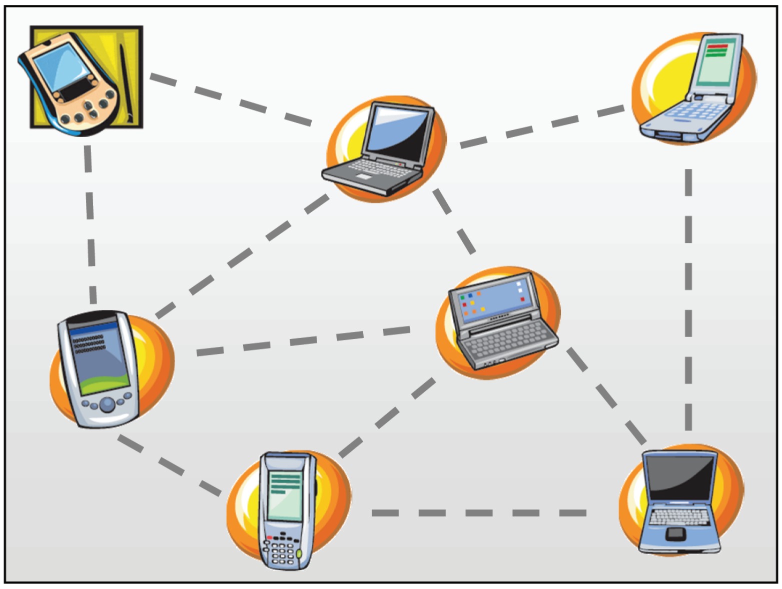
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory
Communication Systems Group

Master's Thesis - Summer Term 2005

Service Management Procedures Supporting Distributed Services in Mobile Ad Hoc Networks



Florian Maurer
maurerfl@ee.ethz.ch
MA-2005-14

August 31, 2005

Tutor:	Károly Farkas	farkas@tik.ee.ethz.ch
Supervisor:	Prof. B. Plattner	plattner@tik.ee.ethz.ch

Abstract

Using real-time applications in mobile environments, e.g., multiplayer games or collaborative working tools, is getting popular as mobile devices and wireless networks are becoming ubiquitous. Especially mobile networked gaming, in regard to the current trends, is considered by game developers, mobile device manufacturers and service providers to be a very attractive source of future revenue. Furthermore, the appearance and evolution of new communication paradigms like mobile ad hoc networking offer new ways and unique features for real-time mobile applications and even for mobile gaming. However, ad hoc networks reserve special challenges mainly due to their self-organized behavior and the resource constraints of the participating mobile devices. One of these challenges is how we can manage applications and support their smooth running in this dynamic and error prone environment.

In this Master's thesis, an algorithm called PBS (Priority Based Selection) will be presented that addresses these challenges. This algorithm is based on graph theory using Dominating Sets to create a distributed service architecture in a self-organized mobile ad hoc, shortly 'self-hoc', network. PBS computes an appropriate Dominating Set of the network graph in a fully distributed manner and it is the first approach in contrast to the existing algorithms that offers continuous maintenance of this set even in dynamically changing network topologies. To get an appreciation about PBS it will be discussed, analyzed and evaluated via simulations and it will be shown how the distributed service architecture created and maintained by applying PBS can be used to manage real-time multiplayer games in self-organized mobile ad hoc networks. Finally, the algorithm has been implemented in a real self-hoc network testbed .

Preface

With this Master's Thesis I will finish my studies at the Department of Information Technology and Electrical Engineering (D-ITET) [1] at the Swiss Federal Institute of Technology (ETH), Zürich [2]. This thesis was performed at the Computer Engineering and Networks Laboratory [3] between March and August 2005.

I would like to express my sincere gratitude to:

- *Károly Farkas* (farkas@tik.ee.ethz.ch), for the opportunity he gave me to conduct this project and his guidance and assistance through the whole project.
- *Dirk Budke* (d.budke@web.de), for the enjoyable teamwork and the permission to use and extend Mobigen, a scene generator for the NS-2 simulator.

Zurich, 31st August

Florian Maurer

Table of Contents

List of Figures

List of Tables

Chapter 1

Task Description

In the first section of this chapter, the topic of this Master's Thesis will be introduced and the structure of this report will be explained. In the second section, a detailed working plan that guided through the different steps of the project will be shown, and the last section contains the general regulations for the project as predetermined by the department for fulfilling the requirements for a report leading to the master's degree.

1.1 Introduction

Real-time applications are attractive candidates to be used in mobile environments as the number of mobile devices and wireless networks are dramatically increasing. Especially mobile networked games can constitute significant source of revenue for the mobile game industry. In 2004, over 170 million people had downloaded games to their mobile phones. This number will triple in 2005. The global market intelligence and advisory firm IDC predicts that wireless gaming will be worth 1.15 billion EUR by 2008 overtaking ring-tones in 2005 [4]. This information reflects the high expectations for the wireless gaming market potential.

New communication paradigms like mobile ad hoc networks (MANETs) can offer new ways and unique features for real-time mobile gaming attracting veteran gamers and new players alike. MANETs are self-organized networks consisting of different mobile devices that are communicating with each other. The network structure of a MANET is altering continuously due to device mobility. Moreover, there exists no central administration in a mobile ad hoc network. Each node participating in the network can act as end host or router or both and must be cooperative in packet forwarding for other nodes. Among the new possible features for MANETS are the abilities to integrate the real-world context of the players into the game, to encourage real-world interaction between players, and to enable ubiquitous and competitive game playing in a continuously changing environment. Existing efforts to enable mobile networked games have either focused on using GPRS, UMTS or WLAN technologies to provide connectivity between players or they were limited to direct short-term communication via Bluetooth or Infrared connections. It is our vision that there will be a strong demand for games running on mobile ad hoc networks where the player neither has to continuously use connectivity through infrastructure-based networks nor is restricted to short-term games with direct communication between players [5]. Market leader companies are also supporting this trend with their latest products like Nokia N-Gage[1] or Sony PSP[2] game consoles.

In order to meet this demand, players' mobile devices need to collaborate as a self-organizing mobile ad hoc system called shortly 'self-hoc' system in our terminology. They will perform tasks such as detecting the real-world context of players, maintaining a persistent game state, and providing connectivity between players that are not in radio range of each other. However, the ad hoc way of communication results in several challenges including the management and support of the applications in this flexible and unreliable environment. Multiplayer games are generally demanding real-time services preferring continuous low latency network connections (in the range of 50-100 ms) with no jitter [6], [7]. In self-hoc networks, network connections are established over feeble wireless links and devices can move and disappear easily which make the management of real-time services especially difficult. Today's online multiplayer game architectures are mainly based on the client-server model [8]. This model is not suitable for a self-hoc network due to the lack of central administration in these networks, point of failure property of the server node, limited scalability and unprepared handling of fluctuating network connections and conditions. In contrast, the zone-based game architecture [9] aids the realization of multiplayer games with a robust, redundant server-client model by increasing fault tolerance and responsiveness. In this model, the player nodes are divided into separate zones. In every zone, a dedicated server node handles the players belonging to the zone and synchronizes with the other zone servers.

However, the zone server nodes should be selected in an efficient and distributed way using the most powerful nodes (concerning available computation and communication resources, position in the network, etc.) as servers. To this end, in this report a distributed Dominating

[1]http://www.n-gage.com/

[2]http://www.us.playstation.com/psp.aspx

Set (DS) computation algorithm called PBS (Priority Based Selection) will be proposed. This graph theory based algorithm computes an appropriate DS of the ad hoc network graph in a fully distributed way containing nodes which can be used as zone servers. Note that the goal in developing PBS was not to create the 'best' distributed DS computation algorithm rather to compute an appropriate DS with reasonable time complexity and signaling overhead. Moreover, to ensure the smooth running of a real-time application in a self-hoc network, the set of zone server nodes must be maintained and recomputed on the fly when it's required (e.g., in case of network topology changes or link failures). PBS is the first algorithm, according to our knowledge, that offers continuous maintenance of the DS even if the network graph changes dynamically. PBS shows a stable performance even in case of high node mobility keeping the DS computation time nearly constant. Note that although this work was mainly motivated by mobile ad hoc multiplayer games, the applied zone-based architecture and the PBS algorithm to select zone server nodes can be used also in case of other mobile applications like collaborative working tools, multimedia entertainment or 'edutainment'.

To be able to get an appreciation about PBS, the algorithm has been analyzed and evaluated via simulations in the NS-2 [10] network simulator and it will be shown how the distributed service architecture created and maintained by applying PBS can be used in managing real-time multiplayer games in self-organized mobile ad hoc networks. Finally, the algorithm's implementation in a real self-hoc network testbed called SIRAMON (Service provIsioning fRAMework for self-Organized Networks) [11] will be described. This framework has been developed at the Computer Engineering and Networks Laboratory [3] in a previous Master's Thesis [12] to support provisioning of services, i.e., the description, indication, deployment and management, in MANETs.

The report is divided into the following chapters:

- *Chapter 2* - *Related Work* - Introduces the Zone-based Game Architecture and gives an overview about the current state of the art concerning ad hoc networks, zone servers and choosing Dominating Sets.

- *Chapter 3* - *Zone Server Selection* - Introduces the Priority Based Selection (PBS) algorithm for the Zone Server selection supporting a Zone-based Game Architecture.

- *Chapter 4* - *Simulations and Evaluation* - Evaluates the PBS algorithm based on the simulation results.

- *Chapter 5* - *Implementation* - Gives an overview about the implementation of the PBS algorithm in the SIRAMON framework.
- *Chapter 6* - *Conclusions and Outlook* - Concludes the work of this Master's Thesis and gives an outlook for further projects.
- *Appendix A* - *NS-2 Implementation* - Gives an overview about the implementation of the PBS algorithm in the NS-2 simulator.
- *Appendix B* - *SIRAMON Implementation* - Gives some more detailed insights of the PBS implementation in the SIRAMON framework.
- *Appendix C* - *Finite State Machine (FSM)* - Explains the Finite State Machine used by the implementations of the PBS algorithm.
- *Appendix D* - *Used Abbreviations* - Index of the used abbreviations of this report.

1.2 Tasks and Working Plan

The project has been divided into the following tasks:

1. *Literature exploration*

   Collect the available material and documentation about algorithms choosing Dominating Sets and get the current state of the art concerning ad hoc networks and zone servers.

2. *Requirements to select Zone Server nodes*

   Define the requirements to select the zone server nodes. What is the ”best” solution for the choice of the server nodes in an ad hoc network?

3. *Evaluation criteria for comparing the modified (C)DS algorithms*

   Define the criteria that can be used to evaluate and compare different algorithms determining Dominant Sets in ad hoc networks.

4. *Develop an own C(DS) algorithm*

   Devlop an own algorithm that is appropriate for selecting the Zone Server nodes according to the elaborated criteria from the previous tasks.

5. *Analytical evaluation and comparison*

   Perform an analytical evaluation and comparison of the different algorithms according to the defined criteria and the own developed algorithm.

6. *Implementation and evaluation in a simulator*

   Implementation and evaluation of the developed algorithm in a network simulator. The used simulator will be NS-2 [10] that is an event driven network simulator and can be used to analyze the algorithm.

7. *Implementation and evaluation in the SIRAMON framework*

   Implementation and evaluation of the developed algorithm in the SIRAMON framework using Java language.

In Table 1.1, the working plan of the project is shown. The numbers in brackets refer to the defined tasks above.

1.3 General Regulations

The project will be guided by Károly Farkas. At the end of the project, a written thesis report describing the work and the outcomes as well as the documentation of the developed code have to be delivered. The master student understands and accepts the terms and regulations of ETH in regard to the developed code which will be published as open source under the terms of the GNU General Public License (GPL) [13]. In the course of the work an intermediate and a final presentation have to be given. An accepted thesis report and successfully accomplished presentations are the prerequisites for getting the final grade.

Start:	Tuesday, 1st March 2005
End:	Wednesday, 31st August 2005

Week	Date		Task
1	March	1st - 6th	Start of the work
2		7th - 13th	Literature exploration (1)
3		14th - 20th	
4		21st - 27th	Requirements to select zone server nodes (2)
5		28th - 3rd	
6	April	4th - 10th	Evaluation criteria for comparing the modified (C)DS algorithms (3)
7		11th - 17th	
8		18th - 24th	Modify the C(DS) algorithms (4)
9	May	25th - 1st	
10		2nd - 8th	Analytical evaluation and comparison (5)
11		9th - 15th	
12		16th- 22th	Implementation and evaluation in a simulator (6)
13		23rd - 29th	
14	June	30th - 5th	
15		6th - 12th	
16		13th - 19th	
17		20th - 26th	Implementation and evaluation in SIRAMON (7)
18	July	27th - 3rd	
19		4th- 10th	
20		11th- 17th	
21		18th - 24th	
22		25th- 31th	Writing Master Thesis
23	August	1st - 7th	
24		8th - 14th	
25		15th - 21st	
26		22th - 28th	
27		29th - 31st	Hand in the Thesis

Table 1.1: Working Plan

Chapter 2

Related Work

In this chapter, the related work to the task of this thesis is presented and the current state of the art concerning gaming architectures and computing Dominating Sets in a distributed manner is shown. In the first section, a new Zone-Based Gaming architecture well suited for real-time multiplayer games is introduced and compared to the traditional peer-to-peer and centralized server architectures. The second section gives an overview about existing distributed algorithms that computes Dominating Sets of nodes based on an existing graph.

2.1 Game Architectures

Most of the commonly used game architectures nowadays follow one of two approaches: First, a central server design where the server receives the state change events of the game from the users, recalculates the overall state and distributes the changes in the game state back to the user. Or second, a completely distributed model often referred to as peer-to-peer, where every player sends state changes directly to all the other players. In this thesis, a new concept as defined in [9] will be used: the concept of a 'Zone Server'. There exist similar approaches called 'Mirror Server Architecture' [14] or 'Proxy Server Architecture' [15], but in this thesis the concept of Zone Servers will be preferred.

The Zone-Based Gaming Architecture means some players are elected as Zone Servers and each of them receives the state changes of only a group of players. These Zone Servers communicate also with the other Zone Servers to propagate the game state changes to all the other players. The Zone Servers are topologically distributed across the network and the clients connect to their closest Zone Servers. In Fig. 2.1, an example of this architecture for a typical ad hoc network is shown in which each group has its own Zone Server. Nodes 1 and 3 are connected to Zone Server 2 while nodes 8 and 9 are connected to Zone Server 10. The white nodes 4, 5, 6, and 7 are not participating in the game session and are called auxiliary nodes, because it can happen that they have to forward the game traffic. In the shown network, this is the case for node 4 and 6.

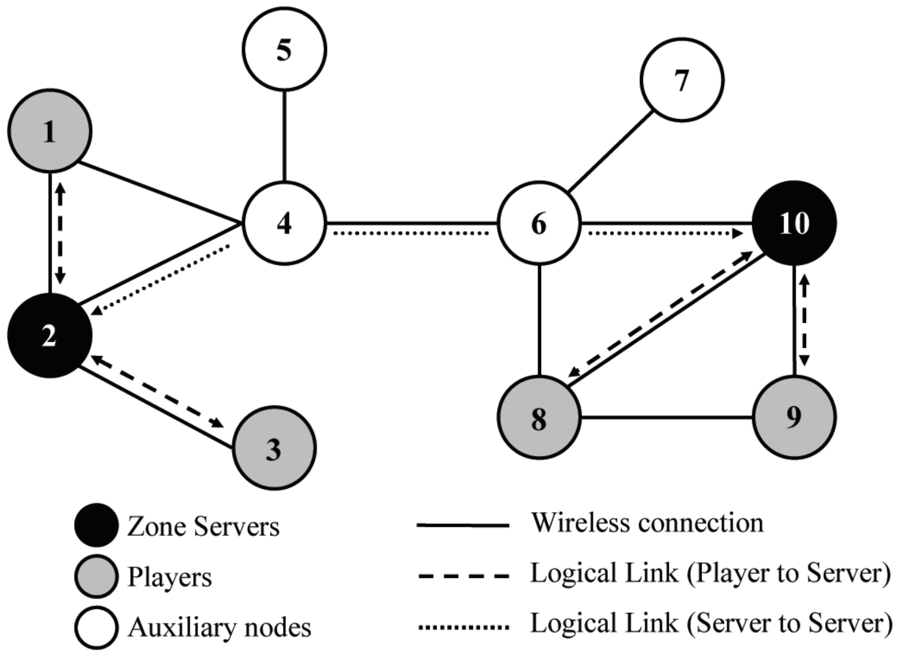


Figure 2.1: Example Zone-Based Game Architecture in a Self-Hoc Network

2.1.1 Comparing Different Architectures

In case of the *centralized* architecture, every player connects to a single central machine, the game server, that knows the game rules and acts as a master authority on the game state. Clients send state updates to the server and the server sends authoritative updates (based on the game logic) back to each client. In this model, the server makes sure that the rules of the game are followed. This centralized architecture also allows the implementation of different security features to avoid cheating. The main problems with this model, which make it unsuitable to be used in self-hoc networks, are the reduced fault tolerance (a centralized server represents a single point of failure for the game), the limited scalability (computation and bandwidth problems may arise if too many players are connected to one server) and the required central administration.

The *peer-to-peer* architecture follows the opposite philosophy. Here the device of each player maintains a local copy of the game state and informs every other player whenever the game state changes. With this architecture, a good fault tolerance level can be achieved because there is no single point of failure (if one player has technical difficulties, still the others will be able to keep playing). The main problems with this architecture are the relatively easy cheating possibility (players are able to cheat by modifying their local copies of the game) and limited scalability (as everybody communicates to everybody else, the bandwidth required at every player can be pretty high).

The *zone-based* game architecture proposed in [9] provides a robust, redundant server-client model that is more appropriate for the self-hoc environment. In this approach, some nodes act as Zone Servers and each Zone Server is in charge of a small group of players. For efficiency reasons, this group should be close to its Zone Server. The Zone Server receives updates from its players and propagates the game state change to all other players via their Zone Servers. If a zone server looses connection or is shut down or disappears, its players will be able to keep playing by using another Zone Server. In Table 2.1 the three different architectures are compared to each other from different view points.

