2013

ALGORITHMS FOR FAULT TOLERANCE IN DISTRIBUTED SYSTEMS AND ROUTING IN AD HOC NETWORKS

Qiangfeng Jiang
University of Kentucky, qiangfeng.jiang@uky.edu

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained and attached hereto needed written permission statements(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine).

I hereby grant to The University of Kentucky and its agents the non-exclusive license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless a preapproved embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's dissertation including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Qiangfeng Jiang, Student

Dr. D. Manivannan, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

ALGORITHMS FOR FAULT TOLERANCE IN DISTRIBUTED SYSTEMS AND ROUTING IN AD HOC NETWORKS

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By
Qiangfeng Jiang
Lexington, Kentucky
Director: Dr. D. Manivannan, Associate Professor of Computer Science
Lexington, Kentucky
2013

ABSTRACT OF DISSERTATION

ALGORITHMS FOR FAULT TOLERANCE IN DISTRIBUTED SYSTEMS AND ROUTING IN AD HOC NETWORKS

Checkpointing and rollback recovery are well-known techniques for coping with failures in distributed systems. Future generation Supercomputers will be message passing distributed systems consisting of millions of processors. As the number of processors grow, failure rate also grows. Thus, designing efficient checkpointing and recovery algorithms for coping with failures in such large systems is important for these systems to be fully utilized. We presented a novel communication-induced checkpointing algorithm which helps in reducing contention for accessing stable storage to store checkpoints. Under our algorithm, a process involved in a distributed computation can independently initiate consistent global checkpointing by saving its current state, called a tentative checkpoint. Other processes involved in the computation come to know about the consistent global checkpoint initiation through information piggy-backed with the application messages or limited control messages if necessary. When a process comes to know about a new consistent global checkpoint initiation, it takes a tentative checkpoint after processing the message. The tentative checkpoints taken can be flushed to stable storage when there is no contention for accessing stable storage. The tentative checkpoints together with the message logs stored in the stable storage form a consistent global checkpoint.

Ad hoc networks consist of a set of nodes that can form a network for communication with each other without the aid of any infrastructure or human intervention. Nodes are energy-constrained and hence routing algorithm designed for these networks should take this into consideration. We proposed two routing protocols for mobile ad hoc networks which prevent nodes from broadcasting route requests unnecessarily during the route discovery phase and hence conserve energy and prevent contention in the network. One is called Triangle Based Routing (TBR) protocol. The other routing protocol we designed is called Routing Protocol with Selective Forwarding (RPSF). Both of the routing protocols greatly reduce the number of control packets which are needed to establish routes between pairs of source nodes and destination nodes. As a result, they reduce the energy consumed for route discovery. Moreover, these protocols reduce congestion and collision of packets due to limited number of nodes retransmitting the route requests.

KEYWORDS: Routing, Ad hoc networks, Communication-induced, Checkpointing, Distributed systems.

Qiangfeng Jiang
Nov. 15, 2013

ALGORITHMS FOR FAULT TOLERANCE IN DISTRIBUTED SYSTEMS AND ROUTING IN AD HOC NETWORKS

By
Qiangfeng Jiang

Director of Dissertation
Dr. D. Manivannan

Director of Graduate Studies
Dr. Miroslaw Truszczynski

ACKNOWLEDGMENTS

I thank many people. Without their help, I could not have completed this dissertation.

I would like to express the deepest appreciation to Dr. D. Manivannan, my Ph.D advisor and the chairman of my Ph.D committee. He continually and convincingly conveyed a spirit of adventure in regard to research and scholarship, and an excitement in regard to teaching. Without his guidance and persistent help this dissertation would not have been possible.

I am also thankful to my thesis committee members Dr. Zongming Fei, Dr. Mukesh Singhal and Dr. James Robert Heath for their valuable comments.

Finally, I thank my wife, Yan Sun for her encouragement and support. I have learned a lot from our discussions and work together.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis makes contribution in the following two areas, namely, (i) checkpointing and rollback recovery in distributed systems and (ii) routing in mobile ad hoc networks. In this chapter, we briefly describe the problems addressed and solutions proposed in this dissertation in these areas.

1.1 Checkpointing and Rollback Recovery in Distributed systems

A distributed system is a set of computers connected by a communication network. A distributed computation is a set of processes running in a distributed system trying to solve a specific problem. Processes involved in a distributed computation communicate with each other by sending messages to each other over the communication network. Current day supercomputers are distributed systems and applications running on these supercomputers are distributed computations.

As the number of processors in a distributed system grows, failure rate also grows. So, handing failures efficiently in such systems is an important problem. Checkpointing and rollback recovery are established techniques for handing failures in such systems. To recover from failures, the state of the processes involved in a distributed computation are saved to stable storage periodically; when a failure occurs, all the processes involved in the computation are restarted from a previously saved state that represents a consistent state of

the computation.

1.1.1 Optimistic Checkpointing and Recovery Algorithm

Based on how checkpoints of processes are taken, existing checkpointing algorithms can be classified into three main categories – *uncoordinated*, *coordinated* [11, 33, 36, 41, 59], and *communication-induced* [2, 34, 43, 45]. In *uncoordinated* checkpointing, processes take local checkpoints without any coordination. To recover from a failure, the failed process determines a consistent global checkpoint by communicating with other processes and all the processes rollback to that consistent global checkpoint. Since multiple checkpoints are stored, *uncoordinated* checkpointing is not a storage resource efficient approach. In order to achieve domino-free recovery, *coordinated* checkpointing schemes have been proposed [11, 33, 36, 41, 59]. In this approach, processes synchronize their checkpointing activities by passing explicit control messages so that a globally consistent checkpoint is always maintained in the system. *Communication-induced* checkpointing is a hybrid of *uncoordinated* and *coordinated* checkpointing schemes. Under *communication-induced* checkpointing algorithms [2, 34, 43–45], processes are allowed to take local checkpoints independently, and the number of useless checkpoints is minimized by forcing processes to take communication-induced (forced) checkpoints under certain situations. Hence, this class of algorithms overcome the disadvantages of *uncoordinated* and *coordinated* checkpointing algorithms, and have the advantages of both coordinated and uncoordinated checkpointing algorithms.

Communication-induced checkpointing appears to be an attractive approach for checkpointing in distributed systems. However, existing algorithms in this category have the following drawbacks: Several processes may take checkpoints simultaneously which can cause network contention and hence impact the checkpointing *overhead* and extend the overall execution time [65, 66]. In general, communication-induced checkpoints have to be taken before processing a received message, which may significantly prolong the re-

sponse time for processing the corresponding received messages. Moreover, communication pattern may induce large number of communication-induced checkpoints since processes have to take their local checkpoints (including communication-induced checkpoints) immediately after specified conditions hold. We address this issue and propose an "Optimistic" [1, 19] checkpointing algorithm.

The optimistic checkpointing algorithm allows processes to save checkpoints and message logs in memory first and then flush them to stable storage when there is no contention for accessing stable storage. Each checkpoint taken by our algorithm is composed of a *tentative* checkpoint representing the state of the process and a set of messages logged after taking the *tentative* checkpoint. This mechanism gives processes the liberty of choosing the time to take tentative checkpoints and hence no checkpoint needs to be taken before processing any received message. Furthermore, processes are able to choose their convenient time for writing the tentative checkpoints and the associated message logs to stable storage at the network file server. This helps in minimizing network contention for accessing stable storage.

1.2 Routing in Mobile ad hoc Networks

With recent advances in wireless communication technology, wireless networks have become increasingly popular. There are several types of wireless networks including wireless local area networks, mobile ad hoc networks, sensor networks and cellular networks. In this dissertation, we focus on routing algorithms for mobile ad hoc networks.

Mobile ad hoc networks generally have the following characteristics: dynamically changing network topology, limited network bandwidth, energy constrained nodes, and limited physical security. Due to the dynamic nature of the topology, there are no fixed routers in mobile ad hoc networks; every node in the network acts as a router also. Designing efficient routing algorithms that take the energy constraint of nodes and the dynamically changing topology of the network into consideration is important.

Routing algorithms can be broadly classified as table driven algorithms, demand driven algorithms, and position based algorithms. Table driven routing algorithms maintain routes from each node to every other node in the network proactively. When network topology changes, the updates are propagated throughout the network in order to maintain accurate routing tables. This type of algorithms are not suitable for networks with nodes moving frequently due to the communication costs involved in topology changes. On the other hand demand driven algorithms requires nodes to establish routes only when a source node needs a route to a destination node. In this case, the source node initiates a route discovery process within the network. The process is completed once a route is found or is terminated when no possible routes to the destination exists. An established route is maintained until it is no longer needed or the route breaks due to the mobility of the nodes on the route. Position based routing algorithms rely on geographic position information to discover routes to destinations. In this dissertation, we present two demand driven routing algorithms we designed for mobile ad hoc networks. Next, we briefly discuss the characteristics of these two routing protons.

1.2.1 Triangle Based Routing

Many of the existing demand driven routing algorithms for ad hoc networks employ simple flooding mechanism to disseminate route request messages during route discovery phase. Under these algorithms, a source node needing a route to a destination, broadcasts a route request message to all nodes within its transmission range. Each node receiving this message rebroadcasts the message if it has not already done so and this process continues until all nodes that are reachable from the source receive this message. When the destination node receives this message, it sends a route reply message which travels along the route traversed by the route request in the reverse direction and reaches the source, establishing the route from the source to the destination. Under this approach each node reachable from the source forwards the route request message once, which leads to redundant rebroadcast-

ing of route request messages. In dense networks, this approach will result in high network contention, high network load, and high network delay.

We developed two routing algorithms which address this issue. Our algorithms reduce the redundant rebroadcasting of the route requests. In both algorithms, we assume that all nodes lie in the same plane and they all have the same transmission range R. In the first algorithm, we divide the plane into a number of equilateral triangle areas as shown in Figure 1.1. Each triangle area is assigned a unique identifier called Absolute Location Identifier. All nodes in a triangle area know the identifier and exchange it with their neighbors periodically. This way, each node in the network has approximate knowledge about its neighbor locations. Based on this information, a node b is able to decide whether and when to forward a route request message received from a node. Redundant messages are greatly suppressed under this approach. We call this algorithm Triangle based Routing Algorithm.

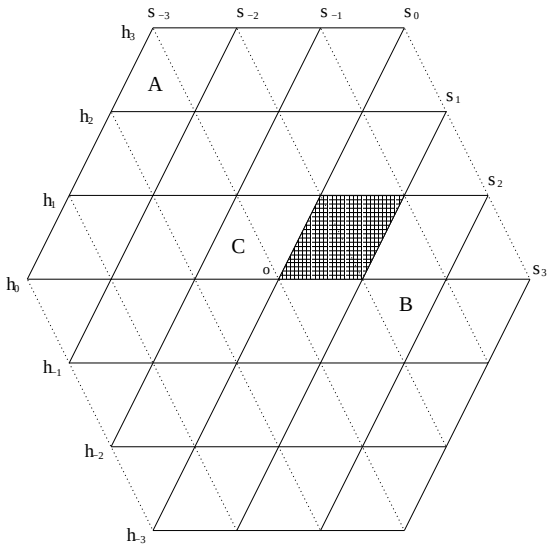


Figure 1.1: Plane divided into triangle areas

1.2.2 A Routing Algorithm with Selective Forwarding for ad hoc Networks

The other algorithm which we call Routing Algorithm with Selective Forwarding (RPSF), allows each node to select a subset of its neighbors to forward route requests. Since only a

small subset of the nodes receiving a route request forward the route request, this approach significantly reduces the routing overhead, especially in dense networks. We theoretically prove that this approach is guaranteed to find a route to the destination if one exists. We compare the performance of our algorithm with the well known Ad hoc On-demand Distance Vector (AODV) routing algorithm. On average, our algorithm needs less than 12.6% of the routing-control packets needed by AODV. Simulation results also show that our algorithm has a higher packet-delivery ratio and lower average end-to-end delay than AODV.

1.3 Organization of this Dissertation

Rest of this dissertation is organized as follows. In Chapter 2, we present an optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. This allows a process to independently initiate consistent global checkpointing by saving its current state, called a tentative checkpoint. Recording of a consistent global checkpoint of the distributed computation is complete when all the processes involved in the computation have taken and saved their tentative checkpoints and the associated message logs to stable storage. In Chapter 3, we present a routing algorithm, called Triangle Based Routing (TBR) algorithm, which utilizes Relative Location Identifier (RLI) to limit the number of route requests sent over the network and hence improves the efficiency in routing for static sensor networks. We present another routing algorithm, Routing Algorithm with Selective Forwarding for MANETs (RPSF) in Chapter 4. RPSF employs relative neighborhood information to suppress the number of route requests propagated. This is similar to the TBR algorithm. The two algorithms differ in the way in which they try to reduce the redundant rebroadcasting of route requests. Under RPSF, a node forwards a received route request packets only when the packets tell them to do so while nodes running TBR algorithm elect to forward / discard the packets based on local information it has about the nodes which were already covered by the route request. Finally, Chapter 5

concludes this dissertation and discusses future research directions.

Chapter 2

Checkpointing and Recovery in Distributed Systems

2.1 Introduction

Checkpointing and rollback recovery are popular approaches for handling failures in distributed systems. Existing checkpointing algorithms can be classified into three main categories – *uncoordinated*, *coordinated* [11, 33, 36, 41, 59], and *communication-induced* [2, 34, 43, 45]. In *uncoordinated* checkpointing, processes involved in a distributed computation take local checkpoints without any coordination. To recover from a failure, the failed process determines a consistent global checkpoint by communicating with other processes and all the processes rollback to that consistent global checkpoint. Message logging [28, 30, 60, 61] has been suggested in the literature to cope with the domino effect. Since multiple checkpoints are stored, *uncoordinated* checkpointing is not a storage resource efficient approach. Moreover, some or all of the checkpoints taken may not be part of any consistent global checkpoint and hence are useless. Hence, in the worst case, all processes may have to be restarted from the beginning when one process fails. *coordinated* checkpointing schemes have been proposed [11, 33, 36, 41, 59] to prevent useless checkpoints. In this approach, processes synchronize their checkpointing activities by passing explicit control messages so that a globally consistent checkpoint is always maintained in the system. *Communication-induced* checkpointing is a hybrid of *uncoordinated* and

coordinated checkpointing schemes. Under *communication-induced* checkpointing algorithms [2, 34, 43–45], processes are allowed to take local checkpoints independently, and the number of useless checkpoints is reduced by forcing processes to take communication-induced (forced) checkpoints under certain situations. Hence, this class of algorithms overcome the disadvantages of *uncoordinated* and *coordinated* checkpointing algorithms, and have the advantages of both coordinated and uncoordinated checkpointing algorithms.

Communication-induced checkpointing appears to be an attractive approach for checkpointing in distributed systems. However, existing algorithms in this category have the following drawbacks: Several processes may take checkpoints simultaneously which can cause network contention for accessing stable storage and hence impact the checkpointing *overhead* and extend the overall execution time [65, 66]. In general, communication-induced checkpoints have to be taken before processing a received message, which may significantly prolong the response time of those corresponding received messages. Moreover, communication pattern may induce large number of communication-induced checkpoints. Processes have to take their local checkpoints (including communication-induced checkpoints) immediately after specified conditions hold.

We use the term "Optimistic" [1, 19] because our algorithm saves checkpoints and message logs in memory first and then flushes them to stable storage to prevent contention for accessing stable storage. Each checkpoint taken by our algorithm is composed of a *tentative* checkpoint representing the state of the process and a set of messages logged after taking the *tentative* checkpoint. This mechanism gives processes the liberty of choosing the time to take tentative checkpoints and hence no checkpoint needs to be taken before processing any received message. Furthermore, processes are able to choose their convenient time for writing the tentative checkpoints and the associated message logs to stable storage at the network file server. This helps in reducing network contention for access to stable storage. Moreover, our algorithm does not incur additional overhead due to communication-induced checkpoints, unlike many other existing algorithms.

The rest of this chapter is organized as follows. First, we discuss related work in Section 2.2. In Section 2.3 we present the system model and background. Then, Section 2.4 describes our *communication-induced* checkpointing algorithm and the recovery algorithm. We present the performance evaluation of our checkpointing algorithm and also compare our algorithm with one other algorithm in Section 2.5. Thereafter, we conclude in Section 2.6.

2.2 Related Work

In this section, we briefly review previously proposed algorithms related to our checkpointing algorithm.

Barigazzi and Strigini [3] presented a *coordinated* checkpointing algorithm in which they assume that all communications between processes are atomic. Koo and Toueg [36] relaxed this assumption. Some other works [33, 36] have focused on reducing the number of synchronization messages and the number of forced checkpoints during checkpointing. These algorithms force relevant processes to block during the checkpointing process, which will degrade system performance [20]. Chandy and Lamport [13] presented the first non-blocking algorithm for *coordinated* checkpointing. However, it leads to a message complexity of $O(n^2)$. Silva et al. [59] also addressed this issue and presented another non-blocking algorithm.

Cao and Singhal [11] presented a min-process and non-blocking algorithm. This non-blocking algorithm is based on the concept of "mutable checkpoint", which can be saved anywhere, e.g., the main memory or the local disk. Therefore, the algorithm avoids the overhead of transferring "mutable checkpoints" to the stable storage at the file server across the network. Moreover, it forces only a minimum number of processes to save their checkpoints on the stable storage. Mandal and Mukhopadhyaya [42] presented a checkpointing algorithm in which processes are arranged in a ring. Processes are allowed to take checkpoints independently anytime in a predetermined time interval, called total checkpointing

time (TCT). Once a process takes a checkpoint, it sends a checkpoint request to the next process in the ring. A process receiving the checkpoint request has to take a checkpoint if it did not take a checkpoint in that interval so far and then forwards the checkpoint request to the next process in the ring and this continues. There are two drawbacks with this algorithm. One is that clocks need to be synchronized so that each process has the same view of the checkpoint interval. The other problem is, if a process takes a checkpoint early in the interval TCT, it will force all other processes to take checkpoints sequentially which will cause contention at stable storage. In our algorithm, a process does not send any control message for taking checkpoints unless it is necessary. Moreover, when a process receives a message from a process that already took a tentative checkpoint, it does not have to take a checkpoint immediately; it can take checkpoint after processing the message. In addition, the checkpoint taken need not be flushed immediately to stable storage, thus preventing contention for stable storage.

Network contention that arises due to multiple processes simultaneously trying to store local checkpoints to the stable storage simultaneously can significantly increase the checkpointing *overhead* and extend the total execution time of the distributed computation [65, 66]. Contention for stable storage can be mitigated by *staggering* the checkpoints [57]. *Staggered* checkpointing attempts to prevent two or more processes take checkpoints at the same time, thereby reducing contention for stable storage. To the best of our knowledge, checkpoint *staggering* has previously been proposed only for *synchronous*, or coordinated, checkpointing algorithms [57, 66]. These algorithms are referred to as staggered checkpointing algorithms. Plank [57] proposed a variation of the Chandy-Lamport algorithm [13] that staggers a *limited* number of checkpoints depending on the network topology. However, a completely connected topology would subvert staggering in this algorithm. Based on Plank's observation, Vaidya [66] proposed another coordinated checkpointing algorithm that staggers *all* checkpoints. Like Plank [57] and Chandy-Lamport [13], Vaidya's algorithm [66] uses a coordinator to initiate the checkpointing process. It has two phases. In

the first phase, the coordinator P_0 takes a physical checkpoint (i.e., saves its current state in stable storage) and sends a *take_checkpoint* message to the next process P_1. Upon receipt of the *take_checkpoint* message, process P_i takes a physical checkpoint and resends it to process P_j, where $i>0$ and $j = (i+1) \bmod n$. The phase is terminated when the coordinator P_0 receives the *take_checkpoint* message from the last process P_{n-1}. In the second phase, the channel states, called logical checkpoints, are recorded. The set of logical checkpoints, together with the physical checkpoints, form a consistent global state. The algorithm successfully staggers all physical checkpoints. However, as shown in our simulation results, contention for stable storage always exists for taking the logical checkpoints. In terms of the number of collisions due to the logged messages, Vaidya's algorithm [66] always performs worse, compared to our algorithm.

2.3 Background

2.3.1 System Model

A distributed computation consists of N sequential processes denoted by P_0, P_1, P_2, $\cdots$, and P_{N-1} running concurrently on a set of computers in the network. Processes do not share a global memory or a global physical clock. Message passing is the only way for processes to communicate with one another. The computation is asynchronous: each process evolves at its own speed and messages are transmitted through communication channels, whose transmission delays are finite but arbitrary. Channels are assumed to be FIFO and the computation is assumed to be piecewise-deterministic [19, 21]. Elnozahy et al. [19] present an excellent survey of the state of the art in checkpointing and recovery. Messages generated by the underlying distributed computation will be referred to as *application messages*. Explicit control messages generated by checkpointing algorithm will be referred to as *control messages*. In our algorithm, limited amount of control messages are generated for the collection of consistent global checkpoint, only when necessary.

2.3.2 Consistent Global Checkpoints

The execution of a process is modeled by three types of events – the send event of a messages, the receive event of messages and internal events. The states of processes depend on one another due to interprocess communication. Lamport's *happened before* relation [39] on events, $\xrightarrow{\text{hb}}$, is defined as the transitive closure of the union of two other relations: $\xrightarrow{\text{hb}} = (\xrightarrow{\text{xo}} \cup \xrightarrow{\text{m}})^+$. The $\xrightarrow{\text{xo}}$ relation captures the order in which local events of a process are executed. The i^{th} event of any process P_p (denoted $e_{p,i}$) always executes before the $(i+1)^{st}$ event: $e_{p,i} \xrightarrow{\text{xo}} e_{p,i+1}$. The $\xrightarrow{\text{m}}$ relation shows the relation between the send and receive events of the same message: if a is the send event of a message and b is the corresponding receive event of the same message, then $a \xrightarrow{\text{m}} b$ [45, 48].

A local checkpoint of a process is a recorded state of the process. A checkpoint of a process is considered as a local event of the process for the purpose of determining the existence of happened before relation among checkpoints of processes. Each checkpoint of a process is assigned a unique sequence number. The checkpoint of process P_p with sequence number i is denoted by $C_{p,i}$.

The send and the receive events of a message M are denoted respectively by $send(M)$ and $receive(M)$. So, $send(M) \xrightarrow{\text{hb}} C_{p,i}$ if message M was sent by process P_p before taking the checkpoint $C_{p,i}$. Also, $receive(M) \xrightarrow{\text{hb}} C_{p,i}$ if message M was received and processed by P_p before taking the checkpoint $C_{p,i}$. $send(M) \xrightarrow{\text{hb}} receive(M)$ for any message M. The set of events in a process that lie between two consecutive checkpoints is called a checkpointing interval.

A global checkpoint of a distributed computation is a set of checkpoints containing one checkpoint from each process involved in the distributed computation. An orphan message M with respect to a global checkpoint is a message whose $receive(M)$ event is recorded in the global checkpoint but the corresponding $send(M)$ event is not recorded. A global checkpoint is said to be consistent if there is no orphan message with respect to that global checkpoint. Figure 2.1 shows two global checkpoints S_1 and S_2. Clearly S_1 is a consistent

global checkpoint while S_2 is NOT a consistent global checkpoint since M_5 is an orphan message with respect to S_2.

Next, we present our algorithm.

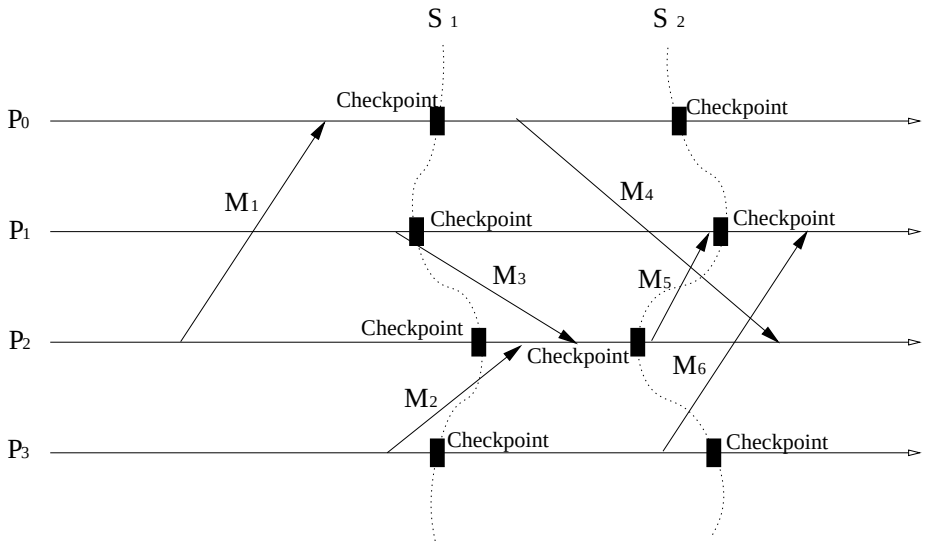


Figure 2.1: Global checkpoints

2.4 Algorithm

2.4.1 Notations

Following are the notations used in describing the algorithm and its correctness proof.

- $C_{i,k}$ denotes the (permanent) local checkpoint taken by P_i. It is composed of two parts – a tentative checkpoint $CT_{i,k}$ recording the state of the process and a set of logged messages $logSet_{i,k}$ associated with the checkpoint.
  - $CT_{i,k}$ denotes the tentative checkpoint taken by P_i with checkpoint sequence number k. It is usually saved in memory first and then flushed to stable storage after recording the associated log, namely, $logSet_{i,k}$ or whenever there is no contention for accessing stable storage.

– $logSet_{i,k}$ denotes the set containing all messages sent and received by P_i after taking the tentative checkpoint $CT_{i,k}$ and before the checkpoint $C_{i,k}$ is finalized. We refer to the operation of flushing the tentative checkpoint and the log of messages to stable storage as *finalizing* the tentative checkpoint. We explain the steps taken for finalizing a tentative checkpoint in Section 2.4.4.

Thus, we have $C_{i,k} = CT_{i,k} \cup logSet_{i,k}$.

- $CFE_{i,k}$ denotes the event that represents the finalizing operation of checkpoint $C_{i,k}$. Therefore, all sending and/or receiving events of messages in $logSet_{i,k}$ happen before $CFE_{i,k}$. For any event e of P_i, we have

$$e \xrightarrow{\text{hb}} C_{i,k} \iff e \xrightarrow{\text{hb}} CFE_{i,k}. \tag{2.1}$$

- S_k denotes the global checkpoint composed of checkpoints with sequence number k from each process. Thus, $S_k = \{C_{i,k} | i \in \{0, 1, \cdots, N-1\}\}$.

2.4.2 Basic Idea

The basic idea behind our algorithm is as follows: Any process can initiate taking a consistent global checkpoint. A process accomplishes this by saving its state (called a tentative checkpoint) and then piggy-backing this information with each application message it sends after that. When a process P_i receives a message from a process P_j, it comes to know whether P_j has taken a tentative checkpoint as a result of its own consistent global checkpoint initiation or as a result of the initiation of some other process. When P_i comes to know about a new initiation of consistent global checkpoint, it takes a tentative checkpoint. Each checkpoint taken is assigned a sequence number which is one more than its previous checkpoint. After a process takes a tentative checkpoint, it continues logging all the messages sent and received in its local memory until it comes to know that all other processes have taken a tentative checkpoint corresponding to its current tentative checkpoint. When

a process comes to know that all the processes have taken a tentative checkpoint that corresponds to its own current tentative checkpoint, it flushes its current tentative checkpoint (if it has not already done so) and the associated message log to stable storage. We call the process of flushing a tentative checkpoint and its associated message log into stable storage as "*Finalizing the Checkpoint*". A process is not allowed to initiate a new consistent global checkpoint until it finalizes its current tentative checkpoint. A process, initially, starts in the *normal* status. After a process takes a tentative checkpoint, its status changes from normal to *tentative*. After a tentative checkpoint is finalized, its status changes back to normal. The set of finalized checkpoints with a given sequence number m, denoted by S_m, forms a consistent global checkpoint as proved in Theorem 2.2. Next, we illustrate the basic idea behind our algorithm with an example.

