ACM Symp. on Parallel Algorithms and Architectures*, pp. 251-260, 1999.

[79] J. Helary, A. Mostefaoui, and M. Raynal, A General Scheme for Token- and Tree-Based Distributed Mutual Exclusion Algorithms, *IEEE Transactions on Parallel and Distributed Systems*, vol.5 no.11, pp.1185-1196, 1994.

[80] J. Helary, N. Plouzeau, and M. Raynal, A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network, *Computer Journal*, vol.31 no. 4, pp. 289-295, 1988.

[81] S.-T. Huang, Detecting Termination of Distributed Computations by External Agents, *Proc. of the 9th Int'l Conf. on Distributed Computing Systems* (*ICDCS'89)*, pp. 157-172, 1989.

[82] M. Hurfin, A. Mostefaoui, and M. Raynal, A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors, *IEEE Trans. on Computers*, vol.51 no. 4, pp. 395-408, 2002.

[83] M. Hurfin, and M. Raynal, A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector, *Distributed Computing*, vol. 12 no. 4, pp. 209-223, 1999.

[84] T. Imielinski, and B. R. Banrinath, Mobile Wireless Computing: Challenges in Data Management, *Communications of the ACM*, vol. 37 no. 10, pp. 18-28, 1994.

[85] T. Imielinski, and H. F. Korth, Mobile Computing, *Kluwer Acdemic Publishers*, 1996.

[86] J. Jiang, T. Lai, and N. Soundarajan On Distributed Dynamic Channel Allocation in Mobile Cellular Networks, IEEE Transactions on Parallel and Distributed Systems, vol. 13 no. 10, pp. 1024-1037, 2002

[87] D. Johnson, and D. Maltz, Dynamic Source Routing in Ad Hoc Wireless Networks, Mobile Computing, Chapter 5, *Kluwer Academic Publishers*, 1996.

[88] A. Kshemkalyani, M. Raynal, and M. Singhal, An Introduction to Snapshot Algorithms in Distributed Computing, *Journal of Distributed Systems Engineering*, vol. 2 no. 4, pp. 224-233, 1995.

[89] L. Lamport, Paxos Made Simple, *ACM SIGACT News*, *Distributed Computing Column*, vol. 32 no. 4, pp. 34-58, 2001.

[90] L. Lamport, The Part-time Parliament, *Technical Report 49, Systems Research Center, Digital Equipment Corp*, Palo Alto, Sep. 1989 (A revised version of the paper also appeared in *ACM Transaction on Computer Systems*, vol. 16 no. 2, May 1998).

[91] L. Lamport, Time, Clocks and Ordering of Events in Distributed Systems, *Communications of the ACM*, vol. 21 no. 7, pp. 558-565, 1978.

[92] M. Larrea, A. Fernandez, and S. Arevalo, Eventually Consistent Failure Detectors, *Proc. of ACM Symp. on Parallel Algorithms and Architectures*, pp. 326-327, 2001.

[93] M. Larrea, A. Fernandez, and S. Arevalo, On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems, *IEEE Trans. on Computers*, vol. 53 no. 7, pp. 815-828, 2004.

[94] W.C.Y. Lee, Overview of Cellular CDMA, *IEEE Transactions on Vehicular Technology*, vol. 40 no. 2, pp. 291-302, 1991.

[95] Q. Lu, and M. Satyanarayanan, Isolation-Only Transactions for Mobile Computing, *ACM Operating Systems Review*, vol. 28 no. 2, pp. 81-87, 1994.

[96] W.-S. Luk, and T.-T. Wong, Two New Quorum Based Algorithms for Distributed Mutual Exclusion, *Proc. of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, pp100-106, 1997.

[97] N. Lynch, Distributed Algorithms, *Morgan Kaufmann*, 1996.

[98] M. Maekawa, A √N Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Trans. on Computer Systems*, vol. 3 no. 3, pp. 145-159, 1985.

[99] S. K. Madria, and B. Bhargava, A Transaction Model for Mobile Computing, *Proc. of Database Engineering and Applications Symposium (IEDAS'98)*, pp. 92-102, 1998.

[100] D. Malkhi, F. Oprea, and L. Zhou, Meets Paxos: Leader Election and Stability without Eventual Timely Links, *Proc. of the 19th Int'l Symp. Distributed Computing (DISC'05)*, Springer-Verlag LNCS #3724, pp. 199-213, 2005.

[101] N. Malpani, N. Vaidya, and J. Welch, Distributed Token Circulation on Mobile Ad Hoc Networks, *Proc. of the 9th International Conference on Network Protocols (ICNP'01)*, pp. 4-13, 2001.