2.2 Zone-Based Game Architecture

2.2.1 Characteristics of Zone Servers

The characteristics of wireless devices are very different from the wired counterparts. There are devices with very tight resources, like mobile phones, and also devices with more of power like PDAs or big machines like laptops. Each device has different computing power, memory, batteries etc. For this reason, there are many things that should be taken into account when deciding if a player is able to provide a server service or not. For example, it has to be considered if this device has spare power to run not only the game but also to act as Zone Server. On one hand it is not eligible to have single point of failure, thus, all players should be able to act as Zone Servers. On the other hand, from the network point of view it is difficult to know which host is a better suited server than another, because the prediction of the movement of a node is difficult. After all Zone Servers should be devices with enough spare resources since the overhead created by the server should not be noticeable by the players at the Zone Server. But there are also other factors that are important. In order for a node to know whether it can be a Zone Server or not, some kind of benchmark should be run to measure the appropriate factors.
The Zone Servers should be prepared not only to handle the players under their control, but also the players from other Zone Servers. This could become necessary in order to keep playing when a player loses the connection to its Zone Server.

	Centralized	**Peer-to-Peer**	**Zone-based**
Performance	The performance is determined by the resources of the central server and influences all players	The performance is determined by the local device and influences only the local player	The performance is determined by the Zone Server and influences the players of a zone
Bandwidth	Bandwidth problems may arise if too many players are connected to the server	As everybody communicates to everybody, the bandwidth required at every player is higher than in the server based approach	The bandwidth is shared among the players of one zone. If the resource is running out a new zone can be created
Fault Tolerance	The centralized server is a single point of failure	If a player's device fails, it does not affect the other players	If a Zone Server fails, the players should be able to connect to another Zone Server
Synchronisation	No synchronisation needed	Synchronisation needed between all players	Synchronisation needed between the Zone Servers
Latency	Low Latency	High Latency	High Latency
Scalability	The scalability is limited by the resources of the central server	Because every player has to send the information to all other players, scalability is an issue as well	The architecture scales well because new zones can be created at every time
Cheating	The centralized architecture allows the implementation of security features	It is very easy for players to cheat by modifying their local copy of the game	Cheating is still possible if a specific Zone Server can be manipulated

Table 2.1: Properties of Game Architectures

2.2.2 Task of Zone Servers

The task of a Zone Server will be discussed on the example of a real-time game. Such a game is a continuous application, but the game simulation, i.e. its processing by the participating nodes, is usually done at certain points in time which discretizes the application. During the processing of a real-time game, the state of the application, i.e. the state of all dynamic entities in the game like their position and the currently performed action, is periodically altered by the server. Therefore, the game can be modeled as a sequence of state transitions. At each transition, the new state S_{i+1} is calculated from the current state S_i and the user inputs occur during the time the game simulation was in the state S_i. In [8] a Game Scalability Model (GSM) is introduced that provides a possibility to compare the scalability of different network topologies. This model can mainly be used for studying the scalability of network topologies, but it gives as well a deep insight into the tasks and the required performance by a Zone Server. In the work itself the three different architectures (Client-Server, Peer-to-Peer, and Proxy Server Network) have been compared. Note that the used Proxy Server Network is similar to the Zone Server Architecture. However, in this thesis the GSM model will be applied with slight changes to the Zone Server Architecture.

The situation for a Zone Server is outlined in Fig. 2.2. Zone Server 1 has k connected clients that are participating in the game session (note that in most of the cases the Zone Server itself is also one of these clients) and there exist in total l Zone Servers that are connected with each other. In the considered game session it is assumed that there are in total $n = k \cdot l$ participating clients and m game-controlled entities. Such entities are the dynamic parts of the game not controlled by participating users. Of course, this is the ideal case, if the number of clients are distributed equally between the different Zone Servers. In the worst case there is one server with $n - l$ clients, and all other servers have only one client. It is further assumed that the administration of the server-controlled entities is also equally distributed among the Zone Servers. With $m = l \cdot s$, each of the l servers have to process s entities.

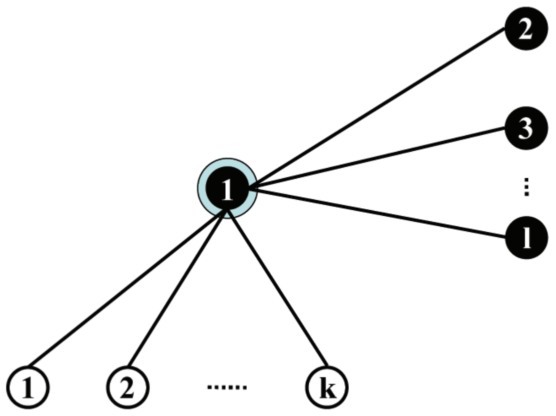


Figure 2.2: Zone Server Architecture from the View of a Zone Server

The following steps are necessary in a state transition at a Zone Server:

- Receiving, validating, and processing of k actions from the different clients, which requires the maximum time for processing of $t_{ca}(n, m)$ per client, depending on the total number of players and game entities. Each received message has a constant size d_{cin}.

- Receiving and processing of $n-k$ remote client actions and state updates from other Zone Servers. Each message has constant size d_{rsu} with constant processing time t_{rsu}.
- General processing of the game world. Process the s game-controlled entities, which requires $t_{se}(n,m)$ for each entity in the worst case. Additionally, this requires each Zone Server to receive information about $m-s$ remotely managed server entities, with a constant size of d_{rsu} for each message.
- Filtering and transmission of the new state S_{i+1} to clients. This takes $t_{fc}(n,m)$ for each client; each message has size $d_{cout}(n,m)$.
- Transmission of updated state information S_{i+1} to $l-1$ Zone Servers. Information about the k local clients and the s server-controlled entities has to be sent, each with an amount of data d_{rsu}. Note that if IP multicast is available between the Zone Servers, this information has to be sent only once.

The overall time for the calculation of a single state transition at a Zone Server results in

$$T_{ZS}(l,n,m) = \frac{n}{l} \cdot t_{ca}(n,m) + (n - \frac{n}{l}) \cdot t_{rsu} + \frac{m}{l} \cdot t_{se}(n,m) + \frac{n}{l} \cdot t_{fc}(n,m) \tag{2.1}$$

The used bandwidth for incoming traffic amounts to

$$D^{in}_{ZS}(l,n,m) = \frac{n}{l} \cdot d_{cin} + (n - \frac{n}{l}) \cdot d_{rsu} + (m - \frac{m}{l}) \cdot d_{rsu} \tag{2.2}$$

while outgoing traffic requires a bandwidth of

$$D^{out}_{ZS}(l,n,m) = \frac{n}{l} \cdot d_{cout}(n,m) + (l-1) \cdot (\frac{n}{l} + \frac{m}{l}) \cdot d_{rsu} \tag{2.3}$$

where the term $(l-1)$ will be reduced to 1 if multicast is supported.

It can be seen from equation (2.1) that with the increasing number l of Zone Servers the second term increases as well, in contrast all other terms decrease. With the increasing number of clients and game-controlled entities the t_{rsu} remains constant and all other processing times will increase. In the worst case consideration where one server has $n-l$ clients, the term $\frac{n}{l}$ have to be replaced by $n-l$ in the equations above. These equations will be used in section 3.1.2 to evaluate different possible selections of Zone Servers.

Because lost or delayed messages can cause inconsistencies between game states a consistency protocol is needed to deliver the messages to their destination, and a synchronization mechanism must be used to detect and resolve inconsistencies. Both of them have to be supported by the Zone Server and influence the processing times. A possible consistency protocol and synchronization mechanism called trailing state synchronization (TSS) can be found in [14].

2.2.3 Detection of Zone Servers

A client can detect existing Zone Servers with the help of a service location protocol. For example, Konark [16] is a service discovery protocol designed for mobile ad hoc networks and targeted toward device independent services. To describe a wide range of services, Konark defines an XML-based description language, based on WDSL, that allows services to be described in a tree-based human and software understandable form. The service advertisements contain name and address of the service, and a time-to-live (TTL) information, as well. Based on this information, a client is able to detect the nearest available Zone Server.

2.2.4 Selection of Zone Servers

The Zone Server nodes must be selected carefully and this selection must be maintained even in case of changes in the network topology or available resources. The most powerful devices (concerning available computation and communication resources) also taking into account their positions in the network should act as Zone Servers. Since the player nodes should be close to their Zone Servers to decrease the latency of their responsiveness, a distributed Dominating Set computation algorithm called PBS (Priority Based Selection) for selecting the Zone Server nodes will be presented in chapter 3.2. In this sense, a self-hoc network can be considered as a graph and the problem can be mapped into the computation and maintenance of an appropriate Dominating Set of this graph containing the most suitable (most powerful with ‘good’ position in the network) nodes. In chapter 3.1, the requirements of this selection of Zone Servers will be discussed.

2.3 Existing Dominating Set Computation Algorithms

In this section, an overview of the existing distributed algorithms building a (connected) dominating set of nodes in an existing graph is given. Note that most of these algorithms have been developed for the purpose of providing routing functionality in ad hoc networks. This means that these algorithms do not fulfill all the requirements, as described in chapter 3.1, for selecting the Zone Servers.

2.3.1 Problem Definition

A Dominating Set and its different variations can be defined as follows:

Dominating Set (DS): A dominating set of a graph G = (V, E) is a subset $S \subseteq V$ of the nodes such that for all nodes $v \in V$, either $v \in S$ or a neighbor u of v is in S.

Connected Dominating Set (CDS): If all the nodes of the subset S induce a connected subgraph, S is spanning a Connected Dominating Set.

Minimum Dominating Set (MDS): A dominating set is called a Minimum Dominating Set if the number of nodes in S is minimal. Note that finding a dominating set of minimum size is NP hard ([17], [18]).

Node Weighted Dominating Set (NWDS): If the nodes have weights, a Node Weighted Dominating Set is the smallest weighted subset S of the nodes forming a Dominating Set of the graph.

Edge Weighted Connected Dominating Set (EWDS): Find the smallest weight tree with the subset S those edges have smallest weight.

Steiner Tree problem: The Steiner Tree problem is defined as follows: Given a graph $G = (V, E)$ and a set $R \subseteq V$ of required nodes in an edge weighted graph, find a minimum weight tree connecting the nodes in R. Note that the tree may include other nodes that are not in R. The Steiner Tree problem can be used, for example, to build a CDS based on a DS.

2.3.2 Notations and Evaluation

In general, the following notations as listed in Table 2.2 will be used.

Unit Disk Graph (UDG): The topology of a wireless ad hoc network can be modeled as a Unit Disk Graph (UDG) [19]. This is a geometric graph in which there is an edge between two nodes if and only if their distance is at most one. In Fig. 2.3 an example is shown. The dashed circles represent the transmission range of the node in the center of the circle. If a node is inside the circle of another node, a edge between these nodes can be drawn, because the nodes are in the range of each other. In a UDG it is assumed that every node's transmission range can be represented by a unit circle.

V	the set of nodes
E	the set of edges
$G(V,E)$	the graph defined by nodes and edges
Δ	maximum degree in graph G
n	number of nodes
$N(v_i)$	open neighborhood (contains all neighbor nodes) of node v_i
$N[v_i]$	closed neighborhood (contains all neighbor nodes including node v_i itself) of node v_i

Table 2.2: Used Notations

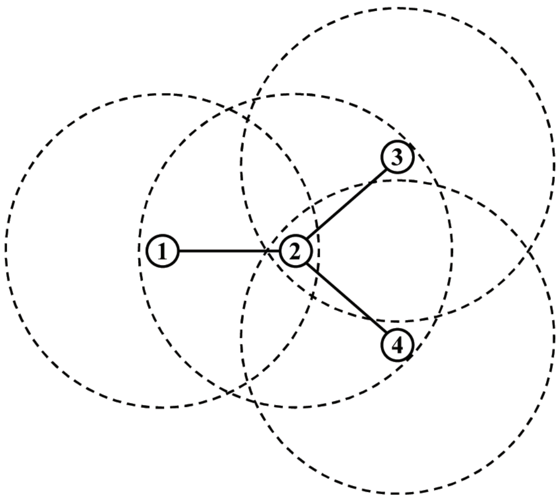


Figure 2.3: Example of a Unit Disk Graph (UDG)

The discussed algorithms will be evaluated according to some quality and construction cost factors. The quality factors are:

- **Connected:** Indicates whether the Dominating Set is connected or not.
- **Approximation:** As already mentioned to find a MDS is a classic NP-complete optimization problem and has to be approximated. The approximation factor is defined as the ratio of the computed Dominating Set's size to that of the MDS:

$$\frac{|DS|}{|DS_{MDS}|} = Approximation$$

where $|DS|$ is the number of nodes in the DS and $|DS_{MDS}|$ the number of nodes in the MDS, respectively.

The construction costs can be characterized by:

- **Time Complexity:** The time complexity is measured in rounds. One round consists of sending a message, receiving a message and some local computation.
- **Message Complexity:** Indicates how many messages have to be sent and their size.

2.3.3 Largest-ID Algorithm

The Largest-ID algorithm [20] is a simple algorithm. Each node in the graph has a unique ID. On every node the algorithm performs:

```
1: send ID to all elements of N(v)
2: tell node with largest ID in N(v) that it has to join the DS
```

An example of the algorithm is outlined in Fig. 2.4. The red arrows show to which node the different nodes communicated to join the DS. For example, node 1 told node 3, or node 2 told node 4 to join the DS. In Fig. 2.5 the same graph is shown but with different node IDs. It can be seen that in this case the algorithm gives the optimal solution. Generally it can be said, that for typical settings the algorithm produces very good Dominating Sets. If the nodes know the distances to each other, there is an iterative variant which computes a constant approximation in $O(\log \log n)$ time. Table 2.3 shows the quality and construction cost factors for the Largest-ID algorithm.

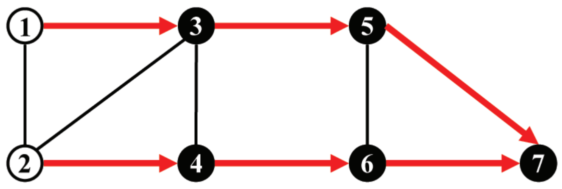


Figure 2.4: Largest-ID algorithm

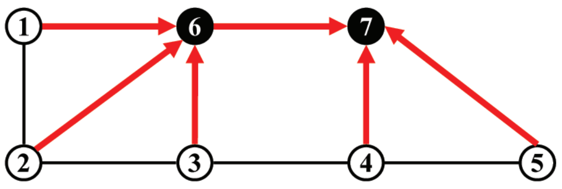


Figure 2.5: Largest-ID algorithm for another arrangement of the node IDs

Connected DS:	No
Approximation:	$O(\sqrt{n})$ for UDG. $Constant$ for iterative algorithm
Time Complexity:	2 rounds
Message Complexity:	n messages

Table 2.3: Largest-ID Algorithm

2.3.4 Local Randomized Greedy Algorithm

The main idea behind the greedy algorithm is to choose the ’good’ nodes into the Dominating Set. The nodes in the algorithm get different colors: black nodes are in the Dominating Set, gray nodes are neighbors of nodes in the DS, and white nodes are not yet dominated and also referred to as candidates. Initially all nodes are white. The greedy algorithm chooses those nodes in the DS that color most white nodes. The number of white neighbors (including itself) is called span value. A basic implementation of the distributed greedy algorithm at node v looks like:

```
1: while v has white neighbors do
2:    compute span d(v)
3:    if d(v) largest within 2 hop counts (ties broken by ID) then
4:       join DS
5:    fi
6: od
```

An illustrated example of this algorithm is shown in Fig. 2.6 and 2.7. After the first round node 3 joins the DS because it has with $d(v_3) = 5$ the highest span value. All neighbors of node 3 becomes gray (Fig. 2.6). In the next round, nodes 6 and 7 have the same span value $d(v_6) = d(v_7) = 2$, but because of the smaller ID node 6 joins the DS. Now there are no white nodes left, and nodes 3 and 6 are in the dominating set. The final result is shown in Fig. 2.7.

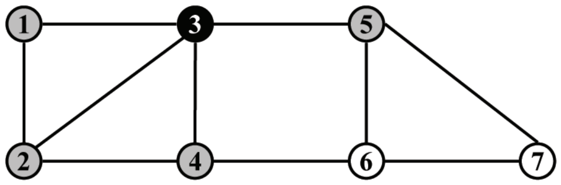


Figure 2.6: Greedy Algorithm - After the first Round

As mentioned in [21], this simple implementation has some considerable drawbacks in the needed calculation time for some special graphs like a caterpillar or star-complete graphs. Therefore, the algorithm has been enhanced and is called *Local Randomized Greedy (LRG)*

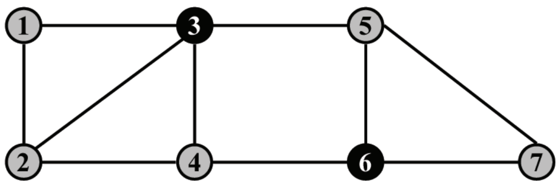


Figure 2.7: Greedy Algorithm - Final Dominating Set

algorithm. The LRG algorithm proceeds in rounds. Every round contains the following steps:

Span calculation: Calculate for each v the span $d(v)$, which is as already mentioned above the number of uncovered white nodes that are adjacent to v (including v itself if uncovered). Let the rounded span $\hat{d}(v)$ of node v be the smallest power of base b that is at least $d(v)$, where $b > 1$ is a constant integer (often $b = 2$). The parameter b represents a tradeoff between time complexity and approximation ratio.

Candidate selection: Node $v \in V$ is a candidate if $\hat{d}(v) \geq \hat{d}(w)$ for all $w \in V$ within 2 hop counts of v. For each candidate v, let cover $C(v)$ denote the set of candidate neighbors of node v.

Support calculation: For each uncovered node u, its support $s(u)$, which is the number of candidates that cover u, is calculated.

Dominator selection: Each candidate v joins the DS with probability $1/med(v)$, where $med(v)$ is the calculated median of the support values of the nodes in $C(v)$.

The quality and construction cost factors of the LRG algorithm are summarized in Table 2.4. There are no information available about the message complexity.

Connected DS:	No
Approximation:	$O(log\Delta)$
Time Complexity:	$O(lognlog\Delta)$ rounds
Message Complexity:	N/A

Table 2.4: LRG Algorithm

Node weighted LRG:

The LRG algorithm can also be extended to build a node weighted Dominating Set. Instead of comparing the rounded span of the nodes, the ratio of the span to the weight of the node can be compared. This value is again rounded to the nearest power of a constant $b > 1$ (allowing negative powers) and is referred to as normalized span. In the candidate selection phase, a node selects itself as a candidate only if the rounded normalized span is the maximum among all the nodes within a distance of two hop counts. The remaining phases in each round are identical to the LRG. Note that the number of different values for the rounded normalized span is $O(log(W\Delta))$, where W is the ratio of the maximum weight to the minimum weight. The quality and construction cost factors of a node weighted LRG algorithm are summarized in Table 2.5.