An Example

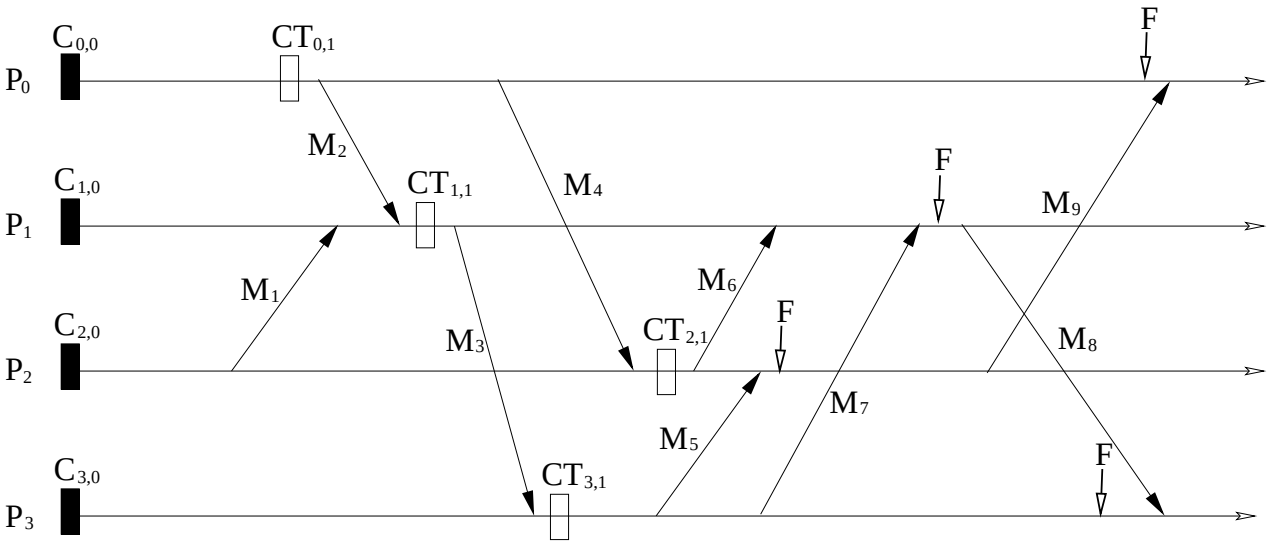


Figure 2.2: An example illustrating the basic idea behind our algorithm

For explaining the basic idea behind our algorithm, we use the space-time diagram of a distributed computation consisting of four processes shown in Figure 2.2. P_0, P_1, P_2 and P_3 are the four processes involved in the computation. Initially, the status of each pro-

cess is *normal* and their initial checkpoints, with sequence number 0, are marked by solid rectangular boxes in the figure. Suppose P_0 initiates consistent global checkpointing by taking a tentative checkpoint $CT_{0,1}$. After taking checkpoint $CT_{0,1}$, it changes its status from *normal* to *tentative* and starts logging in memory all messages sent and received by it until it finalizes this checkpoint. Then, P_0 sends a message M_2 to P_1. Upon receiving M_2, P_1 notices that P_0 has taken $CT_{0,1}$. Therefore, P_1 takes a tentative checkpoint $CT_{1,1}$ after processing M_2 and P_1's status changes from *normal* to *tentative*. Similarly, P_2 and P_3 take tentative checkpoints $CT_{2,1}$ and $CT_{3,1}$ after receiving messages M_4 and M_3 respectively. P_1 knows that the status of P_0 and P_1 is *tentative* before sending the message M_3; P_1 piggy-backs this information with M_3. Therefore, P_3 knows that the status of P_0, P_1, and P_3 is *tentative* before sending the message M_5. Upon receiving M_5, P_2 knows that the status of all processes is *tentative*. At this point, P_2 finalizes the checkpoint with sequence number 1 by flushing the tentative checkpoint $CT_{2,1}$ (if it has not already done so) and the set of logged messages $\{M_5, M_6\}$ into the stable storage. And we have $C_{2,1} = CT_{2,1} \cup \{M_5, M_6\}$. An "F" mark in the figure indicates the event of finalizing the current tentative checkpoint. After a process finalizes its tentative checkpoint, its status becomes *normal* (after a process takes a tentative checkpoint, it is allowed to take another tentative checkpoint only after finalizing the already taken tentative checkpoint). Similarly, P_1 finalizes its tentative checkpoint after the message M_7 is received. When message M_8 is received, P_3 knows that P_1 has finalized its checkpoint, which indicates that all processes have taken a tentative checkpoint corresponding to its current tentative checkpoint. Therefore, P_3 finalizes its current tentative checkpoint. Note that M_8 should not be included in the set of logged messages in $C_{3,1}$ since it was sent after P_1 finalized $C_{1,1}$. Similarly, P_0 finalizes the checkpoint $C_{0,1}$ upon receiving M_9 without including M_9 in the message log. Now, a consistent global checkpoint $S_1 = \{C_{0,1}, C_{1,1}, C_{2,1}, C_{3,1}\}$ has been recorded.

Some Comments

In the example given above, there is only one initiator of the consistent global checkpoint S_1. This is primarily to make the example easily understandable. However, under our algorithm, multiple processes can concurrently initiate consistent global checkpointing by taking a tentative checkpoint. A problem with this basic algorithm is that a tentative checkpoint may never be finalized by a process if it does not receive (sufficient) messages from other processes. For example, messages such as M_5, M_7, M_8 and M_9 are needed for the four processes to finalize their checkpoints in Figure 2.2. So, the basic checkpointing algorithm will not work in the absence of sufficient number of application messages that help each process know the status of every other process in a timely manner. We call this as a consistent global checkpoint *convergence* problem and explain in Section 2.4.5 how it can be addressed by using limited number of control messages when necessary. Next, we introduce the data structures needed for presenting the basic algorithm.

2.4.3 Data Structures

Each process P_i maintains the following data structures.

1. csn_i: An integer variable containing the sequence number of the current checkpoint of process P_i. The checkpoint representing the initial state of P_i has sequence number 0. P_i sets csn_i to 0 initially. csn_i is incremented by one when a new tentative checkpoint is taken.

2. $stat_i$: A variable representing the current status of process P_i. The status of a process can be *tentative* or *normal*. The status of a process P_i is updated as follows: P_i's status is set to *normal* initially. P_i's status changes to *tentative* immediately after P_i takes a tentative checkpoint. After P_i knows that the status of all processes is *tentative* (through the information piggy-backed on the application messages), P_i sets its status back to *normal* after finalizing its current tentative checkpoint.

3. $logSet_i$: The set of messages logged at P_i after it takes a tentative checkpoint. When $stat_i$ is set to *tentative*, P_i sets $logSet_i$ to empty and starts logging messages sent and received by P_i into $logSet_i$. Thus, $logSet_i$ contains messages sent and received by P_i after a tentative checkpoint is taken and before that checkpoint is finalized. When the status of the process changes from *tentative* to *normal*, the tentative checkpoint and the corresponding $logSet_i$ are flushed to the stable storage.

4. $tentSet_i$: The *tentative process set* maintained at P_i. When $stat_i$ is set to *normal*, $tentSet_i$ is set to empty. When P_i takes a tentative checkpoint, P_i sets $tentSet_i$ to $\{P_i\}$. Upon receiving a message, P_i sets $tentSet_i$ to be the union of $tentSet_i$ and the *tentative process set* piggy-backed in the message. Thus, this set contains the set of processes that have taken a tentative checkpoint, to the knowledge of P_i.

5. $allPSet$: This is the set of all processes, namely, $\{P_0, P_1, \cdots, P_{N-1}\}$.

2.4.4 The Checkpointing Algorithm

We assume that each process takes an initial checkpoint representing the initial state of the process. The sequence number of the initial checkpoint is set to 0. Moreover, no process is allowed to take a new checkpoint when its status is *tentative*.

Consistent Global Checkpointing Initiation

Any process whose status is *normal* can take a new tentative checkpoint, thereby initiating consistent global checkpointing. When a process P_i takes a tentative checkpoint, it changes its status from *normal* to *tentative*, increases the checkpoint sequence number csn_i by one and assigns it as the sequence number for the tentative checkpoint, sets $logSet_i$ to empty, and initializes $tentSet_i$ to $\{P_i\}$. At any time, $tentSet_i$ is the set of all processes that have taken a tentative checkpoint corresponding to the current tentative checkpoint of P_i, to the knowledge of P_i. After P_i takes a tentative checkpoint, it starts logging into $logSet_i$ all the

messages sent and received until its status changes back to *normal*. Csn_i and $tentSet_i$ are piggy-backed with each application message.

Sending Messages

Each process P_i piggy-backs with each application message the current value of csn_i, $stat_i$ and $tentSet_i$. The value of csn_i, piggy-backed with messages, helps the receiver determine if the sender took a new tentative checkpoint, thereby initiating a concurrent or new consistent global checkpoint collection. These values piggy-backed with a message M are denoted by $M.csn$, $M.stat$ and $M.tentSet$ respectively. A process receiving message M uses this piggy-backed information to find out whether it is a new consistent global checkpoint collection initiation or a concurrent global checkpoint initiation; it also comes to know the processes that have already taken a tentative checkpoint corresponding to this initiation.

Receiving Messages

Under our algorithm, each process can take a tentative checkpoint independently and concurrently. Once a process comes to know that all the other processes have taken tentative checkpoints corresponding to its most recent tentative checkpoint (through a message received from a process), it finalizes the tentative checkpoint (Section 2.4.4 explains the procedure of finalizing a tentative checkpoint). After finalizing its most recent tentative checkpoint $C_{i,k}$, process P_i can take the next tentative checkpoint $C_{i,k+1}$ before every other process has finalized the tentative checkpoint corresponding to $C_{i,k}$. In such situations, if P_i sends a message M after taking $C_{i,k+1}$ and M is received by process P_j before it finalized $C_{j,k}$, then P_j needs to finalize $C_{j,k}$ first and then process the message M to prevent orphan messages. Next, we describe how process P_i handles a message M received from process P_j.

Case (1) $M.stat = stat_i = normal$. In this case, no additional action needs to be taken except processing M because neither P_i nor P_j is aware of any new consistent global

checkpoint initiation.

Case (2) $M.stat = stat_i = tentative$. In this case, both P_i and P_j have taken a new tentative checkpoint concurrently. The following four subcases arise:

Subcase (a) $M.csn < csn_i$. In this case, P_i has already taken and finalized a tentative checkpoint with sequence number $M.csn$ at the time of receiving M and P_j was not aware of this while sending M. Therefore, no additional action needs to be taken except processing the message.

Subcase (b) $M.csn = csn_i$. In this case, P_i and P_j have taken checkpoints that belong to the same global checkpoint S_{csn_i}. In this case, first M is processed and then in order to know how many processes have taken a tentative checkpoint that belongs to the global checkpoint S_{csn_i}, P_i updates $tentSet_i$ to be the union of $tentSet_i$ and $M.tentSet$. If the updated $tentSet_i$ equals to $allPSet$, P_i logs the message and then finalizes (Section 2.4.4 gives the detailed procedure for finalizing a tentative checkpoint) its tentative checkpoint since all processes have taken a tentative checkpoint with the same sequence number (i.e., tentative checkpoints that belong to the global checkpoint S_{csn_i}) and sets its status to normal (i.e., $stat_i = normal$).

Subcase (c) $M.csn = csn_i + 1$. In this case, P_j finalized the checkpoint with sequence number csn_i before sending M and also has taken a tentative checkpoint with sequence number $M.csn$. Therefore, P_i knows that all processes already took a tentative checkpoint that belongs to the global checkpoint S_{csn_i}. Recall that a process is not allowed to take a new tentative checkpoint until it has finalized its current tentative checkpoint. Thus, P_i finalizes its current tentative checkpoint with sequence number csn_i without including M in the message log because M would be an orphan message with respect to the consistent global checkpoint S_{csn_i}. Then, it processes the message M and initiates next consis-

tent global checkpointing by taking a new tentative checkpoint with sequence number $M.csn$ and also logs the message M.

Subcase (d) $M.csn > csn_i + 1$. In this case, P_j has finalized the checkpoint with sequence number $csn_i + 1$. Since P_j could have finalized that checkpoint only after all other processes including P_i have taken a tentative checkpoint with sequence number $csn_i + 1$, P_i must have a checkpoint with sequence number greater than or equal to $csn_i + 1$. This is not possible because csn_i is the sequence number of the last tentative checkpoint of P_i. So, this case does not arise. Thus, this case is not shown in the formal description of the algorithm.

Case (3) $M.stat = normal$ and $stat_i = tentative$. In this case, $P_j's$ latest checkpoint has been finalized before sending M and P_i has taken a tentative checkpoint which is yet to be finalized. The following three subcases arise:

Subcase (a) $M.csn < csn_i$. In this case, P_i has already taken and finalized a tentative checkpoint with sequence number $M.csn$ at the time of receiving M. Therefore, no further action needs to be taken in this case except processing the message.

Subcase (b) $M.csn = csn_i$. In this case, P_j has finalized taking the checkpoint with sequence number csn_i. This means P_j knows that all processes have taken a tentative checkpoint with sequence number csn_i. Hence P_i finalizes its current tentative checkpoint without including M in the message log (since M would be an orphan message), changes its status back to *normal* and then processes the message.

Subcase (c) $M.csn > csn_i$. This means P_j has taken a new checkpoint with sequence number $M.csn > csn_i$ and has finalized that checkpoint before P_i finalized the checkpoint with sequence number csn_i. This is impossible because a process cannot finalize a checkpoint with sequence number $M.csn$ before

other processes finalize their checkpoint with sequence number $M.csn - 1$. So, this case does not arise.

Case (4) $M.stat = tentative$ and $stat_i = normal$. This means P_j's latest checkpoint taken before sending M has not been finalized while sending M and P_i's latest checkpoint has been finalized. In this case, M is processed first and then the following actions are taken. The following three subcases arise:

Subcase (a) $M.csn \leq csn_i$. In this case, P_i has already taken and finalized a tentative checkpoint with sequence number $M.csn$ at the time of receiving M. So, the message is simply processed without taking any additional action.

Subcase (b) $M.csn = csn_i + 1$. In this case, P_j has taken a new tentative checkpoint about which P_i comes to know through M. Therefore, P_i takes a tentative checkpoint with sequence number $M.csn$. The procedure for taking a new tentative checkpoint is same as that in Section 2.4.4. In addition to that, P_i logs the message and updates $tentSet_i$ to be the union of $tentSet_i (= \{P_i\})$ and $M.tentSet$. Thus, P_i gets P_j's knowledge about the processes that have taken a tentative checkpoint with sequence number $csn_i + 1$.

Subcase (c) $M.csn > csn_i + 1$. This case is similar to **subcase (d)** under **case (2)** and does not arise.

Finalizing a Tentative Checkpoint that belongs to a Consistent Global Checkpoint with a Given Sequence Number

If the status of a process P_i is *tentative* and it knows through the messages received from other processes that the status of all other processes involved in the computation are tentative (i.e., $tentSet_i = allPSet$), it flushes its current tentative checkpoint (the most recent tentative checkpoint taken), if it has not already done so, and also the associated message log $logSet_i$ into the stable storage and makes it permanent. Note that the tentative checkpoint can be flushed to stable storage any time before finalizing the tentative checkpoint.

However, the message log associated with the tentative checkpoint needs to be flushed as soon as a process comes to know that all other processes have taken a tentative checkpoint corresponding to its latest checkpoint. **The tentative checkpoint together with the message log stored is called a checkpoint of the process and it is assigned the same sequence number as the tentative checkpoint stored**. Checkpoints with same sequence number from all the processes form a consistent global checkpoint, as proved in Theorem 2.2.

Formal description of the basic checkpointing algorithm is given in Figure 2.3.

2.4.5 Optimizations

A Convergence Problem

As we noted earlier, the basic checkpointing algorithm presented in the previous section may not converge if not enough messages are exchanged among processes. To address this problem, we present a mechanism that utilizes control messages to expedite convergence when necessary. So, control messages are used only if a tentative checkpoint has not been finalized within a predetermined period of time. In the following, we discuss a mechanism to introduce limited amount of control messages to expedite convergence when necessary. We introduce three type of control messages – checkpoint begin (*CK_BGN*) message, checkpoint request (*CK_REQ*) and checkpoint end (*CK_END*) messages. A process P_i sets a timer when it takes a tentative checkpoint. If P_i does not finalize its tentative checkpoint before the timer expires, it sends a *CK_BGN* message to a pre-specified process, say P_0. Upon receiving the message, P_0 takes a tentative checkpoint if it has not yet taken and then sends a *CK_REQ* message to P_1, P_1 does the same and sends it to P_2, etc. and finally *CK_REQ* reaches back to P_0. After P_0 receives the message back, it sends *CK_END* message to all the processes. When a process receives the *CK_END* message, it finalizes its local tentative checkpoint with the sequence number contained in the *CK_END* message if it has not already finalized it. It ignores the message if it has already finalized. Control

When P_i starts
$csn_i = 0$; $stat_i = normal$; /* Initialization */

Procedure: takeTentativeCheckpoint(i: integer)
$csn_i = csn_i + 1$; $stat_i = tentative$;
$tentSet_i = \{P_i\}$; /* Include the process id in the set */
$logSet_i = \emptyset$; /* Initialize the message log to empty set */
Take tentative checkpoint CT_{i,csn_i};

When P_i starts to take a checkpoint
takeTentativeCheckpoint(i);

When P_i sends a message M to P_j
$M.csn = csn_i$; /* Piggy-back current value of csn_i, $stat_i$, and $tentSet_i$ with the message */
$M.stat = stat_i$;
$M.tentSet = tentSet_i$;
if $stat_i == tentative$ **then** $logSet_i = logSet_i \cup \{M\}$;
$Send(M)$;

When P_i receives a message M from P_j
if $stat_i == normal$ **then**
  Process M;
  if $M.stat == tentative$ **then**
    if $M.csn == csn_i + 1$ **then** /* P_j has initiated a new consistent global checkpoint */
      takeTentativeCheckpoint(i);
      $logSet_i = logSet_i \cup \{M\}$; /* Log the received message */
      $tentSet_i = M.tentSet \cup tentSet_i$;
else /* $stat_i == tentative$ */
  $logSet_i = logSet_i \cup \{M\}$; /* Log the received message */
  if $M.stat == normal$ **then**
    if $M.csn == csn_i$ **then** /* P_j has finalized the checkpoint C_{j,csn_i} */
      Flush $logSet_i - \{M\}$ and CT_{i,csn_i} to the stable storage; /* P_i finalizes its checkpoint C_{i,csn_i} */
      $stat_i = normal$;
    Process M;
  else /* $M.stat == tentative$ */
    if $M.csn == csn_i$ **then** /* P_j has taken the checkpoint CT_{j,csn_i} before sending the message */
      Process M;
      $tentSet_i = M.tentSet \cup tentSet_i$;
      if $tentSet_i == allPSet$ **then** /* Each process P_k has already taken CT_{k,csn_i} */
        $stat_i = normal$;
        Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;
    else if $M.csn == csn_i + 1$ **then** /* P_j has finalized C_{j,csn_i} and took a new tentative checkpoint after that */
      $stat_i = normal$; /*So, P_i finalizes C_{i,csn_i}, excludes M from the log and takes a new tentative checkpoint */
      Flush $logSet_i - \{M\}$ and CT_{i,csn_i} to the stable storage;
      Process M;
      takeTentativeCheckpoint(i);
      $logSet_i = logSet_i \cup \{M\}$;
      $tentSet_i = M.tentSet \cup tentSet_i$;

Figure 2.3: The Basic Checkpointing Algorithm

messages are not sent if each global checkpoint can be finalized within the timeout interval. The *tentative process set* can be used to further reduce the number of control messages as follows:

Case (1) Limiting the number of *CK_BGN* messages. As we know, one *CK_BGN* message is enough to notify P_0 to initiate *CK_REQ* messages for each global checkpoint. In the method described above every process that times out sends *CK_BGN* to P_0. Such redundant messages can be reduced using the information contained in *tentative process set*. Suppose it is time for P_i to send a *CK_BGN* message to P_0. Before sending the message, it checks if there is a process P_j that belongs to $tentSet_i$ and j is less than i. If P_j exists, P_i does nothing since it knows that P_j or some other process with process id smaller than j will send a *CK_BGN* message to P_0. Otherwise, P_i sends a *CK_BGN* message to P_0. Clearly, this method reduces the number of *CK_BGN* messages. However, it introduces a new problem, namely, the process with lower process id may have finalized the checkpoint already and has not exchanged any message afterwards. This way, P_i may not be able to finalize the checkpoint. This problem can be solved by requiring P_0 always broadcast a *CK_END* message to all other processes when it finalizes a checkpoint.

Case (2) Reducing *CK_REQ* messages. Under the above approach, every process needs to forward the *CK_REQ* message once. However, the number of *CK_REQ* messages can be further reduced by the following method. Suppose it is time for P_i to forward the message. If it has finalized this checkpoint, it forwards the message to P_0 directly. Otherwise, P_i looks for a process P_j for which the following condition holds.

$$(j > i)\ AND\ (P_j \notin tentSet_i)\ AND\ (\forall k \in \{z | i < z < j\}, P_k \in tentSet_i)$$

If such a process P_j is found, P_i forwards the message to P_j because all processes with process ids greater than i and less than j have already taken a tentative checkpoint and there is no need to ask them to take it again. Otherwise, all processes with

process ids greater than i have already taken a tentative checkpoint. Therefore, P_i forwards the message to P_0 directly.

Figure 2.4 gives the formal description of how control messages can be used to augment the basic algorithm to help convergence. In this we use CM to denote a control message. A CM has two fields, namely, *type* and *csn*. *CM.type* can have one of the three values, namely, *CK_BGN*, *CK_REQ* or *CK_END*. *CM.csn* is the sequence number of the current tentative checkpoint of the sender when it sends the control message CM. $CM(atype, acsn)$ refers to the control message CM with $CM.type = atype$ and $CM.csn = acsn$. For example, $CM(CK\_BGN, 3)$ refers to a control message *CK_BGN* with $csn = 3$ piggy-backed with it.

A timer is used by each process to determine when to send control messages as follows: A process sets a timer when it takes a tentative checkpoint. When the timer expires, it initiates sending a control message CM. The timer is canceled when a process finalizes the checkpoint or it receives a CM with sequence number equal to that of its current tentative checkpoint.

We illustrate how control messages help in convergence with an example shown in Figure 2.5. Suppose P_1 takes a tentative checkpoint $CT_{1,1}$ first and sends a message M_2 to P_2. Upon receiving M_2, P_2 takes a tentative checkpoint $CT_{2,1}$. When the timer set for $CT_{1,1}$ expires, P_1 sends a *CK_BGN* message (CK_BGN_1) to P_0 (P_2 does not send a *CK_BGN* message since it knows that P_1 will send such message to P_0). Upon receiving CK_BGN_1, P_0 takes a tentative checkpoint $CT_{0,1}$ and sends a *CK_REQ* message CK_REQ_1 to P_1. Thereafter, P_1 sends a *CK_REQ* message CK_REQ_2 to P_3 since it knows that P_2 has already taken $CT_{2,1}$. Finally, the *CK_REQ* message CK_REQ_3 returns to P_0. Now, P_0 knows that all processes have already taken a tentative checkpoint with sequence number 1. Therefore, it finalizes its current tentative checkpoint and broadcasts a *CK_END* message to every other process and flushes logged application messages and $CT_{0,1}$ to the stable storage. Upon receiving CK_END, P_1, P_2 and P_3 flush their logged messages and ten-

When the timer for finalizing the tentative checkpoint on P_i expires

if $i == 0$ **then** /* P_0 *initiates* CK_REQ *messages directly without sending a* CK_BGN *message* */
  forwardCheckpointRequest(P_0, CM);
else /* $i = 1, 3, \cdots, or N-1$ */
  for each $P_k \in tentSet_i$ **do**
    if $k < i$ **then** return; /* P_k *or other process with process number less than* k *will send* CK_BGN *message to* P_0 */
  Send $CM(CK\_BGN, csn_i)$ to P_0; /* *Sending* CK_BGN *message to* P_0 */

Procedure: forwardCheckpointRequest(P_i, CM)

if $i == N-1$ **then** k = 0; /* P_{N-1} *forwards* CK_REQ *message to* P_0 *directly* */
else /* P_i *looks for process* P_j *such that the status of* $P_{i+1}, P_{i+2}, \cdots, and\ P_{j-1}$ *is tentative* */
  for $k = i+1$ **to** $N-1$ **do**
    if $P_k \notin tentSet_i$ **then** break;
  if $P_k \in tentSet_i$ **then** $k = 0$; /* *The status of all processes with process number greater than* i *is* tentative */
Send $CM(CK\_REQ, csn_i)$ to P_k;

When P_i receives CM from P_j

if $CM.csn == csn_i + 1$ **then**
  if $stat_i == tentative$ **then**
    Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;
  takeTentativeCheckpoint(i);
  forwardCheckpointRequest(P_i, CM);
else if $CM.csn == csn_i$ **then**
  if $CM.type == CK\_BGN$ **then**
    if $stat_i == tentative$ **then**
      if $CM(CK\_REQ, csn_i)$ has been sent **then** return; /* *Send the* CK_REQ *message at most once* */
      forwardCheckpointRequest(P_i, CM);
    else if $CM(CK\_END, csn_i)$ has not been sent **then** /* P_0 *has finished taking* C_{i,csn_i} */
      Send $CM(CK\_END, csn_i)$ to $P_1, P_2, \cdots,$ and P_{N-1};
  else if $CM.type == CK\_REQ$ **then**
    if $i == 0$ **then** /* P_0 *initiates* CK_END *if necessary* */
      if $CM(CK\_END, csn_i)$ has been sent **then** return;
      Send $CM(CK\_END, csn_i)$ to $P_1, P_2, \cdots,$ and P_{N-1};
      if $stat_i == tentative$ **then**
        $stat_i = normal$;
        Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;
      else forwardCheckpointRequest(P_i, CM);
    else if $stat_i == tentative$ **then** /* $CM.type == CK\_END$ */
      $stat_i = normal$;
      Flush $logSet_i$ and CT_{i,csn_i} to the stable storage;

Figure 2.4: Augmenting the Basic Algorithm with Control Messages to Speed up Convergence

tative checkpoints with sequence number 1 respectively. This way, all processes finalize the checkpoints with sequence number 1 and return to *normal* status in finite time. Without these control messages, the original algorithm does not converge in this example. Although P_3 sends out messages such as M_5 and M_6, it does not receive any message. Therefore, P_3 is unable to obtain the status information of other processes, and hence P_3 can not finalize its tentative checkpoint $CT_{3,1}$ without the help of control messages.

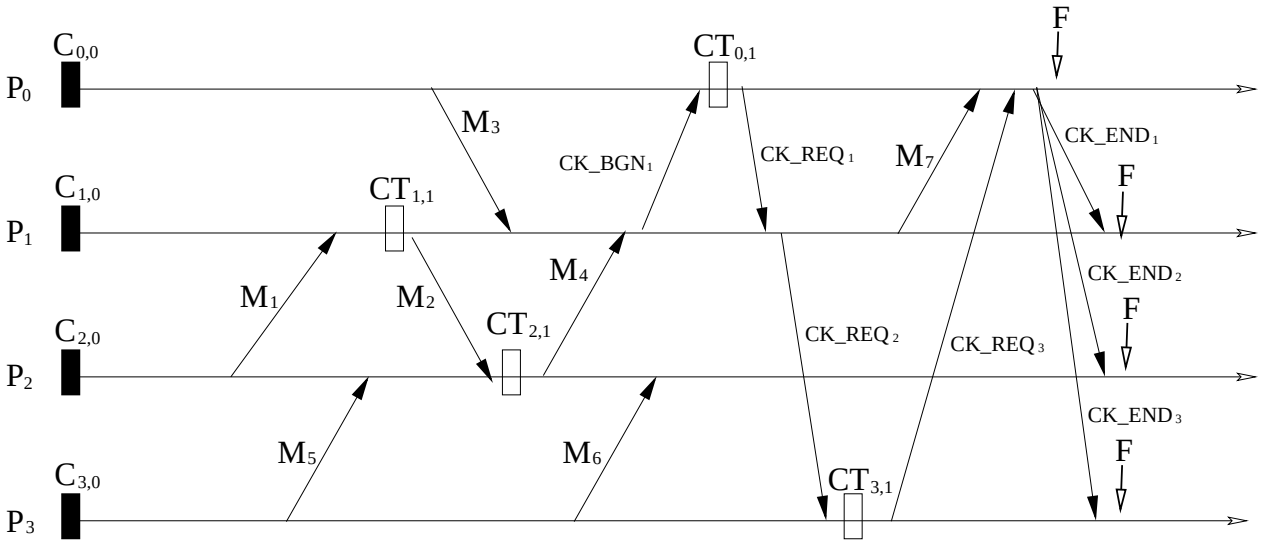


Figure 2.5: An example illustrating the use of control messages in the algorithm

2.4.6 Correctness Proof

We refer to the checkpointing algorithm with control messages as the generalized checkpointing algorithm. With this definition, we have Theorem 2.1.

Theorem 2.1 *The generalized checkpointing algorithm converges, i.e., after a process takes a tentative checkpoint with a given sequence number csn, every process eventually finalizes a checkpoint with sequence number csn.*

Proof. We prove this by contradiction. Suppose the generalized checkpointing algorithm does not converge. In other words, there is at least one process, say P_i, that took a tentative checkpoint $CT_{i,k}$ but never finalized the checkpoint $C_{i,k}$.

Depending upon why P_i takes $CT_{i,k}$, the following two cases arise.

Case (1) P_i takes $CT_{i,k}$ because it receives a message $CM(CK\_REQ, k)$ from a process P_i. Upon receiving such a message, P_i needs to forward the message to a process P_h and assure that all processes with process number greater than i and less than h have already taken a tentative checkpoint with sequence number k. This is repeated until the message returns to P_0 (P_{N-1} forwards the message to P_0 or some process

P_j ($j < N-1$) forwards it to P_0 directly since P_j knows that all processes with process number greater than j have taken a tentative checkpoint with sequence number k). Once P_0 receives the message, it finalizes $C_{0,k}$ and broadcasts a message $CM(CK\_END, k)$ to all other processes. Upon receiving this message, each process finalizes its tentative checkpoint with sequence number k if it has not already done so. In particular, P_i finalizes $C_{i,k}$ which is a contradiction to our assumption.