[102] N. Malpani, J.Welch, and N. Vaidya, Leader Election Algorithms for Mobile Ad Hoc Networks, *Proceedings of the 4th international workshop on Discrete Algorithms and*

Methods for Mobile Computing and Communications (Dial-M'00), pp. 96-103, 2000.

[103] A. Martin, Distributed Mutual Exclusion on a Ring of Processes, *Science of Computer Programming*, vol. 5, pp. 265-276, 1985.

[104] F. Mattern, Global Quiescence Detection Based on Credit Distribution and Recovery, *Information Processing Letters*, vol. 30, pp. 195-200, 1989.

[105] M. Merritt, and G. Taubenfeld, Computing Using Infinitely Many Processes, *Proc. of the 14th Int'l Symp. on Distributed Computing (DISC'00)*, Springer-Verlag LNCS #1914, pp. 164-178, 2000.

[106] B. P. Miller, J. Choi, Breakpoints and Halting in Distributed Programs, *Proc. of the 8th Int'l Conf. on Distributed Computing Systems (ICDCS'88)*, pp. 316-323, 1988.

[107] J. Misra, Detecting Termination of Distributed Computations Using Markers, *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing* (*PODC'83)*, pp. 290-295, 1983.

[108] A. Mostefaoui, S. Rajsbaum, and M. Raynal, A Versatile and Modular Consensus Protocol, *Proc. of Int'l IEEE Conf. Dependable Systems & Networks (DSN '02)*, pp. 364-373, 2002.

[109] A. Mostefaoui, E. Mourgaya, and M. Raynal, Asynchronous Implementation of Failure Detectors, *Proc. Int'l IEEE Conf. on Dependable Systems and Networks (DSN'03)*, IEEE Computer Society Press, pp. 351-360, 2003.

[110] A. Mostefaoui, and M. Raynal, Leader-based Consensus, *Parallel Processing Letters*, vol. 11 no. 1, pp. 95-107, 2001.

[111] A. Mostefaoui, M. Raynal, and C. Travers, Crash-Resilient Time-Free Eventual Leadership, *Proc. of the 23r Symp. on Reliable Distributed Systems (SRDS'04)*, pp. 208-217, 2004.

[112] A. Mostefaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. El Abbasi, From Static Distributed Systems to Dynamic Systems. *Proc. of the 24th Symp. on Reliable Distributed Systems (SRDS'05)*, pp. 109-118, 2005.

[113] A. Mostefaoui, M. Raynal, and F. Tronel, The Best of Both Worlds: A Hybrid Approach to Solve Consensus, *Proc. Int'l IEEE Conf. Dependable Systems & Networks (DSN '00)*, pp. 513-522, 2000.

[114] A. Murphy, Algorithm Development in the Mobile Environment, *Proc. of Int'l Conf. on Software Engineering*, pp. 728-729, 1999.

[115] V. Murthy, Mobile Computing: Operational Models, Programming Modes and Software Tools, *Proc. of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS'01)*, pp. 2016-2025, 2001.

[116] M. Naimi, and M. Trehel, How to Detect a Failure and Regenerate the Token in the *log*(*n*) Distributed Algorithm for Mutual Exclusion, *Proc. of the 2nd Int'l Workshop on*

Distributed Algorithms (DISC'87), LNCS#312, pp. 155-166, 1987.

[117] K. Nakano, and S. Olariu, Randomized Leader Election Protocols for Ad Hoc Networks, *Proc. of the 7th Int'l. Colloquium on Structural Information and Communication Complexity (SIROCCO'00)*, pp. 253-267, 2000.

[118] M.L. Neilsen, and M. Mizuno, A DAG-based Algorithm for Distributed Mutual Exclusion, *Proc. of the 11th Int'l. Conference on Distributed Computing Systems (ICDCS'91)*, pp. 354-360, 1991.

[119] S. Nishio, K.F. Li, and E.G. Manning, A Resilient Mutual Exclusion Algorithm for Computer Networks, *IEEE Transactions on Parallel and Distributed Systems*, vol. 1 no. 3, pp. 344-355, 1990.

[120] R. Oliveira, Solving Consensus: From Fair-Lossy Channels to Crash-Recovery of Processes, *PhD Thesis 2139, Swiss Federal Inst. of Technology (EPFL)*, 2000.

[121] Sung-Hoon Park, An Election Protocol in a Mobile Environment, *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, Jun. 2000.

