1-1-2006

Self-stabilizing cache placements in Manets

Narenda Ganguru
University of Nevada, Las Vegas

SELF-STABILIZING CACHE PLACEMENTS IN MANETS

by

Narendra Ganguru

Bachelor of Science
Andhra University, India
2003

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2006

UNLV

Thesis Approval

The Graduate College
University of Nevada, Las Vegas

Summer Term 2006

This Thesis prepared by **Narendra Ganguru**

Entitled

SELF-STABLIZING CACHE PLACEMENTS IN MANETS

was approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

By the undersigned:

Ajoy K. Datta, Examination Committee Chair

Maria Gradinariu, Examination Committee Member

John T. Minor, Examination Committee Member

Yoohwan Kim, Examination Committee Member

Stephen Carpenter, Graduate Faculty Representative

Dean of the Graduate College

ABSTRACT

Self-Stabilizing Cache Placements in MANETs

by

Narendra Ganguru

Dr. Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

Dr. Maria Gradinariu, Examination Committee Co-Chair
IRISA, Campus de Beaulieu, France

In ad-hoc networks mobile nodes communicate with each other using other nodes in the network as routers. Each node acts as a router, forwarding data packets for other nodes. There are many dynamic routing protocols to find routes between the communicating nodes. The bandwidth and power are limited in MANETs. Although routing is important in MANETs, the final task of MANETs is Data accessing. So, there is need to implement new techniques apart from routing for data access to save bandwidth and power. If some of the nodes in MANET is provided some of the services from internet Service Provider, then the other nodes also want to access these services. Then, there is a need for caching these services to reduce bandwidth and power.

Caching the internet based services in MANETs is an important technique to reduce bandwidth, energy consumption and latency. If some of the nodes store the object data and code and acts as a cache proxies, then nodes near the cache proxies can get the requested data from the cache proxy rather than from a far away server node saving bandwidth and

access latency.

In this thesis research, we design a distributed self- stabilizing algorithm to place the caches in MANETs. If a node requests the service, it will search for the service and if that service is located in a node that is at a distance greater than D, then the requested node caches the data. In our algorithm, nodes that cache the same data will be at a distance greater than D. We also describe an algorithm to have the shortest path from the source of the data object to all the nodes that cache the same data in the network. This path is used to update the DATA that is cached in the nodes. We propose the algorithm for a single service or DATA. We can implement this algorithm in parallel for all the services available in the MANET.

A self-stabilizing system has the ability to automatically recover to normal behavior in case of transient faults without a centralized control. The proposed algorithm does not require any initialization, that is, starting from an arbitrary state, it is guaranteed to satisfy its specification in finite steps. The protocol can handle various types of faults.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS vii

CHAPTER 1 INTRODUCTION 1
1.1 Contributions 2
1.2 Outline of the Thesis 2

CHAPTER 2 SEARCH TECHNIQUES 5
2.1 Routing 5
2.2 Search Techniques 6
2.2.1 Lookup Services 6
2.2.2 Overlay Maintainence 9

CHAPTER 3 PRELIMINARIES 12
3.1 Distributed system 12
3.1.1 Communication 12
3.1.2 Timing Model 13
3.1.3 Scheduler 13
3.1.4 Our Model 14
3.2 Self Stabilization 15

CHAPTER 4 CACHE PLACEMENT ALGORITHM 17
4.1 Data Structures 17
4.2 Outline of the Algorithm 18
4.3 System Model 18
4.4 Cache Placement Module 20
4.5 Checking Module 21
4.6 Error Corrections 24
4.7 proof of correctness 25
4.7.1 Proof of closure 25
4.7.2 Convergence 27
4.7.3 Self-Stabilizing 29

CHAPTER 5 FURTHER EXTENSIONS 30
5.1 Spanning tree 30
5.1.1 Constructing Spanning tree 32
5.1.2 Spanning forest 33
5.2 Steiner tree 34
5.3 self-stabilization of the Update Algorithm 35

CHAPTER 6 CONCLUSION AND FUTURE RESEARCH 36

BIBLIOGRAPHY 38

VITA 40

ACKNOWLEDGMENTS

I would like to offer a special thanks to my thesis advisor, Dr. Ajoy K. Datta, for chairing my committee and advising me throughout my thesis work. His patience, support, enthusiasm, and most importantly, his confidence in my abilities, have helped me greatly throughout my graduate study. I would also like to offer a special thanks to the co-chair of my examination committee, Dr. Maria Gradinariu at IRISA/Universite Rennes 1, France, for her guidance and numerous contributions to this research. The support and advise offered by her for this thesis research was very valuable and important. I am also very grateful to Dr. John Minor, Dr. Yoohwan Kim, and Dr. Venkatesan Muthukumar for their participation in my committee.

This thesis is dedicated to my parents. Their support, love, and faith in my abilities were very important for completing this thesis research. I also want to thank my friends for their support.

CHAPTER 1

INTRODUCTION

In this thesis we present a self stabilizing solution of caching the services in the MANETs.

MANETs are dynamically created and maintained by the individual nodes comprising the network (for example see Figure 1.1). They do not require a pre-existing architecture for communication purposes and do not rely on any type of wired infra structure[12]. They rely on the wireless transmitters of the participating devices. That is, each devices's transmitter has a limited range. However, if some devices are willing to serve as ad-hoc routers, and forward other devices's messages, it is possible to obtain transitive connectivity or in other words a network.

There is an increasing range of applications and distribution of wireless communication technologies, cellular phones and personal Digital Assistants(PDA) are becoming more common among users. Ad hoc networks emerge when these mobile systems are connected in infrastructure less environment. This means that these networks do not utilize any stationary infrastructure, instead they are restricted to use services provided by the participating units. Service providers would be benefited from integrating web services with these kind of networks.

Web services are a set of standards and a programming method for sharing data between software applications. In other words, web services is a standardized way to distribute

services on the Internet. Web services simplifies for service providers due to their interoperability and extensibility. Programs providing simple services can interact with each other in order to deliver sophisticated added value services.

When considering network based services offered to mobile clients, it is likely that multiple clients in same MANET or even the same region of a MANET, will try to access the same service concurrently. This suggests that caching such services within the MANET would be beneficial. Caching Internet based services is a potentially Internet based application for MANETs, as it can improve mobile users perceived quality of service, reduce their energy consumption and lower their air time costs.