Case (2) P_i takes $CT_{i,k}$ due to other reasons. Then a timer is set when $CT_{i,k}$ is taken at P_i. If the timer is canceled due to receiving a *CK_REQ* or *CK_END* message with sequence number k, P_0 has initiated a message $CM(CK\_REQ, k)$. Otherwise, P_i or some process with process number smaller than i will send a message $CM(CK\_BGN, k)$ to P_0. Therefore, P_0 will receive at least one *CK_BGN* message with sequence number k. Then P_0 initiates the process of forwarding *CK_REQ* messages. Similar to **Case(1)**, P_i finalizes the checkpoint $C_{i,k}$ which is a contradiction to our assumption.

Hence the theorem. □

Theorem 2.2 *For each k, the set $S_k = \{C_{i,k} | i \in 0, 1, \cdots, N-1\}$ is a consistent global checkpoint.*

Proof. We prove this by contradiction. Suppose S_k is not consistent. Then, there exists a message M, sent from P_i to P_j (for some $i, j \in \{0, 1, \cdots, N-1\}, i \neq j$), such that $C_{i,k} \xrightarrow{\text{hb}} send(M)\ AND\ receive(M) \xrightarrow{\text{hb}} C_{j,k}$.

Depending on the receiving time of the message M, the following two cases arise.

Case (1) $receive(M) \xrightarrow{\text{hb}} CT_{j,k}$ (a). Since $C_{i,k} \xrightarrow{\text{hb}} send(M)$, $CFE_{i,k} \xrightarrow{\text{hb}} send(M)$ (b). Since P_i has finalized $C_{i,k}$, P_i has known that each process P_j has taken tentative checkpoint $CT_{j,k}$. Therefore, $CT_{j,k} \xrightarrow{\text{hb}} CFE_{i,k}$ (c). From (a), (b) and (c), we have $receive(M) \xrightarrow{\text{hb}} CT_{j,k} \xrightarrow{\text{hb}} CFE_{i,k} \xrightarrow{\text{hb}} send(M)$, i.e., $receive(M) \xrightarrow{\text{hb}} send(M)$, a contradiction.

Case (2) $CT_{j,k} \xrightarrow{\text{hb}} receive(M) \xrightarrow{\text{hb}} CFE_{j,k}$ (a). Similar to **Case (1)**, we have $CFE_{i,k} \xrightarrow{\text{hb}} send(M)$. Upon receiving M, P_j knows that P_i has finalized the checkpoint $C_{i,k}$. Therefore, it knows that all other processes have taken a tentative checkpoint with sequence number k. Based on this information, P_j finalizes the checkpoint $C_{j,k}$ not including message M in the checkpoint. Therefore, we have $CFE_{j,k} \xrightarrow{\text{hb}} receive(M)$ (b). From (a) and (b) we have $receive(M) \xrightarrow{\text{hb}} receive(M)$ which is a contradiction.

Hence the theorem. □

2.4.7 Recovery Algorithm

In this section, we present a recovery algorithm based on the checkpointing algorithm. We make the following assumption for the recovery algorithm.

- At most one process fails at any given time. No other process fails until the recovery due to a failed process is complete.

We need to add the following data structures to the checkpointing algorithm presented in Sections 2.4.4 and 2.4.5.

- Each process P_i has a variable rsn_i, initialized to 0, to keep track of the total number of times recovery took place. Each time P_i initiates recovery, this variable is incremented by 1.

Informal Description of the Recovery Algorithm

When a process P_i fails, it increments rsn_i by 1 and sends $ROLLBACK(rsn_i, csn_i)$ message to all the processes; here csn_i represents the sequence number of the latest finalized checkpoint of the process P_i. When a process P_j receives $ROLLBACK(rsn_i, csn_i)$ message from process P_i, it finalizes the checkpoint with sequence number csn_i if it has not already done so, and then sends $OKTOROLLBACK(rsn_i, csn_i)$ to P_i. After a process sends $OKTOROLLBACK$ message, it blocks (i.e., it neither sends/receives any application message nor does any local computation). After P_i receives $OKTOROLLBACK$

reply from all the processes, it sends $CONFIRMROLLBACK(rsn_i, csn_i)$ message. After a process P_j receives $CONFIRMROLLBACK(rsn_i, csn_i)$ message, it retrieves the finalized checkpoint C with sequence number csn_i, rolls back to the tentative checkpoint with sequence number csn_i stored in C, and replays the messages in the log associated with C and then sends $ROLLBACKFINISHED(rsn_i, csn_i)$ message to P_i and blocks. After P_i receives $ROLLBACKFINISHED(rsn_i, csn_i)$ from all processes, it sends $PROCEED(rsn_i, csn_i)$ message to all the processes. Upon receiving the $PROCEED$ message, each process resumes its computation normally.

Formal description of the recovery algorithm is presented in Figure 2.6.

When P_i fails and initiates recovery process
$rsn_i = rsn_i + 1$;
Sends $ROLLBACK(rsn_i, csn_i)$ to all processes; // csn_i is the sequence number of the latest finalized checkpoint of P_i;

When P_j receives $ROLLBACK(rsn_i, csn_i)$ from P_i
if $rsn_j < rsn_i$ **then** // this is a new recovery initiation
$rsn_j = rsn_i$;
Finalizes the tentative checkpoint with sequence number csn_i
if it has not already done so;
Sends $OKTOROLLBACK(rsn_i, csn_i)$ reply to P_i;
Blocks;

After P_i receives $OKTOROLLBACK(rsn_i, csn_i)$ from all processes
Sends $CONFIRMROLLBACK(rsn_i, csn_i)$ to all processes;

When P_j receives $CONFIRMROLLBACK(rsn_i, csn_i)$ from P_i
Finds the finalized checkpoint C with sequence number csn_i;
Rolls back to the tentative checkpoint contained in C;
Replays the messages in the message log associated with C;
Sends $ROLLBACKFINISHED(rsn_i, csn_i)$ to P_i;
Blocks;

After P_i receives $ROLLBACKFINISHED(rsn_i, csn_i)$ from all processes;
Sends $PROCEED(rsn_i, csn_i)$ to all processes;

When P_j receives $PROCEED(rsn_i, csn_i)$
P_j resumes computation;

Figure 2.6: Recovery algorithm

Correctness of the Recovery Algorithm

A process finalizes its tentative checkpoint with a given sequence number only after it comes to know that all the other processes have taken their tentative checkpoints with the same sequence number. When a process fails, all processes roll back to the checkpoint with

the same sequence number. Note that a checkpoint of a process consists of the saved state of the process (tentative checkpoint) and the log of messages sent and received after the tentative checkpoint was taken and before the tentative checkpoint was finalized. The fact that the checkpoints of all the processes with the same sequence number forms a consistent global checkpoint has been proved in Section 2.4.6. Thus rolling back the processes to checkpoints with same sequence number takes the state of the processes to a state represented by a consistent global checkpoint. However, messages lost due to rollback such as those whose receive event was undone while the corresponding send event has not been undone are not taken care of. They can be handled using sequence number and message logging. Moreover, we do not discuss ways for handling concurrent failures. However, methods similar to the ones used in [44] can be used for handling concurrent failures as well as handling lost messages, duplicate messages and in-transit messages during recovery.

2.5 Performance Evaluation

In this section, we present the performance evaluation of our algorithm. We denote our algorithm as OCML (Optimistic Checkpointing and Message Logging approach) for short. We evaluated our algorithm with respect to the following two aspects: 1) under what scenarios our algorithm converges without using additional control messages and what is the overhead induced by the control messages; 2) how does it perform compared to Vaidya's algorithm [66], which we refer to as Vaidya_Stagger. The comparison focuses on the latency and network contention for accessing stable storage.

2.5.1 Simulation Model

We consider distributed computations running in an environment that has the following features.

- **Network environment**. All processes run on nodes in a local area network (LAN). We assume that the average end-to-end message delay is 5 milliseconds.

- **Clock drift**. We assume that the maximum drift of local clocks at various sites is 100 milliseconds per hour.

- **Simulation time**. It is set to 100 minutes.

- **Checkpoint initiation**. We divide the simulation time into 10-minute intervals. These intervals are called checkpoint intervals. Thus, each process has 10 checkpoint intervals during its life time. Each process chooses the time to take tentative checkpoints randomly in each interval. When control messages are used for convergence, we set the value of timeout for finalizing a checkpoint to be 5 minutes. That is, a process initiates sending control messages if it does not finalize its tentative checkpoint in 5 minutes.

- **Communication model**. We simulated under two types of Checkpoint and Communication Patterns (CCPAT), namely, RANDOM and GROUP, described below:

  - **RANDOM Communication Pattern**: Each process $P_i \in P_0, P_1, \cdots, P_{N-1}$ is able to send an application message to any other process $P_j \in P_0, P_1, \cdots, P_{N-1}$ and $P_i \neq P_j$. The destination of each message m is randomly chosen. Messages sent are uniformly distributed during the entire simulation time of a process.

  - **GROUP Communication Pattern**: Each process $P_i \in P_0, P_1, \cdots, P_{N-1}$ sends/receives messages only to/from its two neighbor processes $P_{(i-1) \bmod N}$ and $P_{(i+1) \bmod N}$. This basically means that processes are logically arranged in a ring and each process sends messages only to its two neighbors.

    We choose these two CCPATs mainly because they are representatives of many long-running, compute-intensive applications [22]. For example, in the implementation of Gaussian elimination, in each iteration, a process receives a row

of the matrix from its predecessor and sends the results of its computation to its successor. Its communication model among processes is fits into our GROUP Communication Pattern. Moreover, these two models have been regarded as two extreme representatives for distributed applications in [17]. So we ran our simulations under these two extreme models to evaluate the performance of our algorithm. In all the simulation runs, we varied the rate of messages sent per second by each process from 0.01 to 0.40, on average. Our goal is to study not only the number of control messages needed under sparse communication pattern but also the network contention for accessing stable storage under dense communication pattern.

2.5.2 Simulation Results

In this section, we first present our simulation results regarding (i) under what scenarios our algorithm converges without using additional control messages and (ii) what is the overhead induced by the control messages. We also evaluate the number of messages logged for the purpose of determining consistent global checkpoint. Then we compare the performance of our algorithm with the algorithm of Vaidya.

1. **OCML with control messages vs. OCML without control messages**

   We evaluated the performance of our algorithm with control messages and without control messages under the RANDOM communication model. We simulated a distributed computation involving 20 processes. Figure 2.7(a) shows the number of finalized global checkpoints for various message patterns. Ideally, our algorithm should take 10 consistent global checkpoints since the simulation time is 100 minutes and the checkpoint interval is 10 minutes. Irrespective of the rate at which messages are exchanged, our algorithm takes exactly 10 consistent global checkpoints if control messages are used. This verifies that the use of control messages helps in convergence, especially when application messages are exchanged at a low rate. However,

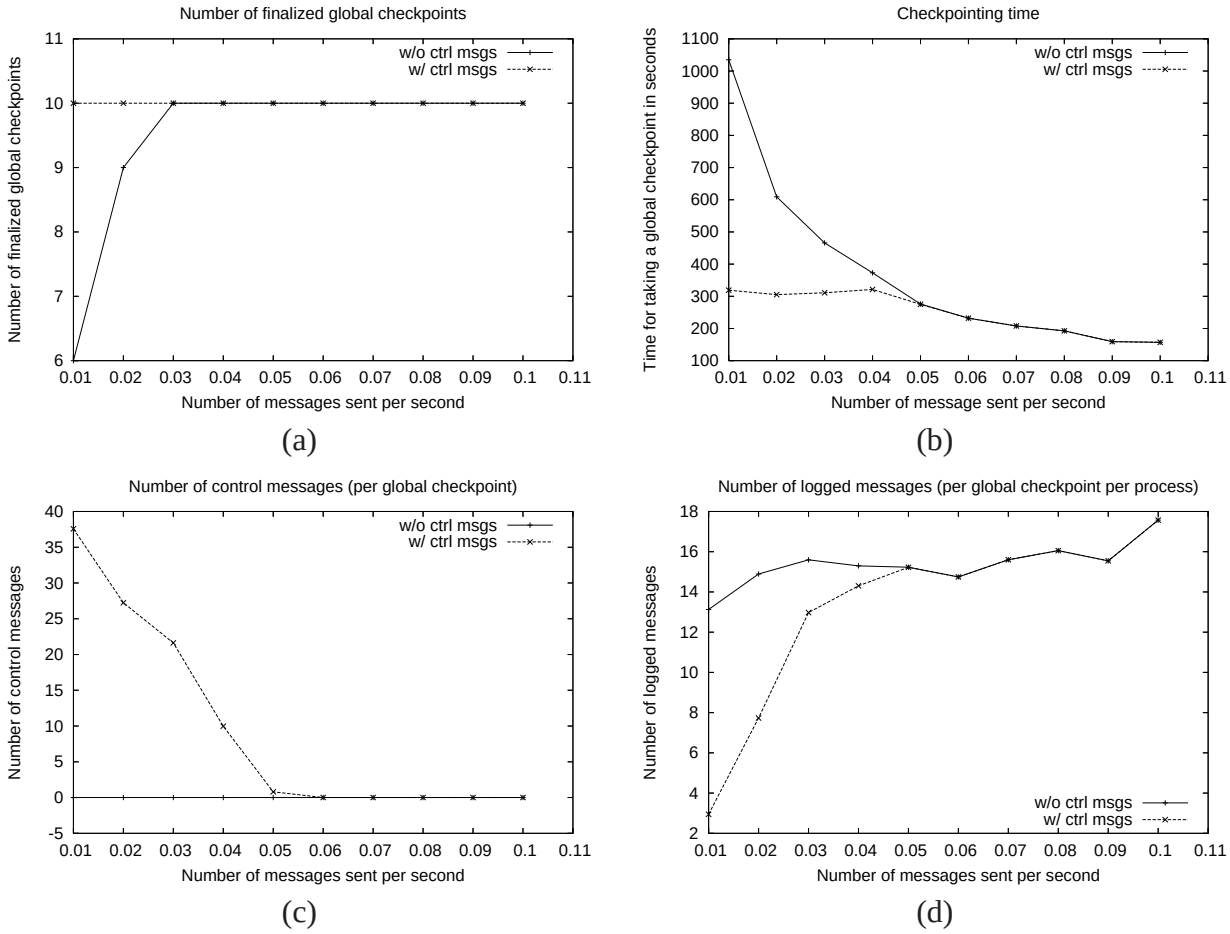


Figure 2.7: Statistics by varying number of messages sent per second

without control messages, only 6 consistent global checkpoints are finalized if each process sends only 0.01 messages per second. This means that processes have to wait for a long time for finalizing a checkpoint. As the rate of messages sent per second by each process increases, our algorithm converges quickly; it only requires 0.03 messages or more per second to converge without any control messages.

Figure 2.7(b) shows the average amount of time (in seconds) needed for taking a consistent global checkpoint, this time being calculated from the time some process initiates consistent global checkpointing to the time at which all processes finalize their tentative checkpoints belonging to this global checkpoint. The average time for taking a consistent global checkpoint is a little more than 300 seconds if less than

0.05 messages are sent by each process per second, in which case control messages are used. If more than 0.05 messages are sent by each process per second, processes finalize their tentative checkpoints before the timer expires. Therefore, no control messages is sent in this case. Figure 2.7(c) verifies this observation. We also note that the number of control messages sent are less than 2 times the number of processes even when only 0.01 messages are sent by each process per second.

Figure 2.7(d) shows the number of logged messages for each global checkpoint at each process. In the figure, the number of logged messages for the case when no control message is sent does not change much as the rate of messages sent per second by each process increases. This also reveals the approximate number of messages needed for the convergence of our algorithm under this communication model. Since the logged messages contain messages sent and received at each process, our algorithm requires each process send only 6 to 9 messages per checkpoint interval for it to converge when 20 processes are involved.

2. **Performance of our algorithm compared to Vaidya's algorithm**

   Next, we present the performance analysis of our algorithm (denoted as OCML) compared to Vaidya's staggered checkpointing algorithm [66] (denoted as Vaidya_Stagger) in this section. We choose Vaidya's algorithm mainly because (1) it represents the staggered checkpointing algorithms which attempt to prevent two or more processes take checkpoints at the same time in order to reduce contention for accessing stable storage; (2) to our knowledge, it is the only algorithm that tries to stagger checkpoints to prevent contention for accessing stable storage; (3) moreover, Vaidya's notion of "physical checkpoint + message log = logical checkpoint" [66], is similar to our notion of "tentative checkpoints + message log = finalized checkpoint".

   We compare the performance of our algorithm with Vaidya's algorithm [66], under both RANDOM and GROUP communication models.

First, we compare our algorithm with Vaidya's algorithm with respect to the average number of checkpoints (note here checkpoints refer to physical checkpoints under Vaidya's algorithm and tentative checkpoints under our algorithm respectively) taken at the same time by each process. Table 2.1 shows the results as the rate of messages sent per second by each process varies from 0.01 to 0.10. Since Vaidya's algorithm successfully staggers all physical checkpoints, the average number of physical checkpoints taken at the same time under all cases for this algorithm are zero. However, this goal has been achieved at the cost of large increase in checkpoint latency in Vaidya's algorithm [66]. On the other hand, although the average number of tentative checkpoints taken at the same time in our algorithm is not zero, since each process is able to store the tentative checkpoint in memory first and choose its convenient time for writing the tentative checkpoints to stable storage at the network file server, it doesn't incur any contention for stable storage in the tentative checkpointing phase of our algorithm while at the same time decreasing the checkpoint latency.

Table 2.1: Physical checkpoints taken by Vaidya_Stagger vs. tentative checkpoints taken by OCML

Average number of checkpoints taken at the same time in each process											
# messages/Sec		0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
RANDOM	OCML1[a]	3.15	4.65	5.1	4.95	6.3	6.4	7.25	7.25	7.2	7.4
	OCML2[b]	4.8	4.6	4.65	5.3	6.3	6.4	7.25	7.25	7.2	7.4
	Vaidya[c]	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
GROUP	OCML1	4.15	6.25	8.5	8.15	7.95	7.7	7.95	7.15	7.3	6.75
	OCML2	8.45	9	8.4	8.2	8.2	8.2	7.4	7.5	7.3	6.75
	Vaidya	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

[a] OCML algorithm without control messages
[b] OCML algorithm with control messages
[c] Vaidya_Stagger algorithm [66]

Next, we compare the performance of our algorithm with Vaidya_Stagger with respect to the number of logged messages under both RANDOM and GROUP communication models. Under the RANDOM communication model, Figure 2.7(d) shows the number of logged messages under OCML with CtrlMessages and OCML without CtrlMessages. Figures 2.8(a) and 2.8(c) show the performance results of our algorithm compared to Vaidya_Stagger under RANDOM and GROUP communication

models respectively, as the rate of messages sent per second by each process varies from 0.02 to 0.20. Figure 2.8(d) shows the result under GROUP model, as the rate of messages sent per second by each process varies from 0.20 to 0.40. As expected, under both communication models, when the rate of messages sent per second by each process increases, our algorithm converges fast and doesn't need control messages. Under RANDOM model, as the rate of messages sent per second by each process increases, the number of logged messages in our algorithm is always smaller than that of Vaidya_Stagger. Under the GROUP communication model, the number of logged messages under our algorithm continues to be smaller than that of Vaidya_Stagger if the rate of messages sent per second by each process is larger than 0.08. Figure 2.8(b) shows how the number of logged messages changes with respect to the number of processes involved in the computation under RANDOM model. The results indicate a linear increase in the number of logged messages in Vaidya_Stagger with respect to the number of processes. On the other hand, increase in the number of processes has only slight impact on the number of logged messages in our algorithm, which indicates that our algorithm is more scalable.

Finally, under both RANDOM and GROUP communication models, we compare our algorithm and Vaidya_Stagger with respect to the contention for stable storage at the network file server that arises due to storing logged messages. Figures 2.9(a) and 2.9(c) show the results under RANDOM and GROUP communication models respectively, as the rate of messages sent per second by each process varies from 0.02 to 0.20. Figure 2.9(d) shows the result under GROUP communication model, as the rate of messages sent per second by each process varies from 0.20 to 0.40. Since in the second phase of Vaidya_Stagger, each process takes its logical checkpoint by logging messages on stable storage after receiving the *marker* message from the coordinator, it means that the coordinator plays the centralized role of synchronizing the message-logging in each process and it may lead to a single point of failure. It

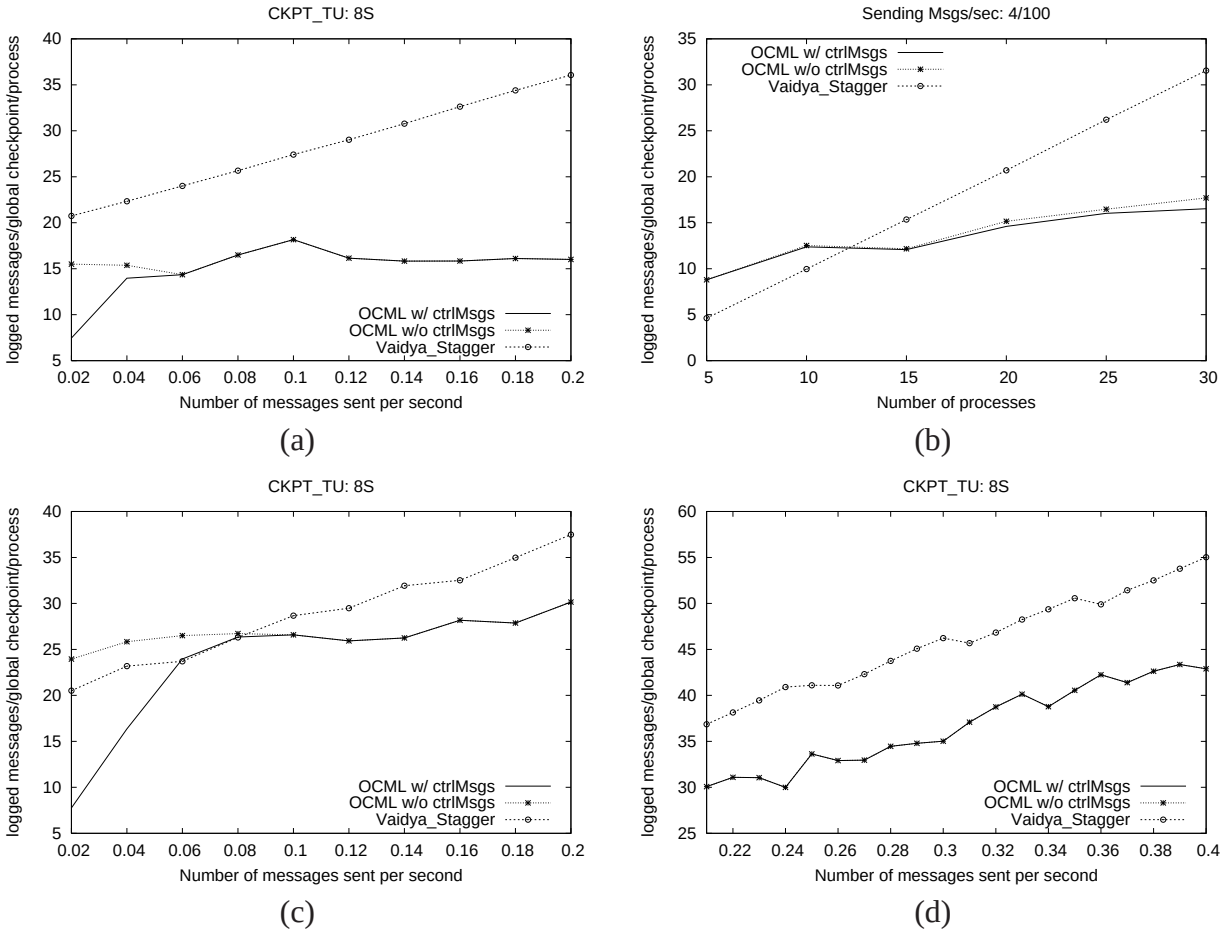


Figure 2.8: Number of logged messages under OCML and Vaidya_Stagger

completely staggers the physical checkpoints, however, contention for access to stable storage still occurs while storing logged messages [66]. As a result, the number of collisions due to logged messages in each process is the same as the number of logical checkpoints taken at each process in Vaidya_Stagger. However, in our algorithm, under the RANDOM model, Figure 2.9(a) shows the average number of collisions due to logged messages is 3.6 without CtrlMessage, which is 64% less than that of Vaidya_Stagger. Under the GROUP communication model, as shown in Figure 2.9(d), as the rate of messages sent by each process varies from 0.21 to 0.40 per second, the average number of collisions due to logged message is 6.3 for both OCML with CtrlMessages and OCML without CtrlMessages, which is 37% less

than that of Vaidya_Stagger. Figure 2.9(b) shows how the number of collisions due to logged messages changes with respect to the number of processes involved in the computation under RANDOM model. As expected, when the number of processes increases, the number of collisions due to logged messages under our algorithm only has slight impact and it is at least 60% less than that of Vaidya's algorithm.

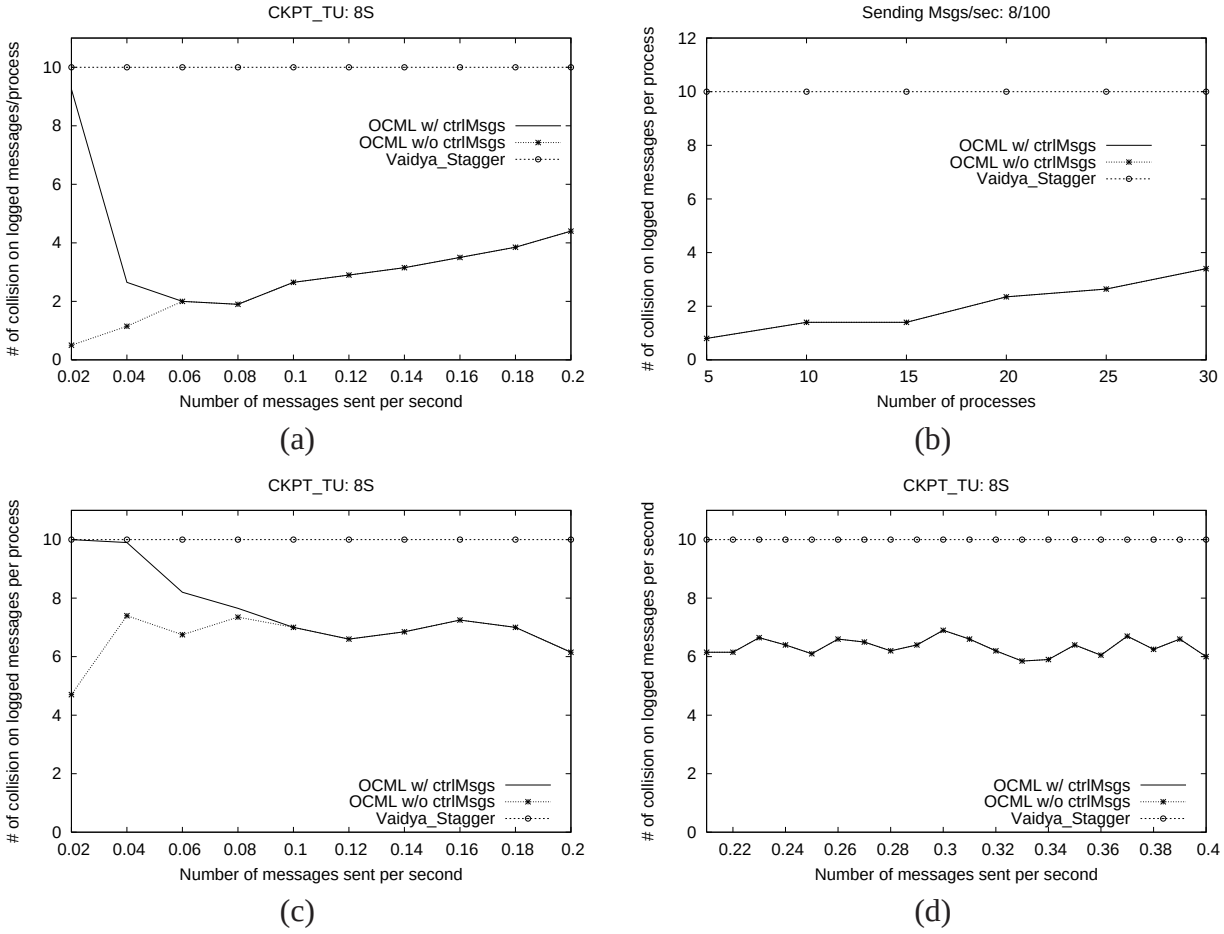


Figure 2.9: Number of collisions due to storing logged messages at the network file server under OCML and Vaidya_Stagger

Vaidya's algorithm [66] successfully staggers all physical checkpoints so that no contention for stable storage occurs while storing physical checkpoints. However, it does incur contention for stable storage when messages are logged in its second phase. Compared to Vaidya_Stagger, although the average number of tentative checkpoints

taken at the same time under our algorithm is not zero, it doesn't incur any contention for stable storage since each process is able to store the tentative checkpoint in memory first and choose its convenient time for writing the tentative checkpoints to stable storage at the network file server. For example, based on our simulation results, we can choose to save the tentative checkpoint together with its corresponding logged messages at the same time when it is finalized or earlier when there is no contention for stable storage. In reducing contention for stable storage at the network file server, our algorithm always performs better than Vaidya_Stagger. And our algorithm also has other desirable features such as low control messages (or even no control messages) and less checkpoint latency compared to Vaidya_Stagger algorithm. Moreover, our algorithm is distributed whereas Vaidya's algorithm is centralized.