[122] V. D. Park, and M. S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks, *Proc. of the 16th Conf. on Computer Communications (INFOCOM'97)*, pp. 1405-1413, 1997.

[123] T. Park, and K. G. Shin, Optimal Tradeoffs for Location-Based Routing in Large-Scale Ad Hoc Networks, *IEEE/ACM Transaction On Networking*, vol. 13 no. 2, pp. 398-410, 2005.

[124] L. M. Patnaik, A. K. Ramakrishna, and R. Muralidharan, Distributed Algorithms for Mobile Hosts, *IEE Proc.Computers and Digital Techniques*, vol. 144 no. 2, pp. 49-56, 1997.

[125] C. E. Perkins, Mobile IP, *IEEE Communications Magazine*, vol. 35 no. 5, pp. 84-99, 1997.

[126] C. Perkins, and P. Bhangwat, Highly Dynamic Destination-Sequenced Distance-Vector (DSDV) Routing for Mobile Computers, *Proc. of ACM SIGCOMM Symp. on Communications, Architectures and Protocols (SIGCOMM'94)*, pp. 234-244, 1994.

[127] C. Perkins, and E. Royer, Ad-hoc On-Demand Distance Vector Routing, *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications(WMCSA'99)*, pp. 90-100, 1999.

[128] E. Pitoura, and B. Bhargava, Dealing with Mobility: Issues and Research Challenges, *Technical Report CSD-TR-93-070, Department of Computing Sciences, Purdue University*, Nov. 1993.

[129] D. Powell, Failure Mode Assumptions and Assumption Coverage, *Proc. of the 22nd Int'l. Conf. on Fault-Tolerant Computing (FTCS'92)*, pp. 386-395, 1992.

[130] K. Raymond, A Tree-based Algorithm for Distributed Mutual Exclusion, *ACM Transactions on Computer Systems*, vol. 7 no.1, pp. 61-77, 1989.

[131] M. Raynal, A Short Introduction to Failure Detectors for Asynchronous Distributed Systems, *ACM SIGACT News, Distributed Computing Column*, vol. 36 no. pp. 53-70, 2005.

[132] M. Raynal, and M. Singhal, Logical Time: Capturing Causality in Distributed Systems, *Computer*, pp. 49-56, 1996.

[133] G. Ricart and A. K. Agrawala, An Optimal Algorithm for Mutual Exclusion in Computer Networks, *Communications of the ACM*, vol. 24 no. 1, pp. 9-17, 1981.

[134] G. Ricart and A.K. Agrawala, Author Response to "On Mutual Exclusion in Computer Networks" by Carvalho and Roucairol, *Communications of the ACM*, vol. 26 no. 2, pp.147-148, 1983.

[135] L. Sampaio, and F. Brasileiro, Adaptive Indulgent Consensus, *Proc. of Int'l IEEE Conf. on Dependable Systems & Networks (DSN '05)*, pp. 422-431, 2005.

[136] B. A. Sanders, The Information Structure of Distributed Mutual Exclusion Algorithms, *ACM Trans. on Computer Systems*, vol. 5 no. 3, pp. 284-299, 1987.

[137] Y. Sato, M. Inoue, T. Masuzawa, and H. Fujiwara, A Snapshot Algorithm for Distributed Mobile Systems, *Proc. of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, pp. 734-743, 1996.

[138] M. Satyanarayanan, Fundamental Challenges in Mobile Computing, *Proc. of the 15th ACM Symp. on Principles of Distributed Computing (PODC'96)*, pp. 1-7, 1996.

[139] A. Schiper, Early Consensus in an Asynchronous System with a Weak Failure Detector, *Distributed Computing*, vol. 10 no. 3, pp. 149-157, 1997.

[140] H. Seba, N. Badache, A. Bouabdallah, Solving the Consensus Problem in a Dynamic Group: an Approach Suitable for a Mobile Environment, *Proc. 7th IEEE Symp. on Computers and Communications (ISCC'02)*, pp. 29-35, 1999

[141] P. Serrano-Alvirado, C. Roncancio, and M. Adiba, Analyzing Mobile Transactions Support for DBMS, *Proc. of 12th International Workshop on Database and Expert Systems Applications*, pp. 595-600, 2001.

[142] P. Serrano-Alvarado, C. Roncancio, M. Adiba, and C. Labbe, Context Aware Mobile Transactions, *Proc. of IEEE International Conference on Mobile Data Management (MDM'04)*, p. 167, 2004.