1.1 Contributions

We use the concept of self stabilization [1, 4, 5, 8, 6] to design a self stabilizing algorithm to place the cache of a service (data object) in an ad hoc network. A self-stabilizing system has the ability to automatically recover to normal behavior in case of transient faults. Regardless of the starting system state, a set of nodes cache the data object and these nodes are apart by at most D. Being self stabilizing our algorithm can deal with the topology changes. We also describe a self stabilizing algorithm to construct a shortest path from source of DATA to all the nodes that cache the DATA in the network, which will serve to update the caches in the network when the DATA in the source is changed.

1.2 Outline of the Thesis

In chapter 2, we decribe different search techniques for context based routing in the network.These services will be used to locate the DATA in the network. In chapter3, we decribe the distributed model of our system and concept of self stabilization. Chapter4 includes

the Cache Placement Algorithm. In chapter 4, we also describe the Data structures,system model, Cache placement and checking Modules. Error corrections are described in section 4.6 and proof of correctness is described in section 4.7. In the chapter5, we describe an algorithm to update the caches in the network, by constructing a spanning tree and on that spanning tree, a steiner tree is constructed. Finally, the thesis ends with conclusion and some future work in chapter 6.

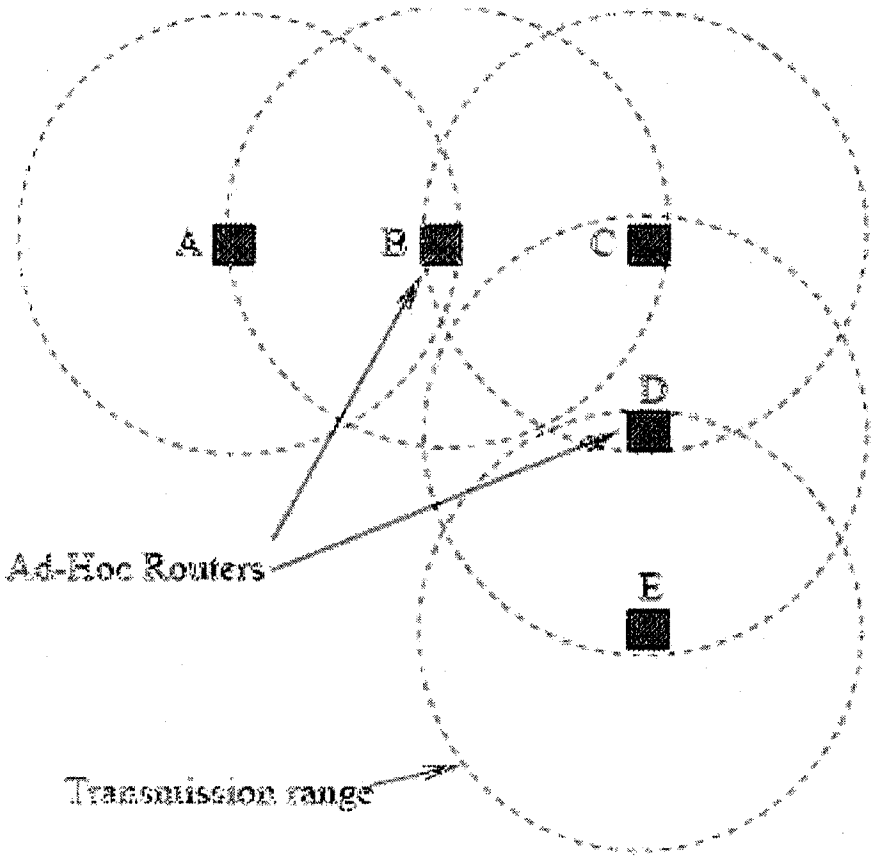

Figure 1.1: **An Ad-Hoc Network: In this example, the transmission ranges are such that A can only communicate directly with B,B with A and C,C with B and D, D with C and E. Here B and D act as ad-hoc routers , allowing for full connectivity between all nodes.**

CHAPTER 2

SEARCH TECHNIQUES

In this chapter we introduce Routing and describe different search techniques called lookup services[13] which will be used in our algorithm.

2.1 Routing

It is necessary to hop several hops to reach the packet to the destination.so, a Routing Protocol is needed.The routing protocol has two main functions,selection of routes for various source-destination pairs and the delivery of messages to their correct destination.The main object of the routing protocol is to find routes.

Largely speaking, there are three types of routing strategies for MANETs, namely, proactive protocols, reactive protocols, and hybrid protocols. Proactive protocols, e.g., OLSR , periodically update their routing tables, and thus always maintain (possibly implicit) routes from any node to any node. Reactive protocols, e.g., AODV and DSR , discover a route only when it is needed. This way, they do not waste resources on routes that are not needed and would never be used, but their response time is much slower and their route discovery process is typically communication inefficient. Hybrid protocols, e.g., ZRP , have a proactive behavior in the neighborhood of a node up to a distance k. After that distance, the protocol acts as a classical reactive protocols. Thus, they attempt to enjoy both worlds.

The proactive lookup assumes that the neighborhood of a node is known in advance, e.g., due to a proactive heartbeat mechanism, while the reactive class constructs the neighborhood at every reincarnation of a query. Broadcast is one of the most popular services on top of MANETs and is highly related to context based search services. The main difference between these two is their goal: broadcast tries to ensure delivery of messages to all processors in the ad-hoc network, while lookup should provide a path to the service or proxy if one exists in the network.

2.2 Search Techniques

This section first examines several existing and a few novel search techniques including flooding, constrained flooding, a novel dynamic variation of probabilistic flooding, and BFS. These are superimposed on a Maximal Independent Set (MIS), a Connected Dominating Set (DS), and a novel adaptation of BFS-tree based overlays.

2.2.1 Lookup Services

2.2.1.1 (Constrained)Flooding

The flooding scheme is the simplest way to disseminate a request in a network . A requester sends a search message to all its neighbors in the network. When a node receives a request for the first time, if it does not have the data, it forwards the request to all its neighbors while recording the path the request has traveled so far on the forwarded message. If a node that receives a request has a copy of the data, it immediately sends a response along the reverse path and does not forward the request any further. Nodes also discard redundant copies of the same request. Alternatively, when an overlay exist, then only nodes that are part of the overlay backbone forward the message to their overlay neighbors. Other nodes only reply if they have a copy of the data and drop the request otherwise. We refer

to running flooding on top of an overlay as constrained flooding. Among the advantages of flooding, we note that the time latency is minimal. Also, this is the most robust scheme and has the highest probability of reaching all network nodes, and therefore find an alternative if one exist. However, the price for this is that unconstrained flooding generates a large number of messages. This means that the energy consumption of flooding is very high, and so is the bandwidth utilization. Since the network is shared between the lookup service and the application, this comes at the expense of the bandwidth left for application messages. Also, this increases the chances of collisions, which could degrade the overall performance of the system. There are two common ways to limit this bad behavior of flooding. One option is to limit flooding with a Time-To-Leave (TTL) parameter, which specifies the maximum number of hops a search request can be forwarded on. This technique solves some scalability problems, but is still not very efficient. Alternatively, constrained flooding greatly reduces the number of messages sent on each search request, since only the overlay backbone nodes participate in the forwarding. However, this also imposes several challenges, since computing the overlay backbone also requires some resources, and one needs to be careful to balance the cost of the overlay maintenance with the benefits it brings. This is particularly true in MANETs due to their dynamic nature, which means that the overlay is continuously evolving.