2.6 Conclusion

In this chapter, we presented a novel *communication-induced* checkpointing algorithm that makes every checkpoint belong to a consistent global checkpoint. Under this algorithm, every process stores the tentative checkpoint in memory first and then flushes it to stable storage when there is no contention for accessing stable storage or after finalizing the tentative checkpoint. Messages sent and received after a process takes a tentative checkpoint are logged into memory until the tentative checkpoint is finalized. Since a tentative checkpoint can be flushed to stable storage any time before finalizing it, contention for stable network storage that arises due to several processes storing the checkpoints simultaneously is reduced/eliminated. Moreover, unlike existing communication-induced checkpointing algorithms, our algorithm, in general, does not force a process to take a checkpoint before processing any received message in order to prevent useless checkpoints. Thus, a process can first process the received message and then take the checkpoint. This improves the response time for messages. It also helps a process take the regularly scheduled basic checkpoints at those times. If messages are not frequently exchanged among processes, additional control

messages may be required for the algorithm to collect consistent global checkpoints in a timely manner. We augmented the basic algorithm with control messages to speed up the collection of consistent global checkpoints in a timely manner for applications in which processes do not communicate frequently. We conducted a performance evaluation of the algorithm and studied the overhead induced by the control messages which also helps in determining when control messages are needed. We also compared the performance of our algorithm with Vaidya's algorithm [66]. In reducing the contention for stable storage at the network file server, our algorithm always performs better than Vaidya's algorithm. Our algorithm also has other desirable features such as the scalability, low control messages (or even no control messages) and less checkpoint latency compared to Vaidya's algorithm algorithm.

Chapter 3

Triangle-based Routing for Mobile ad hoc Networks

3.1 Introduction

In recent years, mobile ad hoc (MANET) and wireless sensor networks (WSN) have attracted a lot of attention. These networks are composed of mobile nodes which communicate with each other wirelessly without the support of any fixed infrastructure. Unlike traditional networks, mobile ad hoc and wireless sensor networks do not have dedicated routers. Each participating node acts as an end system as well as a router. A node may directly communicate with its immediate neighbors within its transmission range. When two nodes that are not within the transmission range of each other need to communicate with each other, intermediate nodes act as routers to forward the packets. The design of efficient routing algorithm for mobile ad hoc and wireless sensor networks could be challenging due to the infrastructureless nature.

Routing algorithms for mobile ad hoc and sensor networks can be classified into two categories: topology-based and position-based. Topology-based routing algorithms use the information of the existing links in the network to route packets. Examples of topology-based routing algorithms are AODV [56], WRP [47], DSR [29], and DSDV [54]. In topology-based routing algorithms, a node typically floods route request message in the network to find a route to a given destination node. Position-based routing algorithms use

the geographic position information of nodes in the network to perform packet forwarding. Examples of position-based routing algorithms are Compass [37], MFR [63], Face-2 [7], GPSR [31], and AFR [38].

Under topology-based routing, a node wishing to establish a route to a destination broadcasts a route request message; each node receiving this route request message re-broadcasts this request once and this process is repeated by every node in the network except the destination node which upon receiving the route request broadcasts a route reply and route reply travels along the path travelled by the route request in the reverse direction and reaches the source which initiated the route request. This approach leads to great number of redundant rebroadcasting of route request messages. In dense networks, this duplication may result in high network contention, high network load, and high network delay. To reduce the number of redundant messages, many algorithms have been developed. They use different graph models such as unit disk graph [10, 14], relative neighborhood graph (RNG) [12, 58, 62, 64], and dominating sets [6, 70, 71]. However, these algorithms do not work well for networks with mobile nodes.

With these considerations in mind, we propose an algorithm that reduces the redundant rebroadcasting of route request messages. In the proposed algorithm, we assume that all nodes lie in the same plane and they all have the same transmission range R. We divide the plane into a number of equilateral triangular regions as shown in Figure 3.1. Each triangular region is assigned a unique identifier called Absolute Location Identifier (ALI). All nodes in a triangular region know the identifier and exchange it with their neighbors periodically. This way, each node in the network has a knowledge about the approximate location of its neighbors. Based on this, a node b is able to decide whether and when to forward a received route request message. Therefore, redundant messages are greatly suppressed when the knowledge is updated in a timely manner and used appropriately. Before explain this in detail, we outline related works in Section 3.2 followed by the algorithm preliminaries in Section 3.4. We then present the algorithm in Section 3.5. Simulation results are discussed

in Section 3.6. Section 3.7 concludes this chapter.

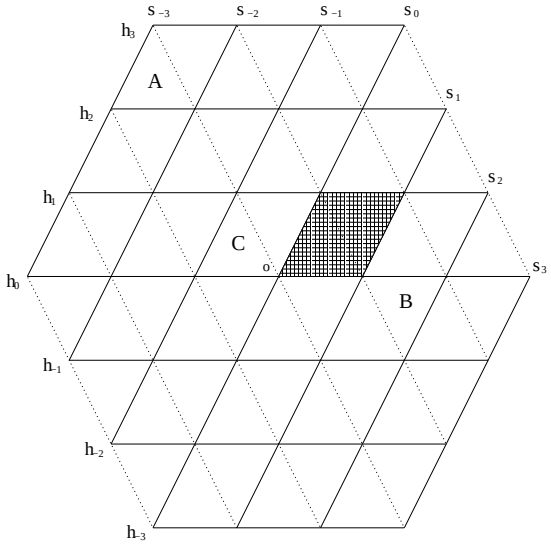


Figure 3.1: Plane divided into triangular regions

3.2 Related Works

Routing algorithms in mobile ad hoc networks [9, 15, 18, 23, 29, 51, 54–56, 67] have been extensively studied in recent years. Many topology-based routing algorithms for mobile ad hoc and sensor networks use a simple broadcasting mechanism that floods the entire network with route request messages, which leads to redundant propagation of route-request messages, contention, and collision. Well known algorithms such as AODV [56], DSR [29], DSDV [54] and TORA [51] use this flooding approach. Broch et al. [9] studied the performance of DSDV, TORA, DSR, and AODV. Their results show that the routing overhead of these algorithms increases quickly as the number of nodes in the network increases.

A Dynamic MANET On-demand (DYMO) [55] routing algorithm, a descendant of AODV and DSR, was proposed by Perkins et al. [55], which is suitable for sparse networks. TBRPF [50] and OLSR [15] are suitable for networks in which a large number of routes are needed and for applications that can not tolerate the delay due to route discovery. However, TBRPF reports updates reactively when a link state changes while OLSR reports them

periodically. Therefore, TBRPF and OLSR may not work well in networks where nodes move quickly. In such a scenario, TBRPF may send a large number of updates into the network and nodes may have too many outdated links in its route table if OLSR is used.

The Zone Routing Protocol (ZRP) [23] uses a hybrid approach for maintaining routes. Under this algorithm, each host proactively updates its routing table for all destinations within its zone. For destinations outside its zone, a node employs a reactive approach to find routes on demand.

Some routing algorithms use a connected dominating set [67] as a backbone network to minimize the number of nodes that participate in forwarding route-request packets, and hence reduce overlapping route-request propagation. A disadvantage of this approach is that the selected "core" or "backbone" nodes may drain their battery quickly. A solution to overcome this problem is to periodically change the set of "backbone" nodes. However, the complexity of computing an approximate minimal dominating set of a wireless network (computing a truly minimal dominating set is known to be NP-complete) may result in high overhead. Moreover, maintaining this dominating set may incur large overhead if nodes are highly mobile.

Position-based routing algorithms [4, 5, 8, 27, 32, 35] have been proposed to limit the propagation of redundant route-request messages during route discovery. Unlike usual greedy position-based algorithms, NADV [40] takes both distance and link cost (measured in terms of delay, power consumption, or other metrics) into account in forwarding data packets. The main drawback of position-based algorithms is that it requires every node know the position of the destination to which it needs a route, which would require additional location service.

Other algorithms also try to reduce redundant propagation of route request packets [49, 52,53]. Williams et al. [69] classify broadcasting techniques into simple flooding, probability-based [49] flooding, area-based [49] flooding, and neighbor knowledge-based [52, 53] flooding. Other algorithms use pruning methods such as self pruning and dominant pruning

to minimize redundant propagation of packets [52, 53].

The basic idea behind many of the position-based routing algorithms [4, 5, 8, 27, 32, 35] is to limit the search for the destination to a portion of the network based on estimating the location of the destination based its last known position and velocity or with the help of a location service. Extra overhead is incurred when the estimation turns out to be incorrect. These algorithms require each node in the network to know its own position and the position and velocity of every other node at some point in time. This information is not practical to maintain in a real ad hoc network environment. Moreover, each node in the search range is required to forward route-request packets, which can result in propagating redundant route-request messages. Our algorithm addresses both problems. It only requires each node to know the relative position of nodes in its neighborhood. A node trying to establish a route to a destination does not need to know the position or velocity of the destination.

3.3 Basic Idea Behind Our Algorithm

Our aim is to reduce the redundant rebroadcasting of route request messages during route discovery. To achieve this goal, we require:

- Each route request message carry the information about what nodes have been already covered by the route request.
- Each node has its two-hop neighbor information.

With the above information, a node is able to make informed decision regarding whether or not to forward a received route request message. However, these requirements are not practical considering the message size and the overhead involved in obtaining two-hop neighbor information. With these considerations, we require:

- Each route request message carry information about which area has already been covered and

- Each node has its one-hop neighbor information.

To accomplish this, we split the network area into triangular regions as shown in Figure 3.1; we also present a method for assigning addresses to each of the triangular regions in Section 3.4.

3.4 Preliminaries

In this section, we present methods for assigning fixed as well as relative address to triangular regions. For this, we introduce several terms and data structures used in the algorithm. They include Absolute Location Identifier (ALI), Relative Location Identifier (RLI), and bit vectors.

3.4.1 Absolute Location Identifier

We assume that the nodes move in a planar area. We divide the planar area into a number of equilateral Triangular Areas (TAs) as shown in Figure 3.1. We assign each TA a unique identifier called Absolute Location Identifier (ALI). Two TAs that share a side make up a rhombus. Without loss of generality, we only take into account the rhombuses whose sides are shown with solid line segments in Figure 3.1. We assign ALIs to TAs in two steps. We first assign ALIs to rhombuses and then we assign ALIs to TAs based on the ALIs of the rhombuses.

The ALI of a Rhombus

Given that all rhombuses are of same size and shape, the coordinates of any one of the vertices of a rhombus uniquely identifies the rhombus. We use the coordinates of the left-bottom corner of a rhombus to identify the rhombus. For instance, point o uniquely identifies the shaded rhombus in Figure 3.1. Hereafter, we will identify a rhombus by the coordinates of its left-bottom vertex. We next describe the coordinate system used for identifying the vertices of rhombuses.

Similar to rectangular Cartesian coordinate system, we choose the left-bottom corner of one rhombus as origin (point o in Figure 3.1). The X-axis is the horizontal line passing through the origin o and the Y-axis is the slant line (which makes 60 degrees with the X-axis) passing through the origin. With reference to these two axes, any point in the plane can be represented by an ordered pair of real numbers (s, h). We divide the network area into rhombuses so that the coordinates of their vertices are integers as shown in Figure 3.1. The coordinates (s, h) assigned to the left bottom vertex of a rhombus is called the ALI of the rhombus. Next, we discuss how we assign an ALI to a TA.

The ALI of a TA

A rhombus is split into two TAs by one of its diagonal lines, shown as dotted lines in Figure 3.1. We assign ALIs to each of the two TAs belonging to the rhombus with ALI (s, h) as follows. The ALIs of the TAs belonging to the rhombus with ALI (s, h) are of the form (s, h,$flag$) where $flag$ is 0 for the left TA and 1 for the right TA. For example, $(-3, 2, 0)$ and $(1, -1, 1)$ are the ALIs of TAs A and B in Figure 3.1 respectively.

Transformation from a Coordinate to an ALI

How to find the ALI of the rhombus that contains a given point? Suppose the length of each side of a TA is the transmission range R and the location of a node b is (x_b, y_b). Let the point (x_0, y_0) in the plane be the origin point. We show how node b computes the ALI (s, h) of the TA in which it lies. Then, the two equations in Equation 3.1 represent the horizontal solid line and slant solid line bounding the rhombus containing the point (x_b, y_b) in Figure 3.1.

$$\begin{cases} y &= y_0 + \frac{\sqrt{3}Rh}{2} \\ y &= y_0 + \sqrt{3}(x - x_0 - Rs) \end{cases} \tag{3.1}$$

Then the coordinates of the left bottom vertex of the rhombus containing the point (x_b, y_b), namely (s, h) are given by Equation 3.2.

$$\begin{cases} h &= \left\lfloor \frac{2(y_b - y_0)}{\sqrt{3}R} \right\rfloor \\ s &= \left\lfloor \frac{x_b - x_0}{R} - \frac{y_b - y_0}{\sqrt{3}R} \right\rfloor \end{cases} \tag{3.2}$$

Each rhombus is divided into two TAs by the slanted dotted line. We identify them using a flag defined by Equation 3.3. The TA on the left/right has the flag of $0/1$.

$$flag = (\sqrt{3}(x_b - x_0) + y_b - y_0 - \sqrt{3}R(h + s + 1) > 0) \tag{3.3}$$

This way, given the coordinates of a node with respect to the origin (x_0, y_0), any node in a given TA is able to determine the ALI in which the TA lies as it knows the transmission range R. Therefore, each node is able to compute the coordinates of the ALI of the TA in which it lies. In the rest of this chapter, we refer the ALI of a node to be the ALI of the TA in which the node lies.

The Representation of an ALI

To reduce the overhead involved in exchanging information about ALIs, we use 32-bit integers to represent them. Figure 3.2 shows how the three fields of an ALI are stored in a 32-bit integer.

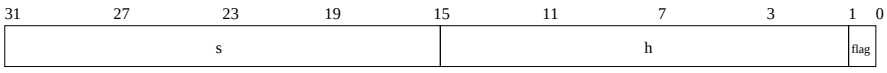


Figure 3.2: Representation of an absolute location identifier (ALI)

One might wonder if this representation of ALIs limits the network area. However, this is not the case for the following reasons. Suppose the transmission range R is 250 meters(m) and the origin is the center of the network. Clearly, there are $2^{16} \times 2^{15} \times 2$ TAs and the area of each TA is $\frac{\sqrt{3}R^2}{4}$. Therefore, this representation is able to cover a network area of size upto $10,781,278m \times 10,781,278m$, which is large enough for mobile ad hoc networks.

3.4.2 Relative Location Identifiers

The bandwidth used for exchanging location information with neighbors may be high when nodes use ALIs to represent their physical locations. Suppose a node has neighbors that lie in 16 different TAs. It has to use a message with length over 16×4 bytes to let its neighbors know which TAs contain its neighbors. Therefore, we define a new term Relative Location Identifier (RLI) to identify neighboring TAs. We show how RLIs help in saving network bandwidth for communication in Section 3.4.5.

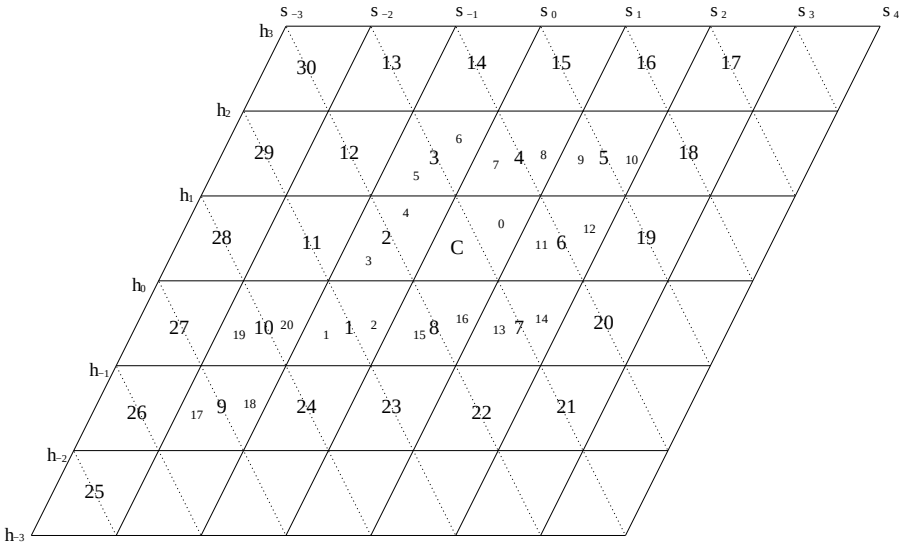


Figure 3.3: Assigning RLTs to neighboring TAs

A RLI is a unique nonnegative integer assigned by a TA, say C in Figure 3.3, to another near TA. In this section, we describe how RLIs are assigned by TA C to other TAs. First of all, TA C picks the TA that lies in the same rhombus and assigns it a RLI of 0. Then TA C assigns RLIs to other TAs in two steps:

Step 1 Assign RLIs to rhombuses (the rhombus that contains TA C is excluded). We first define a new term. The distance between two rhombuses is the max distance between the lines that are parallel to one of the sides and go through the centers of the rhombuses respectively. For instance, the distance between rhombus $(-1, 3)$ and $(-2, -2)$ is 4 times the height of a rhombus.

Then, TA C puts the rhombuses into a number of groups based on their distance to the rhombus that contains TA C. Clearly, each group of the rhombuses forms a ring of rhombuses. Each ring is assigned an integer that is the max distance between the rhombuses on the ring and the rhombus containing TA C, divided by the height of a rhombus. This way, the rings from the nearest to the furthest are assigned numbers $1, 2, 3, \cdots$ respectively. It is easy to see that the i^{th} ring contains $8i$ rhombuses.

Finally, TA C counts rhombuses on the i^{th} clockwise ring one by one starting from the rhombus $(s_C - i, h_C - i)$ where (s_C, h_C) is the ALI of the rhombus that contains the TA C. After finishing counting, TA C assigns RLI $4i(i-1)+j$ to the j^{th} rhombus on the i^{th} ring.

Step 2 Assign RLIs to TAs based on the RLIs of rhombuses. Suppose the RLI of a rhombus is i. Then TA C assigns the left and right TAs in the rhombus with RLI $2i - 1$ and $2i$ respectively.

Figure 3.3 shows how TA C assigns RLIs to the neighboring rhombuses and TAs. The numbers in the larger font size are the RLIs assigned to the rhombuses while the numbers in the smaller font size are the RLIs assigned to the TAs. Clearly, TA C uniquely assigns RLIs to TAs using consecutive integers starting from 0. We discuss why this is important in saving network bandwidth for exchanging neighborhood information in Section 3.4.5.

3.4.3 Transformation between ALIs and RLIs

RLIs are identifiers assigned by a TA (a node) to its neighboring TAs. According to the rules for assigning RLIs, different TAs (nodes) may assign different RLIs to the same TAs. Therefore, a RLI needs its assigner to uniquely identify a TA. RLIs assigned to TAs are relative to a TA but ALIs are global identifiers of TAs. We give equations for determining ALI of a TA from its RLI relative to another TA and vice versa.

We first give the equations for transforming a RLI to an ALI. Suppose a TA B $(s_b, h_b, flag_b)$ assigns its neighboring TA D $(s_d, h_d, flag_d)$ a RLI of rli. The question becomes how to

represent s_d, h_d and $flag_d$ using s_b, h_b, $flag_b$ and rli. If rli is 0, we have $(s_d, h_d, flag_d) = (s_b, h_b, !flag_b)$. Otherwise, more effort is needed to determine the ALIs. As shown in Figure 3.3, a ring (based on its definition given in Section 3.4.2) of rhombuses has four (left, top, right and bottom) wings. Let the left, top, right and bottom wings are the 0^{th}, 1^{st}, 2^{nd} and 3^{rd} wings respectively, and TA D lie in the k^{th} rhombus on the j^{th} wing of the i^{th} ring around the rhombus containing TA B. We give an example in Figure 3.3 to show how we use k, j, and i here. Rhombus $9/4/20/24$ (in the larger font) is the $0^{th}/1^{st}/3^{rd}/3^{rd}$ rhombus on left/top/right/bottom ($0^{th}/1^{st}/2^{nd}/3^{rd}$) wing on the $2^{nd}/1^{rd}/2^{nd}/2^{nd}$ ring. As we showed in Section 3.4.2, the i^{th} ring is made up of $8i$ rhombuses. Therefore, $4i(i-1) < rli \leq 4i(i+1)$, namely $\frac{\sqrt{rli+1}-1}{2} \leq i < \frac{\sqrt{rli+1}+1}{2}$ since $i \geq 0$. We have $i = \left\lceil \frac{\sqrt{rli+1}-1}{2} \right\rceil$ since i is also an integer and $\frac{\sqrt{rli+1}+1}{2} - \frac{\sqrt{rli+1}-1}{2} = 1$. Clearly, the rli in the i^{th} ring starts with $4i(i-1)+1$ and each wing has $2i$ rhombuses. Therefore, the wing number $j = \left\lfloor \frac{rli-4i(i-1)-1}{2i} \right\rfloor$. Similarly, $k = [rli - 4i(i-1) - 1]\,\%(2i)$. Therefore, we have:

$$\begin{cases} i &= \left\lceil \frac{\sqrt{rli+1}-1}{2} \right\rceil \\ j &= \left\lfloor \frac{rli-4i(i-1)-1}{2i} \right\rfloor \\ k &= [rli - 4i(i-1) - 1]\,\%(2i) \end{cases} \tag{3.4}$$

Since the ALI of the center rhombus of the ring is (s_b, h_b) (obtained from the ALI of TA B), we have:

$$\begin{cases} (s_d, h_d, flag_d) = (s_b - i,\ h_b - i + k,\ (rli+1)\%2), & \text{if } j = 0 \\ (s_d, h_d, flag_d) = (s_b - i + k,\ h_b + i,\ (rli+1)\%2), & \text{if } j = 1 \\ (s_d, h_d, flag_d) = (s_b + i,\ h_b + i - k,\ (rli+1)\%2), & \text{if } j = 2 \\ (s_d, h_d, flag_d) = (s_b + i - k,\ h_b - i,\ (rli+1)\%2), & \text{if } j = 3 \end{cases} \tag{3.5}$$

We are able to obtain the ALI of a TA given its RLI and the assigner's ALI using Equation 3.4 and 3.5. Next, we present how a TA determines RLIs from ALIs. This question can be described as how to represent the RLI rli assigned by TA B to TA D in terms of TA D's ALI $(s_d, h_d, flag_d)$ and TA B's ALI $(s_b, h_b, flag_b)$. Again, let TA D lie in the k^{th} rhombus on the j^{th} wing of the i^{th} ring. Clearly the rhombus number is

$4i(i-1)+2ij+k+1$. Then we have:

$$rli = 8i(i-1)+4ij+2k+flag_d+1 \tag{3.6}$$

Based on the definition of the ring and the rules for numbering j and k, we have:

$$\begin{cases} i = \max\left(|s_b - s_d|, |h_b - h_d|\right) \\ (j,k) = \begin{cases} (0,\ h_d - h_b + i) & \text{if } s_b - s_d \geq |h_b - h_d| \text{ and } h_d - h_b \neq i \\ (1,\ s_d - s_b + i) & \text{if } |s_b - s_d| \leq h_d - h_b \text{ and } s_d - s_b \neq i \\ (2,\ h_b - h_d + i) & \text{if } s_d - s_b \geq |h_b - h_d| \text{ and } h_b - h_d \neq i \\ (3,\ s_b - s_d + i) & \text{if } |s_b - s_d| \leq h_b - h_d \text{ and } s_b - s_d \neq i \end{cases} \end{cases} \tag{3.7}$$

Thus, any TA is able to compute RLIs of its nearing TAs from their ALIs using Equation 3.6 and 3.7.

3.4.4 Notations

Before we further discuss the preliminaries and the algorithm, we outline the notations used in the description of the algorithm:

- TA_i refers to the TA whose ALI is i.
- TA_{node_a} refers to the TA in which node a lies.
- $TA_{i,j}$ refers to the TA with RLI of j assigned by TA_i.
- $TA_{node_a,j}$ refer to $TA_{i,j}$ where node a lies in TA_i.
- NTA_i refers to the set of TAs that share one or more vertices with TA_i. And $NTA_{i,j}$ refers to the set of TAs that share one or more vertices with $TA_{i,j}$. For example, NTA_C (let C stand for an ALI) in Figure 3.3 contains $TA_{C,0}$, $TA_{C,2}$, $\cdots$, $TA_{C,7}$, $TA_{C,11}$, $TA_{C,13}$, $\cdots$, and $TA_{C,16}$.
- NTA_{node_a} and $NTA_{node_a,j}$ refer to NTA_i and $NTA_{i,j}$ respectively where node a lies in TA_i.

- C_{node_a} refers to the set of TAs that contains one or more direct neighbors of node a. Suppose TA_{node_a} contains nodes $a, a_1, \cdots$, and a_n, and no others. Then $C_{TA_{node_a}}$ refers to $C_{node_a} \cup C_{node_{a_1}} \cup \cdots \cup C_{node_{a_n}}$. Therefore, $C_{node_a} \subseteq C_{TA_{node_a}}$.

- C_{TA_i} refers to $C_{TA_{node_a}}$ when node a lies in TA_i. C_{TA_i} is empty if no node lies in TA_i.

- $C_{TA_{node_a,j}}$ refers to C_{TA_i} where i is the ALI of $TA_{node_a,j}$.

3.4.5 Bit Vectors

So far, we have introduced two new terms, ALI and RLI. Like absolute and relative path in file systems, an ALI uniquely specifies a group of nodes that lie in the same TA while a RLI is a label assigned to neighboring TA. In the algorithm, we employ ALIs and RLIs to exchange information between neighbors. An ALI used is a 32-bit integer while a RLI could be a very small integer. As we mentioned earlier, we assume that the length of each side of each TA is same the transmission range of the nodes. As shown in Figure 3.3, a node inside TA C may reach some nodes lying in the rhombuses in the 2^{nd} ring but not any ones in 3^{rd} or beyond. Similarly, one hop neighbors of the node in TA C may reach some nodes lying in the rhombuses in the 3^{rd} ring but not any ones in the 4^{th} or beyond. The greatest rhombus number in the $2^{nd}/3^{rd}$ ring is $24/48$ and hence the greatest RLI of a TA in the ring is $48/96$. Therefore, a node can never have a one-hop (two-hop) neighbor that lies in a TA whose RLI is greater than 48 (96). Therefore, a 6-bit (7-bit) RLI is good enough to specify which TAs form a node's one-hop (two-hop) neighbor(s). How does this serve the algorithm? Before answering this question, we briefly describe bit vectors first.

A bit vector is essentially a vector of boolean values. We often use it to represent a set since it is optimized for space efficiency. Many set representations require one byte or more per element while a bit vector needs only one bit per element. A drawback of this representation is that the bit vector could be huge if there are a large number of possible elements. Next we explain how bit vector is used in the algorithm.

Under the algorithm, a node is able to determine whether or not to forward a route request (RREQ) message and also determine the best time to forward the message based on approximate information about neighbors such as which TAs contain one or more nodes that already received or will receive the RREQ message. Each node needs to store the set of TAs locally and update the set whenever a new copy of the same message is received. When it is time for a node b to forward the RREQ message, it takes the union of the stored set of TAs and C_{node_b} and piggybacks it with RREQ message. Based on the updated set of TAs, each node make its own decision regarding when to forward the RREQ message. We encode the set of TAs using bit vectors to reduce the message overhead.

To overcome the drawback of bit vectors, we need to make the number of candidate elements as small as possible. ALIs are not good to identify elements since there are too many possible ALIs. This is why we defined RLIs. Because we are more interested in the coverage information about its one-hop neighbors, we employ a 64-bit vector (2 32-bit words) to transmit the set. As we mentioned earlier, possible RLIs are a sequence of nonnegative integers. Moreover, a node can not have a one-hop neighbor in a TA whose RLI is greater than 48. Therefore, a 64-bit vector serves well for this purpose. The remaining 16 bits carry part of two-hop neighborhood information. Because of this representation, the RLI assigning function has to be a one-to-one map from TAs to RLIs.

We limit the size of set transmitted from one node to another to 64. For example, a node may receive many copies of the same route request message. The union of the sets of TAs carried by those copies is stored locally. When the node decides to forward a RREQ message it piggybacks with this locally stored set with the RREQ message.