Connected DS:	No
Approximation:	$O(log\Delta)$
Time Complexity:	$O(log(W\Delta)logn)$ rounds
Message Complexity:	N/A

Table 2.5: Node Weighted LRG Algorithm

2.3.5 Marking Algorithm

The Marking Algorithm as described in [22] uses a marker for every node $m(v) \in [T, F]$, whereas T means marked and F unmarked, respectively. The algorithm is defined as follows:

```
 1: Assign m(v)=F to every node
 2: Every node v exchanges N(v) with all neighbors
 3: Every node v assigns m(v)=T if there exist two unconnected neighbors

    Extension 1:
 4: if N[v] <= N[u] AND id(v)<id(u)
 5:    m(v)=F
 6: fi
 7: send status to neighbors

    Extension 2:
 8: if u,w neighbor of v in DS
 9:    if N(v)<=N(u)+N(w) and id(v)=min{id(v),id(w),id(u)}
10:       m(v)=F
11:    fi
12: fi
```

An example of the algorithm is shown in Fig. 2.8 and 2.9. The numbers above the nodes represent the node ID. The closed neighborhoods are:

$$\begin{aligned}
N[v_1] &= \{1,2,7\} \\
N[v_2] &= \{1,2,3,7\} \\
N[v_3] &= \{2,3,4,7\} \\
N[v_4] &= \{3,4,5,6\} \\
N[v_5] &= \{4,5,6\} \\
N[v_6] &= \{4,5,6,7\} \\
N[v_7] &= \{1,2,3,6,7\}
\end{aligned}$$

Node v_2 joins the DS because node v_1 and v_3 are not directly connected. Furthermore, nodes v_3, v_4, v_6, and v_7 are joining the DS as well because they have neighbors that are not connected. It can be seen that the approximation is quite poor (see Fig. 2.8).

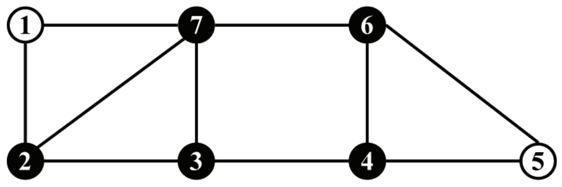


Figure 2.8: Marking Algorithm without Extensions

Using Extension 1, node v_2 marks itself 'F' because $N[2] \subseteq N[6]$ and $id(2) < id(3)$. Because of Extension 2, node v_3 leaves the DS because $N[3] \subseteq N[4] \cup N[7]$ and $id(3) = min\{id(3), id(4), id(7)\}$. Note that if the IDs would have been distributed in another way, this node could not be removed. The final result of the DS for this graph after using Extension 1 and 2 is shown in Fig. 2.9.

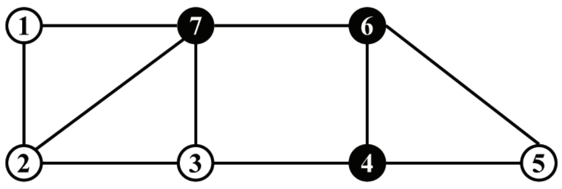


Figure 2.9: Marking Algorithm with Extensions

The quality and construction cost factors of the algorithm are summarized in Table 2.6. Note that there exists only some simulations that investigated the approximation factor of the algorithm. There is no analytical expression available.

Connected DS:	Yes
Approximation:	N/A
Time Complexity:	2 rounds
Message Complexity:	$O(\Delta n)$ messages

Table 2.6: Marking Algorithm

2.3.6 LP-Relaxation Algorithm

In [23] a fully distributed algorithm is presented that approximates the MDS problem by the use of Linear Programming (LP) Relaxation. First, the integer program which describes the MDS problem has to be derived. Let $S \subseteq V$ denote a subset of the nodes of G. To each node $v_i \in V$, a bit x_i is assigned, such that $x_i = 1 \Leftrightarrow v_i \in S$. For S to be a Dominating Set for each node $v_i \in V$ at least one of the nodes in $N(v_i)$ has to be in S. Therefore, S is a Dominating Set of G if and only if $\forall i \in [1, n] : \sum_{j \in N(v_j)} x_j \geq 1$. With the neighborhood matrix N, defined as the sum of the adjacency matrix of G and the identity matrix (N is the adjacency matrix with ones in the diagonal), the MDS problem can be formulated as an integer program:

$$min \sum_{i=1}^{n} x_i \text{ subject to } N \cdot \underline{x} \geq \underline{1} \text{ and } \underline{x} \in \{0,1\}^n$$

By relaxing the condition $\underline{x} \in \{0,1\}^n$ to $\underline{x} \geq \underline{0}$, the linear program can be derived:

$$min \sum_{i=1}^{n} x_i \text{ subject to } N \cdot \underline{x} \geq \underline{1} \text{ and } \underline{x} \geq \underline{0}$$

In [23] two algorithms are shown. The first algorithm solves the LP program and the second transforms the solution from the first algorithm to the Integer Program solution. In Table 2.7, the quality and construction cost factors of these two algorithms are shown. Note that k is a parameter that can be chosen arbitrarily.

2.3.7 Dominator Algorithm

In [24] a distributed approximation algorithm that constructs a Minimum Connected Dominating Set (MCDS) for the Unit Disk Graph has been proposed. The construction of the CDS can be briefly described in two phases:

Connected DS:	No
Approximation:	$O(k\Delta^{\frac{2}{k}} log\Delta)$
Time Complexity:	$O(k^2)$ rounds
Message Complexity:	$O(k^2\Delta)$ messages of size $O(log\Delta)$

Table 2.7: LP-Relaxation Algorithm

- In the first phase, a maximal independent set (MIS), i.e., an independent Dominating Set S is constructed. This means that any pair of nodes in S are separated by at least two hops, and any subset of nodes in S is at most three hops away from the other nodes in S. The nodes in S are referred to as *dominators*, and the nodes not in S are referred to as *dominatees*.

- In the second phase, each dominatee node identifies and broadcasts this information. Using such information from all neighbors, each dominator node identifies a path of at most three hops (not necessarily the shortest one) to each dominator that is at most three hops away from itself and has larger ID than its own ID, and informs all nodes on this path about this selection. The set C then consists of all dominatee nodes on these paths, which are referred to as *connectors*. The DS consists finally of the dominator and connector nodes.

An example of the dominator algorithm is shown in Fig. 2.10, 2.11 and 2.12. White nodes are referred to as candidates, black nodes as dominators, grey nodes as dominatees, and nodes with a point as connectors. Building the MIS of the first phase is based on the node's unique ID. Therefore, in the first step node 1 declares itself as dominator because it has no candidate neighbors with a lower ID and nodes 2 and 7 change their states to dominatee as illustrated in Fig. 2.10.

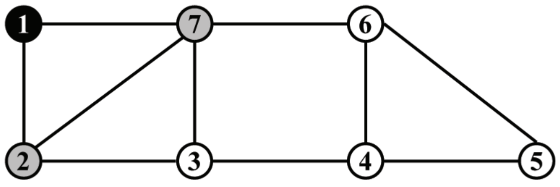


Figure 2.10: First Step of the Dominator Algorithm

In the next step, node 3 becomes dominator because there are no candidate neighbors with lower ID left and node 4 changes to dominatee. Finally, node 5 becomes as well dominator and node 6 dominatee. The first phase of the algorithm is now finished and the built MIS is shown in Fig.

2.11.

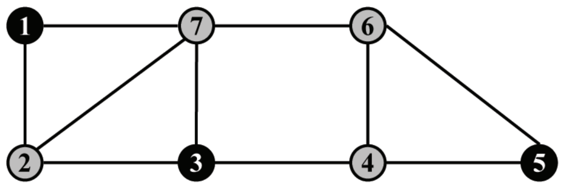


Figure 2.11: MIS of the Dominator Algorithm

In phase 2, the different dominator nodes get now connected, for building the CDS, by nodes 4 and 7 that switch to the connector state. Note that it also would be possible that node 2 becomes a connector. This depends on whether node 7 or 2 announces first to node 1 its neighborhood to dominator node 3. The final CDS consisting of the dominator and connector nodes is shown in Fig. 2.12. The quality and construction cost factors of the Dominator algorithm are summarized in Table 2.8.

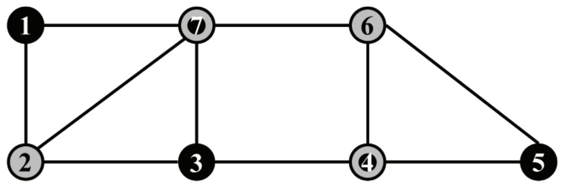


Figure 2.12: Final CDS of the Dominator Algorithm

Connected DS:	Yes!
Approximation:	$O(4)$
Time Complexity:	$O(n)$ rounds
Message Complexity:	$O(n)$ messages of size $O(logn)$

Table 2.8: Dominator Algorithm

2.3.8 Removing Cycles Algorithm

In [25] a fast distributed algorithm is shown. Order the edges of G in some fashion: say each edge has a unique ID (if the edges do not have such IDs, a simple distributed way of achieving this is to make each edge choose a random real number in [0,1] as its ID, this ID has to be agreed of both nodes that are connected to the edge). Each edge, in parallel, drops out if it is the edge with the smallest ID in a cycle of a length less than $1 + 2\log n$. A node is in the CDS if it is not a leaf node in the remaining subgraph. Disadvantage: Every node has to know the number of nodes in the graph, or at least an estimation of it. A simple implementation could look like:

```
1: Assign ID to all edges, send them to all neighbors
2: Accept Edge-ID of neighbor if neighbor has higher Node-ID
3: send N(v) together with Edge-IDs to all neighbors
4: the different N(v) has to be forwarded as far as is needed
   to recognize cycles of length less than 1+2log(n)
4: detect cycles of length less than 1+2log(n) and remove edges
5: join DS if there are still 2 nodes with active edge
```

The quality and construction cost factors of the Removing Cycles algorithm are summarized in Table 2.9.

Connected DS:	Yes!
Approximation:	$O(log\Delta)$
Time Complexity:	$O(n)$ rounds
Message Complexity:	$O(n(n + 2logn))$ messages

Table 2.9: Removing Cycles Algorithm

2.3.9 Steiner Tree Algorithms

As already mentioned, the task of an algorithm that solves the Steiner Tree problem is to build a connected tree containing the nodes of the subset $R \subseteq V$ of a given graph $G = (V, E)$. There exist several approximation algorithms that have been developed for producing graphs that fulfill this Steiner Tree problem.

In [26] an algorithm for node weighted Steiner Trees has been introduced. The nodes of R are referred to as *terminals* and the algorithm maintains a node-disjoint set of trees containing all these terminals. Initially, each terminal is in a tree by itself. The algorithm uses a greedy strategy to iteratively merge the trees into larger trees until there is only one tree. The weight of a tree is defined as the sum of all weights of the nodes that are part of this tree. In each iteration, the algorithm chooses node v to join a tree that minimizes the ratio of the weight of a tree to the number of terminals that it connects. It is proved that this greedy process yields a good approximation.

Based on this algorithm, in [27] some improved methods for approximating Node Weighted Steiner Trees with a better approximation factor has been shown. Unfortunately, this algorithm is computationally intensive and is not well suited for large networks. Therefore, a much faster algorithm has been presented as well with the same approximation factor as in [26]. This algorithm has been transformed to the case if all nodes in a network have the same (or no) weights. An application of this algorithm to build a Connected Dominating Set is shown in [28]: The algorithm runs in two phases. At the start of the first phase all nodes are colored white. Each time a node joins the DS it changes the color to black. Nodes that are dominated and therefore adjacent to a black node are colored gray. In the first phase, the algorithm picks a node at each step and colors it black and all adjacent white nodes gray. A *piece* is defined as a white node or black connected component. At each step a node that gives the maximum reduction in the number of pieces is chosen. At the end of the first phase there are no white nodes left and we have a DS that is not connected. In the second phase, there is a collection of black connected components that need to be connected. This is done by recursively connecting pairs of black components by choosing a chain of nodes, until there is one black connected component.

This procedure can be generalized for the Connected Dominating Set problem:

1. Use an existing algorithm that builds a DS.
2. Use a Steiner Tree approximation algorithm that connects the nodes that are elements of DS.

Another application of the Steiner Tree problem is to build a backbone for multicast applications in wireless ad hoc networks where a tree is needed that contains forwarding nodes and the nodes that joined the corresponding multicast group. In [29] an algorithm called d-hop algorithm is shown that computes a Steiner Connected Dominating Set (SCDS) containing the nodes V_m that are part of the multicast group.

2.3.10 Conclusions

A new Zone-based Gaming Architecture has been proposed that provides a robust, redundant server-client model that is more appropriate for the self-hoc environment than the centralized server or peer-to-peer architecture approaches. In this architecture, there exist several servers that are distributed across the network and the clients will connect to their closest Zone Server. The server will run among each other some synchronization and consistency mechanisms to distribute the game states of the different clients. If a Zone Server looses connection or disappears, its players will be able to keep playing by using another Zone Server. However, the Zone Servers must be selected carefully. Due to the mobility of the devices in an ad hoc network, this selection must be maintained even in case of changes in the network topology or available resources. If the network is considered as a graph, the problem can be mapped into the computation and maintenance of an appropriate Dominating Set of this graph containing the most suitable nodes.

There exist already plenty of distributed algorithms that build a (Connected) Dominating Set of nodes in an existing graph. Most of these algorithms have been developed for the purpose of providing routing functionality in ad hoc networks. In Table 2.10 and 2.11, the evaluation factors and their performance are summarized.

Algorithm	**CDS**	**Approximation**
Largest-ID	No	$O(\sqrt{n})$ for UDG.
LRG	No	$O(\log \Delta)$
Node weighted LRG	No	$O(\log \Delta)$
Marking	Yes	N/A
LP Relaxation	No	$O(k\Delta^{\frac{2}{k}} log\Delta)$
Dominator	Yes	$O(4)$
Removing cycles	Yes	$O(log\Delta)$

Table 2.10: Summary (CDS and Approximation) of DS Computing Algorithms

Algorithm	**Rounds**	**Message Complexity**
Largest-ID	2	O(n) messages
LRG	$O(\log \Delta \log n)$	-
Node weighted LRG	$O(\log(W\Delta) \log n)$	-
Marking	2	$O(\Delta n)$ messages
LP Relaxation	$O(k^2)$	$O(k^2\Delta)$ messages, size $O(log\Delta)$
Dominator	$O(n)$	$O(n)$ messages, size $O(logn)$
Removing cycles	$O(n)$	$O(n(n + 2logn))$ messages

Table 2.11: Summary (Rounds and Message Complexity) of DS Computing Algorithms

It can be seen, that the different algorithms have different properties according to their quality and construction factors. From the view of time complexity, there are two algorithms (Largest-ID [20] and Marking [22]) that perform in the minimal number of 2 rounds. But this is paid with a higher approximation factor (for the Marking algorithm we found only simulation results and not analytical expression). The best approximation $log\Delta$ results can be achieved with a greedy (LRG [21]) or the Removing Cycles [25] algorithms.

The Dominator algorithm [24] is a distributed approach that constructs in the first step an independent Dominating Set. In the second step, every node in the Dominating Set from the previous phase detects the best paths to the other dominator nodes (that consist at most of 2 intermediate nodes), and forces the intermediate nodes to join the DS, too. At the end a Connected Dominating Set is constructed. This procedure can be generalized for building a Connected Dominatig Set: First, use an existing algorithm that builds a DS. Second, use a Steiner Tree approximation algorithm that connects the nodes that are elements of the DS from the first step. This approach could be used by any Zone Server Selection algorithm, if the set of Zone Servers needs to be connected.

Chapter 3

Zone Server Selection

This chapter presents the Priority Based Selection (PBS) algorithm that computes a Dominating Set. This algorithm compares the priorities of the different nodes and choses the high prioritized nodes into the DS that will act as Zone Servers. The chapter is divided into two sections: The first section describes the requirements for the Zone Server Selection (ZSS) and the second section describes the PBS algorithm that fulfills all these requirements.

3.1 Requirements

This section describes the requirements for the Zone Server Selection in an existing ad hoc network based on building a DS. The requirements are split up into the following subsections:

- The section *3.1.1 Prerequisites* describes the requirements for the nodes and edges in an existing ad hoc network and the network itself.

- The section *3.1.2 Properties of Dominating Set* describes the requirements for the Dominating Set that should be built.

- The section *3.1.3 Requirements for Zone Server Selection Algorithm* describes the requirements for the algorithm that computes the DS.

It is obvious that some of the requirements are conflicting with each other. These conflicts are investigated and a strategy is determined how the different requirements should be prioritized to each other. At the end of the section, a summary with the decision for the chosen requirements and properties is given. This summary builds the base for the development of the algorithm.

3.1.1 Prerequisites

This section describes the prerequisites for selecting the Zone Servers with respect to the nodes and edges of a network.

Connected ad hoc network: It is assumed that there is an ad hoc network consisting of several nodes that span a connected network. Thus, any node can communicate with any other node in this network using a routing protocol.

Neighbor Discovery: There is a neighbor discovery functionality required that guarantees that every node knows its neighbors and is able to detect new nodes and nodes that left the network.

Unique IDs: Every node needs an identification number that is unique within the local ad hoc network. This identifier will be used as last tie breaker in the zone server selection procedure.

Node weight: The Zone Servers should be nodes with enough spare resources so that the overhead created by the server functionality will not be noticed by the player itself. It is obvious that a laptop device is better suited to act as Zone Server than a mobile phone. Therefore, every node should be classified by a weight that indicates the node's capability to act as Zone Server. This node weight should be based on:

- CPU power
- Memory
- Remaining battery power
- Mobility (the probability for a laptop that it moves around is less than for a mobile phone or PDA)
- Link connectivity

It can be seen, that the weight is mainly based on the available resources of a node. But it is also important that the servers provide good connectivities to the connected clients. Therefore, the node weight is also based on the link connectivity that it can offer to the clients. For generating this node weight, there should be some kind of benchmark test included in the framework that measures all these factors and generates the weight value. (Note, that there should be a mechanism that avoids the possibility to pretend a false node weight. I.e., a laptop that claims to have resources like a mobile phone to minimize the probability to join the DS. But this is out of scope of this thesis).

Edge weight: Similar to the nodes, it is also possible that the edges of a graph get different weights that indicates the quality of the link. The edge weight can be based, for example, on the Round Trip Delay (RTT) value.

Co-operative behavior: Co-operative behavior from every node as social contribution is assumed. Every node in an ad hoc network must be willing to contribute to a service, even if it is not participating in this specific service session itself (like offering routing functions to other nodes). In the case of multiplayer games, this means that a node can be selected as Zone Server for a game session even if this node will not participate in the game itself. In Fig. 3.1 a network is shown where the nodes colored in gray (1,2,5, and 7) want to build a game session. Although the other nodes will not participate it is possible that they have to act as Zone Server (in the shown graph there is a high probability that node 4 will be selected as server).