2.2.1.2 Probabilistic Flooding

Probabilistic flooding is similar to constrained flooding, except that nodes only forward incoming messages with some probability. In dense networks, when a host decides to forward a message, all of its neighbors might already receive the message. Deciding randomly that some nodes would not forward a message can save unnecessary messages without harming

effectiveness. But, in sparse networks, some nodes will not receive all the messages unless the probability of rebroadcasting a message is high.

One solution is to adapt the probability parameter to the number of neighbors. Another idea is to decide whether to forward a message based on the number of redundant messages a host receives . That is, each node counts the number of times it received each message during a given time interval starting from the arrival of the first copy of the message. At the end of this time interval, if the number is less than a threshold value, the message is forwarded, and it is dropped otherwise. This way, in dense networks, some nodes will not rebroadcast messages, while all nodes rebroadcast in sparse network. However, in this solution messages are delayed at each hop, which greatly increases the delivery latency.

2.2.1.3 BFS

Another scheme is based on the Breadth First Search algorithm . This algorithm can be viewed as successive instantiations of flooding with increasing TTL values ranging from 1 to the expected diameter of the network. More specifically, a requester p $\in$ V first initiates the lookup by sending a request to all nodes in $N(p)$. Upon receiving a request, a node responds with a positive message if the requested data is in its cache and by a negative message otherwise. If the requester does not receive any positive response, it sends another request to all nodes in $N^2(p)$. Nodes that did not receive the message during the first round send their response along the reverse path. This process continues iteratively until either the requester receives a positive answer, or the requester believes it has reached all nodes. In this protocol, the message propagation is controlled by the requester. Once an alternative is detected, the propagation stops, which saves a substantial number of unnecessary messages when there exists an alternative close to the requester. Note that this idea can be applied

over an unstructured network, as been described above, or using an overlay. In the latter case, only the overlay backbone nodes forward the request, while preserving the TTL of the corresponding iteration. The main drawback of this protocol is its potential long latency, as messages are not propagated to nodes at distance i hops until a timeout after which it is assumed that all replies from nodes at distance i 1 have been received. Moreover, it may generate many messages when the proxy is far, since search messages travel repeatedly through nearby nodes in successive invocations of the search.

2.2.2 Overlay Maintainence

2.2.2.1 A Maximal Independent Set (MIS) Based Overlay

We start this section by giving a formal definition of a Maximal Independent Set (MIS) and then discuss how to obtain an MIS based overlay.2 Let G = (V,E) be a communication graph. Two nodes i and j in G are said to be independent if (i, j)$\notin$ E. A subset S $\subseteq$ V of nodes is independent if every pair of nodes in S are independent. A set S is a maximal independent set (MIS) if S is independent, yet for any node k $\in$ V / S, S $\cup$ k is not independent. The MIS based overlay is constructed in two phases that are executed in parallel. In the first phase, the MIS is computed. Since by definition, the set of nodes in an MIS cannot directly communicate with each other, the second phase identifies bridge nodes that connect the MIS nodes. Of course, the goal is to find as few bridge nodes as possible, yet to do this in a completely decentralized manner. So, we are interested in a distributed algorithm for computing an MIS in such a way that every node makes local calculations based only on the knowledge of its neighbors. Recall that the neighbors of p are the nodes that appear in the transmission disk of p, and thus p can communicate directly with them, and every message p sends is received by all of them. Additionally, we would like to influence the

overlay construction process such that the overlay nodes will be the best nodes under a given metric. For example, since in mobile systems nodes are often battery operated, we may wish to use the energy level as the metric, in order to have the nodes with highest energy levels members of the overlay. Alternatively, we might use the number of objects for which a node is proxy as the metric, in order to reduce the average number of hops a search message has to travel. Similarly, we might use bandwidth, transmission range, or local storage capacity, or some combination of several such metrics. This is achieved by having a generic function which associates with each node some value from an ordered domain, which represents the nodes appropriateness to serve in the overlay. We call this value the goodness number. This way, it is possible to compare any two nodes using their goodness number and to prefer to elect the one whose value is higher to the overlay. For example, it is easy to evaluate and compare the battery level of nodes, or to obtain and compare the number of objects for which a node is proxy. The MIS algorithm consists of computation steps that are taken periodically and repeatedly by each node. In each computation step, each node makes a local computation about whether it thinks it should be in the MIS or not, and then exchanges its local information with its neighbors.

2.2.2.2 A Connected Dominating Set (DS) Based Overlay

We start this section by giving a formal definition of a Connected Dominating Set (DS), and then discuss how to obtain a DS based overlay. Let G = (V,E) be a communication graph. A set $S \subset V$ is a dominating set if any node in V is a member of S or has a neighbor in S. S is connected if for any node x in S there is another node y in S such that $(x, y) \in E$.

2.2.2.3 BFS tree protocol

The idea in this protocol is to exploit the synchronous behavior of BFS and the broadcast nature of wireless networks to produce an efficient reactive dissemination tree for each search request. Interestingly, this is obtained without any additional delay or control messages. Specifically, recall that with BFS dissemination, the search propagates in synchronous rounds such that in round i the search messages disseminate until distance i hops from the requester. The requester waits for receiving the reply messages of round i before continuing to round i+1. Thus, when the requester receives all replies for round i, it knows the fastest paths between all nodes at distance i and itself. Based on this, it can calculate the minimal set of nodes that need to forward the search during the next round and ensure that the message will still reach all nodes at distance i + 1.

CHAPTER 3

PRELIMINARIES

3.1 Distributed system

The term distributed system is used to describe a communication network, a multiprocessor computer or a multitasking single computer. A distributed system contains of two types of components: processors and communication channels between the processors. Distributed computing studies the computational activities performed on these systems.In the theory of distributed computing, one usually uses the term model to denote an abstract representation of a distributed system. An algorithm is the program given to the processors to solve a certain problem on a certain model setting. Complexity analysis provides some measurement of the performance of algorithm

Various different models arise depending on assumptions about how both processors and communication behave. The following subsections describe some common alternative choices for several different components of a model of a distributed system

3.1.1 Communication

The communication model describes the mechanism that supports information exchange between processors. Two common interprocessor communication models are the message passing model and the shared memory model.