3.5 The Algorithm

In this section, we present the proposed algorithm. We first outline the drawbacks of some of the existing topology-based routing algorithms. As we mentioned earlier, routing algorithms, such as AODV [56], DSR [29], and TORA [51], that simply flood RREQ mes-

sage for route discovery are known to have high routing overhead, especially in dense networks and hence do not scale well. Many papers have partially addressed these problems by using various graph models, e.g. unit disk graph [10], relative neighborhood graph (RNG) [12, 58, 62], and dominating set [6, 70, 71]. However, they also introduce new problems, e.g. they use too much bandwidth in exchanging neighbor information. Typically, they employ heartbeat messages (a.k.a. hello messages) to exchange neighbor information. When two-hop neighbor information is needed for these algorithms, the size of heartbeat messages is large in dense networks. Moreover, if nodes move fast, the neighborhood information known through heartbeat messages becomes obsolete quickly. Clearly, finding new routes using obsolete neighborhood information increases algorithm complexity. The problem of obsolete information is reduced somewhat when only one-hop neighborhood information is exchanged. Moreover, these algorithms can only suppress a limited number of redundant messages.

To solve these problems, we propose an algorithm that suppresses redundant route request messages. The proposed algorithm allows nodes to determine whether and when to forward a received RREQ message based on its neighborhood information and the information piggybacked on the message. Under this algorithm, it is required that a node has information such as who are its one-hop neighbors, what are their ALIs, and what are the ALIs of the TAs they can reach. It is not required that a node knows the exact locations of its two-hop neighbors although it can derive a rough 'two-hop' neighbor knowledge from its exchanged one-hop information.

When a node initiates a route request, it sends RREQ message piggybacked with the set of TAs it can reach. Upon receiving the message, the receiver knows which TAs have been potentially covered by the RREQ message already; potentially covered means that there is at least one node in each of the TAs that receives the message. When a node receives multiple copies of the same RREQ message from different neighbors, the potentially covered TA set is the union of the sets piggybacked on those messages. Therefore, a node b knows

which TAs in the set $C_{TA_{node_b}}$ may not have been covered. We refer to such set of TAs as $NPC_{TA_{node_b}}$. If $NPC_{TA_{node_b}}$ is empty, node b does not need to forward the message. Otherwise, it starts a timer (We present a method for computing the timeout value for the timer in Section 3.5.2). When the timer expires, it recomputes the $NPC_{TA_{node_b}}$. Node b forwards the message only if the recomputed $NPC_{TA_{node_b}}$ is not empty. Unnecessary forwarding of RREQ message is suppressed further by performing other checks. We present those additional checks in Section 3.5.2.

We next discuss heartbeat messages, route discovery, and route maintenance, step by step. Then we present the performance evaluation results of the proposed algorithm in Section 3.6.

3.5.1 Heartbeat Messages

Heartbeat messages are a special type of messages sent by nodes periodically. It is a commonly used technique for a node to tell its neighbors its status in mobile ad hoc networks. Upon receiving a heartbeat message, the receiver knows the ids of the nodes lying within its transmission range as well as other information piggybacked in the message. In other-words, heartbeat messages help the receiver get to know which nodes are its direct neighbors. Senders typically piggyback relevant information on the heartbeat messages such that the receivers have better knowledge about the senders if necessary. The proposed algorithm employs this information.

A heartbeat message used in the proposed algorithm has three fields: SrcID, ALI and PCTA. The first field, SrcID, refers to the address of the sender. The second field, ALI, contains the ALI of the sender. The third field, PCTA, contains the set of TAs within the transmission range of the sender that contain at least one node. Let s be the sender of a heartbeat message. Then C_{node_s} equals to the field PCTA in the message.

Under the proposed algorithm, each node is required to send out heartbeat messages periodically, say every 2 seconds. A node may piggyback a heartbeat message onto other

types of messages, such as data messages, route request messages, etc. If a node has not sent such messages within the predetermined time period, it sends out a new heartbeat message at the end of the time period.

Upon receiving a heartbeat message, a node b updates its neighbor table (NT) accordingly. NT maintains one entry per neighbor (nbr). Each entry has four fields: NbrID, ALI, PCTA, and ts. NbrID and ALI refer to the neighbor's address and ALI respectively. PCTA equals to $C_{node_{nbr}}$. The ts field stores the time at which node b received the last heartbeat message from neighbor nbr. The following two cases arise when node b receives a heartbeat message from node s: If there is no entry in neighbor table corresponding to s, node b inserts a new entry to NT and updates the time stamp field; Otherwise, node b updates the ALI, PCTA and time stamp in the entry corresponding to node s. Node b scans its neighbor table and removes outdated neighbor entry from its neighbor table before it sends out (or piggybacks) a heartbeat message. A neighbor entry is considered to be outdated if its timestamp has not been updated during the last three time periods; i.e., a node assumes that the corresponding neighbor is not within its transmission range. After initial rounds of exchanging heartbeat messages, each node has the necessary information for running the proposed algorithm.

We indicated earlier that neighbor location information does not work well in mobile environment. How does this heartbeat mechanism work in such environment? Note that this mechanism does not collect exact neighbor location information but the approximate locations (TAs) a node can reach. Network topology changes as nodes move. A node's PCTA remains relatively stable as long as related TAs contain one or more nodes even though nodes may move in/out those TAs. PCTA contains the information we rely on to suppress redundant route request messages and find route to the destination. Therefore, the proposed algorithm works relatively stable in mobile environment. We demonstrate this when presenting simulation results in Section 3.6.

3.5.2 Route Discovery

Similar to existing topology-based on-demand routing algorithms, the proposed algorithm uses route request messages for finding a route to the destination in route discovery stage. Unlike other algorithms, we piggyback a set of TAs onto each route request message. We refer the set of TAs piggybacked of route request messages as PCTA which contains all the TAs the route request message has reached, to the knowledge of its sender. Here is a PCTA example. Suppose node a initiates a route request message which is then forwarded by nodes b and c. Node d later receives the message from node b and c but not from node a. Node a piggybacks the message with C_{node_a} before broadcasting it. Similarly, node b and c piggyback the message with $C_{node_a} \cup C_{node_b}$ and $C_{node_a} \cup C_{node_c}$ respectively. Upon receiving the message from both node b and c, node d knows the message has reached $C_{node_a} \cup C_{node_b} \cup C_{node_c}$. Therefore, node d will piggyback the message with

$$PCTA = \{C_{node_a} \cup C_{node_b} \cup C_{node_c} \cup C_{node_d}\}$$

in case it decides to forwards the message. The PCTA in the messages forwarded by node b and c are $C_{node_a} \cup C_{node_b}$ and $C_{node_a} \cup C_{node_c}$ respectively. Clearly, a node x is able to make right decision on whether and when to forward a route request message by comparing C_{TAnode_x} and the PCTA in the route request message. Node x does not need to forward the message if all the TAs in $C_{TA_{node_x}}$ are present in the PCTA piggybacked on the message; one exception would be when the destination of the route request message resides in the same TA as node x does. In this case, at least one node in the TA would need to forward the message to make sure that the destination receives the message.

In some cases, information contained in the PCTA alone is not sufficient for suppressing redundant route request messages. For example, suppose node a forwards a route request message and all its neighbors receive the message at the same time. Without loss of generality, let node b be one of the neighbors. There is a very good chance that one or more TAs in $C_{TA_{node_b}}$ are missing from the PCTA piggybacked in the message. The route request

message still gets forwarded if all of the neighbor nodes do what b does. One solution to this issue is to let the neighbors forward the message at different times. This solution has a couple of advantages. For example, it reduces contention for media access and avoids unnecessary packet loss due to the collision. It also gives time to the neighbor nodes to learn about the route request message and hence make right decision to reduce redundant route request message propagation.

We now present how a node b responds to a received a route request message with PCTA piggybacked. It first checks if the following conditions hold:

1. Node b forwarded the same route request message earlier.
2. All the TAs in $C_{TA_{node_b}}$ have been covered already, namely $C_{TA_{node_b}}$ is a subset of the union of PCTAs piggybacked on the same route request messages received from other nodes by node b.
3. All the TAs in C_{node_b} have been covered already and a node in the set TA_{node_b} forwarded the message earlier.
4. The destination of the route request message resides in TA_{node_b} and a node inside the TA forwarded the message earlier.
5. Node b already saw a route reply message for this route request message.

If any of the above conditions holds, node b simply ignores the received message since forwarding the message would not help any new node to receive the message. Otherwise, it computes the priorities of its direct neighbors including itself for forwarding the message based on information in its neighbor table, $C_{TA_{node_b}}$ and virtual PCTA (VPCTA). A virtual PCTA is the union of C_{node_b} and PCTAs piggybacked on the same seen route request messages. The priority calculated is proportional to the size of the set $C_{TA_{node_b}} - VPCTA$. The timeout value for a node to forward a route request message is inversely proportional

to this calculated priority. The higher the priority a node has, the faster the node will forward a received route request message. After setting the timeout value, node b waits for the timeout period. If node b receives another copy of the same route request message, before the timeout expires, it will re-evaluate the conditions and re-calculate the priority and timeout value if necessary. If node b still needs to forward the route request message, node b adjusts the timeout value according to newly calculated timeout value.

Clearly, the size of PCTA piggybacked on route request messages directly plays a role in how efficient the algorithm is in suppressing redundant route request messages. However, we cannot let the size of PCTA piggybacked grow indefinitely. Since a node b only uses sets no larger than $C_{TA_{node_b}}$ in calculating the criteria for forwarding a received route request message, it's clear that the TA information inside the PCTA of the message is useless when the TA is two or more hops away from the receiving node b. Therefore, we can limit the size of the bit vector to 96 bits which is good enough to cover all the TAs containing all two-hop neighbors of the node.

In route discovery phase, a node initiates a route request message and the destination upon receiving the route request message, sends a reply message. Upon receiving a route request message, the intermediate nodes record them in a route request table (RRT). The intermediate nodes also record route reply messages in a route table (RT) when they receive a route reply message. Detailed descriptions of the data structures used are as follows.

1. Each route request message has six fields. They are Seq, SrcID, DstID, hopcount, PCTA, and ALI respectively. The Seq field is the sequence number of the route request message assigned by the source node whose address is recorded in the SrcID field. The source node maintains a sequence number and increments it every time it initiates a new route request. Therefore, the Seq field together with SrcID field uniquely identifies a route request message. The DstID field specifies the address of the destination node. The hopcount field contains the numbers of nodes the route request message has traversed from the source so far. The PCTA and ALI fields

together uniquely defines the set of TAs the route request message has reached.

2. Each node maintains a route request table (RRT) which stores information about received route request messages. Each entry in this table contains the first 5 fields of the received route request messages. Note that ALI is not relevant anymore after converting the RLIs in the PCTA to ALIs or the RLIs relative to its own ALI. In addition to these five fields, each entry has three other fields. They are prehop, sameTA, forwarded, and ts. The field prehop indicates the node from which the route request message has been received. This field is updated whenever a better route to the source is detected (i.e., when a route request message with lower hopcount is received). Typically, it contains the node from which it receives the first copy of the route request message. We may also take the hopcount field into account when updating this field. The field sameTA indicates whether a node inside the same TA has forwarded the route request message. The field "forwarded" indicates whether a node has forwarded the same route request message earlier. The field ts records the time at which the route request message has been received. It is updated when it receives the same route request again. Each entry has limited lifetime. We remove an entry in case it expires.

3. Each route reply message is composed of four fields. They are Seq, SrcID, DstID, and hopcount. The first three fields are copied from the corresponding route request message. The forth field hopcount indicates how many nodes the route reply message has traversed from the destination.

4. Each node maintains a route table (RT) which stores the routing entries describing how to get to another node in the network. Each route entry has three fields, namely DstID, nexthop, and ts. The field ts indicates the time at which the route entry was created or updated. The nexthop field contains the id of the node to which it needs to forward data packets destined to the node with id DstID.

Figure 3.4 shows the pseudo code for disseminating route request messages.

When node b initiates a RREQ for destination d
Init $rreq$ = ($Seq = ++Seq, SrcID = b, DstID = d,$
$hopcount = 1, PCTA = C_{node_b}, ALI = ALI_b$);
Store $rreq$ in RRT and broadcast $rreq$;

When b receives a RREQ m
if b has forwarded a copy of m already **then return**;
if b has not received a copy of m before **then**
Store m in RRT;
Get the entry rrt from RRT corresponding to m; /* Operations on rrt are done in RRT as well */
$rrt.PCTA = rrt.PCTA \cup m.PCTA \cup \{TA_{node_b}\}$;
$rrt.SameTA = rrt.SameTA || (ALI_b == m.ALI)$;
if $|C_{TA_{node_b}} - rrt.PCTA| == 0$ **then return**;
if $(rrt.sameTA) \&\& (|C_{node_b} - rrt.PCTA| == 0)$ **then return**;
$priority$ = **getPriority**($rrt.PCTA$, b);
$toVal = priority \times BroadcastSpacing$; /* $BroadcastSpacing$ is a predefined value */
if $rrt.SameTA$ **then** $toVal += SameTAWait$; /* $SameTAWait$ is a predefined value */
Set a timer T with a timeout value of $toVal$;

When timer T at b expires
Get the entry rrt from RRT corresponding to T;
if $rrt.forwarded$ **then return**;
if $|C_{TA_{node_b}} - rrt.PCTA| == 0$ **then return**;
if $rrt.sameTA$ **and** $|C_{node_b} - rrt.PCTA| == 0$ **then return**;
Reconstruct the message m with ALI and $PCTA$ updated to $rrt.PCTA \cup C_{node_b}$;
Broadcast m with a broadcast jitter;

function: getPriority ($PCTA$, b)
$C = C_{TA_{node_b}} - PCTA$;
$priority = 0$;
$nset$ = the set of neighbors that lie in $PCTA$;
while ($|C| > 0$)
Find a node n in $nset$ such that $|C - C_{node_n}|$ is the smallest;
if n is b **then return** $priority$; changed d to n
$C = C - C_{node_n}$; $priority = priority + 1$; $nset = nset - n$;
return $lowest\_priority$; /* $lowest\_priority$ is a predefined value */

Figure 3.4: Algorithm for disseminating route request messages

Once the destination node receives the route request message, it sends a route reply message back to the source node via the node in the prehop field corresponding to this route request message, found in its RRT. In this case, it also updates its route table accordingly for the source node. When an intermediate node receives a route reply message, it simply forwards the message to the prehop corresponding to the route request message entry in RRT and updates its RT. After the route reply message reaches the source node, a route has been established to the destination and the source node may start forwarding data packets

to the destination.

3.5.3 Route Maintenance

Nodes in mobile ad hoc networks may move at will. An established route will break when an intermediate node on the route moves away. Therefore, source needs to establish a new route in such scenario if the route is still being used. Many existing routing algorithms, such as AODV and DSR, use route-error messages to notify source nodes about broken links. The source nodes then re-initiate route discovery to establish a new route to the destination. In triangle based routing, each node may maintain multiple next hops for a given destination, helping it repair a broken link by using other valid next hops. We take this approach for repairing broken links.

The basic idea behind route repair is as follows: When a node b detects a broken link on a route to the destination, if it can not find another available link through which it can forward data to the destination, it first sends a route repair message to its one-hop neighbors. Upon receiving the route repair message, each node updates its own route table by removing appropriate links, and checks if it has a good forwarding node to the destination. If so, it acknowledges the route repair message. Otherwise, nothing needs to be done. The broken route is repaired when node b receives one or more acknowledgments for the route repair message. Otherwise, it initiates a route discovery on behalf of the source node.

3.5.4 Correctness Proof

In this section, we present a correctness proof of the algorithm in a connected network. Before presenting the correctness proof of the algorithm, we prove the following theorems.

Theorem 3.1 *The algorithm for route discovery terminates in finite time assuming message delay is bounded and the given network has finite number of nodes.*

Proof: Each node sets up a timer upon receiving a route request message. It forwards the message only after the timer expires and it has never forwarded the message before.

Since the timeout value is always finite, each node forwards or stops forwarding a received route request message at most once in a finite time. Nodes in the network will stop forwarding any given route request message in finite time since the number of nodes in the network is finite. Therefore, the algorithm terminates in finite time. □

Theorem 3.2 *Assuming the network is connected, for any TA having one or more nodes, there is at least one node inside the TA that receives the route request message initiated by any node.*

Proof: Without loss of generality, let node s initiates a route request message m. Thus, we rephrase the theorem as follows: The message m is received by at least one node in each TA. We prove the theorem by induction on TAs.

Base: Want to prove there is at least one node in each TA that receives the message m. It is clear that node s in TA_{node_s} that receives the message m. Note that we assume that a node will receive a message sent by itself.

Induction: Assume that there is at least one node that lies in TA_i and receives the message m. Want to show that it is also true for each TA in C_{TA_i}. According to the algorithm, a node in TA_i does not forward the message only when all TAs in C_{TA_i} have been covered already (Case 1), or it has forwarded the message before (Case 2). If it is the Case 1, the proof is done. In Case 2, all nodes in TA_i receive the message m. In this case, a node, say x, is able to reach TA_y which does not belong to C_{node_i}. According to the proposed algorithm, node x forwards the message m when TA_y is not covered by a route request message. Therefore, all TAs in C_{TA_i} are covered. □

Theorem 3.1 proves that the proposed algorithm will terminate in finite time, while theorem 3.2 proves that at least one node in each TA receives the route request message initiated by a source node. Clearly, both theorems together prove that any route request message initiated in a connected network will reach a node x inside the TA in which the destination lies. According to the proposed algorithm, node x or some other node in the

same TA forwards the message to the destination. In either case, the destination receives the route request message.

3.6 Performance Evaluation

In this section, we present the results of our performance evaluation of TBR compared to AODV [56]. We first introduce the simulation model and then present our simulation results and analysis of those results.

3.6.1 Simulation Model

We used GloMoSim [74], a widely used network-simulation tool for studying the performance of routing algorithms for mobile ad hoc networks, for evaluating the performance of TBR.

We chose IEEE 802.11 [26] and IP as the MAC (Medium-Access Control) and network-layer algorithms respectively. All nodes have a fixed transmission range of 350m. We used the implementation of AODV that comes with the GloMoSim 2.0.3 package to compare its performance with TBR. This implementation employs expanding-ring search to discover a route from a source to a destination; under expanding ring search, the search neighborhood is enlarged by increasing the TTL (TimeToLive) field in the IP header of the request packets. AODV starts the search for a route to the destination by setting TTL to 1 or to the previously known hopcount and repeats the search by increasing the TTL by 2 (after the TTL reaches 7, it is set to 35, the maximum network diameter) until a RREP message is received from the destination or the timeout for route discovery expires. This phased search reduces the route-establishment overhead for destinations that are close to the source. We simulated TBR also with this mechanism to reduce the propagation of route request messages.

In the implementation of AODV, we set the route-discovery timeout to 10 seconds. The source checks if a route reply message is received within 80 times TTL milliseconds

after the last time it initiated a route request. In our implementation of TBR, each node broadcasts a heartbeat message every 2 seconds. Like AODV, the timeout for checking route replies for TBR is set to 80 times TTL milliseconds. A node re-initiates a new route request if it receives no reply before it times out.

3.6.2 Mobility Model

We adopt the steady state random-waypoint model [9, 18, 73] that is a widely used mobility model for simulations. Under this model, each node travels from a random location to a random destination at a random speed, the speed being uniformly distributed in a predefined range. After a node reaches its destination, it pauses for a predetermined amount of time and then moves to a new randomly chosen destination at a randomly-chosen speed.

In our simulation, we set the speed range to 1 – 19 m/s. In order to study how mobility affects the performance of the routing algorithms, we selected pause times of 0, 30, 60, 90, 120, 200, 300, 500, and 900 seconds. When the pause time is 0 seconds, every node moves continuously. As the pause time increases, the network approaches the characteristics of a fixed network.

In a dense network, a path may always be available between any source-destination pair. On the contrary, if nodes are sparsely distributed, the network may be partitioned; moreover, in this case, node mobility can exacerbate the situation. In our performance evaluation, we simulated the following three scenarios to study the effect of density of the nodes on performance:

- $1500 \times 1500 m^2$ field with 200 nodes
- $1500 \times 1500 m^2$ field with 300 nodes
- $1500 \times 1500 m^2$ field with 400 nodes

We ran the simulation for each of the three scenarios for 15 simulated minutes.

3.6.3 Traffic Model

To measure the effect of network traffic, we used 5, 10, 20 30, 40, or 50 CBR (constant bit-rate) data sources. We selected both the sources and the destinations randomly and uniformly. The sources transmit data between a chosen start time and a corresponding end time; we selected the start and corresponding end times randomly and uniformly within the 15-minute simulated interval in such a way that the start time precedes the end time. We fixed the size of data packets at 512 bytes and had each source generate packets at the rate of 4 packets per second. Measurements were taken after a settling time [73] of 150 simulated seconds.

3.6.4 Performance Metrics

We evaluated the performance of our algorithm with respect to the following three metrics:

- Packet-delivery ratio: The ratio of the number of data packets delivered to the destinations to the number of data packets generated by the CBR sources.
- End-to-end delay of data packets: This figure includes all possible delays, including those caused by buffering due to route discovery, queuing delay at the interface queue, retransmission delays at the MAC layer, and propagation and transfer time.
- Normalized routing overhead: The ratio of the number of routing control packets transmitted to the number of data packets delivered to the destinations. We count each time a node sends a routing control packet to its next-hop neighbor.

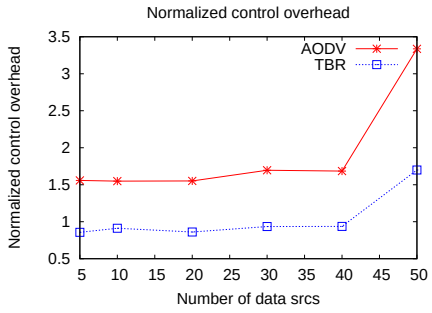
Next, we present the performance evaluation results of our algorithm.

3.6.5 Performance Results

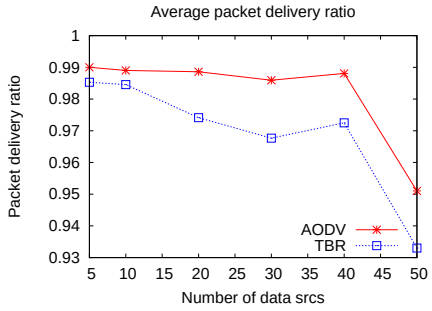
We evaluated the performance of our algorithm with respect to the above-mentioned metrics under three scenarios.

Scenario I

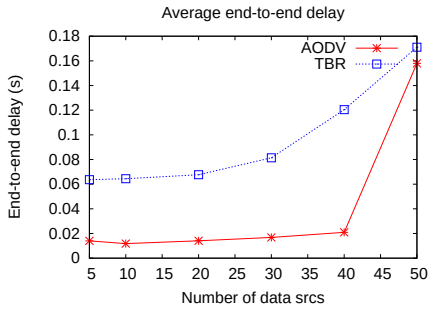
Under this scenario, we used a total of 200 nodes randomly distributed across the simulated region.



(a)



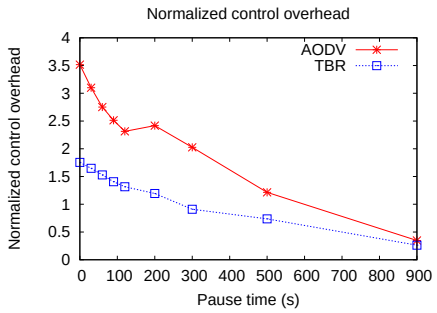
(b)



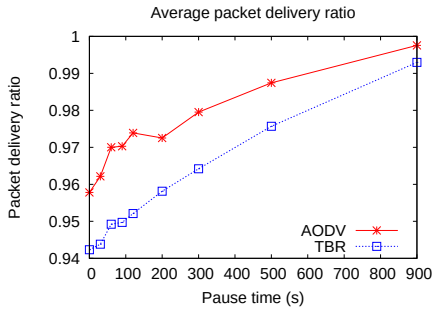
(c)

Figure 3.5: Varying number of data sources in scenario I (200 nodes)

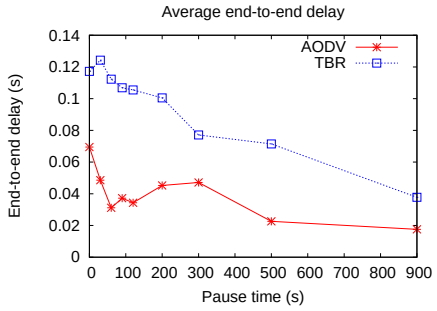
Figure 3.5 and 3.6 show the performance of TBR compared to AODV with respect to the three metrics for varying numbers of data sources and pause times. In Figure 3.5, the values plotted are the average values taken over various pause times ranging from 0 to 900 seconds for different number of data sources. Figure 3.6 the values plotted are for various



(a)



(b)



(c)

Figure 3.6: Varying the pause time in scenario I (200 nodes)

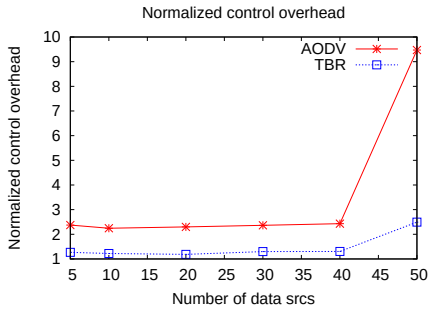
pause times, averaged over 5 to 50 CBR sources.

Under scenario I, the simulation results show that the average normalized routing overhead of AODV and TBR is 2.23 and 1.20 respectively. As expected, TBR uses fewer nodes for forwarding route requests than AODV, resulting in lower routing overhead. TBR has slightly higher average end-to-end delay, on average; however end-to-end delay of AODV increases sharply as the number of CBR sources increases beyond 40. In summary, performance of TBR is more stable than AODV when nodes with high mobility are involved or

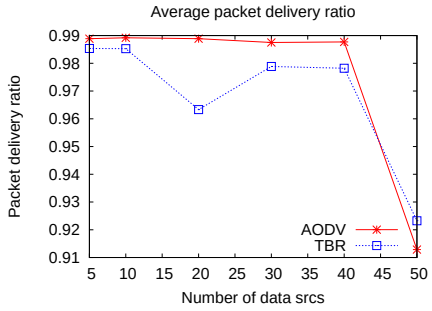
the number of CBR sources are high. The results obtained in scenario II and III (described next) also confirm this observation.

Scenario II

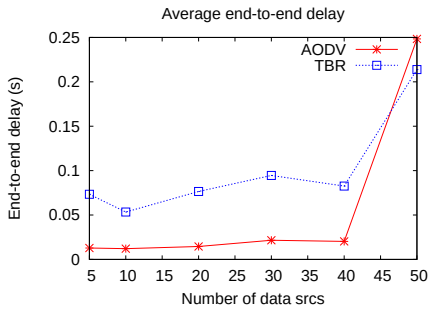
This scenario has 300 nodes.



(a)



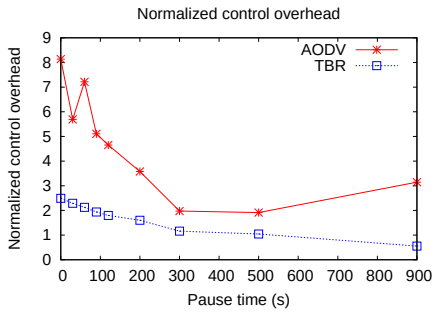
(b)



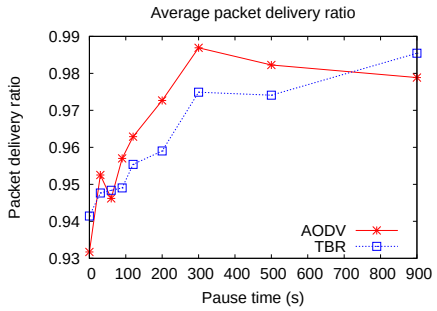
(c)

Figure 3.7: Varying number of data sources in scenario II (300 nodes)

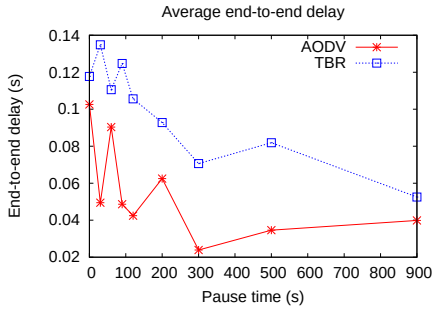
Figures 3.7 and 3.8 show the performance of TBR compared to AODV with respect to the three metrics for varying numbers of data sources and pause times. In this scenario, TBR performs better than AODV with respect to all three metrics. For instance, the average



(a)



(b)



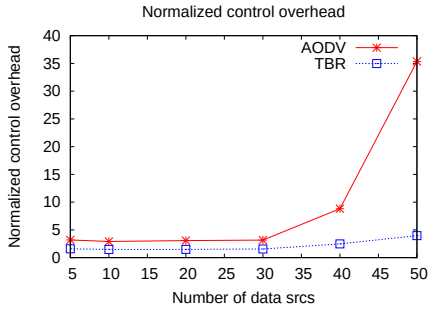
(c)

Figure 3.8: Varying the pause time in scenario II (300 nodes)

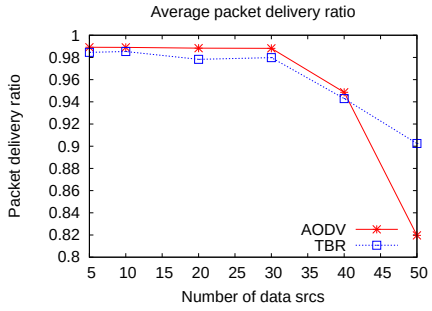
values of the normalized routing overhead, packet-delivery ratio, and end-to-end delay of TBR are 1.67, 0.959, and 0.099 respectively, while the three measurements for AODV are 4.57, 0.963, and 0.055 respectively. In this scenario, AODV has slightly lower end-to-end delay when fewer CBR sources are involved. However, it has higher end-to-end delay when there are 50 CBR sources, which makes its average value higher than that of TBR. Again, as results in Figure 3.8 indicate, the performance of TBR is much more stable than AODV with respect to node mobility.