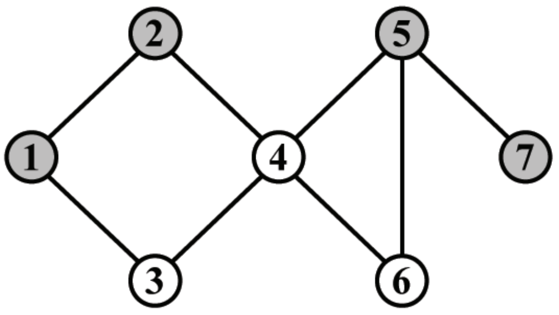


Figure 3.1: An Ad Hoc Network Where the Gray Nodes Want to Build a Game Session

3.1.2 Properties of Dominating Set

This section describes the desired properties of the DS. Some of them are conflicting with each other. These conflicts will be investigated at the end of the section.

Connected DS (optional): Does the DS need to be connected and to build a direct connected backbone of Zone Servers? Or seen from another point of view: In a non-connected DS, there are maximum two nodes between two ’neighboring’ Zone Servers. Does it make sense that such interconnecting nodes of two Zone Servers do not join the DS? An example is illustrated in Fig. 3.2 where nodes 7 and 9 are interconnecting nodes. Some arguments pro and contra are listed in the following:

Pro:

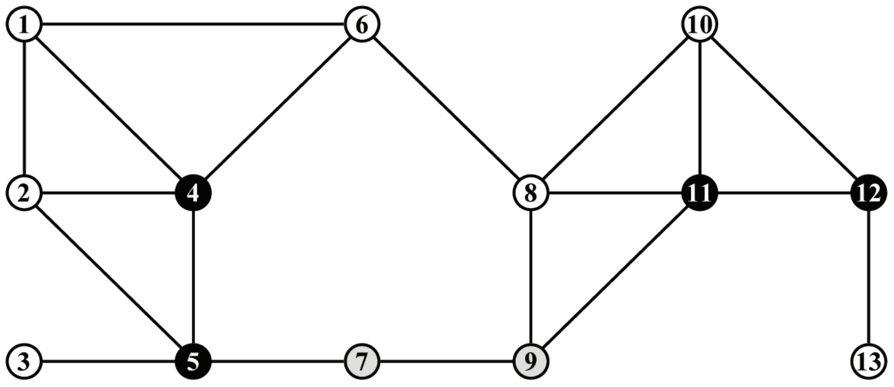


Figure 3.2: Example for Interconnecting Nodes. Should Node 7 and 9 Join the DS?

- An intermediate node has anyway to forward the whole game traffic between the Zone Servers. Whereas there are several other functions besides traffic forwarding for a Zone Server.
- In some cases synchronization can be simplified, because every Zone Server has to forward the changes of its zones only to its neighbors and those will forward it to their neighbors. Otherwise the different Zone Servers have to build a full mesh of connections to synchronize their game states. Note that this simplification can lead to higher latency, because if the chain of Zone Servers is quite long, the game state information has to travel from the first server in the chain to the last server.
- Additional redundancy is created if interconnecting nodes join the DS. Because every node has already a neighbored node that is acting as server, the neighbors of the interconnecting nodes will now have more than one server neighbor. In Fig. 3.2, node 8 is already connected to node 11 that is in the DS. If node 9 joins the DS as well, it has now 2 neighbor nodes (node 9 and 11) that are acting as server.
- If the DS is connected, it is easier to monitor the status of the other Zone Servers. Every server checks the connectivity to its neighboring Zone Servers and does not need to wait until the routing protocol reports the fail of a Zone Server node.

Contra:

- Acting as Zone Server creates some extra overhead, especially for the consistency and synchronization mechanisms. If the interconnecting node is a device with less resources this can lead to some considerable problems and slow down the whole game session.

It can be seen that the decision whether the DS should be connected or not is mainly based on how expensive the overhead of the consistency and synchronization mechanisms are if there are some additional Zone Servers. At the moment there exist no deeper investigations of the performance requirements of these mechanisms, and some simulations and deeper investigations of this question are required. It can be assumed that the cost will mainly be dependent on the complexity of the game itself. It is assumed in this thesis that it is not required to build a Connected Dominating Set, rather it is an optional possibility.

Node weighted: The algorithm should prefer to put nodes in the DS that have high weights and are therefore better suited to act as Zone Server.

Minimum number of nodes: For every graph there have to be at least two nodes in the DS, also for full connected graphs (this is different to the conditions when the DS is used for the purpose of routing where the DS is empty for full connected graphs). In Fig. 3.3, the situation is illustrated for the simple case of a three-node network. Even in this case, minimum two Zone Servers are needed, because if node 1 is leaving the network the two remaining nodes should still be able to continue playing.

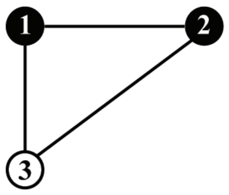


Figure 3.3: Full Connected Graph with Two Nodes in the DS

MDS approximation: With the increasing number of Zone Servers the produced overhead as needed for the synchronization increases as well. This results in lower game performance and increases, for example, the latency time. Therefore, the Dominating Set should contain as few nodes as possible and should have a quite good approximation of the MDS.

The Dominating Set should consist of nodes with only high weights as well as the DS should be the best approximation of the MDS. It is obvious that these properties are in conflict with each other. In most of the graphs it is not possible to fulfill both requirements at a time. To determine the order of the priority of these two requirements, a worst case scenario will be discussed based on a graph, where it is required to build a Connected DS. In Fig. 3.4 this graph of an ad hoc network is shown. The numbers written inside the circles represent the node IDs, and the numbers above the nodes indicate the calculated node weights. A high weight indicates high capabilities to act as Zone Server. It can be seen that node 7 with the highest degree in the graph has unfortunately the lowest node weight of the whole graph. How this case should be handled is discussed and evaluated considering three possible choices of the CDS:

In the first case, outlined in Fig. 3.5, only the requirement to build a CDS containing nodes of highest weight has been considered. That's why first the nodes 3 and 5 with weight 5 are chosen. Afterward node 1 because that's the node with the highest weight of the uncovered node. Node 4 and 6 are connecting the dominated set.
In the second case, as shown in Fig. 3.6, again a Dominating Set choosing the nodes 1, 3, and 5 with the highest weight has been chosen first. But second, for building a Connected Dominating Set the condition of using as few nodes as possible has been used, and only node 7 joins the DS as well.
In the third and last case the first priority is to get a good approximation of the MCDS and second, if there is a choice between several nodes, it takes the node with the highest weight. In Fig. 3.7 node 7 has been chosen, because it covers most other nodes, afterward node 3 joins the DS because it has the highest weight (tie broken by ID) among the remaining nodes.

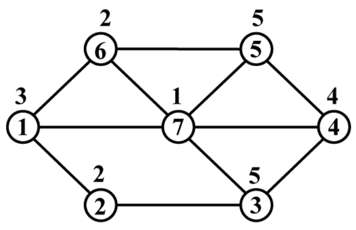


Figure 3.4: Ad Hoc Network Shown as a Graph

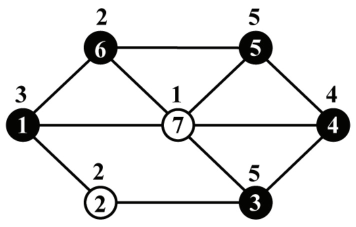


Figure 3.5: Case 1: CDS Based Only on Weights

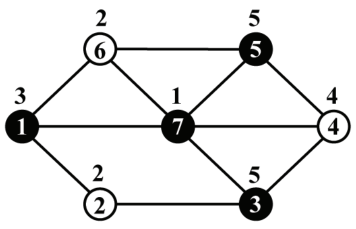


Figure 3.6: Case 2: CDS Based on Weights, then on Minimum Number of Nodes

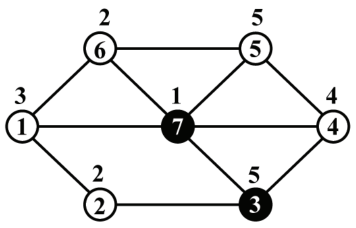


Figure 3.7: Case 3: CDS Based on Minimum Number of Nodes, then on Weights

The chosen Dominating Sets for the three different cases can be summarized as follows:

1. Only node weights (Fig. 3.5): It can be seen that the "danger" to choose node 7 in the CDS is avoided because of having a low node weight. Although due to its central position and the high degree this node is predestined as Zone Server. This is paid with having a lot of nodes in the CDS.

2. **First node weight, then minimum number of nodes** (Fig. 3.6): In this case node 7 is in the CDS as well and reduces the number of nodes. But all nodes have the possibility to connect to another node with higher weights.

3. **First minimum number of nodes, then node weight** (Fig. 3.7): A minimal CDS is selected but with the disadvantage that most of the nodes outside the CDS have to connect to the weakest node 7.

Which of these three cases is the most suited solution? If the weakest node 7 is chosen into the Dominating Set due to its central position it can seriously slow down the whole game session because of its constraint resources. On the other hand in the shown example, as already mentioned, not choosing node 7 as Zone Server is paid with having more Zone Servers that can slow down as well the performance. Considering the GSM model (as introduced in section 2.2.2) for a fixed number n of clients and constant number m of game-controlled entities, the time required to calculate a game state transition dependent on the number l of Zone Servers can be expressed as follows:

$$T_{ZS}(l,n,m) = \frac{n}{l} \cdot (t_{ca} + t_{fc}) + (n - \frac{n}{l}) \cdot t_{rsu} + \frac{m}{l} \cdot t_{se} \tag{3.1}$$

Whereas t_{ca}, t_{fc}, t_{rsu}, and t_{se} are the used times for processing the arriving messages inside the nodes. It is evident that the values of these times are determined by the weakest Zone Server. Considering the shown examples (Fig. 3.5, 3.6, and 3.7), the calculation times result into the following values (n=7; l=5,4,2; m=7 arbitrarily chosen):

$$T^1_{ZS}(5,7,7) = 1.4 \cdot (t^1_{ca} + t^1_{fc}) + 5.6 \cdot t^1_{rsu} + 1.4 \cdot t^1_{se} \tag{3.2}$$

$$T^2_{ZS}(4,7,7) = 1.75 \cdot (t^2_{ca} + t^2_{fc}) + 5.25 \cdot t^2_{rsu} + 1.75 \cdot t^2_{se} \tag{3.3}$$

$$T^3_{ZS}(2,7,7) = 3.5 \cdot (t^3_{ca} + t^3_{fc}) + 3.5 \cdot t^3_{rsu} + 3.5 \cdot t^3_{se} \tag{3.4}$$

The message processing times in case 2 and 3 are equal because in both cases they are determined by node 7, in case 1 they are determined by node 6. Because node 7 has a lower weight than node 6, it can be assumed that its processing times will be proportionally higher. For $x > 1$ it can be stated that $t^2_{ca} = x \cdot t^1_{ca}$, $t^2_{fc} = x \cdot t^1_{fc}$, $t^2_{rsu} = x \cdot t^1_{rsu}$, and $t^2_{se} = x \cdot t^1_{se}$. It can be shown that T^2_{ZS} has better values than T^1_{ZS} only if x is less than 1.1. But it can be assumed that the node weights indicate real differences in the resources and therefore a higher value for x can be expected and $T^1_{ZS} < T^2_{ZS}$.

With focus on the calculation time for a game state transition, the first case is the better solution than case two and three. The disadvantage of the first case is that due to the high number of nodes a higher game latency can result, because it takes time to distribute the game state data to all servers. But this depends on the used synchronization mechanism and the complexity of the game and on the available bandwidth and delay of the edges. On the other hand, due to its restraint resources it is not guaranteed that the synchronization mechanism is faster if node 7 is part of the DS.

It can be concluded that for the network shown in Fig. 3.4 it is assumed that case 1 as outlined in Fig. 3.5 shows the best selection of Zone Servers. And the situation is still improved, if it is not required to build a Connected DS. Because then, only nodes 1,3, and 5 will join the DS in the first case, and this is only one additional server compared to the ideal situation from the view of the MDS approximation in the third case. Therefore choosing nodes with high weights into the DS will be prioritized over the requirement to get a good MDS approximation.

3.1.3 Requirements for Zone Server Selection Algorithm

Completely Distributed: The algorithm has to run locally on every node in a distributed manner because there is no pre-established infrastructure in ad hoc networks that can be used for central administration.

Time Complexity: The time complexity is measured in rounds consisting of sending messages, receiving messages, and performing some local computations. It is evident that the number of rounds that the algorithm needs to determine the DS should be kept minimal. Especially if the algorithm needs to be recalculated during the game, the interruption of the game should not be noticeable to the players.

Message Complexity: The message complexity is measured in the number of messages that have to be sent per round and the size of the message. In order to keep the time complexity low, the number of the messages and their size should be kept low as well.

Mobility Maintenance: Due to the possible host's movements there has to exist some *update* and *recalculation* mechanisms for modifying the DS according to the topological changes that occur. Note, that an update mechanism should only change the status of some individual hosts in contrast to a recalculation mechanism that recalculates the status of all nodes in the entire network. It is obvious that the DS should be recalculated as seldom as possible, because moving from one Zone Server to another produces a lot of overhead. There exist 3 types of ”movement” that cause topological changes in an ad hoc network: A host joining the game, a host leaving the game, or a host that moves around. These three cases can be combined in one problem case: There are links (respectively edges from the view point of a graph) going down and up between the different nodes in the network. These topological changes can be reported by the routing protocol that underlies to the framework. Based on these reports the concerning update or recalculation algorithms can be started.

Zone Splitting (optional): If the resources of a Zone Server are running out and the server gets overloaded, it should be possible to split the zone and to determine a new additional Zone Server. An example is shown in Fig. 3.8 and 3.9.

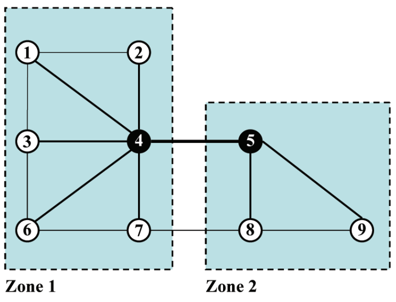


Figure 3.8: Zone Servers Before the Splitting

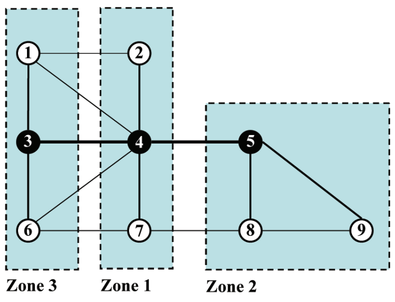


Figure 3.9: Zone Servers After the Splitting

It is evident that based on the graph structure it is not always possible to determine a new Zone Server that keeps the DS connected. Maybe the structure of sub zones within a zone has to be investigated.

3.1.4 Summary

The following requirements for the Zone Server Selection will be considered:

Ad Hoc Network: It is assumed that there is a connected ad hoc network and a routing protocol running for the communication between the nodes.

Nodes: Every node needs a unique ID and a node weight. The ID is used as tie breaker in the Zone Server selection procedure and the weight indicates the node’s capability to act as Zone Server. Furthermore, every node must be willing to contribute to the service and act as server, even if it is not participating in this specific service.

Dominating Set: There is a minimum number of 2 nodes that should be in the DS for every ad hoc network. The DS should consist of nodes with high weights as first priority. If there are several nodes with the same weight, those nodes should be chosen that give a good MCDS Approximation. As final tie breaker the node's unique ID can be used. It is assumed, that there is no gain in building a Connected Dominating Set. It could be shown that the decision whether the DS should be connected or not is mainly based on how expensive the overhead of the consistency and synchronization mechanisms are. Therefore, it is an optional requirement to extend a built Dominating Set to a Connected Dominating Set.

Algorithm: Because there is no central infrastructure that can be used, the algorithm has to run locally on every node. The time complexity (i.e. the number of rounds that the algorithm needs to determine the DS) should be kept minimal as well as the message complexity for saving resources. The algorithm should provide a mobility maintenance and being able to react to links that are going down or up. There are some optional requirements that can be considered like zone splitting if a Zone Server becomes overloaded.

Table 3.1 shows a collected overview of the requirements:

Ad hoc Network	Not fragmented net
	Routing protocol
Nodes	Unique ID
	Node weight
	Co-operative behavior
Dominating Set	Minimum 2 nodes
	Node weighted
	Tie breaker 1: MDS approximation
	Tie breaker 2: unique ID
	Connected (optional)
Algorithm	Completely Distributed
	Low time complexity
	Low message complexity
	Update mechanism for mobile maintenance
	Zone splitting (optional)

Table 3.1: Zone Server Selection Requirements

3.2 Priority Based Selection (PBS) algorithm

In section 2.3, a lot of existing distributed algorithms that choose a Dominating Set in a given network have been presented. Nearly all of them have been developed for the purpose of providing routing functionality in ad hoc networks. Providing functionality for a Zone-based Architecture and the required selection of Zone Servers is faced with partly different requirements as described in section 3.1. Therefore, a new modified algorithm is required. In this section, the Priority Based Selection (PBS) algorithm is presented for choosing the Zone Servers in an ad hoc network that fulfills these requirements. The section is structured into the following subsections:

- The section *3.2.1 Notations and Prerequisites* introduces the used notations in the algorithm and the needed prerequisites.
- The section *3.2.2 Dominating Set Computation* presents a Zone Server Selection algorithm based on comparing priorities.
- The section *3.2.3 Extensions* presents an enhancement to the PBS algorithm that constructs a Connected Dominating Set.
- The section *3.2.4 Examples* shows some examples of the PBS algorithm.
- The section *3.2.5 Performance Analysis* analyzes the algorithm and checks whether all requirements have been considered.

3.2.1 Notations and Prerequisites

In the algorithm, the following notations and definitions will be used:

Status: A node can be in one of the following four states:

- DOMINATOR
  The node is in the DS and will act as Zone Server.
- DOMINATEE
  The node is not in the DS but is covered by one or more DOMINATOR nodes (it has at least one DOMINATOR neighbor).
- INT_CANDIDATE
  The node participates in the game session and still does not have a DOMINATOR or DOMINATEE status but it is an internal candidate to become one of them.

- EXT_CANDIDATE

  The node does not participate in the game but based on the co-operative behavior assumption (see section 3.1.1) it is possible that the algorithm chooses the node as DOMINATOR in the DS. Thus, the node can be considered as an external candidate.