In the message passing model, processors communicate by exchanging messages. A

processor sends a message by adding it to its outgoing message queue, and receives a message by removing it from its incoming queues.A communication link is either unidirectional or bidirectional

In the shared memory model, processors communicate through globally shared objects. Typically these objects are atomic registers. An atomic register is a shared variable that can be either read or written in one indivisible (atomic) step. An atomic register can be multi-reader/multi-writer, or multi-reader/singer-writer, or single-reader/single-writer.

3.1.2 Timing Model

The two basic models of timing in distributed systems are called the Synchronous model and the Asynchronous model .

The synchronous model, where each processor simultaneously executes one step of its program in each time step, is the simplest model to describe, to program and to analyze.

In the asynchronous model, processors execute their programs at different speeds. Both the absolute speed of each processor and the relative speed between processors may vary arbitrarily during the computation. The asynchronous model is harder to program than the synchronous model. Without timing restrictions, problems are more general and interesting and more realistic.

3.1.3 Scheduler

In an asynchronous system, these differences in the speeds of the processors are simulated with the use of a scheduler, alternatively called a daemon. It is assumed that at each time step a scheduler determines which processors execute the next step of their program subject only to synchronization enforced by explicit program constraints.

The distributed daemon and the central daemon are two types of scheduler. In each step,

the central daemon activates only one processor at a time. In each step, the distributed daemon selects a nonempty set of processors and activates all the processors in the set simultaneously.

The scheduler is typically constrained by a fairness assumption that provides some minimum guarantee on the interval between successive steps of a processor. There are many different strengths of fairness. Weak fairness only ensures that in an infinite execution, each processors takes an infinite number of steps. k-fairness ensures that no processor executes more than k steps between any two successive steps of any other processor. A round robin scheduler constrains processors to take a fixed order under a 1-fairness assumption.

3.1.4 Our Model

We use the asynchronous message-passing system model. The asynchronous systems are the most common systems, and the hardest to design algorithms for. Every node can execute its code at its own pace, and the message delivery can take an arbitrary time.

All nodes execute the same distributed program (*uniform*). The program is a finite set of *guarded actions* of the form:

$< label >::< guard > \rightarrow < statement >$

A statement can be executed if and only if its guard, a boolean expression, evaluates to *true*. The selected statement is executed in one atomic step. If a process has at least one true guarded command, then it is called *enabled*.

We consider a *distributed daemon*: In every execution step, if one or more processes are enabled, then the daemon chooses at least one (possibly more) of these enabled processes to execute. Once the process is selected, then non-deterministically one of its enabled actions is selected and its statement is executed. We assume a *weakly fair* daemon: A continuously

enabled process will be eventually chosen by the daemon.

Each node has a local state defined by its current variables. The global state of the system (configuration) is the union of the local state of its nodes and the messages on the links.

3.2 Self Stabilization

The idea of self-stabilization in distributed computing first appeared in a classic paper by E.W.Dijkstra in 1974 [4]. In this short paper published in the Communications of ACM, he proposed the idea of stabilization of a distributed system: the system should be able to converge to a legitimate state with in a bounded amount of time by itself without any outside intervention. In this paper he showed three examples of a self-stabilizing token ring systems: one with K states where K is greater or equal to the number of processors in the ring, the other two with *three* and $four$ states respectively. The global states of the token ring in which there are multiple tokens or there is no token are defined to be illegitimate states. There has been considerable amount of interest on analyzing these protocols and proving the correctness of these protocols.

A self-stabilizing system S guarantees that, starting from an arbitrary global state, it reaches a legal global state within a finite number of state transitions, and remains in a legal state unless a change occurs. In a non-self-stabilizing system, system designers need to enumerate the accepted kinds of faults, such as node/link failures, and they must add special mechanisms for recovery. Ideally, a system should continue its work by correctly restoring the system state whenever a fault occurs. In a non-self-stabilizing system, system designers need to enumerate the accepted kinds of faults, such as node/link failures, and they must add special mechanisms for recovery. Ideally, a system should continue its work

by correctly restoring the system configuration whenever a fault occurs.

Each node has a local state defined by its current variables. The global state of the system (configuration) is the union of the local state of its nodes and the messages on the links. Given $\mathcal{C}$, the set of all possible states, and a predicate $\mathcal{P}$ over $\mathcal{C}$, we denote by $\mathcal{L}_{\mathcal{P}} \subseteq C$ the set of all *legitimate states with respect to* $\mathcal{P}$, or simply, the set of all *legitimate states.*

An execution e is a maximal sequence of configurations, $e = c_1, c_2, ...$ such that $\forall i \geq 1, c_i \in \mathcal{C}$, and c_i is reached from c_{i-1} by executing some guarded action, or c_i is a terminal configuration (no nodes are enabled).

We now define self-stabilization. Let X be a set. The notation $x \vdash Q$ means that an element $x \in X$ satisfies the predicate Q defined on the set X. We define a special predicate true as follows: *for any* $x \in \mathcal{X}$, $x \vdash$ true.

Let P be a distributed system and R, S predicates on the configurations of P. R is closed if every configuration of the computation of P that starts in a configuration satisfying P also satisfies R. R converges to S if both R and S are closed in P, and any computation starting from a configuration satisfying R contains a configuration satisfying S.

Definition 3.1 (Self-Stabilization) P **stabilizes** *to a configuration satisfying predicate* R *iff* true *converges to* R *in* P.

CHAPTER 4

CACHE PLACEMENT ALGORITHM

In this chapter we include our *algorithm* and describe the data structures that are used in our algorithm. We provide a brief outline of the algorithm and system model, describe the cache placement and checking modules, explain error corrections, followed by the proof of correctness. In our algorithm we implement that a node requesting a DATA can cache the DATA if there is no cache within or at distance D. In this way we place the caches in the network. This caching algorithm will implement to cache a single data object or service in MANETS.

4.1 Data Structures

The Variables used in our algorithm are:

- UID: unique identifier .every node in the network has unique ID, which is a positive integer
- N_i: set containing all the neighbors of node i.
- P_i: its a pointer points to parent of i
- RS: responsibility set construct tree function return a RS in a requested node. RS represents a BFS tree. For example RS=$< 2, < 1, < 15, 5 >>, < 3, < 12, 9 >>>$

represents a tree with root 2 and 1, 3 are his children.15, 5 are children of 1 and 12, 9 are the children of 3.