[143] M. Singhal, A Dynamic Information Structure Mutual Exclusion Algorithm for Distributed Systems, *IEEE Trans. on Parallel and Distributed Systems*, vol. 3 no. 1, pp. 121-125, 1991.

[144] M. Singhal, A Heuristically-aided Algorithm for Mutual Exclusion, *IEEE Transactions on Computers*, vol. 38 no. 5, pp. 651-662, 1989.

[145] M. Singhal, A Taxonomy of Distributed Mutual Exclusion, *Journal of Parallel and Distributed Computing*, vol. 18 no. 1, pp. 94-101, 1993.

[146] M. Singhal, and D. Manivannan, A Distributed Mutual Exclusion Algorithm for Mobile Computing, *Proc. of IASTED International Conference on Intelligent Information Systems (IIS '97)*, pp. 557-561, 1997.

[147] M. Singhal, and N. G. Shivaratri, Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems, *McGraw-Hill*, 1994.

[148] W. Stallings, Wireless Communications and Networks (2nd Edition), *Pearson/Prentice Hall*, 2005.

[149] K. Sundaresan, V. Anantharaman, H. Hsieh, and R. Sivakumar, ATP: a Reliable Transport Protocol for Ad-hoc Networks, *IEEE Transactions on Mobile Computing*, vol. 4 no. 6, pp. 588- 603, 2005.

[150] I. Suzuki, and T. Kazami, A Distributed Mutual Exclusion Algorithm, *ACM Transactions on Computer Systems*, vol. no. 4, pp344-349, 1985.

[151] A. S. Tanenbaum and M. V. Steen, Distributed Systems–Principles and Paradigms, *Prentice-Hall*, 2002.

[152] G. Tel, Introduction to Distributed Algorithms, 2nd edition, *Cambridge University Press*, 2000.

[153] B. Thai, and A. Seneviratne, IPMOA: Integrated Personal Mobility Architecture, *IEEE Symposium on Computers and Communications (ISCC'01)*, pp. 485-490, 2001.

[154] Y. Tseng, Detecting Termination by Weight-Throwing in a Faulty Distributed System, *J. Parallel and Distributed Computing*, vol. 25 no. 1, pp. 7-15, 1995.

[155] Y. Tseng, and Cheng-Chung Tan, Termination Detection Protocols for Mobile Distributed Systems, *IEEE Trans. on Parallel and Distributed Systems*, vol. 12 no. 6, pp. 558-566, 2001.

[156] UCLA Parallel Computing Lab., GloMoSim Manual v1.2, http://pcl.cs.ucla.edu/

[157] U. Varshney, Networking Support for Mobile Computing, *Communications of AIS*, vol. 1 article. 1, pp. 1-30, 1999.

[158] S. Vasudevan, J. Kurose, and D. Towsley, Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks, *Proc. of 12th IEEE International Conference on Network Protocols (ICNP'04)*, pp. 350-360, 2004.

[159] M. G. Velazquez, A Survey of Distributed Mutual Exclusion Algorithms, *Technical report CS- 93-116*, Colorado State University, 1993

[160] E. Vollset, and P. D. Ezhilchelvan, Design and Performance-Study of Crash-tolerant Protocols for Broadcasting and Reaching Consensus in MANETs, *Proc. of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pp. 166-175, 2005.

[161] J. Walter and S. Kini, Mutual Exclusion on Multihop, Mobile Wireless Networks,

Technical Report TR97-014, Texas A&M Univ., College Station, Dec. 1997.

[162] J. Walter, J. Welch, and N. Vaidya, A Mutual Exclusion Algorithm for Ad Hoc Mobile Networks, *Wireless Networks* , vol. 9 no. 6, pp. 585-600, 2001.

[163] P. Waters, and A. Walter, Trusted Transactions in a Mobile Environment, *Proc. of 3G Mobile Communication Technologies*, pp. 359- 363, 2003.

[164] J. Wei, T. He, and T. Huang, Challenges of Communication in Mobile Computing, *Proc. of Technology of Object-Oriented Languages* (*TOOLS Asia'98)*, pp. 196-203, 1998.

[165] X. Yu, Improving TCP Performance over Mobile Ad Hoc Networks by Exploiting Cross-layer Information Awareness, *Proc. of the 10th Int'l Conf. on Mobile Computing and Networking (MobiCom'04)*, pp. 231-244, 2004.

[166] Q. Zhao, and L. Tong, Energy Efficiency of Large-Scale Wireless Networks: Proactive Versus Reactive Networking, *Journal on Selected Areas in Communications*, vol. 23 no. 5, pp. 1100-1112, 2005.