In the beginning, every node will have the status of an INT_CANDIDATE or EXT_CANDIDATE node. The task of the algorithm is to put at least every INT_CANDIDATE node of the graph into DOMINATOR or DOMINATEE status. In the figures shown in this chapter, INT_CANDIDATEE nodes are white, DOMINATEE nodes gray, and DOMINATOR nodes black colored. EXT_CANDIDATE nodes will have gray dashed circles.

Span: The $span(v)$ of a node v is the number of INT_CANDIDATE neighbors (including itself).

Fully connected node: If a node has a link to all other nodes in the network, the node is fully connected.

Neighborlist: Every node is tracking a neighborlist that contains all relevant information about its neighbors. For every neighbor node the following information will be stored in this list about this node:

- ID

  The unique ID of the neighbor node.

- Address

  The network address of the neighbor node.

- Node weight

  The weight of the neighbor node.

- Span

  The span value of the neighbor node.

- Status

  The status of the neighbor node.

- Fullconnected

  A flag used to indicate that a node is fully connected and a further DOMINATOR node is required.

In the beginning all values of these fields are unknown except the address field.

Coverage: All the nodes that are directly connected to a DOMINATOR node are covered by this node. Every DOMINATEE node is covered by minimum one DOMINATOR node.

3.2.2 Dominating Set Computation

Building a Dominating Set using the PBS algorithm is based on comparing priorities. A node has a **higher priority** than another node if

1. the node has a higher node weight;
2. if tie: the node has a higher span value;
3. if tie: the node has more neighbors with DOMINATOR status;
4. if tie: the node has a lower ID.

As described in chapter 3.1, the node weight indicates the node's capability to act as Zone Server and having high weighted nodes in the DS has the highest preference. Thus, this is the most important criterion to determine the priority of a node. The span value is an indication of how many nodes will lose their INT_CANDIDATE status if this node becomes a DOMINATOR and is used as the second criterion to compare the priority of two nodes. This criterion leads to a better MDS approximation. Note that if all nodes have the same node weights and their priority decisions are based on the span value, the algorithm will be quite similar to the greedy algorithms presented in section 2.3.4. If no decision can be made based on the span values the number of DOMINATOR neighbors will be considered. It is useful to have as less hops as possible between the DOMINATOR nodes due to the consistency and synchronization mechanisms required by the applications. The number of DOMINATOR neighbors of a node can be calculated based on the received neighborlist. The final tie breaker for comparing priorities of different nodes is the node ID which is unique within the network. A node with lower node ID has higher priority.

The choice of criteria and their order to compare the priorities of different nodes are based on the discussions from chapter 3.1. It is easy to see, that they are the most influencing factors of the quality and property of the DS built by the PBS algorithm. One advantage of the algorithm is that the Dominating Set can be influenced by purely changing these criteria and their order. For example, if it is important to get a Connected DS fast the criterion about the number of DOMINATOR neighbors (criterion 3) can be put at the top, and the algorithm constructs a CDS. In this case, not compulsory nodes with highest weights will be chosen into the DS.

The PBS algorithm is performed in rounds. Every round consists of three steps, such as sending

the own neighborlist to the neighbors, receiving the neighborlist from the neighbors and recalculation of the own status. Based on the exchange of the neighborlists, every node knows the network up to a distance of 2 hops and is therefore able to determine the nodes with the highest priorities. In the beginning, the rounds are performed as long as there are any INT_CANDIDATE neighbors left (including the node itself) within distance of 2 hops. Afterward, a node starts a round again, if it detects lost or new links to some neighboring nodes. Note that not all nodes perform compulsory the same amount of rounds because it is possible that a node has already no INT_CANDIDATE neighbors left whereas other nodes still have some. But the rounds are synchronized between the nodes, because a node always have to wait until all neighbor nodes sent their neighborlist. Nodes that already have no INT_CANDIDATE neighbors left will also send a neighborlist back if it receives one from a node.
Every node sets its initial state to EXT_CANDIDATE when it joins the self-hoc network. If the node wants to join the game session, it changes its status to INT_CANDIDATE and starts sending the neighborlist.

In a very general way, the algorithm for a node v can be outlined as:

```
1: set own status to INT_CANDIDATE or EXT_CANDIDATE
2: while v has INT_CANDIDATE neighbors within distance 2 do
3:    send neighborlist
4:    receive neighborlist
5:    change to DOMINATEE if a direct neighbor has status DOMINATOR
6:    change to DOMINATOR if v has highest priority within distance 2
7:            among the nodes with INT_CANDIDATE status
8: od
```

In the beginning, every node set its own status to INT_CANDIDATE or EXT_CANDIDATE depending on whether the node wants to join the game session or not, and sends in the first round the neighborlist to its neighbors. This means after the first round every node has already a two hop view of the network. Every node compares its priority with the node neighbors (based on the received neighborlists) and if it has the highest priority among all neighbors up to a distance of two hops it changes the own status to DOMINATOR. Note that there always a node exists with the highest priority among the INT_CANDIDATEE neighbors, because the last tie breaker for the priority comparison is the node with lower ID and the ID is unique within the network. A node needs to have the highest priority within a distance of two hops because it is possible, that a candidate node has two neighbors with higher priority and therefore both of them will change to DOMINATOR. An example of such a constellation is shown in Fig. 3.10 where node

1 already has been determined as DOMINATOR, and nodes 2 and 3 as DOMINATEE. Node 4 has still INT_CANDIDATE status and has node 2 and 3 as its neighbors. All nodes have the same node weights. If the nodes 2 and 3 are only looking at the direct neighborhood, both nodes will change to DOMINATOR to cover node 4 because both of them has higher priority than node 4 due to the number of neighbored DOMINATOR nodes. But if the nodes compare the priority within the distance of two hops, only node 2 will change to DOMINATOR and cover node 4 because based on the ID it has higher priority than node 3. In the end, the DS consists of node 1 and 2, which is the best possible solution for this network topology.

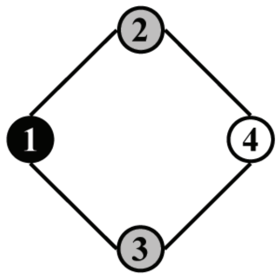


Figure 3.10: Sample Graph Where Node 1 is DOMINATOR, Nodes 2 and 3 DOMINATEE, and Node 4 still INT_CANDIDATE

But this very basic implementation of the first phase of the PBS algorithm does not fulfill all requirements. Therefore, some modifications are required that consider the requirements and that improve the time complexity of this first phase:
A neighbor node of a node that changed its status to DOMINATOR will be informed in the next round about this change and will change the own status to DOMINATEE. If the node has some common neighbors with the DOMINATOR node, it will also change immediately their status to DOMINATEE. By doing this the node gets faster the correct information and do not need to wait one round more until these nodes informed it about their change to DOMINATEE and wrong decisions can be avoided.
Different to the DOMINATOR status that can not be changed during the DS building process, a node that has status of a DOMINATEE can still switch to DOMINATOR. Because a DOMINATEE node is already connected to a DOMINATOR, and if it has the highest priority among the other INT_CANDIDATE, EXT_CANDIDATE, and DOMINATEE neighbors it is a good choice if it joins the DS.

A requirement to the algorithm is that the DS has to contain minimum two nodes regardless of the network topology due to redundancy reasons. This causes some problems especially if a node is fully connected. Consider the simple network shown in Fig. 3.11 consisting of four nodes. Nodes 1 and 2 has node weight 3 and nodes 3 and 4 node weight 2. The nodes 1 and 3 are fully connected with the other nodes.

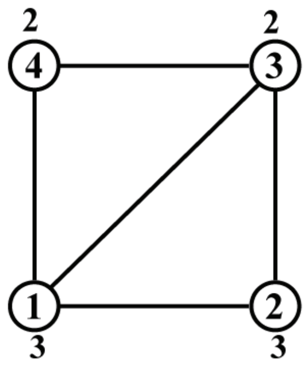


Figure 3.11: A Graph with Two Full Connected Nodes

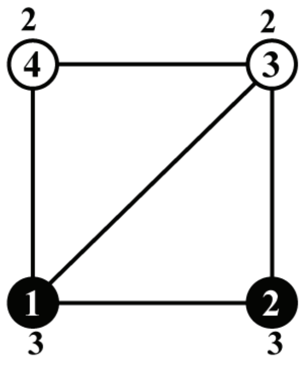


Figure 3.12: The Chosen CDS for this Graph with Two Full Connected Nodes

Based on the node weight and on the higher span value node 1 will change its status to DOMINATOR in the first round of the algorithm. In the second round, all other nodes change to DOMINATEE and without the requirement of having a minimum number of two Zone Servers, the DS could be considered as built. But caused by this requirement, the algorithm needs to be extended: It is very easy for a node to detect if it is fully connected or not by receiving the neighborlist from the neighbor nodes. If none of the neighbors has a neighbor that is not already an own neighbor, then the node is fully connected. Being such a full connected node has the big advantage to know the whole network. Therefore, if a full connected node is selected as DOMINATOR it checks all neighbors if there are any other DOMINATOR nodes. If this is not the case it determines the node with the highest priority among its neighbors and sends in the next round the neighborlist with a flag called ”fullconnected” that forces this neighbor node to join the DS as well. In the considered example of Figure 3.11, after switching to DOMINATOR in round 1, node 1 sends in round 2 node 2 the ”fullconnected” flag in the neighborlist, because it is the node with the highest priority among the neighbors of node 1. Node 2 will also switch to DOMINATOR and the DS consists of nodes 1 and 2 as shown in Fig. 3.12. The requirement of two Zone Servers is fulfilled.

With this modifications, a round of the PBS algorithm can be summarized as follows:

Step 1: Send neighborlist to all neighbors;

Step 2: Receive neighborlist from all neighbors;

Step 3: Determine status:

- If the node's status is INT_CANDIDATE:
  - change status to DOMINATEE if at least one neighbor has DOMINATOR status. Update also the status of the common neighbors of the DOMINATOR node to DOMINATEE.
  - change status to DOMINATOR if
    - the node has the highest priority among its 1-hop INT_CANDIDATE neighbors and the neighbors of these INT_CANDIDATE nodes which are two hop away from the original node;
    - a DOMINATOR neighbor reported "fullconnected".

    If the node is fully connected and changed to DOMINATOR, check the number of other DOMINATOR nodes among its neighbors. If there are not any other DOMINATOR node set the "fullconnected" flag in the neighborlist that is sent to the node with the highest priority among its 1-hop neighbors.
- If the node's status is DOMINATEE or EXT_CANDIDATE:
  - change status to DOMINATOR if
    - the node has the highest priority among its 1-hop INT_CANDIDATE neighbors and their two hop neighbors;
    - at least one DOMINATOR neighbor reported "fullconnected".

    If the node is fully connected and changed to DOMINATOR, check the number of other DOMINATOR nodes among its neighbors. If there are not any other DOMINATOR node set the "fullconnected" flag in the neighborlist that is sent to the node with the highest priority among its 1-hop neighbors.
- If the node's status is DOMINATOR:
  - change status back to DOMINATEE if there are no DOMINATEE neighbors left, but there is still another DOMINATOR neighbor.

For the better understanding, Fig. 3.13 shows the pseudo code of the algorithm and the logic is shown in Fig. 3.14. The black boxes contain the conditions for the different branches, and the gray boxes indicate the status of the node at the given point.
Due to the device's mobility and the possibility that nodes leave or new nodes join the mobile ad hoc network, the network and especially its links can not be considered as static. It is one of the real contribution of the PBS algorithm that it can react properly to links that are going

```
myStatus = INT_CANDIDATE or EXT_CANDIDATE

do until (!all_neighbors_determined)
  send_neighborlist
 receive_neighborlist

 if (myStatus==INT_CANDIDATE)

    if (isNeighbor(DOMINATOR)==true)
        myStatus=DOMINATEE
        updateCommonNeighbors()
     fi

     if(highestPriority or haveLeafNeighbor
                        or fullConnected_reported)
         myStatus=DOMINATOR
         if(fullconnected)
            v=neighborWithHighestPriority
            reportFullConnected(v)
         fi
     fi
  fi

   else if(myStatus==EXT_CANDIDATE or myStatus==DOMINATEE)
      if(highestPriority() or haveLeafNeighbor
                           or fullConnected_reported)
          myStatus=DOMINATOR
          if(fullconnected)
             v=neighborWithHighestPriority
             reportFullConnected(v)
          fi
     fi
  fi

  else if(myStatus==DOMINATOR)
     if(!stillRequired)
        myStatus=DOMINATEE
     fi
  fi
od
```

Figure 3.13: Pseudo Code of the PBS Algorithm

down or new links that are coming up. It is the task of the algorithm to rebuild the Dominating Set if there are changes detected in the network, because the interruption of the service should not be noticeable to the users of the service itself. Therefore, it is also preferable in general to keep the existing Zone Servers because the transfer of the Zone Server functionality and the data creates a lot of overhead. In general, if a node detects some changes in its neighborhood it starts a new round of the algorithm by sending a neighborlist as shown in Fig. 3.14 where the node is waiting in the Wait state until a new or lost link is detected. Similar, if a node that is in the EXT_CANDIDATE status and wants to join the game session, it will change its status to INT_CANDIDATE and starts sending out neighborlists as well. In the following, some cases for new and lost connections from the view of the nodes and their current status will be discussed:

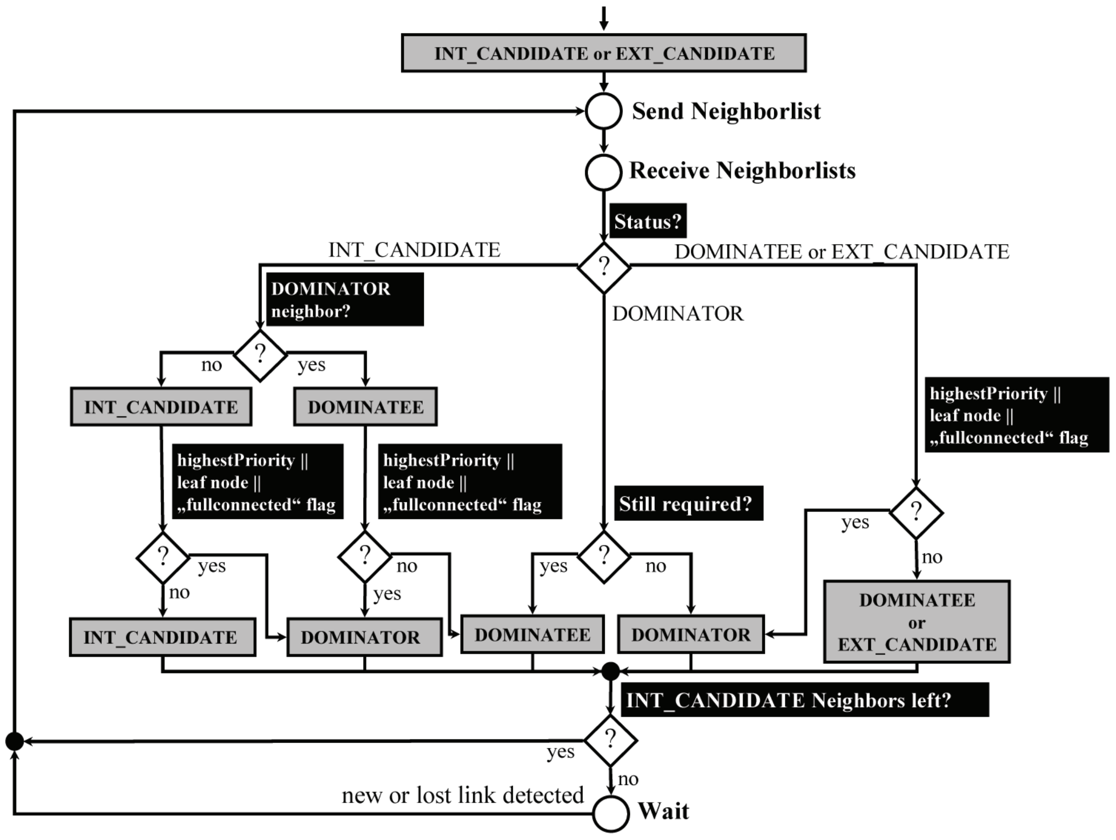

INT_CANDIDATE or EXT_CANDIDATE
Send Neighborlist
Receive Neighborlists
Status?
INT_CANDIDATE
DOMINATEE or EXT_CANDIDATE
DOMINATOR
DOMINATOR neighbor?
no
yes
INT_CANDIDATE
DOMINATEE
highestPriority || leaf node || „fullconnected“ flag
highestPriority || leaf node || „fullconnected“ flag
Still required?
highestPriority || leaf node || „fullconnected“ flag
yes
no
INT_CANDIDATE
DOMINATOR
DOMINATEE
DOMINATOR
DOMINATEE or EXT_CANDIDATE
INT_CANDIDATE Neighbors left?
yes
no
new or lost link detected
Wait

Figure 3.14: Flow Chart of the PBS Algorithm

EXT_CANDIDATE node:

If an EXT_CANDIDATE node detects some new or lost nodes, or a new INT_CANDIDATE node it simply starts a new round of the PBS algorithm to inform the other nodes about the changes.

INT_CANDIDATE node:

If a node still have INT_CANDIDATE status, the DS computation process is still going on and the node is still sending out neighborlists. The network changes will be updated in the neighborlists that are sent out.

DOMINATEE node:

After the detection of the changes in the network, the node checks if it is still connected to a DOMINATOR node. If this is not the case, it switches its status back to INT_CANDIDATE before it resends the neighborlist and starts a new round of the algorithm.

DOMINATOR node:
Note that a DOMINATOR node will remain a DOMINATOR node, even if its clients are covered by other DOMINATOR nodes. A DOMINATOR node will only switch back to DOMINATEE status if there are no DOMINATEE nodes left in its neighborhood. This is due to the shown oscillation problem in section 4.2 that can arise if the DOMINATOR nodes switch back to DOMINATEE if its connected DOMINATEE nodes are covered by other DOMINATOR nodes with a higher ID.

3.2.3 Extensions

In section 3.1.2, it has been assumed that it is not mandatory to build a Dominating Set that is connected for the support of the selection of Zone Servers. But anyway, based on the computation of a DS from the previous section and with some modification a Connected Dominating Set can be built. The following modifications are required:

Neighborlist: A new field has to be added to the exchanged neighborlists, called 'Dominator Neighbors'. This field contains the DOMINATOR nodes that are reachable within 1 or 2 hops for the neighbor node. For the DOMINATOR nodes reachable within 2 hops the intermediate node that has to be traversed for reaching the DOMINATOR node is stored as well.