- N : N is a set of tuples $< N_i, i >$.It is send in response message.For the above tree, the field N of the response message send by node 1 to its parent has the $<< N_1, 1 >, < N_1 5, 15 >, < N_5, 5 >>$

4.2 Outline of the Algorithm

In this chapter we present an algorithm to select the nodes to place the cache of DATA in MANETs. The nodes having a cache of DATA should be at most distance D with each other, so that, it would be memory efficient. It would be unnecessary to cache the same DATA in different nodes which are not far from each other.

This algorithm consists of two modules- cache placement module and checking module. Cache placement module contains actions related to selection of nodes to place the caches in the network. The idea for selection is, if a node requests a DATA, the search techniques(lookup Services) described in the previous chapter will find a node having a DATA. If the distance between the requester and node having the DATA is greater than D, then the requester caches the DATA in its memory. Checking module will be activated only in cached nodes. This module when activated checks whether there is any two nodes cache the same DATA within or at distance D. If there is, it will make sure that one of the two nodes drop the cache.

4.3 System Model

In this work we focus on wireless mobile systems. A *node* in the system is a device owning an omni directional antenna that enables wireless communication. A transmission of a node

i is received by all nodes within a disk centered on p whose radius depends on transmission power, referred to in the following as the *transmissiondisk*; the radius of the transmission disk is called the *transmissionrange*. Thus, there is a single communication primitive *broadcast*(m), allows a node to transmit a message m to all nodes inside its transmission disk. The combination of the nodes and the transitive closure of their transmission disks forms a wireless ad-hoc network.

Practically, in order to obtain external services, we assume that all nodes have another mean for accessing the Internet directly e.g., using a cellular connection or a Wireless Access Point(WAP); such an access to the Internet passes through an Internet Service Provider(ISP). We say that a node i is a *localproxy* for a data object d when the buffer cache of i contains d.

The network described above can also be modeled as a graph G=(V,E) where V is the set of network nodes and E models the one-to-one neighboring relations. A node j is a *neighbor* of another node i if j is located within the transmission disk of i. In the following, $N(i)$ refers to the set of neighbors of a node i. By considering $N(i)$ as a relation, we say that a node i has a *path* to a node j if j appears in the transitive closure of $N(i)$ relation.

As nodes can physically move, there is no guarantee that a neighbor at time t will remain in the transmission disk at a later time t+Δ. Messages are not guaranteed to be delivered ,but we assume the most of them are delivered with high probability. Additionally, devices might be turned off or on at any time, so the set V of alive nodes varies with time and has no fixed size.

4.4 Cache Placement Module

In this section we present a normal execution of the cache placement module and we describe how the BFS tree constructed in this module is maintained. We use the BFS lookup service over an BFS overlay which is described in chapter 2 in this module.

The messages used in this module are request and response. The request message is defined by request(requester, DATA, round, dist, RS, k) where requester is the requester, DATA is the requested data, round is the distance of nodes the requester want to reach, dist is the number of hops the search has traveled so far, RS is the responsibility set, k is the node from which this message was received from, in that hop. The response message is defined by response(j,presence(j),N), where j is the node from which this message is initiated, presence(j) represents true or false depending on whether DATA is cached in j or not, N is a set of tuples of form $< N_p, p >$ where node p belongs to the subtree of the node that sending this message.

When node i requests a DATA, it sends request message to all its neighbors. It will get a response message from all its neighbors. These nodes also send their neighbors in the N field in the response message. At the end of first round, the requester has N1- the nodes at distance 1 hop from the requester, N2- the nodes at distance 2 from the requester. So, the requester constructs a BFS tree of depth 2. In the second round, the requester send the request message to all the nodes at distance 2 along the BFS tree constructed. The nodes at distance 2 send a response message. In the response message N field carries N3 and N2. At the end of second round, the requester has N1,N2 and N3. It constructs a BFS tree of depth3. Similarly, at the round k, the requester send the request message to all the nodes at distance k, along the BFS tree of depth k constructed before this round. The response

message along the path to the requester attach all the sets of neighbors of all the nodes along the path. At the end of this round, the requester constructs a BFS tree of depth k+1.

This requester stops sending request messages, when it receives response(j,true,...) or the authorized levels are reached. If it receives response(j,true,...) message, then it caches the DATA if the distance between requester and j is greater than D, otherwise it does not cache the DATA because there is already a node that caches the same DATA within or at distance D.

If a new node that entered in the region may be connected in the middle of the tree at the end of a round. If that new node has the DATA , then the search has to stop. To include this case, in the algorithm, response(j,true,....) can be send by any node in the BFS tree if it has the DATA and response(j,false,...) is send by only the nodes, that are leaves of the BFS tree constructed by the requester at that round.

4.5 Checking Module

Module checking is activated in each node when that node caches a DATA object. This checker checks whether there is any node that caches the same data within or at distance D from this node. There are different coloring algorithms which implements that no two nodes have the same color within distance d[9, 16, 10]. When a node caches a DATA after executing the cache placement module, then the BFS tree constructed is of depth greater than D. so, the detect messages are send to all the nodes within or at distance D along the BFS tree. After this one time check, this checker checks at regular intervals of time (more likely when it moves to a different neighborhood) using flooding technique as described in the chapter 2.

The messages used in this module are detect and dropcache. The detect message is

Algorithm 4.4.1 Cache Placement Module

Parameters:

UID:: i
N_i :: Set of i's neighbors;
P_i :: Parent of i;
$constructtree(N)$ :: returns of a BFS tree with the input N

Actions:

$\mathcal{R}_1$ :: node i requests a DATA $\longrightarrow$
round:=1;RS=Φ;
repeat until authorized levels are reached:
send $request(i, DATA, round, 1, RS, i)$
wait $response(j, prescence_j, N)$ from all
$j \in N_i$or timeout
if received at least one $response(j, true, N)$ then
cache the DATA in its memory,
if distance between i and j is greater than D
endif
return 1;
endif
RS= $constructtree(N)$;
$round = round + 1$
end repeat
return 0;

$\mathcal{R}_2$ :: Upon first rec. of $request(j, DATA, round, dist, RS, k)$ $\longrightarrow$
P_i=k;
if DATA is present at i,then send $response(i, true, ...)$ to j
else
if dist= round, then send $response(i, false, << N_i, i >>)$ to root j
else,
send $request(j, DATA, round, dist + 1, RS, i)$ to their children if there is any,
otherwise send $response(i, false, << N_i, i >>)$ to root j
endif
endif

$\mathcal{R}_3$ :: upon rec. $response(j, false, << N_j, j >, ... >)$ from all its children $\longrightarrow$
send $response(j, false, N)$ to its parent, where N contains the tuples of
the neighbors of all the nodes in that subtree including the tuple $< N_i, i >$

defined by detect(DATA, checker, dist), where DATA is the DATA item they are detecting, checker is the process having a DATA checks if there any other node within or at distance D that caches the DATA,dist is the distance traveled so far. The dropcache message is defined dropcache(checker), where checker is the process from which the detect message is received.