Scenario III

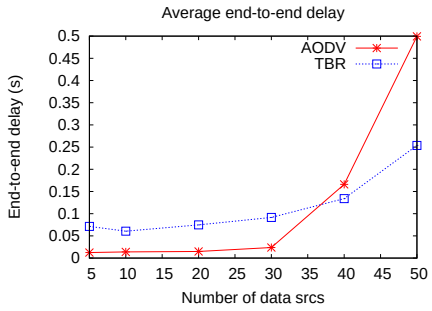
This scenario has 400 nodes.



(a)



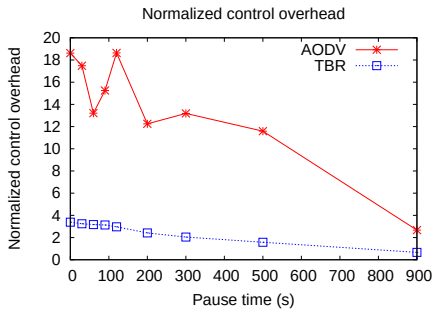
(b)



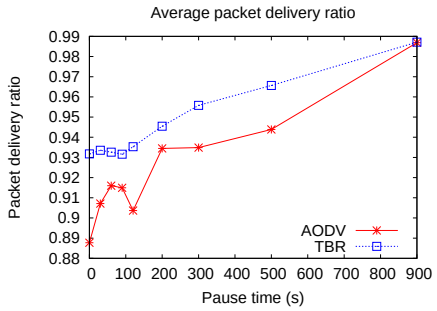
(c)

Figure 3.9: Varying number of data sources in scenario III (400 nodes)

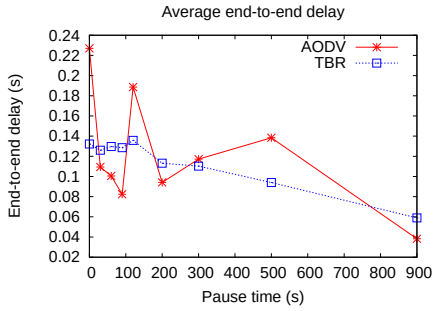
Figures 3.9 and 3.10 show the performance of TBR compared to AODV with respect to the three metrics for varying numbers of data sources and pause times. The simulation results under this scenario are similar to the simulation results under scenario II. TBR has much lower routing control packet overhead than AODV in this case. TBR has higher packet-delivery ratio, and lower end-to-end delay than AODV in this case when the number



(a)



(b)



(c)

Figure 3.10: Varying the pause time in scenario III (400 nodes)

of data sources reaches 40. Even in such a dense network, the average normalized routing overhead of TBR is 2.51, which is only 1.31 more than that in scenario I and 0.84 more than that in scenario II. This case also demonstrates that TBR is much more stable than AODV.

3.6.6 Analysis

We make the following observations based on the simulation results.

Routing Overhead

In sparse networks, the two algorithms have similar packet-delivery ratio. Since TBR tries to guarantee the delivery of generated data packets, it issues many useless RREQ messages searching for non-existent paths in a partitioned network. However, as the network becomes denser, the number of route-control packets issued by TBR does not greatly increase. This gentle rise is due to TBR's selective forwarding mechanism in flooding RREQ messages. This mechanism is very efficient in controlling routing overhead by limiting the number of nodes that forward the RREQ messages in dense networks.

The average normalized routing overhead under all three scenarios for AODV and TBR are 6.81 and 1.79 respectively. TBR has relatively constant overhead as the number of nodes in the network increases from 200 to 400. On the contrary, AODV incurs much more routing overhead as the number of nodes increases. The same thing happens as the number of CBR sources increases or the nodes become more mobile (or pause time decreases). Thus, TBR performs much better than AODV with respect to routing overhead in networks with highly mobile nodes, networks in which nodes are densely distributed, or heavily loaded networks.

End-to-end Delay

The overall average end-to-end delay for AODV and TBR are 0.072 and 0.10 respectively. TBR has highest end-to-end delay in a sparse network. This result arises because it is hard to repair a broken route in a sparse network. As the density of the network increases, more routes become available, and the end-to-end delay is more dependent on the number of hops and the network load. There the end-to-end delay under TBR is comparable to AODV. In high-density and high-load networks, TBR has lower end-to-end delay than AODV because TBR has much lower routing overhead. Another reason TBR has higher end-to-end delay is that nodes need to wait certain amount of time before forwarding a route request message in order to suppress more redundant messages.

Network Load

As we expect, as network load increases, both algorithms show increasing normalized routing overhead and end-to-end delay. However, TBR is relatively stable as the number of data sources increases, but performance of AODV degrades greatly.

3.7 Conclusion

This chapter proposes a novel mechanism for suppressing redundant route request messages when broadcasting them in mobile ad hoc networks. It presents the triangle based routing algorithm that employs that mechanism. In a dense network, we have demonstrated that the algorithm efficiently selects a limited, but sufficient, set of forwarding nodes to flood the route requests. We compared the performance of our algorithm with a well known routing algorithm AODV. Simulation results show that TBR always has much lower normalized routing overhead than AODV.

Chapter 4

A Routing Algorithm with Selective Forwarding for MANETs

4.1 Introduction

A Mobile Ad Hoc Network (MANET) consists of a set of mobile hosts that can form a network automatically without the aid of any infrastructure or human intervention. This feature of ad hoc networks facilitates its deployment in a variety of environments such as battlefields, disaster areas, and natural habitats. The limited battery life of mobile hosts implies a need for energy-efficient routing algorithms on such networks.

Depending on when the sender of a message gains a route to the receiver, routing algorithms for mobile ad hoc networks can be classified into three categories: proactive [15,50,54], reactive [29,55,56], and hybrid [24]. Proactive routing algorithms compute all routes before they are needed. Reactive algorithms compute routes on demand. Hybrid algorithms use a combination of proactive and reactive approaches. A reactive routing algorithm consists of a route-discovery phase and a route-maintenance phase. Many of the existing reactive routing algorithms flood the network with redundant route-request messages in order to find a route to the destination. In this chapter, we propose a reactive routing algorithm under which a node can select its neighbors to forward route requests, lowering the routing overhead. Moreover, our routing algorithm can help in maintaining multiple routes to a destination.

4.1.1 Related Work

Routing in MANETs has been extensively studied in the literature [9, 15, 18, 23, 29, 51, 54–56, 67]. Many of the existing on-demand routing algorithms for MANETs use a simple broadcasting mechanism that floods the entire network with route-request messages. This mechanism can lead to a high redundancy of route-request messages, contention, and collision. Well known algorithms such as AODV [56], DSR [29], DSDV [54] and TORA [51] use the flooding approach. Broch et al. [9] studied the performance of DSDV, TORA, DSR, and AODV. Their results show that the routing overhead of these algorithms increases quickly as the number of of nodes in the network increases. Perkins et al. [55] proposed Dynamic MANET On-demand (DYMO) routing algorithm, a descendant of AODV and DSR. DYMO is suitable for sparse networks. TBRPF [50] and OLSR [15] are two proactive, link-state routing algorithms. Both of them are suitable for networks in which a large number of routes are needed and for applications that can not tolerate the delay due to route discovery. TBRPF reports updates reactively when a link state changes while OLSR reports them periodically. Therefore, TBRPF and OLSR may not work well in networks where nodes move quickly. In such a scenario, TBRPF may send a large number of updates into the network and nodes may have too many outdated links in its route table if OLSR is used.

The Zone Routing Algorithm (ZRP) [23] uses a hybrid approach for maintaining routes. Under this algorithm, each host proactively updates its routing table for all destinations within its zone. For destinations outside its zone, a node employs a reactive approach to find routes on demand. Some routing algorithms use a connected dominating set [67] as a backbone network to minimize the number of nodes that participate in forwarding route-request packets, and hence reduce overlapping route-request propagation. A disadvantage of this approach is that the selected “core” or “backbone” nodes may drain their battery quickly; a solution to overcome this problem is to periodically change the set of “backbone” nodes. However, the complexity of computing an approximate minimal dominating set of a

wireless network (computing a truly minimal dominating set is known to be NP-complete) may result in high overhead.

Researchers have proposed position-based routing algorithms to limit the propagation of redundant route-request messages during route discovery [4,5,8,27,32,35]. DREAM [4] proactively maintains at each node the location information of all the nodes in the network and floods data packets to nodes in the direction of the destination. Location-Aided-Routing (LAR) [35] floods route-request packets only in a request zone, which it calculates based on the last known position and velocity of the destination. The quality of unicast routes obtained by LAR is improved in [16]. GPSR [32], GFG [8], and GRA [27] use similar greedy methods for forwarding data packets. Under these algorithms, upon receiving a data packet, each node forwards it to a neighbor that is closer to the destination. This process is repeated until the data packet reaches the destination. However, they use different mechanisms to route data packets when the greedy method fails. GPSR and GFG use perimeter-mode packet forwarding, while GRA uses breadth-first or depth-first route discovery to handle such failures. The path found by perimeter-mode packet forwarding may not be optimal if the source and destination do not lie on a path that closely follows a straight line. Breadth-first or depth-first route discovery may result in very high routing overhead for large ad hoc networks. Xing et al. [72] propose Bounded Voronoi Greedy Forwarding (BVGF). Mauve et al. [46] present a good survey of many routing algorithms such as DREAM, LAR, and GPSR. Terminode routing [5] combines location-based routing and link-state routing and uses anchors to optimize the quality of routes. CLR [25] partitions the network into interlaced gray and white districts according to location; only nodes in gray districts participate in re-transmitting control packets. Unlike usual greedy position-based algorithms, NADV [40] takes both distance and link cost (measured in terms of delay, power consumption, or other metrics) into account in forwarding data packets.

Other algorithms also try to reduce redundant propagation of route request packets [49, 52,53]. Williams et al. [69] classify broadcasting techniques into simple flooding, probability-

based [49] flooding, area-based [49] flooding, and neighbor knowledge-based [52, 53] flooding. Other algorithms use pruning methods such as self pruning and dominant pruning to minimize redundant propagation of packets [52, 53].

The basic idea behind many of these position-based routing algorithms is to limit the search for the destination to a portion of the network based on estimating the location of the destination based on its last known position and velocity. Extra overhead is incurred when the estimation turns out to be incorrect. These algorithms require each node in the network know its own position and the position and velocity of every other node at some point in time. This information is not practical to maintain in a real ad hoc network environment. Moreover, each node in the search range is required to forward route-request packets, which can result in propagating redundant route-request messages. Our algorithm addresses both problems. It only requires each node to know the relative position of nodes in its neighborhood. A node trying to establish a route to a destination does not need to know the position or velocity of the destination.

4.1.2 Assumptions

We make the following assumptions about the nodes in the network.

- Nodes communicate via omni-directional antennas. The transmission range R is the same for all nodes in the network. Nodes within range R of n are called neighbors of n; Any message sent by n is received by all its neighbors.

- Nodes can determine the direction and distance of their neighbors. Nodes can determine their location by GPS and include it in the hello algorithm, or they can estimate the location of their neighbors by a combination of smart antennas and signal-strength measurements [68].

- Nodes are mobile.

4.1.3 Chapter Objectives

None of the many existing position-based routing algorithms for routing in ad hoc networks, to our knowledge, uses the relative position of neighbors to reduce route-discovery overhead. Furthermore, many of the existing reactive routing algorithms use simple flooding for sending route requests, which may result in redundant messages, contention, and collision. To address these faults, we propose **RPSF** (**R**outing **P**rotocol with **S**elective **F**orwarding), a novel algorithm for route discovery in mobile ad hoc networks. RPSF tries to minimize the propagation of redundant route requests by limiting the number of nodes that forward any route-request message. Performance evaluation shows that RPSF has significant advantages over AODV.

4.1.4 Organization of the Chapter

The remainder of this chapter is organized as follows. The basic idea behind route-request propagation under RPSF in an ideal network is presented in Section 4.2. In Section 4.3, we extend this algorithm to a general network and present RPSF, a general route-discovery algorithm. In Section 4.4, we evaluate the performance of RPSF and compare its performance with AODV [56]. In Section 4.5, we discuss the merits and shortcomings of our routing algorithm. Finally, Section 4.6 concludes this chapter.

4.2 Route-Request Propagation under RPSF in the Ideal Case

In many of the existing reactive routing algorithms for ad hoc networks, when a source node wants to find a route to a destination, it broadcasts a route-request (RREQ) message to all its neighbors; every node that receives the route-request message rebroadcasts the request to all its neighbors. This method results in overlapping broadcasts and incurs a lot of network overhead. In order to minimize such overlapped broadcasts, RPSF chooses only a subset of the nodes in its neighborhood to forward the RREQ message. However,

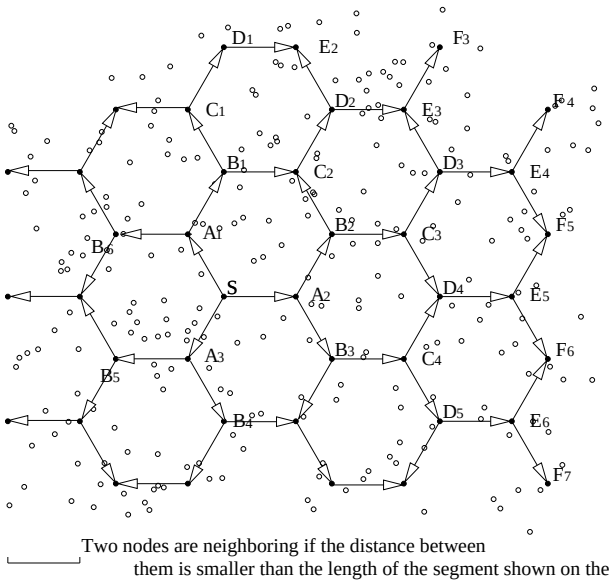


Figure 4.1: Route-Request Propagation in the Ideal Case

it ensures that the RREQ message eventually reaches the destination unless the network is partitioned. In the ideal situation, under RPSF, a source node only needs to select three of its neighbors to forward its RREQ message. Each node forwarding the RREQ message needs to select only two nodes in its neighborhood to relay the RREQ and this continues until the RREQ reaches the destination.

The *ideal case* occurs when the source node is able to select three nodes that are at distance R and are 120° apart with respect to the source to forward the RREQ message. Moreover, every forwarding node (unless it lies at the edge of the network) receiving the route request from a node n is able to select two nodes that are at distance R and are 120° apart with respect to n. Figure 4.1 illustrates this ideal situation in which the source *S* initiates route discovery by sending a route request. Solid circles represent the nodes forwarding the RREQ message; arrows point from nodes broadcasting the RREQ message to nodes selected to forward the RREQ message. Node *S* initiates the route-discovery process by sending a RREQ message. S selects A_1, A_2 and A_3 as its forwarding nodes. A_2, for example, selects B_2 and B_3 as forwarding nodes. D_4 receives the route request from both C_3 and C_4 in some order. Depending on this order, D_4 selects either $\{C_4, E_5\}$

or $\{C_3, E_5\}$ as its set of forwarding nodes. The same node can be selected as a forwarding node by more than one node. However, a node forwards the message only once even if it has been selected as a forwarding node by more than one node. From Figure 4.1 and the properties of regular hexagons, it is clear that the route request sent by a source eventually reaches the destination in the ideal case if the network is not partitioned. To be precise, in the ideal case, the entire geographical region can be partitioned into hexagons as shown in Figure 4.1 with the source lying at the vertex of one of these hexagons. As a result of the source broadcasting the route request each node on the vertex of each of these hexagons will rebroadcast the message. Each node inside a hexagon is at a distance less than or equal to R from one of the vertices of the hexagon, where R is the length of each side of the hexagons which is also the transmission range of each node. Thus, every node in the system is within the transmission range of at least one node broadcasting the route request and hence will receive the route request.

4.3 Route Discovery in the General Case

The ideal situation of Section 4.2 may not be present in a general ad hoc network, especially if the nodes are sparsely distributed. Based on the intuition gained from the algorithm in the ideal case, we now present an algorithm that is suitable for general ad hoc networks.

4.3.1 Selecting Forwarding Nodes for Route-Request Propagation

The key difference between route-request propagation in the ideal case and the general case lies in how a node selects its forwarding nodes. We first present the criteria used by a node for selecting its forwarding nodes. We then present the route-discovery algorithm in the general case and prove its correctness.

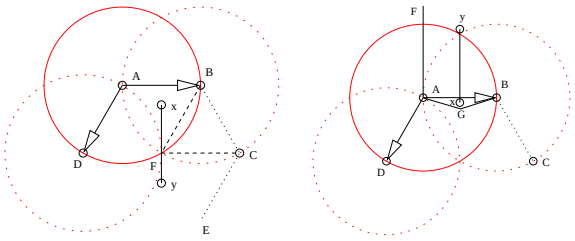
Definition 4.1 *A node A* **covers** *B if B lies within the transmission range R of A.*

Definition 4.2 *In an ad hoc network, a node d is* **reachable** *from node s if either (i) d is*

within the transmission range R of s or (ii) there exists a sequence of nodes $x_1, x_2, ..., x_n$ such that x_i is within distance R from x_{i-1} for $1 < i \leq n$ and s and d are within distance R from x_1 and x_n respectively. In other words, there exists a path $s, x_1, x_2, ..., x_n, d$ from s to d in the network.

In the general case, a node n uses the following steps to determine the list of forwarding nodes: It selects as distant a node as possible within its transmission range. It then selects further nodes that are mutually as far apart as possible, as far away as possible from n, and subject to the constraint that the angle made by any two successive nodes and n is at most $120°$. This last constraint is relaxed if no node can be found that satisfies it.

This method to select forwarding nodes may fail to cover the entire network in the general case. We now discuss some special scenarios that a node needs to take into consideration while selecting forwarding nodes and also propose rules for handling such special scenarios.



(a) (b)

Figure 4.2: Selection of forwarding nodes in the general case

Scenario (i): The ad hoc network consists of six nodes, A, B, C, D, x and y, as shown in Figure 4.2(a). The distance between y and x is $\leq R$; the distance between y and every other node is $> R$. B and D are at distance R from A and $\angle DAB \leq 120°$. F is a point on the bisector of $\angle ABC$. E is a point at distance R from C, and $\angle BCE = 120°$. E and F do not represent any node in the ad hoc network; they are just points of reference. Suppose A is the source node that initiates the route discovery. A selects B and D as forwarding nodes and sends them an RREQ message. B selects C as its only forwarding node and

forwards the RREQ message. All nodes in the network receive the route request except y. This failure would not happen if B also chooses x as a forwarding node, because x covers y. This observation leads to the following additional rule for the selection of forwarding nodes.

Rule 1: When a node B receives a route request from a node A, it chooses a forwarding neighbor C such that (i) C is as far as possible from B and (ii) $\angle ABC$ is as large as possible but $\leq 120°$ (if there is no such node, $\angle ABC$ could be $> 120°$). After choosing C, B may find one or more neighbors in the sector $\angle ABF$ (where F is a point on the bisector of $\angle ABC$), and the shortest of the distances between the neighbors in sector $\angle ABF$ and neighbors in sector $\angle FBC$ (including C) may be greater than R. If so, B replaces its choice of C with the neighbor x in $\angle ABF$ that is closest to the line segment BF; it resolves ties by picking the neighbor farthest from B. After choosing x as a forwarding node, B continues selecting other forwarding nodes, which may include C.

The intuition behind this rule for selecting forwarding nodes is that there could be nodes such as y in Figure 4.2(a) (that are exterior to the two circles with center A and B) that are not covered by A, B, C or D but can be covered by some node within the range of B lying inside the sector $\angle ABC$. B needs to choose the farthest such node as a forwarding node to cover nodes such as y.

Scenario (ii): The ad hoc network consists of six nodes: $A, B, C, D, x \; and \; y$ shown in Figure 4.2(b). In this figure, the distance between y and x is $\leq R$; the distance between y and every node other than x is $> R$. B and D are at distance R from A. F is a point in the plane such that $\angle FAB = 90°$. G is a point such that $\angle GBA = \angle GAB = \arctan(\frac{2R-\sqrt{4R^2-|AB|^2}}{|AB|}) \leq \arctan(2-\sqrt{3})$. F and G do not represent any nodes in the network; they are just points of reference. Suppose A is the source node that initiates the route discovery. A selects B and D as forwarding nodes and sends them the RREQ message. B selects C as its only forwarding node and forwards the RREQ message. All the nodes in the network receive the route request except y. This failure would not happen if

A or B also chooses x as a forwarding node, because x covers y. This observation leads to the following additional rule for selecting forwarding nodes.

Rule 2: After a node A chooses two nodes B and D as forwarding nodes, if there is a node x that lies in the triangular region $\triangle GAB$ where G is some point in the sector $\angle DAB$ such that $\angle GBA = \angle GAB = \arctan(\frac{2R-\sqrt{4R^2-|AB|^2}}{|AB|}) \leq \arctan(2-\sqrt{3})$, A replaces its choice of B with that x that is close to the line segment AB (and secondarily close to node A) and then continues the selection of forwarding nodes. A simple calculation shows that if y is reachable from x, then x should lie in the triangular region $\triangle GAB$ such that $\angle GAB = \angle GBA = \arctan(\frac{2R-\sqrt{4R^2-|AB|^2}}{|AB|})$.

The intuition behind this node-selection rule is to cover nodes such as y in Figure 4.2(b) (lying between the common tangent lines to the circles with centers at A and B and exterior to these two circles), which may be within the transmission range of some node close to the line AB in the sector $\angle DAB$, in $\triangle GAB$, but are not within the transmission range of A or B.

An Optimization for RREQ forwarding: Because of the delay in RREQ message propagation in various directions, it is possible that a node n is selected as a forwarding node by one of its neighbors even after all the neighbors of n have received the RREQ message. In such a case, n does not forward the message even though it has been selected as a forwarding node by one of its neighbors. Since each node maintains a list of directions from which it receives the same RREQ message, it can determine if all its neighbors have received the RREQ message without it having forwarded the message. For example, if n is selected as a forwarding node by one of its neighbors after n receives the RREQ message from three of its neighbors that are at 120° to each other with respect to n, then n need not forward this message because all its neighbors would have already received this message. So, even if a node n is selected as a forwarding node, it need not forward the message,if it can judge that all its neighbors would have received the message.

When b **initiates a RREQ for destination** d
Select a list L of forwarding nodes;
if L is empty **then return**; /* *No neighbor* */
Init RREQ = (Seq=++Seq, SrcID=b DstID=d,
HopCount=1,FwdIDList=L);
Broadcast RREQ;

When b **receives RREQ from node** n **at direction** dir
if (RREQ.SrcID == b) **then return**;
Store (Seq=RREQ.Seq, SrcID=RREQ.SrcID, DirectionList=dir)
in RRT;
if RREQ is new or the route to RREQ.SrcID in RREQ is shorter **then**
Remove all routes to RREQ.SrcID in RT;
Store(NextHop=n, HopCount=RREQ.HopCount) in the
NextHopInfoList corresponding to RREQ.SrcID in RT;
if b relayed this RREQ before **then return**;
if RREQ.DstID == b **then** /* *Initiate a RREP* */
Init RREP= (Seq=RREQ.Seq, SrcID=RREQ.SrcID,
DstID=RREQ.DstID, HopCount=1);
Broadcast RREP;
return;

if $b \in$ RREQ.FwdIDList **then** /* *It is a forwarding node* */
if b has received but not relayed RREP for the RREQ **then**
Init RREP = (Seq=RREQ.Seq, SrcID=RREQ.SrcID,
DstID=RREQ.DstID, HopCount=1+(HopCount in RT));
Broadcast RREP; **return**;
if b has already relayed RREP for this RREQ **then return**;
if b has already relayed this RREQ **then return**;
Select a list L of forwarding nodes;
if L is empty **then return**;
Set RREQ.FwdIDList = L;
RREQ.HopCount++;
Broadcast RREQ;

When b **receives RREP from** n
/* *Source node maintains multiple routes to a destination* */
/* *Intermediate nodes on a route to a destination maintain* */
/* *only one route to that destination* */
if (RREP.DstID == b) **return**;
if (RREP.SrcID == b) **then**
Append (NextHop=n, HopCount=RREP.HopCount) to the
NextHopInfoList corresponding to RREP.DstID in RT;
return;
elseif RREP is new or RREP has better route **then**
Remove the route entry for RREP.DstID in RT;
Store (NextHop=n, HopCount=RREP.HopCount) to the
NextHopInfoList corresponding to RREP.DstID in RT;
if b has already relayed RREQ but not RREP **then**
RREP.HopCount++;
Broadcast RREP;

Figure 4.3: RPSF route discovery in the general case

The general-case algorithm uses the message types and per-node data structures given in Table 4.1.

- *RREQ:* A route-request message sent to find a route to a destination.
- *RREP:* The route-reply message sent to notify the source of a valid route.
- *NIT:* A table maintained at each node and contains the direction and distance information for each of its neighbors. This table, which is periodically updated, is used for determining forwarding nodes.

Message or table name		Contents
Route-Request packet	(RREQ)	Seq, SrcID, DstID, HopCount, FwdIDList
Route Reply packet	(RREP)	Seq, SrcID, DstID, HopCount
Route Repair packet	(RRPR)	AckFlag, SenderID, DstID, HopCount
Neighbor Information Table	(NIT)	NeighborID, Direction, Distance
Route-Request Table	(RRT)	Seq, SrcID, DirectionList
Route table	(RT)	DstID, NextHopInfoList

Table 4.1: Data structures for the algorithm in the general case

- *RRT:* A table maintained at each node containing information about the RREQ messages received. For each RREQ message, it contains the sequence number (Seq), id of the source node that initiated the RREQ message (SrcID), and a list of directions from which the RREQ message was received.

- *RRPR:* A route-repair message used for repairing a broken route. A route to a destination could be broken due to a node moving outside the transmission range of an adjacent node in the route. RPSF only considers 1-hop repair.

- *RT:* The routing table maintained at each node, containing next-hop information for each destination to which a route has been established.

- *Seq:* The sequence number assigned to a RREQ message by the source. Together with the SrcID, Seq uniquely identifies a RREQ and its corresponding RREP message.

- *ID:* The unique identifier or address of a node, used in fields such as SrcID, DstID, and NeighborID.

- *HopCount:* An integer message field. In the RREQ message, it counts the number of nodes traversed by the RREQ message from the source. In the RREP message, it counts the number of nodes traversed by the RREP message from the destination. In the route repair (RRPR) message, it is meaningful only when the AckFlag is true, when it counts the number of nodes on the path from the destination to the node that have acknowledged the RRPR message.

- *FwdIDList:* The list of nodes that need to forward the RREQ message.
- *Direction:* Information about the direction in which a neighbor lies.
- *Distance:* The physical distance between a node and its neighbor.
- *DirectionList:* The list of directions from which the same RREQ message has been received so far.
- *AckFlag:* A Boolean field of an RRPR message. If it is false, the RRPR message is a request sent for repairing a broken route to some destination. If true, it is an acknowledgment sent by a node that has a route to the destination in response to a RRPR request. .
- *SenderID:* The ID of the node that initiates the RRPR message.
- *NextHopInfoList:* For each destination, RPSF maintains multiple routes. Corresponding to each destination, NextHopInfoList contains the list of next-hop nodes lying on various paths to that destination; it also includes the HopCount to the destination via each such node.

The algorithm for forwarding an RREQ message remains the same as in the ideal case except in the way the forwarding nodes are selected. For selecting the forwarding nodes in the general case, each node uses the selection criteria supplemented with Rules 1 and 2 discussed earlier. We prove below that the RREQ message sent by any source node eventually reaches the destination if the destination is reachable from the source. We now present the basic idea behind route discovery in the general case.