DOMINATOR list: If a node has the status of a DOMINATOR, it will also store a list containing the DOMINATOR nodes that are up to three hops away in the network in addition to the Neighborlist. A DOMINATOR list contains the following entries:

- ID
  Node ID of the DOMINATOR node.
- Path
  Contains the path to this node. I.e., the IDs of the intermediate nodes (max. 2 nodes) that have to be traversed to reach the DOMINATOR node.
- Path Weight
  The weight of this path. Contains the lowest node weight that has to be traversed and the sum of the edge weights.

It is possible that there exist several paths for one DOMINATOR neighbor node. This information will be needed to build a connected DS.

DOMINATOR piece: A DOMINATOR piece is defined by DOMINATOR nodes that are directly connected. In the worst case, a piece consists of only one DOMINATOR node. In the best case, when a CDS is built there is only one piece in the whole graph because all DOMINATOR nodes are connected with each other.

It is obvious that after the computation of the DS from the previous section it is not guaranteed that the nodes of the DS are already connected. But to build a CDS of a given DS is a typical Steiner Tree problem as introduced in section 2.3.9. Based on the given graph $G(V, E)$ a CDS has to be built that includes the nodes $R \subseteq V$, whereas R contains all nodes from the DS built by the main PBS algorithm. To connect these nodes in R some of the DOMINATEE nodes that are intermediate nodes between two DOMINATOR nodes will switch to DOMINATOR status as well and build the final CDS. Note that there are at most 2 intermediate nodes between two DOMINATOR nodes. In section 2.3.9, some algorithms that solve such Steiner Tree problems have been presented. But different to the situations there, where it is assumed that every node has minimal knowledge of the network, the second phase of the PBS algorithm can be based on the information that the nodes got in the first phase of the algorithm: Every node knows its neighborhood up to a distance of two hops. In addition, every DOMINATOR node knows all its DOMINATOR neighbors that are in a distance of 1, 2, and 3 hops away based on the information from the DOMINATOR list.
The main idea of building a connected DS is that the DOMINATOR nodes compare the different possible paths to their DOMINATOR neighbors and choose the best ones that reduce the number of DOMINATOR pieces. Note that the graph and the built DS of the main PBS algorithm can be separated in common DOMINATOR neighborhoods.

A *common DOMINATOR neighborhood* is defined by a set of DOMINATOR nodes that can reach each other with a maximum distance of 3 hops. In Fig. 3.15, a graph is shown where the nodes 2, 4, 6 and 8 joined the DS in the first phase. This graph consists of two common DOMINATOR neighborhoods. The nodes 2, 4, 6 are part of the first common neighborhood, and all of these nodes can reach each other by a distance less or equal 3 hops, and the nodes 6 and 8 build the second common neighborhood. In the first step of this extension, the DOMINATOR nodes will exchange their DOMINATOR lists between the DOMINATOR nodes that are reachable within 3 hops in order to be able to detect these common DOMINATOR neighborhoods. For example, in Fig. 3.15, DOMINATOR node 4 exchanges its DOMINATOR list with nodes 2 and 6. It detects that it is part of the common DOMINATOR neighborhood consisting of nodes 2, 4, and 6. Because all of its DOMINATOR nodes in the DOMINATOR list, nodes 2 and 6, can reach each other as well within 3 hops. Different for node 6, after exchanging the DOMINATOR lists, it detects that it is part of two common DOMINATOR neighborhoods. Because node 8 is only reachable by itself and not from the nodes 2 and 4, and therefore it is part of two common DOMINATOR neighborhoods.

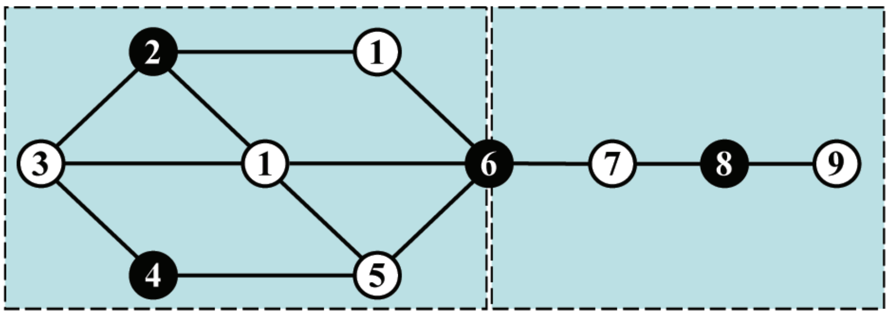


Figure 3.15: A Graph Containing Two Common DOMINATOR Neighborhoods

Every common DOMINATOR neighborhood containing k nodes has to be connected with maximum $k-1$ paths. A path is concerned as a better path than another, if

1. the **lowest node weight** along the path (there are at most two nodes) between two DOMINATOR nodes is higher;

2. if tie: the path has the **lower edge weight**.
   Note that different to the node weight where in the case of two nodes between two DOMINATOR neighbors the lower weight has to be considered, the edge weights in the case of multiple hops between two neighbors have to be summed up.

If the best paths have been determined, for every chosen path the DOMINATOR node with the lower node ID sends a ”connect” message to the first intermediate node on this path. This node will change its status to DOMINATOR and forward the ”connect” message if existing to the second node on the path. This second node will also change to DOMINATOR and discards the message.

To sum up, this enhancement of the PBS algorithm consists of the following actions for a DOMINATOR node:

- Send DOMINATOR list to all DOMINATOR neighbors that are two or three hops away

- Receive DOMINATOR lists from all DOMINATOR neighbors

- Among the DOMINATOR neighborhood where the node is part of it, determine $k - 1$ best paths that reduce the number of pieces.

- If a path is chosen where the node is attached to and it has the lower ID than the DOMINATOR node on the other end, send the ”connect” message to the first intermediate node on this path.

Or in pseudo code:

```
send_dominator_list()
receive_dominator_list()
common_neighborhoods = determine_common_neighborhoods();
for every common_neighborhoods do
    k = numOfDOMINATORS(neighborhood);
    paths = determineBestPaths(neighborhood);

    for all paths do{
        if(attached(path) && lowerID(path)){
           send_connect();
        }
    }
}
```

By exchanging the DOMINATOR lists every DOMINATOR node gets the complete view with the different edge weights inside its DOMINATOR neighborhood and is able to determine the best paths that have to be chosen. The number of $k - 1$ paths can be reduced if there are any common paths (meaning the same intermediate nodes) from several DOMINATOR neighbors to reach another DOMINATOR neighbor as shown, for example, in Fig. 3.16. DOMINATOR node 6 can only be reached from DOMINATOR nodes 1 and 3 via the nodes 4 and 5, and therefore there is only one path required to build a CDS.

Because the DOMINATOR nodes are exchanging only once their DOMINATOR lists it is possible that after building the CDS cycles occur. Consider Fig. 3.17 where the constructed DS after the main PBS algorithm is shown. After using this extension to build a Connected DS all nodes will have DOMINATOR status. Because, for example, node 1 does not know that nodes 4 and 10 are also connected via node 7, and sends a ’connect’ message to nodes 2 and 12 to force them to join the DS. Similar, node 4 sends a ’connect’ message toward nodes 5 and node

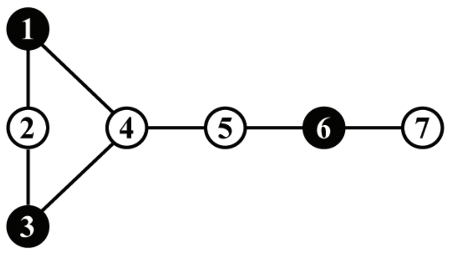


Figure 3.16: Graph Where Only One Path is Required to Build CDS

7 toward node 8. At the end all nodes are DOMINATOR nodes and there is one path too much chosen, and the CDS could consist of two nodes less. Only cycles up to 9 nodes can be detected by exchanging the neighborlists once. It is possible to extend this capability to detect cycles by exchanging the DOMINATOR lists several times. For every extra round it is possible to detect a cycle that consists of 3 more nodes. On the other hand in a cycle of DOMINATOR nodes, there are always maximum 2 nodes that are reducable in the CDS. Therefore, for longer cycles the extra needed time performance is much higher compared to the received improvement of the MCDS approximation and the disadvantage of the possibility of having cycles with more than 9 nodes in the CDS will be neglected.

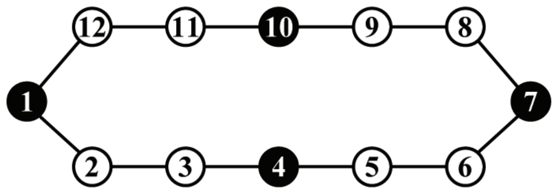


Figure 3.17: The Cycle Cannot Be Detected and All Nodes Will Switch to DOMINATOR

3.2.4 Examples

In this section two examples of the PBS algorithm are shown and should provide a deeper understanding of how this algorithm works. Also the extension of the algorithm is shown, that constructs a Connected Dominating Set.

Example 1:

In Fig. 3.18 a typical graph is shown with different node weights and different delays on the edges. The delays are only used for the extended PBS algorithm. The following actions per round take place:

Round 1: All nodes send their neighborlists to all neighbors. Based on this list node 4 changes its status to DOMINATOR because it has the highest node weight among its neighbors and therefore the highest priority. Also node 7 changes to DOMINATOR status because there are no neighbor nodes with higher priorities. Node 1 has indeed the same node weight but a lower span value. The situation after the first round is outlined in Fig. 3.19.

Round 2: Nodes 4 and 7 inform their neighbors via the sent neighborlists that they changed to DOMINATOR. Due to this message all other nodes change to DOMINATEE because all of them are covered by node 4 or 7.

Round 3: Again the neighborlists are exchanged. Based on this node 7 is informed by node 3 and 6 that they can reach DOMINATOR node 4. Node 7 stores this information in its DOMINATOR list. Node 4 is informed as well by nodes 3 and 6 that they can reach DOMINATOR node 7 and stores this information as well in its DOMINATOR list. But also the other nodes 1, 2, 5, and 6 store the information about the available DOMINATOR nodes. For example, node 2 is informed by node 1 and node 3 that they can reach DOMINATOR node 7 and DOMINATOR node 4, respectively.

Round 4: One further round where the neighborlists are exchanged. Now the DOMINATOR nodes get the information about the other DOMINATOR nodes that are reachable within a distance of 3 hops. For example, node 7 is informed by node 2 that DOMINATOR node 4 is also reachable via nodes 2 and 3.

The main algorithm is now completed (Fig. 3.20). Note that the DOMINATOR list is only used by the extension of the PBS algorithm to build a Connected DS. Due to this list the determined DOMINATOR nodes 7 and 4 know all paths how to reach each other. If required, the extension will now be started and performs some additional rounds:

Round 5: The two nodes 4 and 7 exchange their DOMINATOR lists and detect that there exists only one neighborhood containing only DOMINATOR node 4 and 7. Based on that, both nodes decide that the best path to connect them is via node 6, because it has a higher node weight than for example node 3, and also the time delays of the edges have better values.

Round 6: Because node 4 has the lower ID it sends a ’connect’ message to node 6, and node 6 changes as well to DOMINATOR status. Because this is only a 2 hop path with one intermediate node, node 6 does not have to forward the message, and phase 2 is finished.

The final CDS chosen by the extended PBS algorithm for this graph is shown in Fig. 3.21.

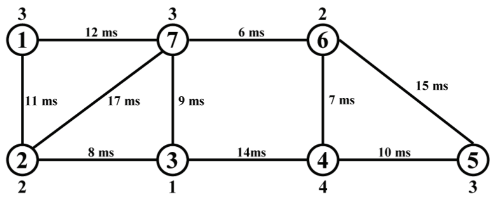


Figure 3.18: Example 1 - Graph

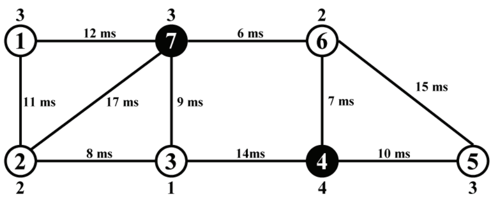


Figure 3.19: Example 1- After first round

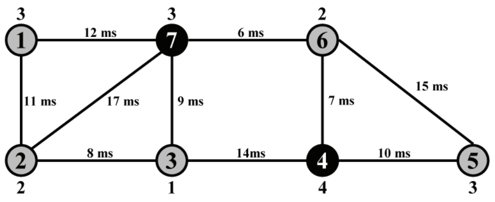


Figure 3.20: Example 1 - Chosen DS

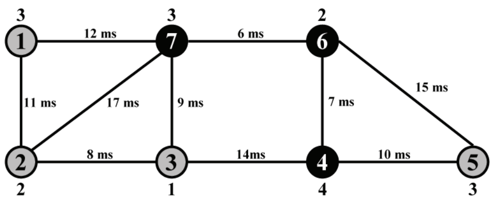


Figure 3.21: Example 1 - Chosen CDS

Example 2:

In this example, the algorithm is applied to a graph consisting of 12 nodes as shown in Fig. 3.22. All nodes have the same weight values and all edges the same time delays. Because the priorities have to be compared based on the ID (except the nodes in the corner that have a less span value), the DOMINATOR nodes have to be determined step by step. The main algorithm requires 9 rounds to determine the DS as shown in Fig. 3.23. Afterward, the DOMINATOR nodes exchange their DOMINATOR lists and the connect message is sent from node 2 via node 3 to node 4. The second phase needs 3 rounds. The final CDS that has been determined by the PBS algorithm in 12 rounds is shown in Fig. 3.24.

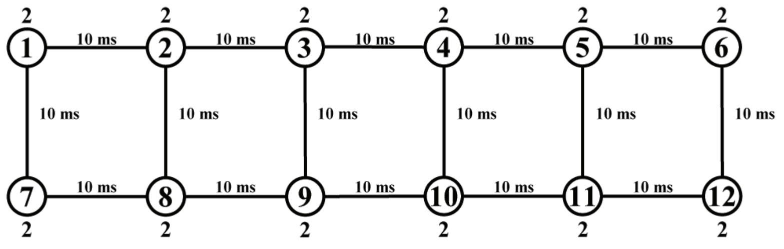


Figure 3.22: Example 2 - Graph

3.2.5 Performance Analysis

In the last section a Zone Server Selection algorithm based on comparing priorities has been presented. In this section the performance and the correctness based on the requirements from chapter 3.1 will be investigated for this PBS algorithm.

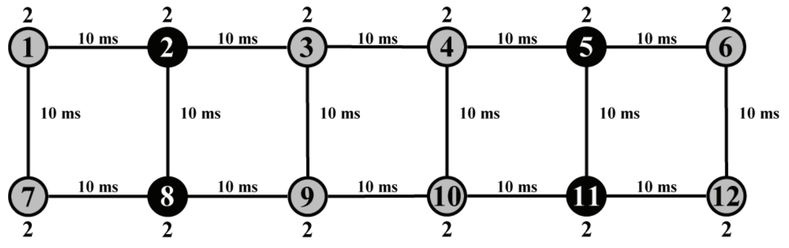


Figure 3.23: Example 2 - Chosen DS

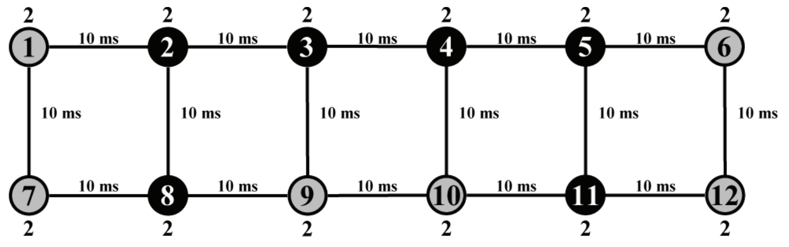


Figure 3.24: Example 2 - Chosen CDS

Node weighted:

If the algorithm compares the priorities, the first criterion to get a high priority is the weight of the node. Therefore, only nodes with high weights will be chosen into the DS.
Even by the extension, the different paths are compared according to the smallest node weight along this path. If there is a choice, the path and consequently the nodes with the higher node weights will be chosen into the DS. If there is only one path between two DOMINATOR nodes, and this path is needed to build a connected DS, the nodes along this path will switch to DOMINATOR nodes independent from their node weight. But there is no other choice due to the requirement of building a CDS.

Minimum number of DOMINATOR nodes:

It is guaranteed that there are at least two DOMINATOR nodes for any network topology (of course, the network should consist of at least two nodes). If there is a node fully connected with every other node, this node will force another node to turn to DOMINATOR status if required. If there is no fully connected node in the graph with n nodes, the maximum possible degree for a node is $n - 2$. This means, that if a node change to DOMINATOR status, there is minimum one node left that will not be covered by this node, and a second DOMINATOR node will be determined by the algorithm.

Completely distributed:

The decisions of a node to turn to DOMINATOR or DOMINATEE status are only based on the received information by exchanging the neighborlists with the direct neighbors. Thus, the algorithm is completely distributed and runs locally on every node.

Connected DS:
Due to the presented extension, it is possible to construct optionally a Connected Dominating Set. The task of the extension is to reduce the number of DOMINATOR pieces until there is only one piece left. Therefore, after the extension it is guaranteed that the DS is connected.

Mobility maintenance: The algorithm can react to links that are going down or coming up. If a node detects new or lost links it will start new rounds of the algorithm to distribute the information about the link changes and to determine the status of possible new INT_CANDIDATE nodes.

The task of the performance analysis is to give a lower and upper bound for the time and message complexity of the PBS algorithm. This means that for both of them a worst case scenario as well as a best case scenario will be investigated.

Good MDS approximation:
The approximation factor of the PBS algorithm is strongly influenced by the distribution of the node weights. In the worst case, the nodes with the lowest span values will have the highest node weights deteriorating the approximation factor. If all nodes have the same weight or the nodes with the highest span values have also the highest weights, the span value will determine the priority between the different nodes. This is similar to the greedy algorithms [21] and the approximation factor at the beginning of the algorithm is $log(\Delta)$. During the game session due to node mobility the approximation factor can change and become worse, because the algorithm tries to keep the existing DS even if there could be a better MDS approximation achieved. However, we assume that changing Zone Servers and the required transfer of the game states are more expensive than having some additional servers.

Low time complexity:
The required number of rounds to determine the DS can be rapidly reduced if as many nodes as possible can change from INT_CANDIDATE status to either DOMINATOR or DOMINATEE status per round. However, there are some worst case scenarios where the higher priority can

only be determined by comparing the node IDs. This means that the DOMINATOR nodes have to be chosen step by step. For example, in Fig. 3.25 a fragment of a graph is shown, where all nodes have the same node weights and all edges are identical. At the beginning, all nodes have INT_CANDIDATE status. The white numbers in the black boxes show the current span values of the nodes. In this case, there is a chain of nodes whose priorities can be compared only based on the unique node IDs.