After regular intervals of time, a node i caching a DATA sends detect(DATA, i, dist=1) to all the nodes within or at distance D. If a node p receives a first detect message and if it caches the DATA, then it sends dropcache(checker) to the checker process, a process from which this detect message is received if UID(p)>UID(checker), otherwise it will drop the cache in its memory. If a node p receives a first detect message, and it did not cache the DATA, then it will send the detect message to all the other neighbors(at the first time check it will send to its children of the BFS tree) by increasing the dist field in detect message by 1, if dist in the detect message is less than D.

Algorithm 4.5.1 Checking Module

Parameters:
UID:: i

Actions:

$\mathcal{C}_1$ :: At regular intervals of time $\wedge$ node i cached the DATA $\longrightarrow$
send $detect(DATA, i, 1)$

$\mathcal{C}_2$ :: Upon first rec. of $detect(DATA, j, dist)$ $\longrightarrow$
if DATA is present at i,then drop the cache in its memory,
if $UID_i < UID_j$
else send $dropcache(j)$ to node j
endif
else,
if (dist < D) then send $detect(DATA, j, dist + 1)$
endif
endif

$\mathcal{C}_3$ :: upon rec. $dropcache(i)$ $\longrightarrow$
drop the cache in its memory

4.6 Error Corrections

There are three illegitimate configurations in the algorithm.We show how this Algorithm corrects each of these situations.

1. A node caches the data when there is any other node already cached the same object within distance d

   A node caches the object only when it receives the response from a node greater than distance d. It will gets the response from that node only when there is no other node within less distance than that node. So, it is not possible to get this configuration

2. Two nodes within distance d caches a same object at the same time.

   These two nodes activates the checking module and checks whether there is any node caches the same object. Then they find out and drop the cache in the node which has lower UID. So, there will be no other node that caches the data within d hops of the node that still caches the data.

3. If a node, cache the data, moves and there is other node cache the data within distance d of new location.

   Checker Module, in each cached node, checks whether there is any node cached the same object within distance d. If there is,the node with a lower UID will drop the cache.

4.7 proof of correctness

Definition 4.1 *:The system is considered to be in a legitimate state(i.e.satisfies the legitimate predicate L)if the following conditions are true with respect to a query region*

(i) node which request the data, caches the data if there is no cached node in d hops from the node

(ii) No two nodes with in or at distance d should not cache the same data

4.7.1 Proof of closure

Lemma 4.1 *:The actions in Module Caching are enabled only when any node requests for a data*

Proof. The responsibility of actions in Module Caching in each node are - when a node requests a data item, it sends request message to the nodes. If a node has a data, it responds to the requester. If requester node is at a distance greater than d from the responder, then the request node caches the data item.

□

Lemma 4.2 *: In the legitimate configuration, the actions in Module checking are not enabled as long as no topology changes occur or there is no request for a data*

Proof. The illegitimate configuration is there is atleast two nodes that cached the data within or at distance d. If it finds anything it will trigger the actions of Module Checking to drop the cache. If these actions are not triggered then there is no two nodes that cached the object within or at distance d. The illegitimate configuration can happen only in two ways.

(i) If two nodes within distance d requests the same data object and they get the response from nodes at a distance greater than d from them.then these two nodes cache cache the data. So, there will be two nodes within or at distance d that cached the same data.

(ii) If the topology changes, there is a possibility that a cached node move to a new neighbourhood so that there is another cached node within or at distance d.

So, if there is no topology changes occur or no node requests for a data, then Module checking actions are not enabled.

□

Theorem 4.1 *:(***Closure***)* *The system defined by a legitimate predicate* **L** *is closure as long as no topology changes occur or no node requests for a data.*

Proof. By Lemma 4.1, the actions of Module caching is not activated if no node requests for a data

By Lemma 4.2, the actions in Module Checking to drop the cache are not enabled if there is no topology changes occur or no node requests for a data.

so, this proves that legitamacy predicate **L** holds. Hence, L is a closed predicate.

□

4.7.2 Convergence

Lemma 4.3 *:The BFS tree constructed in the cache placement module is self stabilizing*

Proof. : The BFS tree we constructed at the end of each round is self stabilizing. To construct a BFS tree of depth round+1 at the end of the *round*, we only need the neighbors of all the nodes at distance round because the BFS tree of depth *round* is already constructed. But it is not self stabilizing because of topology changes.

Due to the mobility of nodes, the BFS tree constructed in the previous round may be destroyed. So, in the response message we carry all the neighbors of the nodes in the BFS tree constructed. Even if the topology change, these nodes will change its neighbors and it will be informed to the requester and a new BFS tree is constructed as of current topology.

Some cases:

i) if a node entered in the region– All the neighbors of this node will change their neighborhood set. This set will be send with the response message and the requester constructs a BFS tree including this node in the tree.

ii) If a node leaves the tree– The neighboring nodes change their neighboring set and this is informed to the requester. The nodes in the subtree of that node will not be in the tree. Now the requester constructs a BFS tree in which that node is not in the tree and all the nodes in the subtree of that node will be connected to the BFS tree constructed at the end of the round.

□

Theorem 4.2 *:(***Convergence***) Starting from any arbitrary configuration, Cache Placement Algorithm reaches a configuration that satisfies the legitimacy predicate* **L**

Proof. The goal is to prove that starting from any arbitrary configuration of the system of nodes,algorithm gurantees that in finite steps, the system will reach a configuration that satisfies legitimacy predicate **L**.

We prove this by contradiction.

=> There exists a configuration in which, after any finite number of steps, the system will never reach a configuration that satisfies the legitimacy predicate **L**.

=> There exists a configuration in which, after any finite steps, the system will never reach a configuration in which any two nodes that caches the same data should be greater than distance d.