4.3.2 Basic Idea Behind RPSF in the General Case

The basic idea behind the route-discovery algorithm is as follows:

When a node wants to find a route to a destination, it assigns a sequence number to the route request (RREQ), selects a set of forwarding nodes using the criteria described in Section 4.3.1, and sends the request. When an intermediate node receives the request, it stores the sequence number, source ID and direction in its route-request table (RRT). If this path is shorter than an earlier path through which the same route request was received or if it is a new request then it stores the source ID and the next hop in the routing table (RT). If the node receiving the RREQ is the destination, it sends an RREP with the same sequence number and with HopCount initialized to 1. If the node receiving the RREQ is an intermediate node that has not already forwarded the same RREQ, it selects a set of forwarding nodes, increments the HopCount and forwards the request. If the intermediate node already knows a route to the destination, obtained through the corresponding RREP sent by the destination, it sends back a route reply (RREP) with a sequence number that is same as the one in the route request. The route reply propagates to the source along the path traversed by RREQ in the reverse direction. When a node receives an RREP corresponding to an RREQ (the ID of the node that sent the RREQ and the sequence number uniquely identifies a RREQ), it increments the HopCount, stores the destination ID, next hop ID and HopCount in its routing table (RT) and broadcasts the RREP if it is a forwarding node for the corresponding RREQ. An intermediate node receiving an RREP does not forward it until it receives the corresponding route request. Route replies propagate backward along the paths traversed by the corresponding RREQs. Thus the source can maintain multiple routes to the destination.

The formal description of the algorithm for route discovery in the general case is given in Figure 4.3. We now prove the correctness of our algorithm.

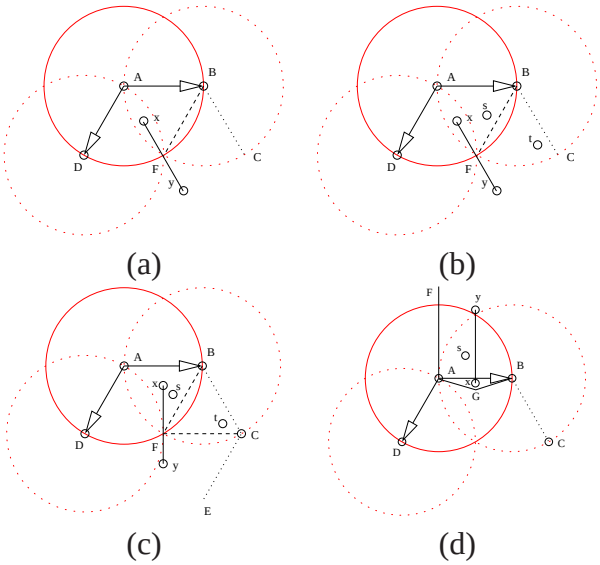


Figure 4.4: Proof of coverage in the general case

Lemma 4.1 *In the general case, a RREQ message sent by a source node s eventually reaches its destination d if d is reachable from s.*

Proof: We prove this lemma by contradiction. Suppose a RREQ message sent by s is not received by d. Then there exists a path $s, ...x, y, ...d$ from s to d and a node $x \neq d$ in the path such that x receives the RREQ message, but none of the nodes in the path beyond x receives the RREQ message. Suppose y is the first node following x in the path that does not receive the RREQ message. It is possible that y is d. y is within the transmission range of x, but y did not receive the RREQ message; that is, x has not forwarded the RREQ message. Two cases arise:

Case (1): x was chosen as a forwarding node by some node but x did not forward the RREQ message because, in its judgment, based on the reception of the RREQ message in various directions, all its neighbors including y would have received it. So, this case does not arise.

Case (2): x was not chosen as a forwarding node by any of the nodes from which x received the RREQ message.

Suppose A is one such node from which x received the RREQ message. Since x is within

the transmission range of A, A must have chosen at least two nodes B and D as forwarding nodes such that x lies in the sector $\angle DAB$ with $\angle DAB \leq 120°$. The following two sub-cases arise:

Case (2.1): x lies within the transmission range of both B and D as shown in Figure 4.4(a), and (b). In this case, since y did not receive the message, y must be outside the transmission range of A, B and D. Moreover, since y lies within the transmission range of x, then y has to lie in the sector $\angle DAB$, because, if y lies outside the sector $\angle DAB$ and within distance R from x, then y must be within distance R from one of the nodes A, B or D, which is a contradiction to our assumption that y did not receive the message. In this case, B would have selected x or some other node s in the sector $\angle DAB$ that is close to the line segment BF and farthest from B as a forwarding node according to the forwarding node selection **Rule 1**, or C would have selected some node that would cover y. Such a forwarding node s, selected by B, would have forwarded the message that would have been received by y if it is covered by s. However, if s is closer to B than y, then s may not cover y. In this case, if s will selects a forwarding node that covers y, then we are done. If s does not select a forwarding node that covers y, then s will do the same thing as B does and y will finally be covered since the next selected forwarding node similar to s will be closer to y than s. This contradicts the fact that y did not receive the message. Hence the Lemma is true in this case.

Case (2.2): x lies within the transmission range of either B or D but not both. Without loss of generality, suppose x lies within the transmission range of B but not D as shown in Figures 4.4(c) and (d) (the case in which x lies within the transmission range of D but not B is similar). Two sub-cases arise.

Case (2.2.1): y lies in the sector $\angle BAD$. (We already know that y is reachable from x but not from A, B or D.) Such a situation is shown in Figure 4.4(c). This case is similar to Case 2.1. Node B chooses x or some other node s as a forwarding node using **Rule 1** that would cover y, or C chooses some node that would cover y.

Case (2.2.2): y does not lie inside the sector $\angle BAD$.

Since y is reachable from x, x is close to the line segment AB, and y lies in the region enclosed by the common tangent line to the two circles with radius R and centers at A and B and the two circles themselves, as is shown in Figure 4.4(d). A simple calculation shows that if y is reachable from x, then x lies inside the triangle $\triangle GAB$ such that $\angle GAB = \angle GBA = \arctan(\frac{2R-\sqrt{4R^2-|AB|^2}}{|AB|}) \leq \arctan(2-\sqrt{3})$. In this case, A would have chosen x or some other node close to A in the triangular region $\triangle GAB$ as a forwarding node according to **Rule 2**. Hence y is covered by such a node, contradicting to the fact that y did not receive the message. Hence the Lemma is true in this case. □

4.3.3 Routing-Table Maintenance

As nodes move in the network, one or more links in an established route may break. In order to transmit the received data to the given destination, a node that detects broken links needs to repair the broken route and update its routing table. In many existing routing algorithms (like AODV and DSR), route-error messages notify source nodes about a broken link in a path, and the source nodes re-initiate route discovery to establish a new path. In RPSF, each node can maintain multiple next hops for a given destination, helping it repair a broken link by using other valid next hops. We take this approach for repairing broken links.

The basic idea behind routing-table maintenance in RPSF is as follows: When a node n detects a broken link, if it can not find another available link through which it can forward data to the destination, it first sends a route repair (RRPR) message to its one-hop neighbors. Upon receiving the RRPR message, each node updates its own route table by removing appropriate links, and checks if it is a good forwarding node to the destination. If so, it acknowledges the RRPR message. Otherwise, nothing needs to be done. If node n receives one or more acknowledgments for its RRPR message, the route has been repaired. Otherwise, it initiates a route discovery on behalf of the source. The formal description of

the route-maintenance algorithm is given in Figure 4.5.

```
Function: HandleBrokenLink(ID, NextHop)
    for each destination d in RT do
        Remove NextHop from NextHopInfoList of RT if appropriate;
        if NextHopInfoList is empty then /* No valid route to d */
            if it has buffered data destined to d then
                Init RRPR = (AckFlag=false, SenderID=ID, DstID=d,
                                HopCount=known hop count);
                Broadcast RRPR;

/* this code is executed periodically */
When b finds that neighboring node n is out of transmission range
    Call Function HandleBrokenLink(b, n);

When b drops a packet due to link failure to n
    Call Function HandleBrokenLink(b, n);

When b receives a RRPR message from n
    if RRPR.AckFlag then /* It is an ack for the RRPR request */
        if RRPR.SenderID == b then
            Append (NextHop=n, HopCount=RRPR.HopCount) to the
                NextHopInfoList corresponding to RRPR.DstID in RT;
        if b has buffered data for RRPR.DstID, and has a valid route
            Transmit the data;
    else /* It is a request for route repair */
        Remove n from the NextHopInfoList of RRPR.DstID in RT;
        if b has a route to RRPR.DstID and RRPR.HopCount > HopCount
        in RT for RRPR.DstID then
            Set RRPR.AckFlag = true;
            Set RRPR.HopCount = HopCount in RT + 1;
            Send RRPR back to RRPR.SenderID; /* Ack RRPR Request */
```

Figure 4.5: Route-maintenance algorithm

Let us follow an example to understand the route-maintenance algorithm. Figure 4.6 shows a network with five nodes, where S and D are the source and destination, respectively. Two paths exist from S to D. Suppose node A moves away. S now forwards data through B. If B also moves away, S no longer has any next hop for D in its route table. Therefore, S broadcasts a RRPR message. E receives and acknowledges the RRPR message. S then updates its routing table to reflect the fact that E is a next hop for destination D. If E also moves away, S re-initiates route discovery to find a path to D.

4.4 Performance Evaluation

We now present the results of performance evaluation of RPSF compared to AODV [56]. We first introduce the simulation model and then present the simulation results and our analysis of those results.

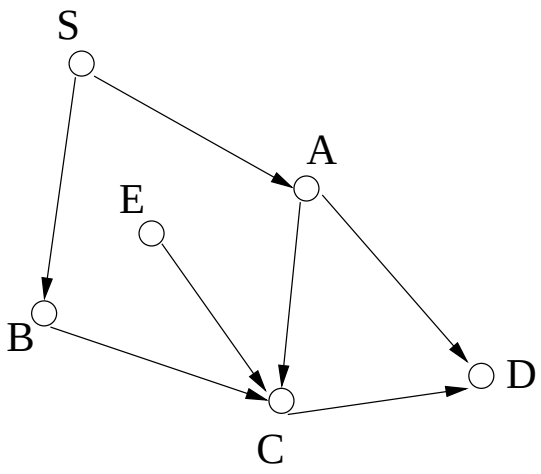


Figure 4.6: An example of route maintenance

4.4.1 Simulation Model

We used GloMoSim [74], a widely used network-simulation tools for studying the performance of routing algorithms for ad hoc networks, for evaluating the performance of RPSF.

We chose IEEE 802.11 [26] and IP as the MAC (Medium-Access Control) and network-layer algorithms respectively. All nodes have a fixed transmission range of 350m. We used the implementation of AODV that comes with the GloMoSim 2.0.3 package to compare its performance with RPSF. This implementation employs expanding-ring search to discover a route from a source to a destination. The neighborhood search range is enlarged by increasing the TTL (TimeToLive) field in the IP header of the request packets. AODV starts the search by setting TTL to 1 or to the previously known HopCount and repeats the search, increasing the TTL by 2 until it reaches 35 (it increments the TTL from 7 directly to 35, the maximum network diameter) or a RREP message is received before the timeout expires. This phased search reduces the route-establishment overhead for destinations that are close to the source. We simulated RPSF also with this mechanism to reduce the propagation of route request messages.

In the implementation of AODV, we set the route-discovery timeout to 10 seconds. The source checks if an RREP message is received within 80 times TTL milliseconds after the

last time it initiated a route request. In our implementation of RPSF, we update the direction and distance of the neighbors of each node every 2 seconds. Like AODV, the timeout for checking route replies for RPSF is set to 80 times TTL milliseconds. A node re-initiates a route request if it receives no reply before it times out.

4.4.2 Mobility Model

We adopt the steady state random-waypoint model [9, 18, 73] that is a widely used mobility model for simulations. Under this model, each node travels from a random location to a random destination at a random speed, the speed being uniformly distributed in a predefined range. After a node reaches its destination, it pauses for a predetermined amount of time and then moves to a new destination at a different randomly-chosen speed.

In our simulation, we set the speed range to 1 – 19 m/s. In order to study how mobility affects the performance of the routing algorithms, we selected pause times of 0, 30, 60, 90, 120, 200, 300, 500, and 900 seconds. When the pause time is 0 seconds, every node moves continuously. As the pause time increases, the network approaches the characteristics of a fixed network.

In a dense network, a path may always be available between any source-destination pair. On the contrary, if nodes are sparsely distributed, the network may be partitioned; moreover, in this case, node mobility can exacerbate the situation. In our performance evaluation, we simulated the following three scenarios to study the effect of density of the nodes on performance:

- $1500 \times 1500 m^2$ field with 200 nodes
- $1500 \times 1500 m^2$ field with 300 nodes
- $1500 \times 1500 m^2$ field with 400 nodes

We ran the simulation for each of the three scenarios for 15 simulated minutes.

4.4.3 Traffic Model

To measure the effect of network traffic on RPSF, we used 5, 10, 20 30, 40, or 50 CBR (constant bit-rate) data sources. We selected both the sources and the destinations randomly and uniformly. The sources transmit data between a start time and an end time; we selected all start and the corresponding end times randomly and uniformly within the 15-minute simulated interval in such a way that the start time precedes the end time. We fixed the size of data packets at 512 bytes and had each source generate packets at the rate of 4 packets per second. Measurements were taken after a settling time [73] of 150 simulated seconds.

4.4.4 Performance Metrics

We used the following three metrics to evaluate performance:

- Packet-delivery ratio: The ratio of the data packets delivered to the destinations to those generated by the CBR sources.
- End-to-end delay of data packets: This figure includes all possible delays, including those caused by buffering due to route discovery, queuing delay at the interface queue, retransmission delays at the MAC layer, and propagation and transfer time.
- Normalized routing overhead: The ratio of the number of routing control packets transmitted to the number of data packets delivered to the destinations. We count each time a node sends a routing control packet to its next-hop neighbor.

4.4.5 Confidence Intervals

As we mentioned earlier, we simulated 324 different scenarios (9 different pause times, 6 different numbers of CBR sources, three scenarios, and two algorithms, $9 \times 6 \times 3 \times 2 = 324$) for the two algorithms and simulated each scenario twenty times. We computed 95% confidence intervals for packet-delivery ratio, end-to-end delay and normalized routing overhead. Table 4.2 gives the distribution of tests and the related error ranges as a percentage

of the mean values. The values obtained for about 89.5% of the runs lie in the interval

$$[(meanvalue - meanvalue * 0.25), (meanvalue + meanvalue * 0.25)].$$

For clarity and simplicity, we do not plot error bars in the graphs.

Error range (%)	0 - 5	5 - 10	10 - 15	15 - 20	20 - 25	25 - 100	100 - 155
Tests (%)	32	23.6	20.6	7.6	4.7	9.9	1.6

Table 4.2: Distribution of tests in terms of confidence intervals

4.4.6 Performance Results

Scenario I

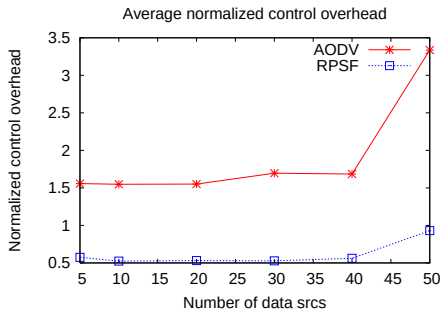
This scenario has 200 nodes.

Figure 4.7 and 4.8 show the performance of RPSF compared to AODV with respect to the three metrics for varying numbers of data sources and pause times. In Figure 4.7, the values of the three metrics are the average values taken over various pause times ranging from 0 to 900 seconds for different number of data sources. Figure 4.8 contains the values of the three metrics for various pause times, averaged over 5 to 50 CBR sources.

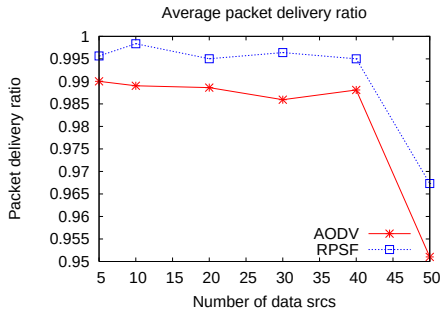
Under scenario I, the simulation results show that the average normalized routing overhead of AODV and RPSF is 2.23 and 0.67 respectively. As expected, RPSF uses fewer nodes for forwarding route requests than AODV, resulting in lower overhead. RPSF also has a better average packet-delivery ratio than AODV. RPSF has slightly higher average end-to-end delay, which becomes much more pronounced as the number of CBR sources increases. The AODV line changes more sharply than RPSF as the pause time increases. Therefore, performance of RPSF is more stable than AODV when nodes with high mobility are involved in the simulation. The results obtained in scenario II and III also conforms with this observation.

Scenario II

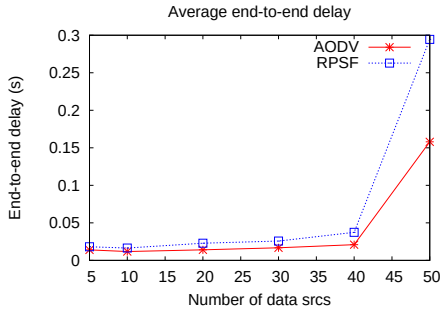
This scenario has 300 nodes.



(a)



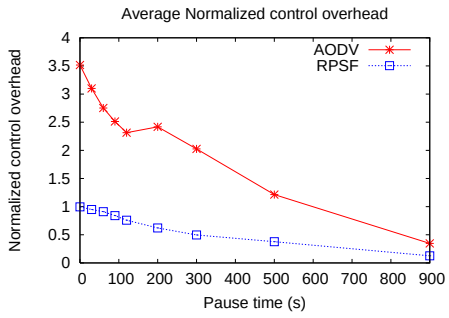
(b)



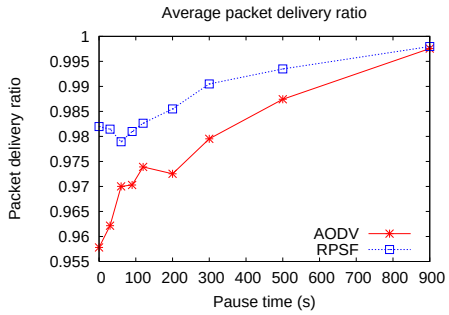
(c)

Figure 4.7: Varying number of data sources in scenario I (200 nodes)

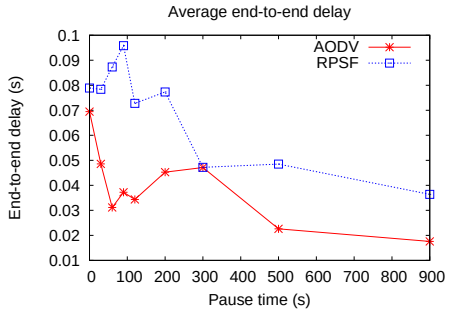
Figures 4.9 and 4.10 show the performance of RPSF compared to AODV with respect to the three metrics for varying numbers of data sources and pause times. In this scenario, RPSF performs better than AODV with respect to all three metrics. For instance, the average values of the normalized routing overhead, packet-delivery ratio, and end-to-end delay of RPSF are 0.81, 0.986, and 0.052 respectively, while the three measurements for AODV are 4.57, 0.963, and 0.055 respectively. In this scenario, AODV has slightly lower end-to-end delay when fewer CBR sources are involved. However, it has higher end-to-end delay



(a)



(b)



(c)

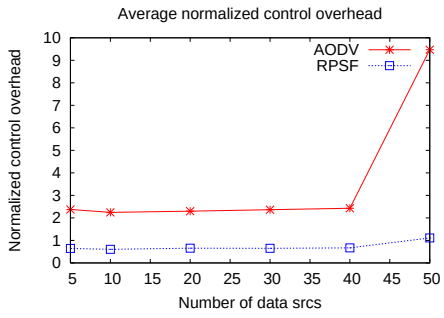
Figure 4.8: Varying the pause time in scenario I (200 nodes)

when there are 50 CBR sources, which makes its average value higher than that of RPSF. Again, as results in Figure 4.10 indicate, the performance of RPSF is much more stable than AODV with respect to node mobility.

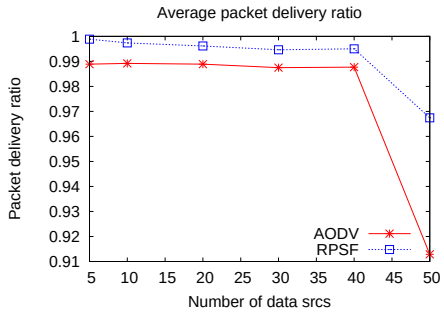
Scenario III

This scenario has 400 nodes.

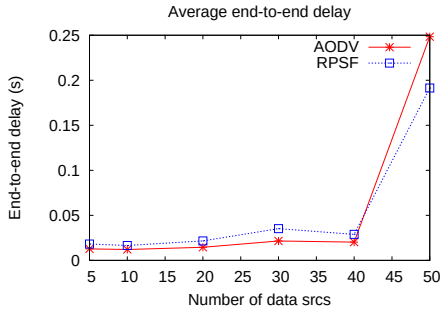
Figures 4.11 and 4.12 show the performance of RPSF compared to AODV with respect to the three metrics for varying numbers of data sources and pause times. The simulation



(a)



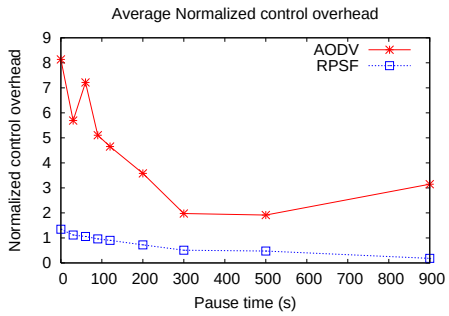
(b)



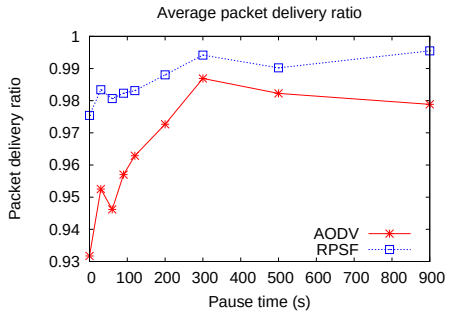
(c)

Figure 4.9: Varying number of data sources in scenario II (300 nodes)

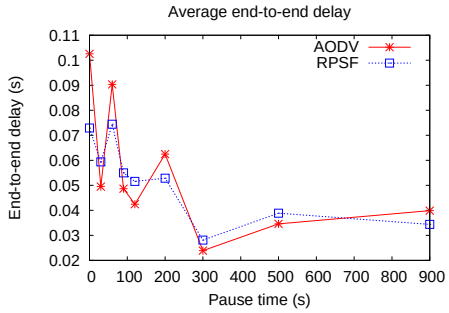
results under this scenario are similar to the simulation results under scenario II. RPSF has much lower routing control packet overhead, higher packet-delivery ratio, and lower end-to-end delay than AODV in this case. Even in such a dense network, the average normalized routing overhead of RPSF is 1.07, which is only 0.4 more than that in scenario I and 0.26 more than that in scenario II. This case also demonstrates that RPSF is much more stable than AODV.



(a)



(b)



(c)

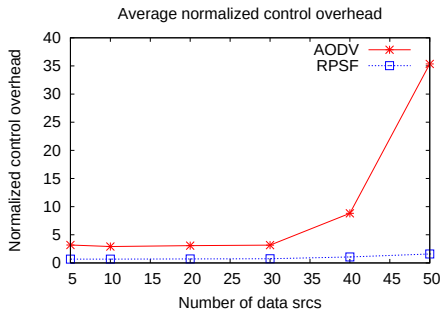
Figure 4.10: Varying the pause time in scenario II (300 nodes)

4.4.7 Analysis

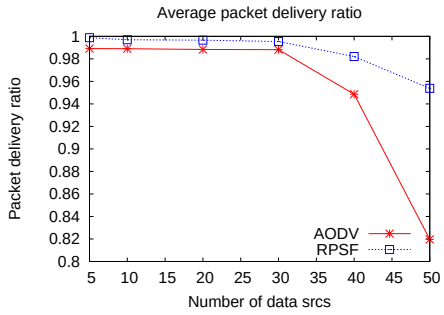
We make the following observations based on the simulation results.

Routing Overhead

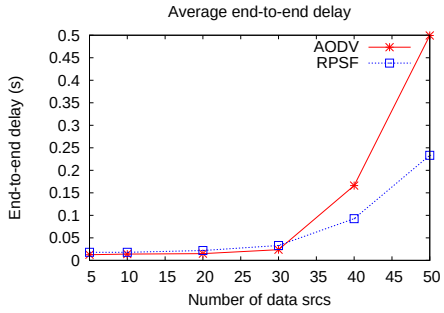
In sparse networks, the two algorithms have similar packet-delivery ratio. Since RPSF tries to guarantee the delivery of generated data packets, it issues many useless RREQ messages searching for non-existent paths in a partitioned network. However, as the network becomes



(a)



(b)

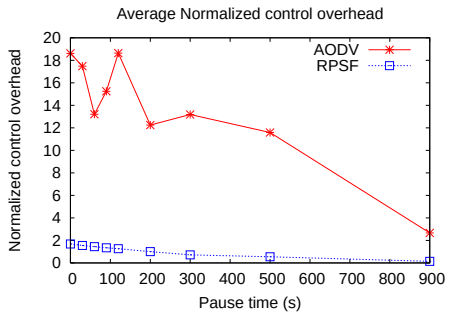


(c)

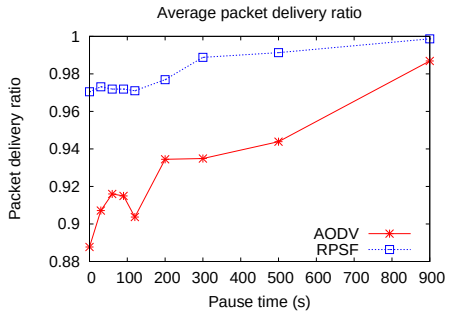
Figure 4.11: Varying number of data sources in scenario III (400 nodes)

denser, the number of route-control packets issued by RPSF does not greatly increase. This gentle rise is due to RPSF's selective forwarding mechanism in flooding RREQ messages. This mechanism is very efficient in controlling routing overhead by limiting the number of nodes that forward the RREQ messages in dense networks.

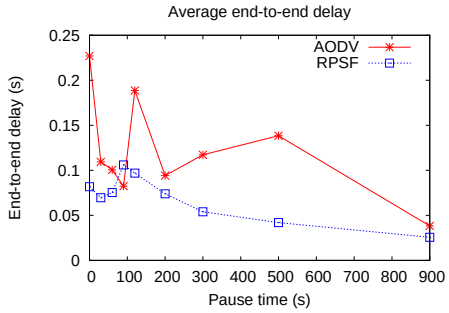
The average normalized routing overhead under all three scenarios for AODV and RPSF are 6.81 and 0.86 respectively. RPSF has relatively constant overhead as the number of nodes in the network increases from 200 to 400. On the contrary, AODV incurs much



(a)



(b)



(c)

Figure 4.12: Varying the pause time in scenario III (400 nodes)

more routing overhead as the number of nodes increases. The same thing happens as the number of CBR sources increases or the nodes become more mobile (or pause time decreases). Thus, RPSF performs much better than AODV with respect to routing overhead in networks with highly mobile nodes, networks in which nodes are densely distributed, or heavily loaded networks.

End-to-end Delay

The overall average end-to-end delay for AODV and RPSF are 0.072 and 0.064 respectively. RPSF has highest end-to-end delay in a sparse network. This result arises because it is hard to repair a broken path in a sparse network. As the density of the network increases, more paths become available, and the end-to-end delay is more dependent on the number of hops and the network load. There the end-to-end delay under RPSF is comparable to AODV. In high-density and high-load networks, RPSF has lower end-to-end delay than AODV because RPSF has much lower routing overhead.

Network Load

As we expect, as network load increases, both algorithms show increasing normalized routing overhead and end-to-end delay. However, RPSF is relatively stable as the number of data sources increases, but AODV degrades greatly.

4.5 Discussion

Our measurements show that RPSF is far superior to AODV, especially for higher node densities. At 400 nodes in a region of $1500 \times 1500m^2$, each node has, on average, about 68 neighboring nodes within transmission distance. This density is typical of a scenario such as attendees of a conference trying to establish an ad hoc network of their laptops. However, this superiority has a price. We require accurate position knowledge, which implies additional hardware (GPS or smart antennas) and its concomitant battery drain. However, unlike existing position-based routing algorithms, our algorithm requires that each node know the relative position only of its neighboring nodes, not of all nodes in the network. We also require that communication links be bidirectional, even at extreme ranges. A practical implementation of RPSF would most likely choose forwarding nodes at perhaps 80% of the transmission range to improve the chance of bidirectionality of links; this choice would increase the average path length, and therefore the end-to-end delay.

4.6 Conclusion

This chapter proposes a novel mechanism for flooding in ad hoc networks. It presents the RPSF routing algorithm that employs that mechanism. In a dense network, we have demonstrated that the algorithm efficiently selects a limited, but sufficient, set of forwarding nodes to flood the route requests. We compared the performance of our algorithm with a well known routing algorithm AODV. Simulation results show that RPSF always has much lower normalized routing overhead than AODV.