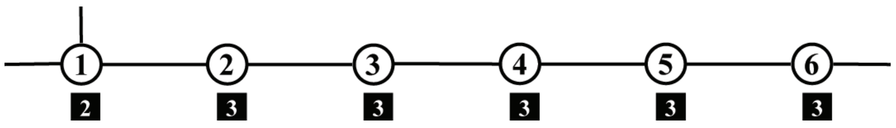


Figure 3.25: Worst Case Scenario Concerning Time Complexity

In the first round, node 2 declares itself as a DOMINATOR because it has the lowest node ID among the nodes with span value 3. In the second round, node 1 and 3 change to DOMINATEE. In the third round, node 4 gets the information about the new status of node 3. Node 5 recognizes the situation as shown in Fig. 3.26 only in the fourth round and changes itself to DOMINATOR because it has the lowest node ID among the remaining nodes with span value 3.

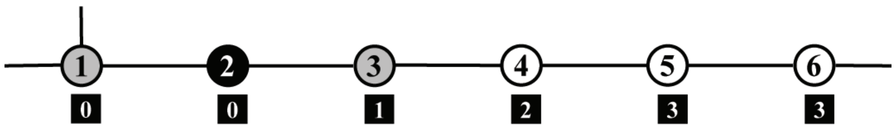


Figure 3.26: Situation in the Worst Case Scenario After 3 Rounds Applying the PBS Algorithm

This means that before node 5 as the next DOMINATOR can be determined 3 rounds have been required. During this time 3 nodes changed their status from INT_CANDIDATE to either DOMINATOR or DOMINATEE. Therefore, in the worst case when a graph consists only of such fragments shown in Fig. 3.25, the PBS algorithm can be upper bounded with $O(n)$ rounds. In general, if

$$DEG = \{deg(v_1), ..., deg(v_n)\} \tag{3.5}$$

is the set of the different node degrees in a graph, and

$$\delta = min(DEG \backslash \{deg(v_i)|deg(v_i) = 1\}) \tag{3.6}$$

the minimum degree (excluding degree 1 for leaf nodes), and it is assumed that in the worst case

$$\forall v_i \in V, deg(v_i) = \delta \tag{3.7}$$

then it is possible to change $\frac{\delta+1}{3}$ nodes from the INT_CANDIDATE status to either DOMINATOR or DOMINATEE in one round. Therefore, in total

$$\frac{n}{\frac{\delta+1}{3}} = \frac{3}{\delta+1}n \tag{3.8}$$

rounds are needed for determining all nodes and the time complexity can be upper bounded with $O(n)$.

The required time for building the DS is influenced by the way the node weights are assigned and the span values that the nodes with high weights have. In the best case, already in the first round DOMINATOR nodes are determined that cover all remaining nodes. Then the PBS algorithm requires only 3 rounds to determine the status of all the nodes: In the first round, the nodes exchange their neighborlists and some of the nodes change to DOMINATOR. In the second round, the remaining nodes change to DOMINATEE and finally in the third round every node recognizes that all nodes changed to either DOMINATEE or DOMINATOR. Therefore, the time complexity can be lower bounded with $\Omega(3)$. For the extension of the algorithm that constructs a Connected DS, the number of rounds can be lower bounded by $\Omega(4)$. There is one round more required, because the Dominator lists need to be exchanged as well. The upper bound is $O(n)$, as well.

Low message complexity:

In general, every node sends a neighborlist to its neighbors and every node can have maximum Δ neighbors. Therefore, per round Δn messages will be sent, and the message complexity can be upper bounded with $O(\Delta n)$. Note that this bound can be drastically reduced, if the different nodes are connected via a broadcast medium like in case of WLAN. In such a medium, all nodes that are in the transmission range of a certain node can receive all messages from this node. Because a node sends the same neighborlists to all its neighbors, it is enough to send the message once to the broadcast address with the TTL (Time To Live) field set to one. For a broadcast medium the message complexity can be upper bounded with $O(n)$ messages that need to be sent per round. In general, only the nodes that detected new or lost links and their neighbors up to a distance of 2 hops will exchange neighborlists. The upper bounds will only be achieved at the beginning of the algorithm and if all nodes need to update their neighborlists due to mobility maintenance.

The size of a message is determined by the number of entries in the neighborlist. A neighborlist contains for all neighbors and the node itself an entry and its size can be upper bounded with $O(k(\Delta + 1))$ bytes, whereas k represents the size of an entry in the neighborlist in bytes. The lower bound of a message size is $\Omega(k(\delta + 1))$. These bounds can be decreased as well, if only incremental updates are sent instead of sending always the full neighborlist.

3.2.6 Summary

In this chapter a distributed Dominating Set computation algorithm called PBS (Priority Based Selection) has been presented. This algorithm provides service management support in self-hoc networks for real-time applications. As the first approach, PBS offers continuous maintenance of the computed DS even if the network graph changes dynamically. Though the development of the PBS algorithm was mainly motivated by multiplayer games supporting the zone-based architecture, the algorithm can even be used in other real-time applications, such as collaborative working, which have similar requirements to multiplayer games.

The Priority Based Selection (PBS) algorithm constructs a Dominating Set based on comparing priorities of different nodes. A node has a higher priority than another node if

1. the node has a higher node weight;
2. if tie: the node has a higher span value;
3. if tie: the node has more neighbors with DOMINATOR status;
4. if tie: the node has a lower ID.

During the algorithm a node can have one of the following states: INT_CANDIDATE, EXT_CANDIDATE, DOMINATOR, or DOMINATEE. In the beginning every node has the status of an INT_CANDIDATE if it wants to join the game session, or EXT_CANDIDATE if it will not participate in the game session. During the DS computing process all INT_CANDIDATE nodes change their status to either DOMINATOR (the node is in the DS and will act as Zone Server) or DOMINATEE (the node is not in the DS but is covered by a DOMINATOR node). Based on the requirement of co-operative behavior (see section 3.1.1) it is also possible that a node with EXT_CANDIDATE status will join the DS and change its status to DOMINATOR.

The main idea of the algorithm is that every node exchanges the information about its neighbors in neighborlists between its neighbors and gets a 2-hop view of the network. Based on these received information every node is able to determine its own status. The algorithm performs in rounds. Every round consists of three steps, such as sending the own neighborlist, receiving the neighborlist from the neighbors and recalculation of the own status. A pseudo code implementation and a flow chart of the algorithm can be seen in Fig. 3.13 and Fig. 3.14, respectively. In addition, an extension to the PBS algorithm has been presented that guarantees that the constructed Dominating Set will be connected.

It could be shown, that the PBS algorithm fulfills all requirements from chapter 3.1 that are needed to support the Zone-based Architecture. The performance calculations of the algorithm are shown in Table 3.2. The MDS approximation factor is only listed for the case if all nodes have the same node weights. Otherwise, the upper bound is dependent on the current distribution of the node weight and the movement of the nodes.There exist two upper bounds for the message complexity. One if the node generates for every neighbor an own message, and one if a broadcast medium is used, and the message is sent only once to the broadcast address. The parameter k used by the Message Size represents the size of an entry in the neighborlist in bytes. The values shown in the table can be compared to the Table 2.10 and 2.11 that contain the same evaluation factors for the existing DS calculation algorithm.

Evaluation	**Upper bound**	**Lower bound**
Connected	Optional	
Approximation	$O(\log n)$ if equal node weights	
Rounds	$O(n)$	$\Omega(3)$
	with Extension:	
	$O(n)$	$\Omega(4)$
Messages	$O(\Delta n)$	$\Omega(n)$
	Broadcast Medium: $O(n)$	
Message Size	$O(k(\Delta + 1))$	$\Omega(k(\delta + 1))$

Table 3.2: Performance Results of PBS Algorithm

Chapter 4

Simulations and Evaluation

The testing and evaluation of protocols and applications for mobile ad hoc networks in a real environment can be rather costly and complex, especially if large networks are considered. Therefore, simulation is an important tool to improve or validate an implementation. In this chapter, the performance of the PBS algorithm will be evaluated based on simulation results. For these simulations the PBS algorithm has been implemented in the Network Simulator 2 (NS-2) [10]. NS-2 is a discrete event driven simulator to support networking research. Its target is for designing new network protocols, comparing different protocols and traffic evaluations. NS-2 follows closely the OSI model and provides substantial support for simulation of TCP, routing and multicast protocols over wired and wireless networks.

The implementation of the algorithm in the NS-2 simulator is shown in appendix A. In the first section of this chapter, the used simulation settings and scenarios are described, and in the second section, the results of the simulations are shown.

4.1 Simulation Settings

For the simulation with the NS-2 simulator, the PBS implementation as described in appendix A has been used, and the wireless extensions developed at CMU [30]. Throughout the simulations, each mobile node shares a 2 Mbps radio channel with its neighboring nodes using a two-ray ground reflection model [31] and IEEE 802.11 MAC [32] protocol. Three different scenarios have been simulated that have been run twice. First with 15 nodes, where 10 nodes are participating in a game session, and second with 35 nodes where 25 of them are participating in a game in average. The following scenarios have been used in the simulations:

School Yard Scenario:

The School Yard scenario can be characterized by a group of people that are standing on a school yard of 400x400 m^2 and are gaming a multiplayer game together. Because in general, there are no huge obstacles on a school yard the transmission range of a device is assumed as 250 meters, which is a typical value for WLAN in a free area. Due to this wide transmission range, most of the players will have 1 hop connections to each other. The movements of the nodes are simulated with Random Way Point (RWP) models [30] in which the non-participating nodes will move freely on the whole area of the school yard between different destination points, since the distance between two destination points of the participating nodes is uniformly distributed in the range of 0-15 meters. It is assumed, that people that are in a current game session will only slightly move, because it is quite difficult to move and play at the same time for most of the existing games nowadays. The speed of the nodes is uniformly distributed in the range of 0-6 km/h.

Train Scenario:

In the second scenario, a train is assumed where some passengers are playing with each other. For the simulation with 15 nodes the geometrical dimensions are 240x5 m^2 (about 8 wagons), and 450x5 m^2 (about 15 wagons) for 35 nodes. Due to the narrow but long area and the assumption that the disjunction between the wagons will reduce the transmission range of the devices (it is assumed as 40 meters), most of the nodes will be connected by several intermediate hops between each others. Because every playing passenger is expected to sit most of the time during a game session, we use the probability value of 50% that they will move around. This movement can be characterized by moving towards a destination point like the toilet or restaurant wagon, spending some time there and moving back to the seat. Non-participating nodes will move around more and will not implicitly move back to the original starting-point. The speed of the nodes are again uniformly distributed in the range of 0-6 km/h.

Test Scenario:

For testing the robustness and reliability of the PBS algorithm we have used a testing scenario that faces the algorithm with some more challenging conditions than in the previous scenarios. This scenario has the geometric dimension of 400x400 m^2 for the simulation with 15 nodes and 800x800 m^2 for 35 nodes, respectively. All the nodes are moving around with uniformly distributed speeds in the range of 0-30 km/h using again the RWP models. The increased speed leads to much more mobility of the nodes and causes the algorithm to recalculate the Dominating Set more frequently.

In every scenario 10 game sessions have been simulated with a duration of 900 seconds using different seed values and then averaged the results. The game joining and leaving points in time of the players were randomly distributed during the simulations. To simulate a real environment some traffic between the nodes has been generated, as well. For every client participating in the game session a constant bitrate data flow (20 packets/second with 64 byte packet size) simulating the game traffic [8] has been used. Some background traffic with the following parameters has also been added: In the scenarios with 15 nodes, the background traffic was generated by 5 parallel connections being active at the same time during the whole simulation between any two random nodes. Per connection, the sender produced 10 packets/second with 64 byte packet size for 30 seconds, then a new connection was established. In the scenarios with 35 nodes, the number of parallel connections being active at the same time has been increased to 15. Moreover, the node weights have been set randomly using uniform distribution between 0 and 99, and 9 bytes have been used to describe a node in the PBS neighborlist packet. The simulation settings are summarized in Table 4.1.

Scenarios	School Yard, Train, Test
Number of nodes	35 (with 25 participants), 15 (with 10 participants)
Node weight	Uniform distribution between 0-99
Number of game sessions	10
Duration of game sessions	900 s
Game joining and leavig points	Randomly distributed
Game traffic	Constant bitrate: 20 packets/s of 64 bytes
Background traffic	Constant bitrate: 10 packets/s of 64 bytes
	5 (15 nodes) and 15 (35 nodes) parallel connections
Size of neighborlist entry	9 bytes

Table 4.1: Simulation Settings

4.2 Simulation Results

To evaluate the performance of the PBS algorithm, the time the PBS algorithm requires to compute and maintain a Dominating Set of the network graph and the signaling traffic generated during this time have been investigated. The following metrics have been used:

Bandwidth: It indicates the percentage of the used bandwidth by the PBS algorithm comparing to the total available bandwidth (2 Mbps in these simulations) from the viewpoint of a node. The used bandwidth is strongly influenced by the size of the sent messages and will increase as more neighbors a node has. To avoid counting some messages several times, only the sent messages have been considered. The results of the bandwidth measurements are shown in Table 4.2 (we calculated the minimum, maximum, average and standard deviation values based on the simulation traces). The traffic characteristics of the nodes contain some peaks, because the nodes only start exchanging neighborlists, if some changes in the network topology have been detected and no messages will be sent if nothing changes in the neighborhood up to a distance of 2 hops. In Fig. 4.1 the amount of sent data of a single node during a game session is shown. It can be seen that at the beginning of the game session a lot of data will be sent, but once the DS is built, the peak behavior can be detected. There are some time periods where no data is sent, because there are no changes in the network's topology. From the table it can be seen that in the School Yard scenario the required bandwidth is increasing from 0.020% to 0.661% on average if the number of nodes increases from 15 to 35 nodes. The reason is, as we already mentioned, that there are more 1 hop neighbors in the second case. In the Test scenario with 35 nodes the simulation area has been expanded from 400x400 m^2 to 800x800 m^2 and the required bandwidth has decreased to 0.091% comparing with the School Yard scenario because there are less 1 hop connections. Moreover, using higher speed wireless interfaces in the self-hoc network (e.g., 11 or 54 Mbps according to the IEEE 802.11x standards [32]) the required bandwidth by the PBS algorithm won't cause any serious problems.

Determination Delay: This delay indicates the time that is needed until a node gets a neighboring DOMINATOR node or becomes itself a DOMINATOR. This happens at the beginning of a game session, or during the game when the connection to a DOMINATOR node gets lost due to the node's mobility and a new DOMINATOR node needs to be selected in order to rebuild

Scenario	min	max	avg	σ
School Yard w/ 15 nodes	0.005	0.040	0.020	0.008
Train w/15 nodes	0.001	0.015	0.006	0.003
Test w/15 nodes	0.008	0.086	0.045	0.015
School Yard w/35 nodes	0.138	1.006	0.661	0.180
Train w/35 nodes	0.001	0.049	0.021	0.011
Test w/35 nodes	0.004	0.268	0.091	0.053

Table 4.2: Used Bandwidth [%]

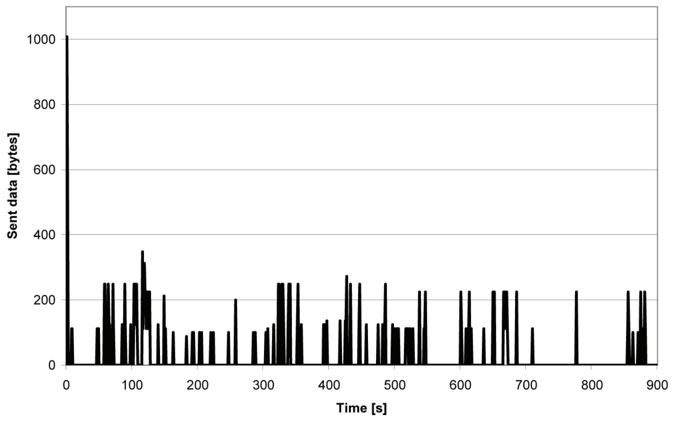


Figure 4.1: Sent Data of a Node During a Game Session

the Dominating Set. The results of the determination delay measurements are shown in Table 4.3. We can see that in all the three scenarios with 15 nodes, the average delay is in the same order between 0.111 and 0.185 seconds. However, the maximum delay in the Train scenario is much higher (1.039 seconds) than in the other two scenarios. This is, because the narrow but long area in a train creates a situation similar to the worst case scenario as shown in Fig. 3.25 of section 3.2.5. This ‘chain’ topology can lead to situations where the DOMINATOR nodes need to be determined step by step. With the increasing number of nodes, the determination delay will increase as well. In the scenarios with 35 nodes, there are considerable differences in the average values of the determination delay that are caused by the different number of 1 hop neighbors, but the maximum values are all in the same order.

Number of Required Changes: This metric indicates from the view of a node how often it looses the connection to a neighboring DOMINATOR node and a new node has to be determined as DOMINATOR. Obviously, this number should be as small as possible. As we can see

Scenario	min	max	avg	σ
School Yard w/ 15 nodes	0.018	0.690	0.159	0.172
Train w/15 nodes	0.007	1.039	0.111	0.178
Test w/15 nodes	0.021	0.674	0.185	0.188
School Yard w/35 nodes	0.003	1.358	0.510	0.311
Train w/35 nodes	0.004	1.175	0.153	0.193
Test w/35 nodes	0.012	1.130	0.283	0.251

Table 4.3: Determination Delay [sec]

observing the determination delay it can take more than one second until a new DOMINATOR is determined. During this time the client needs to connect to a server that is more than 1 hop away if this is possible. In Table 4.4, we indicated how often a node had lost the connection to its DOMINATOR node in the scenarios being used. We can notice that the average values are between 0.1 and 0.4. This means that most of the nodes will have constant connection to a neighboring DOMINATOR node during the whole game session. In the Train scenarios, however, a node can loose its DOMINATOR node more frequently because of the narrow but long geometrical shape of the train and the limited transmission range if the DOMINATOR node starts moving around. Clearly, the higher level of mobility also has essential influence on the number of required changes. For example, in the Test scenarios this number is relatively high because the topology can change much faster and the Dominating Set needs to be recomputed more frequently. An interesting observation is that there are nodes in all scenarios never losing their DOMINATOR nodes.