There exista a configuration where a requested cannot find a cached node even there is a cached node that is at a distance less than the BFS tree constructed so far in the cache placement module. This can be contradicted by Lemma 4.3

There exists a configuration that two nodes caches the same object within or at distance d. Module Checking in each cached node i checks whether there is any node with in or at distance d. If it finds another node j that caches the data within or at distance d and $UID_i > UID_j$ then the cached node j drop the cache in its memory. So, there will be no other cached node within or at distance d from each cached node.

This is a contradiction.

□

4.7.3 Self-Stabilizing

Theorem 4.3 *Algorithm Cache Placement is self stabilizing.*

Proof. The proof follows from Theorem 4.1 and Theorem 4.2 □

CHAPTER 5

FURTHER EXTENSIONS

In this chapter we decribe an algorithm to have the shortest path from all the nodes that cached the DATA to the source of the DATA for updating the DATA in the cached nodes. The better idea is to construct a spanning tree with root as source of the DATA. On this spanning tree, we construct a steiner tree where steiner nodes are the nodes that cached the DATA or it is the source of DATA. If there is more than one source node in the network, we construct a spanning forest in such a way the root of each tree in the forest is the source node and construct a steiner tree for each tree separately. The steiner tree constructed between cached nodes and source of DATA should be self stabilizing due to the mobility nature of nodes in MANETS.

In the first section we describe a self stabilizing spanning tree construction algorithm where the source node is a root of the tree. In the next section we describe an self stabilizing algorithm to construct a steiner tree on the constructed spanning tree.

5.1 Spanning tree

There are different self stabilizing spanning tree algorithms as mentioned in [7]. One of the first papers appered in self stabilizing spanning tree construction in 1990 is by Dolev, Israeli and Moran [14, 15]. In the algorithm, every node maintains two variables: (1) a

pointer to one if its incoming edges (this information is kept in a bit associated with each communication register), and (2) an integer measuring the distance in hops to the root of the tree. The distinguished node in the network acts as the root. The algorithm works as follows: The network nodes periodically exchange their distance value with each other. After reading the distance values of all neighbors, a network node chooses the neighbor with minimum distance dist as its new parent. It then writes its own distance into its output registers, which is dist + 1. The distinguished root node does not read the distance values of its neighbors and simply always sends a value of 0. The algorithm stabilizes starting from the root process.

In the same year as Dolev, Israeli and Moran [14] published their algorithm, Afek, Kutten and Yung [17] presented an self-stabilizing algorithm for a slightly different setting. Their algorithm also constructs a BFS spanning-tree in the read/write atomicity model. However, they do not assume a distinguished root process. Instead they assume that all nodes have globally unique identifiers which can be totally ordered. The node with the largest identifier will eventually become the root of the tree. The idea of the algorithm is as follows: Every node maintains a parent pointer and a distance variable like in the algorithm above, but it also stores the identifier of the root of the tree which it is supposed to be in. Periodically, nodes exchange this information. If a node notices that it has the maximum identifier in its neighborhood, it makes itself the root of its own tree. If it learns that there is a tree with a larger root identifier nearby, it joins this tree by sending a join request to the root of that tree and receiving a grant back. The subprotocol together with a combination of local consistency checks ensures that cycles and fake root identifiers are eventually detected and removed.

Also in 1990, Arora and Gouda [2, 3] published a self-stabilizing BFS spanningtree algorithm Similar to Afek, Kutten and Yung, they also assume unique identifiers and the node with maximum identifier eventually acts as the root of the system. In contrast to Afek, Kutten and Yung, the algorithm needs a bound N on the number n of nodes in the network to work correctly. The bound N is necessary because the algorithm uses a different technique to detect and remove cycles. Again, every node maintains variables for distance, parent and root identifier. Periodically, every node compares its own distance and root identifier setting with the values stores in the node pointed to by the parent variable. In the finished spanning tree, the root identifiers should be the same and the distance should be the distance of the parent incremented by 1.

5.1.1 Constructing Spanning tree

To construct a spanning tree, we use the algorithm of Arora and Gouda [2, 3] with some modifications. This algorithm is to maintain a rooted spanning tree. In the solution given below, we accommodate such changes by ensuring that the tree layer performs its task irrespective of which state its starts from.

Each process maintains a $f.i$ variable which represents a parent of i. Since, it can start in any state, the initial graph of the father relation is arbitrary. In particular, the initial graph may be a forest or it may contain cycles.

For the case where initial graph is a forest of rooted trees, all trees are collapsed to one by giving to precedence to the tree which root is source of DATA followed by whose root has maximum ID. This module contains a variable $root.i$ whose value is the current root process of i. If $root.i$ and $root.j$ is not the source of DATA and $root.i$ is less than $root.j$ for some adjacent processes i and j, $root.i$ is set to $root.j$ and $f.i$ is sets to j. If $root.i$ or

root.j is the source of the DATA then the tree whose root is not the source will attach the tree to the tree whose root has the source and changes it *root* variable to the source node.

For the case, where the initial configuration has cycles, each cycle is detected and removed by using a bound on length of the path from each node to its root process in the spanning tree. This length in our module would be the authorized levels allowed to search for a cache by the requester in the *cacheplacement* module described in the last chapter. Each process has the variable $d.i$ whose denote the length of path from i to its current root. If $d.i$ exceeds K-1,where K is the upper bound on length. Since the length bound by k, a cycle is detected. To remove the cycle that it has detected, i make itself as a father.

Because, our assumption that initial configuration is arbitrary, we need to consider all other cases where the initial values are "locally" inconsistent, that is, one of the following hold:$root.i < i$,$f.i=i$ but $root.i \neq i$ or $d.i \neq 0$, or $f.i$ is not i nor in N_i. In these cases, the module makes itself locally consistent by setting $root.i$ to i,$f.i$ to i and $d.i$ to 0. Another possibility is, root of process i is inconsistent to the state of the father, that is $root.i$ is different from $root.(f.i)$. In this case, $root.i$ is corrects its value to $root.(p.i)$.

G is a legitimate predicate, where G is, *root* variable is same in all the nodes and that value is the source node. In the paper [2, 3], it is shown that the legitamte predicate is reached in finite steps. So, a spanning tree rooted to the source node exists.

5.1.2 Spanning forest

If there are more than one source node in a network, then cached node having a path to one of the source nodes is enough. So, we need to construct a spanning forest in which each tree in the forest has the root which is source of the DATA.

This can be acheived with the same module described above. The idea is, if there are

two adjacent nodes with different *root* values then by precedence, one of the tree will attach to another by becoming one, but if both the *root* values are the sources of DATA, then they will not become one. The individual tree that does not join into another tree in the network has to have the root which is the source node. So, a spanning forest is constructed with each tree in the forest has the source as a root.