Chapter 5

Conclusion and Future Work

5.1 Summary

This dissertation focused on designing efficient algorithms for achieving fault tolerance in distributed systems and routing in mobile ad hoc networks.

We presented a novel *communication-induced* checkpointing algorithm that makes every checkpoint belong to a consistent global checkpoint. Under this algorithm, every process stores the tentative checkpoint in memory first and then flushes it to stable storage when there is no contention for accessing stable storage or after finalizing the tentative checkpoint. Messages sent and received after a process takes a tentative checkpoint are logged into memory until the tentative checkpoint is finalized. Since a tentative checkpoint can be flushed to stable storage any time before finalizing it, contention for stable network storage that arises due to several processes storing the checkpoints simultaneously is reduced/eliminated. Moreover, unlike existing communication-induced checkpointing algorithms, our algorithm, in general, does not force a process to take a checkpoint before processing any received message in order to prevent useless checkpoints. Thus, a process can first process the received message and then take the checkpoint. This improves the response time for messages. It also helps a process take the regularly scheduled basic checkpoints at those times. If messages are not frequently exchanged among processes, additional control messages may be required for the algorithm to collect consistent global checkpoints in a timely manner. We augmented the basic algorithm with control messages to speed up the

collection of consistent global checkpoints in a timely manner for applications in which processes do not communicate frequently. We conducted a performance evaluation of the algorithm and studied the overhead induced by the control messages which also helps in determining when control messages are needed. We also compared the performance of our algorithm with Vaidya's algorithm [66]. In minimizing the contention for stable storage at the network file server, our algorithm always performs better than Vaidya's algorithm. Our algorithm also has other desirable features such as the scalability, low control messages (or even no control messages) and less checkpoint latency compared to Vaidya's algorithm algorithm.

We designed two novel methods for suppressing redundant route request messages when broadcasting them in mobile ad hoc networks. We then presented two new routing protocols, namely, triangle based routing (TBR) protocol and routing protocol with selective forwarding (RPSF), for mobile ad hoc networks, that employ the mechanisms. Performance of TBR and RPSF have been evaluated with GloMoSim simulator. We have demonstrated that the protocols efficiently select limited, but sufficient, set of nodes to forward the route requests. We compared the performance of our algorithms with a well known routing algorithm AODV. Simulation results show that both TBR and RPSF always have much lower normalized routing overhead than AODV.

5.2 Future Work

In the future, we will focus on designing better algorithms for achieving fault tolerance in distributed systems and routing in mobile ad hoc networks.

In Chapters 3 and 4, we demonstrated the advantages of TBR and RPSF with respect to routing overhead in mobile ad hoc networks. They can be tuned to be more efficient, more adaptive, and easier to implement. We discuss below the ways in which they can be fine tuned.

Algorithm for Selecting Forwarding Nodes

The algorithms for selecting forwarding nodes in the current implementation of TBR and RPSF may select more forwarding nodes than needed. This gives an opportunity for improving them by designing more efficient forwarding node selection algorithm.

There are two ways in which the method of selecting forwarding nodes can be improved. One approach is to modify the method so that the forwarding nodes lying near the network edge do not select further forwarding nodes if all of their neighbors have been already covered. The other approach is to modify the forwarding node selection algorithm so that the number of selected forwarding nodes are reduced.

The former approach is hard to achieve due to the difficulty in determining the network edge. However, some of the edge nodes can be detected by checking if they have at least one neighbor in any sector of 180^o centered at themselves. They are considered as internal nodes if the condition holds or edge nodes otherwise. This information is exchanged between neighboring nodes. Therefore, each node has basic knowledge about its closeness to the network edge. Although the information does not determine the exact network edge, it is enough for forwarding nodes to determine if they need to select more forwarding nodes further.

The latter approach is more about algorithm design. The algorithm employed in the current implementation of RPSF only chooses forwarding nodes in such a way that each node in a connected network is covered by one or more selected forwarding nodes. However, it may select forwarding nodes more than necessary since it does not fully meet the two sub-criteria, selecting as few forwarding nodes as possible and the selected forwarding nodes are as far away as possible. We plan to develop and implement efficient node selection algorithm which closely matches the criteria.

Multi-path Routing

Multi-path routing is a popular method for addressing reliability issue and balancing network load. In multi-path routing protocols, researchers are more interested in node disjoint and edge disjoint multi-paths for routing. Supporting reliable communications and balancing network load are easier to achieve with the help of node disjoint or edge disjoint paths. Although TBR and RPSF are multi-path routing protocols, they do not exploit any property of multi-path. Therefore, utilizing multi-path properties for reliable communication and studying its performance is one of our goals.

In the current implementation of TBR and RPSF, a multi-path route, if available, is discovered in each route discovery process. However, it can not be used to support reliable communication because the multiple paths may not be node disjoint or edge disjoint. Therefore, we plan to extend them so that they are able to search for node disjoint and edge disjoint paths. In addition, we will measure the reliability of a route in terms of the number of node disjoint or edge disjoint paths. In the future, for TBR and RPSF, the number of paths for each route will be considered as a QoS parameter.

Prerequisite Information for RPSF

In the current version of RPSF, it is required that every node in the network needs the distance and direction information of each of its neighbors. This is possible only if each mobile node is equipped with smart antennas. We will relax this requirement and design efficient forwarding node selection algorithms.

Since obtaining direction information is much more difficult than getting distance information, we will relax the condition of requiring direction information. Under the current RPSF, direction information helps in reducing the number of selected forwarding nodes and guarantees that all of the two-hop neighbors of the selectors are/will be covered by some forwarding nodes. Therefore, to relax the direction information requirement, hosts running RPSF need two-hop neighbor information at least.

By relaxing direction information requirement, the problem of forwarding node selection can be described as: for node *A*, find a set (of one-hop neighbors) as small as possible such that every two-hop neighbor is at least covered by a node in the set. We refer the neighbors in the set to the forwarding nodes of node *A*. This problem has been addressed in many papers in the literature using various graph models, e.g. unit disk graph [10], relative neighborhood graph (RNG) [12, 58, 62], and dominating set [6, 70, 71]. However, we still feel the need to propose a new algorithm for forwarding node selection based on RNG model, and compare it with the developed algorithms with regard to the performance.

Quality of Service

Ad hoc networks are likely to support multimedia applications which require high QoS requirements. To support such applications, routing protocols which ensure the required level of QoS need to be developed. We propose to extend our already developed RPSF to support QoS and also propose to develop new and more efficient protocols that support QoS.

Under the new RPSF, we plan to only allow partial network capacity that are used for QoS control. This is to avoid the scenario in which some nodes reserve all of the network capacity and prevent other nodes from using the network.

Security

As the mobile ad hoc networks have the potential for being deployed in critical areas, such as business meetings, and battle fields, security becomes an important issue. We will extend TBR and RPSF to secure routing protocols. The security issues that need to be addressed related to routing in mobile ad hoc networks are:

1. Ensuring anonymity: preventing attackers from knowing about the source and destination in the routing packets.

2. Privacy: preventing attackers from knowing the found routes.

3. Alteration: preventing attackers from modifying found routes.

4. Masquerading: preventing attackers from providing sources with false routes to destinations.

5. Denial of service: preventing attackers from initiating too many route requests.

Among the security issues discussed above, ensuring anonymity is one of the most important issues to be addressed. To ensure anonymity, TBR and RPSF need to be changed to source routing, since table driven routing can not be used to hide the addresses of destinations. After the change is made, anonymity can be achieved by efficiently changing routes for data communication from time to time and randomly adding redundant nodes before the source node and after the destination node respectively on the routes. The changing routes can complicate the analysis of data traffic pattern. And randomly adding redundant nodes on the routes can hide the source and destination nodes of each data transmission from the attackers.

For adding the other security features, efficient encryption and authentication mechanisms need to be developed. With these mechanisms available, the security features can be easily incorporated. For example, privacy can be achieved by encrypting the source routes for data transmission (and allow each intermediate node on the route to get the next hop easily); alterations can be detected by appropriate hash functions; and masquerading and denial of service attacks can be prevented given well defined authentication mechanisms. Therefore, our future work here mainly focuses in the encryption and authentication mechanisms that are deployable in mobile ad hoc networks.

Fault-Tolerance

For applications in environments such as battlefields, disaster areas, and natural habitats, mobile nodes could fail, which may partition the network. Fault tolerance provides the capability for networks to continue functioning in the presence of failed nodes. Nevertheless, not much work has been done in the area of fault tolerance in mobile ad hoc networks.

In the literature, there are two major fault tolerance schemes, namely, checkpointing-recovery and redundancy. Under the former scheme, the executing states of a process are checkpointed from time to time. When a process fails, the processes involved in the computation can be restarted from the latest consistent saved states. Under the later scheme, nodes prone to failure are deployed with more than one identical copies (nodes), but only one copy (called primary copy) is active and participates in computation. The other copies (called secondary copies) monitor the behavior of the primary copy and change their status according to the executing status of the primary copy. When the primary copy fails, one of the secondary copies will be elected as the new primary copy and take the place of the failed copy and undertake the computation. Clearly, the two schemes have pros and cons in different applications. For example, the former scheme is considered to be better in terms of costs, while the other scheme is preferable in terms of recovery time.

Bibliography

[1] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, and Causal. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 229–236, June 1995.

[2] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. A Communication Induced Algorithm that Ensures the Rollback Dependency Trackability. In *Proceedings of the* 27^{th} *International Symposium on Fault-Tolerant Computing, Seattle*, July 1997.

[3] G. Barigazzi and L. Strigini. Application-transparent Setting of Recovery Points. In *Proceedings of* 13^{th} *IEEE Symposium on Fault-Tolerant Computing*, pages 48–55, June 1983.

[4] S. Basagni, I. Chlamtac, V.R. Syrotiuk, and B.A. Woodward. A Distance Routing Effect Algorithm for Mobility (DREAM). In *MobiCom 1998: Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 76–84, New York, NY, USA, 1998. ACM Press.

[5] L. Blazevic, J.L. Boudec, and S. Giordano. A Location-Based Routing Method for Mobile Ad Hoc Networks. *IEEE Transactions on Mobile Computing*, 4(2):97–110, March/April 2005.

[6] P. Bose, P. Morin, I. Stojmenovi, and J. Urrutia. Routing with Guaranteed Delivery in Ad Hoc Wireless Networks. *Wireless Networks*, 7(6):609–616, 2001.

[7] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *3rd int. Workshop on Discrete Algorithms and methods for mobile computing and communications*, 1999.

[8] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with Guaranteed Delivery in Ad Hoc Wireless Networks. *Wireless Networks*, 7(6):609–616, 2001.

[9] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva. A Performance Comparison of Multi-hop Wireless Ad Hoc Network Routing Protocols. In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 85–97, Dallas, Texas, United States, 1998. ACM Press.

[10] Gruia Calinescu, Ion I. Mandoiu, Peng-Jun Wan, and Alexander Zelikovsky. Selecting forwarding neighbors in wireless ad hoc networks. *ACM/Kluwer Mobile Networks and Applications*, 9(2):101–111, April 2004.

[11] Guohong Cao and Mukesh Singhal. Checkpointing with Mutable Checkpoints. *Theoretical Computer Science*, 290(2):1127–1148, January 2003.

[12] Julien Cartigny, David Simplot, and Ivan Stojmenovic. Localized minimum-energy broadcasting in ad-hoc networks. In *Proc. IEEE Infocom 2003*, San Francisco, USA, 2003.

[13] K. M. Chandy and L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[14] E. Chavez, S. Dobrev, E. Kranakis, J. Opatrny, L. Stacho, and J. Urrutia. Local construction of planar spanners in unit disk graphs with irregular transmission ranges. In *LATIN*, 2006.

[15] T. Clausen, C. Dearlove, and P. Jacquet. The Optimized Link-State Routing Protocol version 2. Mobile Ad-hoc Networks (MANET) Internet Drafts, June 2006.

[16] M. Colagrosso, N. Enochs, and T. Camp. Improvements to Location-Aided Routing Through Directional Count Restrictions. In *International Conference on Wireless Networks*, pages 924–929, 2004.

[17] Om P. Damani, Yi-Min Wang, and Vijay K. Garg. Distributed Recovery with K-optimistic Logging. *J. Parallel Distrib. Comput.*, 63(12):1193–1218, 2003.

[18] S.R. Das, C.E. Perkins, and E.E. Royer. Performance Comparison of Two On-demand Routing Protocols for Ad Hoc Networks. In *Proceedings of the INFOCOM*, pages 3–12, 2000.

[19] E. N. Elnozahy, L. Alvisi, Y.M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[20] E. N. Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[21] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-recovery with Low Overhead, Limited Roll-back and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[22] E. N. Elnozahy and W. Zwaenepoel. On the Use and Implementation of Message Logging. In *FTCS-24: 24th International Symposium on Fault Tolerant Computing*, pages 298–309, Austin, Texas, 1994. IEEE Computer Society Press.

[23] Z. Haas. A New Routing Protocol for the Reconfigurable Wireless Networks. In *Proceedings of the IEEE International Conference on Universal Personal Communications*, pages 562–566, October 1997.

[24] Z.J. Haas and M.R. Pearlman. The Performance of Query Control Schemes for the Zone Routing Protocol. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 23–29, Seattle, Washington, United States, 1999. ACM Press.

[25] Y.K. Ho and R.S. Liu. Checkerboard-Like Routing Protocol for Ad Hoc Mobile Wireless Networks. *Wireless Personal Communications*, 33(2):177–196, April 2005.

[26] IEEE. Wireless LAN Medium Access Control and Physical Layer Specifications. IEEE 802.11 Standard, IEEE Computer Society LAN MAN Standards Committee, August 1999.

[27] R. Jain, A. Puri, and R. Sengupta. Geographical Routing Using Partial Information for Wireless Ad Hoc Networks. *Personal Communications*, 8(1):48–57, February 2001.

[28] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[29] D.B. Johnson and D.A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[30] T-Y. Juang and S. Venkatesan. Crash Recovery with Little Overhead. In *Proceedings of* 11^{th} *International Conference on Distributed Computing Systems*, pages 349–361, 1990.

[31] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *ACM/IEEE International Conference on Mobile Computing and Networking*, 2000.

[32] B. Karp and H.T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Mobile Computing and Networking*, pages 243–254, 2000.

[33] Junguk L. Kim and Taesoon Park. An Efficient Protocol for Checkpointing Recovery in Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):955–960, August 1993.

[34] K. H. Kim. A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes. In *Proceedings of* 16^{th} *IEEE Symposium on Fault-Tolerant Computing*, pages 130–135, June 1986.

[35] Y.B. Ko and N.H. Vaidya. Location-Aided Routing (LAR) in Mobile Ad Hoc Networks. In *Mobile Computing and Networking*, pages 66–75, 1998.

[36] R. Koo and S. Toueg. Checkpointing and Roll-back Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[37] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *11th Canadian Conference on Computational Geometry: an Introduction*, pages 51–54, 1999.

[38] F. Kuhn, R. Wattenhofer, and A. Zollinger. Asymptotically optimal geometric mobile ad-hoc routing. In *6th International workshop on discrete algorithms and methods for mobile computing and communications (DIALM'02)*, 2002.

[39] L. Lamport. Time, Clocks and Ordering of Events in Distributed Systems. *Communications of the ACM.*, 21(7):558–565, July 1978.

[40] S. Lee, B. Bhattacharjee, and S. Banerjee. Efficient Geographic Routing in Multihop Wireless Networks. In *MobiHoc 2005: Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 230–241, New York, NY, USA, 2005. ACM Press.

[41] K. Li, J. F. Naughton, and J. S. Plank. Checkpointing Multicomputer Applications. In *Proceedings of* 10^{th} *Symposium on Reliable Distributed Systems*, pages 2–11, 1991.

[42] P.S. Mandal and K. Mukhopadhyaya. Concurrent Checkpoint Initiation and Recovery Algorithms on Asynchronous Ring Networks. *Journal of Parallel and Distributed Computing*, 64(5):571–685, 2004.

[43] D. Manivannan and M. Singhal. A Low-overhead Recovery Technique using Quasi-synchronous Checkpointing. In *Proceedings of the* 16^{th} *IEEE International Conference on Distributed Computing Systems*, pages 100–107, Hong Kong, May 1996.

[44] D. Manivannan and M. Singhal. Asynchronous Recovery Without Using Vector Timestamps. *Journal of Parallel and Distributed Computing*, 62(12):1695–1728, December 2002.

[45] D. Manivannan and Mukesh Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification . *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999.

[46] M. Mauve, J. Widmer, and H. Hartenstein. A Survey on Position-Based Routing in Mobile Ad-Hoc Networks. *IEEE Network Magazine*, 15(6):30–39, November 2001.

[47] S. Murthy and J.J. Garcia-Luna-Aceves. An Efficient Routing Protocol for Wireless Networks. *ACM Mobile Networks and App. J., Special Issue on Routing in Mobile Communication Networks*, pages 183–197, Oct. 1996.

[48] Robert H. B. Netzer and Jian Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[49] S.Y. Ni, Y.C. Tseng, Y.S. Chen, and J.P. Sheu. The Broadcast Storm Problem in a Mobile Ad Hoc Network. In *MobiCom 1999: Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 151–162, New York, NY, USA, 1999. ACM Press.

[50] R.G. Ogier, F.L. Templin, and M.G. Lewis. Topology Dissemination Based on Reverse-Path Forwarding (TBRPF), February 2004.

[51] V.D. Park and M.S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proceedings of the INFOCOM*, pages 1405–1413. IEEE Computer Society, 1997.

[52] W. Peng and X. Lu. AHBP: An Efficient Broadcast Protocol for Mobile Ad Hoc Networks. *Journal Computer Science Technology*, 16(2):114–125, 2001.

[53] W. Peng and X.C. Lu. On the Reduction of Broadcast Redundancy in Mobile Ad Hoc Networks. In *MobiHoc*, pages 129–130, 2000.

[54] C.E. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, pages 234–244, London, United Kingdom, 1994. ACM Press.

[55] C.E. Perkins and I. Chakeres. Dynamic MANET On-demand (DYMO) Routing. Mobile Ad-hoc Networks (MANET) Internet Drafts, June 2006.

[56] C.E. Perkins and E.M. Royer. Ad Hoc On-Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, February 1999.

[57] J. S. Plank. *Efficient Checkpointing on MIMD Architectures*. PhD thesis, Priceton University, June 1993.

[58] M. Seddigh, J. Solano, and I. Stojmenovic. RNG and internal node based broadcasting in one-to-one wireless networks. *ACM Mobile Computing and Communications Review*, 5(2):37–44, April 2001.

[59] Luis Mourae Silva and Jou ̄ao Gabriel Silva. Global Chedkpointing for Distributed Programs. In *Proceedings of Symposium on Reliable Distributed Systems*, pages 155–162, 1992.

[60] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. In *Proceedings of* 8^{th} *ACM Symposium on Principles Distributed Computing*, pages 223–238, August 1989.

[61] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[62] Kenneth J. Supowit. The relative neighborhood graph, with an application to minimum spanning trees. *Journal of ACM*, 30(3):428–448, 1983.

[63] H. Takagi and L. Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *IEEE transactions on communications*, 32(3):246–257, Mar. 1984.

[64] G. Toussaint. The relative neighborhood graph of finite planar set. *Pattern Recognition*, 4(12):261–268, 1980.

[65] N. Vaidya. On Checkpoint Latency. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1995.

[66] N. Vaidya. Staggered Consistent Checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):694–702, January 1999.

[67] P.J. Wan, K.M. Alzoubi, and O. Frieder. Distributed Construction of Connected Dominating Set in Wireless Ad Hoc Networks. *Mobile Network Applications*, 9(2):141–149, 2004.

[68] R. Wattenhofer, L. Li, P. Bahl, and Y.-M. Wang. Distributed Topology Control for Power Efficient Operation in Multihop Wireless Ad Hoc Networks. In *Proceedings of the INFOCOM*, volume 3, pages 22–26, April 2001.

[69] B. Williams and T. Camp. Comparison of Broadcasting Techniques for Mobile Ad Hoc Networks. In *Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC)*, pages 194–205, 2002.

[70] J. Wu, F. Dai, M. Gao, and I. Stojmenovic. On Calculating Power-Aware Connected Dominating Set for Efficient Routing in Ad Hoc Wireless Networks. *Journal of Communications and Networks*, 5(2):169–178, March 2002.

[71] Jie Wu. Extended dominating-set-based routing in ad hoc wireless networks with unidirectional links. *IEEE Trans. Parallel Distrib. Syst*, 13(9):866–881, September 2002.

[72] G. Xing, C. Lu, R. Pless, and Q. Huang. On Greedy Geographic Routing Algorithms in Sensing-Covered Networks. In *MobiHoc 2004: Proceedings of the 5th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 31–42, New York, NY, USA, 2004. ACM Press.

[73] J. Yoon, M. Liu, and B. Noble. Random Waypoint Considered Harmful. In *Proceedings of the INFOCOM*, pages 1312–1321, 2003.

[74] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

Vita

Qiangfeng Jiang
Place of Birth: Jiangxi Province, China

- **Research Interests**
  - Wireless and sensor networks, distributed and mobile computing systems, and mobile agents.
- **Education**
  - M.S. in Computer Networks, East-China Institute of Computer Technology, Shanghai, China, 2001
  - B.S. in Computer Software, Tongji University, Shanghai, China, 1998
- **Publications**
  - Refereed journal papers:
    * Y. Sun, Q. Jiang and M. Singhal. "A Hill-Area-Restricted Geographic Routing Protocol for Mobile Ad Hoc and Sensor Networks". Computer Journal, 55(8):932-949, 2012
    * Y. Sun, Q. Jiang and M. Singhal. "A Pre-Processed Cross Link Detection Protocol for Geographic Routing in Mobile Ad Hoc and Sensor Networks under Realistic Environments with Obstacles". Journal of Parallel and Distributed Computing, 71(7):1047-1054, 2011
    * Y. Sun, Q. Jiang and M. Singhal. "An Edge Constrained Localized Delaunay Graph for Geographic Routing in Mobile Ad Hoc and Sensor Networks". IEEE Transaction on Mobile Computing, 9(4):479-490, 2011
    * Q. Jiang, Y. Luo and D. Manivannan. "An Optimistic Checkpointing and Message Logging Approach for Consistent Global Checkpoint Collection in Distributed Systems". Journal of Parallel and Distributed Computing, 68(12):1575-1589, December 2008, Elsevier.
    * Q. Jiang, R. A. Finkel, D. Manivannan and M. Singhal. "RPSF: A Routing Protocol with Selective Forwarding for Mobile Ad-Hoc Networks". Wireless Personal Communications journal, 43(2):411-436, October, 2007, Springer Verlag.
    * J. Yang, Q. Jiang and D. Manivannan. "A Fault-Tolerant Channel Allocation Algorithm for Cellular Networks with Mobile Base Stations". IEEE Transactions on Vehicular Technology, 56(1):349-361, January, 2007.

    - J. Yang, Q. Jiang, D. Manivannan and M. Singhal. “A Fault-Tolerant Distributed Channel Allocation Scheme for Cellular Networks”. IEEE Transactions on Computers, 54(5):616-629, May, 2005.

  - Refereed conference papers:

    - Q. Jiang and D. Manivannan. “An Optimistic Checkpointing and Message Logging Approach for Consistent Global Checkpoint Collection in Distributed Systems”. In Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), March 26- 30, 2007, Long Beach, California USA.
    - Y. Sun, Q. Jiang, and M. Singhal. “An Edge Constrained Localized Delaunay Graph for Geographic Routing in Mobile Ad Hoc Networks”. In IEEE Wireless Communications and Networking Conference (WCNC), March 11-15, 2007, Hong Kong, China.
    - D. Manivannan, Q. Jiang, J. Yang, K. Persson and M. Singhal. “Asynchronous Recovery based on Staggered Quasi-Synchronous Checkpointing”. In Springer Lecture Notes in Computer Science Series No.3741 pp 117-128.
    - D. Manivannan, Q. Jiang, J. Yang, K. Persson and M. Singhal. “Asynchronous Recovery based on Staggered Quasi-Synchronous Checkpointing”. In International Conference on Distributed Computing and Networking (ICDCN), 2005.
    - Q. Jiang and D. Manivannan. “Routing Protocols for Sensor Networks”. In Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC 2004), January 5-8, 2004, Las Vegas, Nevada, USA.
    - Q. Jiang and D. Manivannan. “Securing Mobile Agents Against Malicious Hosts”. In Proceedings of 2003 International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC’03), Vienna, Austria, February 12-14, 2003, pages 640-651.

  - Technical reports

    - Q. Jiang and D. Manivannan. “Routing Protocols for Sensor Networks”. University of Kentucky, Computer Science Department, Technical Report No. 383-03.

- **Research Experience**

  - Research Assistant, 08/2002 – 11/2007, Department of Computer Science, University of Kentucky

    - Designed and implemented a network tool using UDP and the Forward Error Correction (FEC) for transferring files reliably in lossy channels. Demonstrated that the tool outperforms TCP in Emulab testbed.
    - Designed and implemented a suite of simulation tools for automating the process of distributing simulation load over multiple network nodes, collecting simulation data, and generating visualized reports.
    - Implemented three routing protocols: TFRC, RPSF, and TBR.

    - Designed and implemented an XML search engine using XPATH for the ARCHway project.
    - Designed and implemented a database tool for importing chemical formula information in RTF files into a given relational database using ODBC.
    - Developed many C libraries, such as file I/O management, simplified socket interface, general sorting interface, and many frequently used data structures, for playing with Linux and making my programming work easy.
  - Software Engineer, 08/2000 – 08/2001, Tekview Technology Co. Ltd., Shanghai
    - Wrote Orbix sample codes for customers. Taught prospective customers Orbix and CORBA.
    - Installed Orbix in various systems, e.g. Linux, AIX, Solaris, NT, and XP.
    - Directed Orbix users to solve their technical problems.
  - Research Assistant, 08/1999 – 08/2000, East-China Institute of Computer Technology, Shanghai
    - Developed service migration and atomic action modules for the Fault Tolerant CORBA Technologies, a key pre-research project under the country's Ninth Five-Year Development program of China.
    - Designed and implemented a communication framework based on CORBA technologies for the Distributed Command System.
  - Software Engineer, 05/1998 – 09/1998, Kinpo Electronics (Shanghai) Co., Ltd., Shanghai
    - Developed an Electronic Dictionary Editor which imports, exports, merges, and edits dictionary entries in a SQL server database.
  - Research Assistant, 07/1996 – 03/1998, Department of Computer Science at Tongji University, Shanghai
    - Developed an efficient configuration reader for the Management System of Personnel in Railway Ministry
    - Developed a network communication support module for the Medical Insurance System in Railway Ministry
    - Developed a Query By Example (QBE) module for the Visualized Report System

- **Professional Skills**
  - Network and Linux Kernel programming
  - Distributed Computing Environment: CORBA (Orbix) and DCE (Gradient DCE)
  - Operating Systems: Linux, Digital UNIX, Solaris UNIX, HP-UX and Windows NT/9x/XP
  - Programming languages: C/C++, Perl, Java (Eclipse), CGI, XML, Tcl, Visual Basic, Delphi, Pascal and SQL
  - Developing methods: OOA&OOD, software engineering and design patterns

- Network simulation tools: GloMoSim, NS2 and Sense-3.0.3
- Database: SQL server, Oracle, DB2 and MySQL

- **Professional Activities**
  - Served as a reviewer for the following journals
    - IEEE Network
    - International Journal of Computers and Applications
    - Journal of Information Science and Engineering
  - Served as a reviewer for the following conferences
    - IEEE International Conference on Communications (ICC)
    - IEEE Wireless Communications and Networking Conference (WCNC)
    - IEEE Symposium on Reliable Distributed Systems (SRDS)

- **Membership in Professional Organizations**
  - Student member, IEEE (Institute of Electrical and Electronics Engineers)
  - Student member, IEEE Computer Society
  - Student member, IEEE Communications Society

- **Awards**
  - Dissertation Year Fellowship, University of Kentucky, 2005–2006
  - Research assistantship, Department of Computer Science, University of Kentucky, 2002–2005
  - Student Travel Support from Graduate School Fellowship Office of the University of Kentucky, 2004
  - Teaching Assistantship, Department of Computer Science, University of Kentucky, 2001–2002

- **References**
  - Dr. D. Manivannan, Associate Professor
    Department of Computer Science, University of Kentucky
    231 Hardymon Building, 301 Rose Street
    Lexington, KY 40506-0195
    Phone: (859) - 257 - 9234, Fax : (859) - 323 - 3740
    Email: manivann@cs.uky.edu
  - Dr. Mukesh Singhal, Chancellor's Professor
    Department of Electrical Engineering and Computer Science
    University of California - Merced
    Lexington, KY 40506-0495
    Phone: (209) 228-4344
    Email: msinghal@ucmerced.edu

– Dr. Zongming Fei, Associate Professor
Department of Computer Science, University of Kentucky
227 Hardymon Building, 301 Rose Street
Lexington, KY 40506-0195
Phone: (859) - 257 - 3202, Fax : (859) - 323 - 3740
Email: fei@netlab.uky.edu

Qiangfeng Jiang