5.2 Steiner tree

Definiton:: Let G=(V,E) be a connected undirected graph, where V is the set of nodes and E is the set of edges. For any non-empty subset Z$\subseteq$ V, G_z=(V_z,E_z) is a steiner tree for Z and G, if and only if G_z is connected and Z $\subseteq$ V_z holds. A self stabilizing steiner tree on a given spanning tree with a root as source is given in [11].We have to construct a steiner of Z, where Z is a set of all cached nodes and source nodes of that DATA. The idea is, the nodes in the path of cached nodes to the root of the spanning tree should be included in the steiner tree. So, we can have the path from the root to all cached nodes. To make it self stabilizing, the nodes will check its children status continuously.

The actions of the algorithm are:

1. Z-members join the tree

2. non-Z-member leaf of the spanning tree leave the steiner tree

3. If every children leave the steiner tree,leave the tree

4. If atleast one Steiner tree's member exist in the children, join the tree.

In any illegitimate state, one of these actions is active. From any configuration and any computation starting from C, eventually no process is previleged. This algorithm is self stabilizing from the above two statements. These are proved in [11].

5.3 self-stabilization of the Update Algorithm

The spanning tree constructed is Self-stabilizing and also steiner tree constructed on the spanning tree is also self-stabilizing. Even if the co-ordination between the up processes is lost(due to node mobility)then each component eventually reaches a state where coordination is reached. The self stabilization of the Update algorithm is reached, when the spanning tree is stabilized in a finite steps, say T1, and from that configuration, the steiner tree will be stabilized beacuse it will stabilize from any arbitrary configuration and the steps, say T2, is the steps of the steiner tree algorithm after spanning tree is stabilized. Finally, the Update algorithm is stabilized in steps T1+T2.

CHAPTER 6

CONCLUSION AND FUTURE RESEARCH

We have presented a self-stabilizing Cache Placement Algorithm to place the cache of a service in the MANETs. The algorithm place the cache of the DATA in a node, when the node requests for DATA and it finds the DATA at a node which is at distance greater than D. Once a node caches the DATA, it has to make sure that no other node within or at distance D caches the same DATA.Overall, the algorithm place the caches in such a manner that there is only one node that cache the DATA within distance D of a node that requests a DATA and if there isn't, then the requested node caches the DATA.

We also described an algorithm to have the shortest path from all the nodes that cached the DATA to the source of the DATA for updating the DATA in the cached nodes. We constructed a self stabilizing spanning tree and a self stabilizing algorithm to construct a steiner tree on a spanning tree. We can update the DATA in the cached nodes if the DATA at the source is changed by having the shortest path from source to all cached nodes in the network using this algorithm.

In this research, the algorithm is to place the cache of a single data object. We can extend this algorithm by implementing it parallel for each data object independently.

In future, further work can be done on caching algorithm we presented in this thesis. Work can be done on how to provide *incentives* to the nodes that host the cache of DATA.

There should be some scheme to provide some special previliges or incentives to the cached nodes, as it is using its power to provide the DATA in that region.

We can also extend the algorithm that caches several services available in the MANETs to be *LoadBalancing*. For a node to be a cache proxy for each of these services should be load balancing among all nodes. A device is willing to serve as a proxy if it enjoys the benefits of other services for which the proxy is located in other devices.

We can also write the algorithm using the cluster based approach to solve the same problem. The idea is, cluster head is the node that cached the DATA and the nodes with in distance D of the clusterhead will belong to that cluster.

BIBLIOGRAPHY

[1] A. Arora and M. G. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19:1015–1027, 1993.

[2] Anish Arora and Mohamed Gouda. Distributed reset. *In the Proceedings of 10th Conference on Foundations of Software Technology and Theoretical Computer Sceince*, pages 316–333, 1990.

[3] Anish Arora and Mohamed Gouda. Distributed reset. *IEEE Transactions on Computers*, pages 1026–1038, 1994.

[4] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[5] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Selected Writings of Computing: A Personal Perspective*, pages 41–46, 1982.

[6] Shlomi Dolev. *Self-Stabilization.* The MIT Press, 2000.

[7] Felix C. Gartner. A survey of self- stabilizing spanning tree construction algorithms. Technical report, Swiss Federal University of Technology(EPFL), 2003.

[8] M. G. Gouda. *Elements of network protocol design.* John Wiley & Sons, Inc., 1998.

[9] Maria Gradinariu and Sebastien Tixeuil. Self stabilizing vertex colorating of arbitrary graphs. *In the 4th International Conference on Principles of Distributed Systems ,OPODIS'2000*, pages 55–70, 2000.

[10] Lisa Higham and Lixiao Wang. Self-stabilizing distance -d distinct labels via enriched fair composition. *23 rd Annual Symposium on Priniciples of Distributed Computing, PODC 2004*, 2004.

[11] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing algorithm for the steiner tree problem. *SRDS*, pages 396–401, 2002.

[12] C. E. Perkins. Ad hoc networking. *Addison-Wesley*, pages 1–2, 2004.

[13] Maria Gradinariu Roy Friedman and Gwendal Simon. Locating cache proxies in manets. *MobiHoc '04*, 2004.

[14] Amos Israeli Shlomi Dolev and Sholmo Moran. Self-stabilizing of dynamic systems assuming only read/write atomicity. *Proc. of 9th Annual ACM symposium on Principles of Distributed Computing*, pages 103–118, 1990.

[15] Amos Israeli Shlomi Dolev and Sholmo Moran. Self-stabilizing of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[16] David P. Jacobs Stephen T. Hedetniemi and Pradip K. Srimani. Fault tolerant distributed coloring algorithms that stabilize in linear time. *IPDPS'2002*, 2002.

[17] Shay Kutten Yehuda Afek and Moti Yung. Memory efficient self-stabilizing protocols for general networks. *Distributed Algorithms, 4th International Workshop,volume 486 of Lecture notes in Computer Science*, pages 15–28, 1990.

VITA

Graduate College
University of Nevada, Las Vegas

Narendra Ganguru

Home Address:
969 E. Flamingo RD. Apt. 167
Las Vegas, NV 89119

Degrees:
Bachelor of Science, Electronics and Communication Engineering,2003
Andhra University, India

Thesis Title: Self-stabilizing Cache Placement in MANETs

Thesis Examination Comittee:
Chairperson, Dr. Ajoy K. Datta, Ph.D.
Co-Chairperson, Dr. Maria Gradinariu, Ph.D.
Committee Member, Dr. John T. Minor, Ph.D.
Committee Member, Dr.Yoohwan Kim, Ph.D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.