Scenario	min	max	avg	σ
School Yard w/ 15 nodes	0	1	0.1	0.2
Train w/15 nodes	0	2	0.2	0.4
Test w/15 nodes	0	2	0.4	0.7
School Yard w/35 nodes	0	2	0.1	0.4
Train w/35 nodes	0	3	0.2	0.6
Test w/35 nodes	0	4	0.2	0.7

Table 4.4: Number of Required Changes

Number of DS Changes: This metric is used to give an indication about the stability of the DS. It shows how often the initial Dominating Set needs to be changed during a game session from a global viewpoint. Note that in our implementation, a DOMINATOR node will remain a DOMINATOR node, even if its clients are covered by other DOMINATOR nodes. A DOMINATOR node will only switch back to DOMINATEE status if there are no DOMINATEE nodes left in its neighborhood. We choose this solution, because it can lead to an oscillation problem if a DOMINATOR node having the lowest priority switches back immediately when it detects other DOMINATOR nodes also covering its DOMINATEE neighbors. In Fig. 4.2, the number of DOMINATOR nodes during the simulated time in the School Yard scenario with 35 nodes is shown when the DOMINATOR nodes don't switch back to DOMINATEE status at all. We can see, that after building the initial Dominating Set consisting of 2 nodes, 4 further nodes were added to this set until the end of the simulation which lead to 4 changes in the DS during the whole session. In Fig. 4.3, the same situation is outlined if the DOMINATOR nodes immediately turn to DOMINATEE status when they detect that their clients are covered by other DOMINATOR nodes. This causes a frequent oscillation in the number of DS nodes massively increasing the number of DS changes. The oscillation can be alleviated if a DOMINATOR node waits a given amount of time than checks again whether it is still not required. Fig. 4.4 depicts the situation when a DOMINATOR node waits 10 seconds before switching back to DOMINATEE status. Concerning the approximation factor of the DS, the results shown in Fig. 4.3 and 4.4 are better than we can see in Fig. 4.2 because the number of DOMINATOR nodes in average is less decreasing the synchronization complexity for the application. On the other hand, the various changes force the clients to switch between servers more frequently which causes a lot of overhead. Therefore, we selected and used in all the scenarios presented in this thesis the solution when a DOMINATOR node never switches back to DOMINATEE status. To avoid the oscillation problem but still get a good approximation of MDS a possible improvement can be the use of some node movement prediction. In this case, the node movement also could influence the node weights forcing only those nodes to join the DS with a high probability that they will not move away. This would guarantee a quite stable DS.

In Table 4.5, the number of DS changes in all the different scenarios is summarized. With the increasing number of nodes and the higher mobility level in the Test scenario the required changes are also increasing.

Number of DOMINATOR Nodes: This metric indicates the number of DOMINATOR nodes

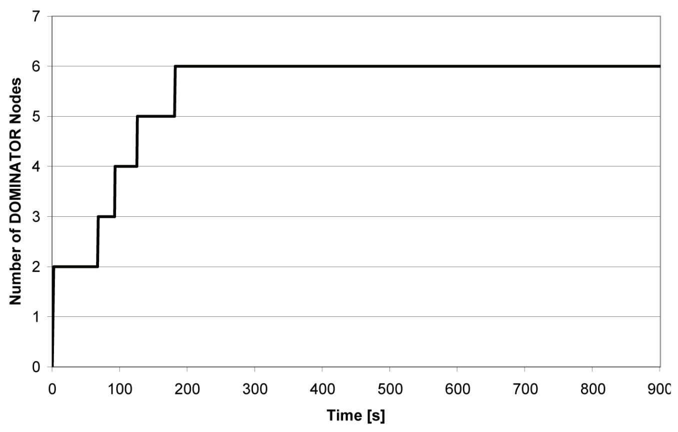


Figure 4.2: Number of DOMINATOR Nodes if a DOMINATOR Doesn't Switch Back to DOMINATEE Status

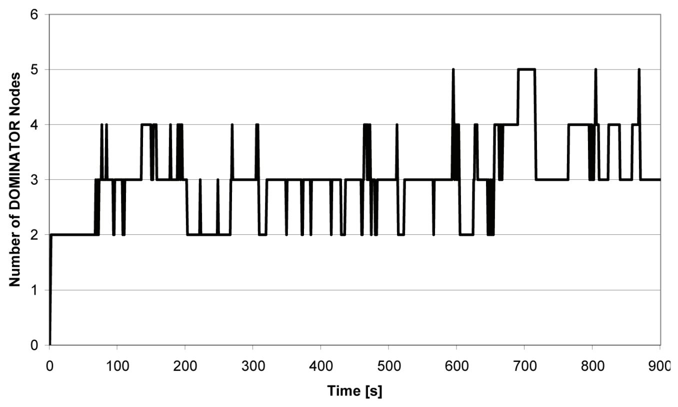


Figure 4.3: Number of DOMINATOR Nodes if a DOMINATOR Switches Back Immediately to DOMINATEE Status

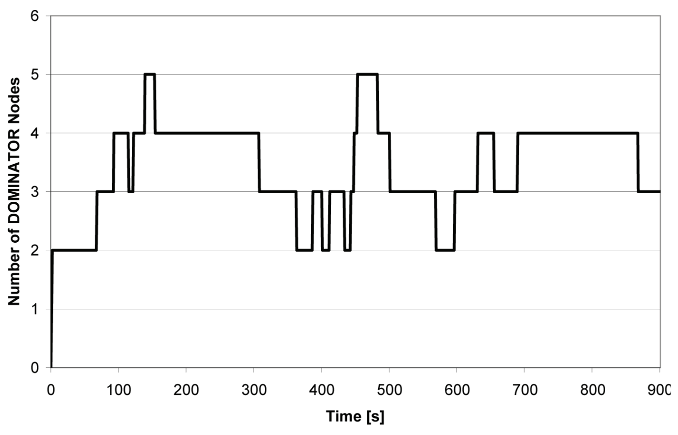


Figure 4.4: Number of DOMINATOR Nodes if a DOMINATOR Waits 10 Seconds Before Switching Back to DOMINATEE Status

in the computed DS. We collected the minimum and maximum number of DOMINATOR nodes in case of the different scenarios (moreover the minimum, maximum, average values and the

Scenario	min	max	avg	σ
School Yard w/ 15 nodes	0	1	0.6	0.5
Train w/15 nodes	0	4	1.6	1.3
Test w/15 nodes	0	2	0.8	0.6
School Yard w/35 nodes	2	5	3.6	0.8
Train w/35 nodes	3	7	4.4	1.2
Test w/35 nodes	1	9	5.6	2.6

Table 4.5: Number of DS Changes

Scenario	min	max	avg	σ
School Yard w/ 15 nodes	2	3	2.4	0.5
Train w/15 nodes	2	4	3.1	0.5
Test w/15 nodes	2	3	2.4	0.8
School Yard w/35 nodes	2	4	3.2	1.1
Train w/35 nodes	6	13	7.6	1.9
Test w/35 nodes	5	8	7.0	0.6

Table 4.6: Minimum Number of DOMINATOR Nodes

Scenario	min	max	avg	σ
School Yard w/ 15 nodes	2	4	2.9	0.9
Train w/15 nodes	3	6	4.5	1.5
Test w/15 nodes	2	4	3.0	0.5
School Yard w/35 nodes	5	9	6.6	0.6
Train w/35 nodes	10	16	12.6	1.7
Test w/35 nodes	10	17	13.0	1.8

Table 4.7: Maximum Number of DOMINATOR Nodes

standard deviation of these numbers through the several runs with different seed values of a given scenario) in Table 4.6 and 4.7. Investigating Table 4.6 we can notice, that in all the scenarios the DS consists always of minimum 2 nodes as required. However, the average value of the Train scenario is a bit higher than in the other two scenarios (3.1 and 7.6) because the reduced transmission range and the narrow but long geometry require more DOMINATOR nodes. In regard to the maximum number of DOMINATOR nodes (cf. Table 4.7) we can observe very similar tendency. Moreover, the number of DOMINATOR nodes is increasing with the increasing size of the geometrical area and the increasing number of nodes, such as in the Test scenario, which is not so surprising. The interesting thing is, however, that the Train and Test scenarios show very similar behavior from this respect. This indicates to us, that the careful selection of the scenario and the mobility pattern are very important and they should capture real world situations as much as possible.

Distribution of the Node Weights: For the shown simulation results above, the node weight has been distributed between 0 and 99. With this choice, most of the nodes will have different weight values and the PBS algorithm can determine the priority based on these weights. This simulates also the situation that every node will have different capabilities to act as server due to different hardware configurations and other factors as remaining battery power. But for investigating the difference to the case where most of the nodes will have the same node weights, the School Yard scenario with 35 nodes has been run as well with the node weights distributed only between 0-4. Now, the other criteria to determine the priority as described in section 3.2.2 get more importance. Such a node weight distribution can be used, if the nodes are only divided in some categories like ’mobile phone’, ’pda’, or ’laptop’ that indicates the node’s server capability. The results of the different node weight distribution is shown in Table 4.8. It can be seen, that the second case where the node weights are distributed between 0-4 requires slightly more time to determine the servers. But the differences are not really significant. From the view of the Determination Delay both node weight distributions are usable but the first one should be preferred. Because with a big range of node weights it is possible to classify the node’s server capability and it is guaranteed that really the ”best” nodes will be chosen in the DS.

Scenario	min	max	avg	σ
Node weights 0-99	0.003	1.358	0.510	0.311
Node weights 0-4	0.016	1.369	0.521	0.296

Table 4.8: Determination Delay for School Yard w/ 35 Nodes Scenario

4.3 Summary

Different to the performance analysis of section 3.2.5, in this chapter the PBS algorithm has been evaluated based on simulations of real world scenarios. Three different scenarios has been used: The School Yard scenario simulates a group of people that are standing on a school yard and are gaming a multiplayer game together. In the Train scenario, a train is assumed where some passengers are playing with each other. And for testing the robustness and reliability of the PBS algorithm a Test scenario has been used that faced the algorithm with some more challenging conditions than the other two scenarios. The simulations have been run with 15 nodes, where 10 nodes participated in a game session, and with 35 nodes where 25 of them participated in the game. The following metrics have been used to evaluate the performance of the algorithm:

Bandwidth: This metric indicates the used bandwidth by the PBS algorithm comparing to the total available bandwidth (2 Mbps in these simulations) from the viewpoint of a node. It could be shown that the created overhead of the algorithm and the required bandwidth will not cause any serious problems.

Determination Delay: This metric indicates the time that is needed until a node gets a neighboring DOMINATOR node or becomes itself a DOMINATOR. In average the delay is in the order of 100-200 ms for 15 nodes and 100-500 ms for 35 nodes for the different scenarios. It could be seen that the Train scenario creates a 'chain' topology that is similar to the worst case scenario as discussed in section 3.2.5 and leads to maximum delay values of 1 second.

Number of Required Changes: This metric indicates from the view of a node how often it looses the connection to its DOMINATOR node and a new DOMINATOR node has to be determined. The averaged values are all between 0.1 and 0.4. This means that most of the nodes had constant connections to the DOMINATOR node. It is evident that the higher level of mobility also influences the number of required changes.

Number of DS Changes: This metric shows how often the initial Dominating Set needs to be changed during a game session from a global viewpoint. Note that in the simulations, a DOMINATOR node will remain a DOMINATOR node, even if its clients are covered by other DOMINATOR nodes. Otherwise, it could be shown that an oscillation problem can arise and nodes always join and leave the DS. Similar to the previous metric, the increasing number of nodes and the higher mobility level (for example in the Test scenario) the required DS changes are also increasing. In average the number of changes are between

0.6 and 1.6 for the simulations with 15 nodes, and 3.6 and 5.6 for 35 nodes.

Number of DOMINATOR Nodes: This metric indicates the number of DOMINATOR nodes in the computed DS. The number of DOMINATOR nodes is increasing with the increasing size of the geometrical area and the increasing number of nodes. For example, the values for the Train scenario are a bit higher because the reduced transmission range and the narrow but long geometry leading to a ’chain’ topology require also more DOMINATOR nodes to build a DS.

In addition to the measured metrics above, different distributions of the node weights has been investigated, as well. In the shown simulation results above, the node weights have been distributed between 0 and 99. With this choice, most of the nodes will have different weight values and the algorithm determines the priority based on these weights. It could be shown, that there are no significant differences to the Determination Delay if the node weights are only distributed between 0 and 4. With this distribution, the other criteria (span value, number of DOMINATOR neighbors, and node ID) for determining the priority of a node become more important. Therefore, the first distribution should be preferred because then the different hardware configurations and other factors as remaining battery power can really be mapped to the node weight and indicate the node’s capability to act as server.

In appendix B, the implementation of the algorithm in a real testbed will be described. This implementation can be used to collect practical experiences.

Chapter 5

Implementation

In this chapter a general overview of the implementation of the PBS algorithm in the SIRAMON framework [11] will be given. More specific information can be found in appendix B.

5.1 About SIRAMON

SIRAMON (Service provIsioning fRAMework for self-Organized Networks) is a service provisioning framework based on a decentralized and modular design. SIRAMON has to integrate the functions required to deal with the full lifecycle of services, such as service specification, lookup, deployment and management of not only trivial but also complex services (e.g., mobile ad hoc group game applications). Every device runs an instance of SIRAMON which handles the control and synchronization among the devices, as well. The different functions of the different service provisioning phases are placed into different modules:

Service Specification: The Service Specifiaction module defines a universal service description language to describe the heterogeneous services and assists applications in the usage of this language. For the service description XML Information Sets [33] are used. Such an XML infoset defines an abstract data set in a tree structure and is normally described using a well-formed XML document [34].

Service Indication: The Service Indication module of the framework enables to advertise and discover services. Due to the mobile environment, the service indication is completely distributed and no central service directory can be used.

Service Deployment: The Service Deployment module is responsible for deploying a service. Deploying a service requires the following actions that are covered by this module:

- Requesting and downloading software according to the specification.
- Discovering and gathering of resources.
- Mapping of this specification to resources.
- Configuring and installing the downloaded software.
- Activating the service in a synchronized manner along with the other service participants.
- Transferring the control to the Service Management module.

Service Management: The Service Management module is responsible for the maintenance of running services. Maintenance includes the control of the service execution, the dynamic service adaptation to resource and environment variations, as well as user triggered service adjustment, and the support of the communication between both local and remote services.

Environment Observer: The Environment Observer module monitors the network, node and user context and makes this information available to the framework and its services. An application can query the Environment Observer about certain resource characteristics. It is also possible to register watch statements, therewith the application is informed when a resource characteristics falls below or rises above a certain threshold.

In Fig. 5.1, a device model for the SIRAMON framework is shown. It can be seen that the framework, integrating the service provisioning functions, is located as a management middleware layer. This layer provides an interface between the device's resources and the applications. The framework is implemented using Java to be independent of the used platform.

5.2 Implementation Overview

The task of this implementation was to port and integrate the evaluated algorithm from the NS-2 simulator implemented in C++ (see appendix A) into the SIRAMON framework that is written in Java. Therefore, in both implementations the main core of the algorithm is implemented using a Finite State Machine (FSM) that can easily be ported between the two programming languages and guarantees that the evaluated algorithm from the simulator is implemented in the framework with the same properties, behavior, and performance results. The description of the FSM can be found in appendix C. The implementation of the PBS algorithm required different changes, respectively additional implementations in the different modules:

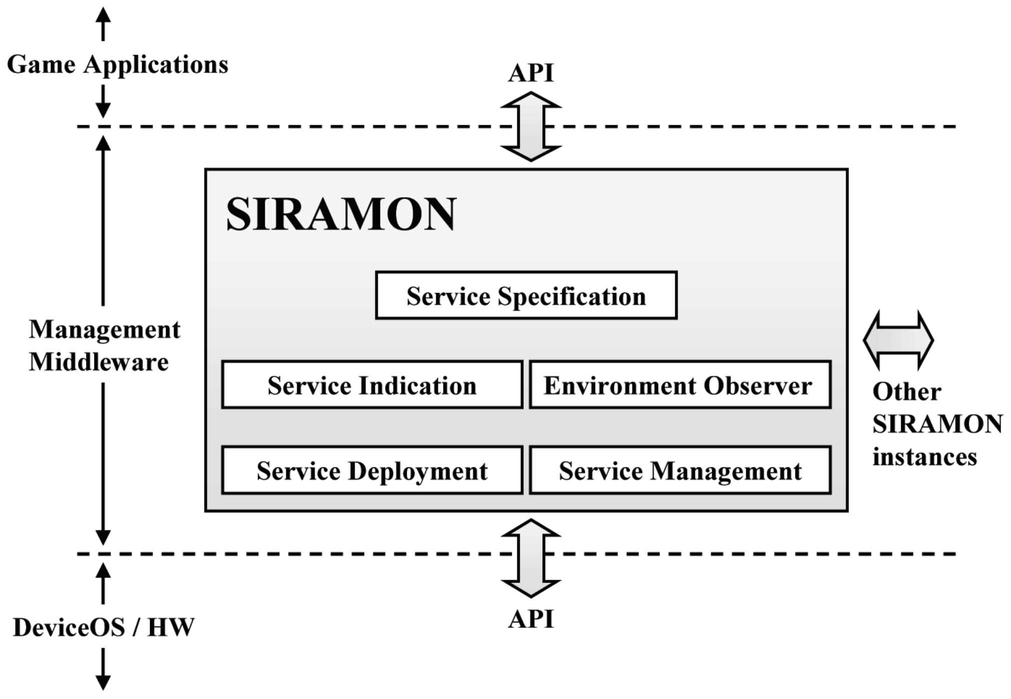


Figure 5.1: Ad Hoc Device Model with SIRAMON

Service Specification:

A service needs the possibility to indicate if it requires the Zone Server Selection functionality of the framework. Therefore, two additional attribute fields have been added to the service description XML document. The fields are shown in Table 5.1 and are added under the DEMANDS element (path SERVICE/IMPLEMENTATION/CODE/ENVIRONMENT/DEMANDS). A description about the structure of the service description files in SIRAMON can be found in [12]. Both fields are optional, and it is assumed, if they are not specified, that no Zone Server Selection is needed.

Attribute	**Description**
zss	Indicates if the service requires Zone Server Selection or not. Possible values are 'true' or 'false'.
zss_server	Contains the server component that will be started, if the node is selected as server respectively DOMINATOR.

Table 5.1: Two New Attribute Fields of the DEMANDS Element

Service Management:

The Zone Server Selection functionality has been implemented in the Service Management module. If a service is started that set the 'zss' attribute field to true, a new instance of the PBS algorithm will be initiated. To distinguish between different instances a unique session ID is generated as well for the different instances that is used in the sent packets between the different nodes. Based on this session ID, the receiver of the framework knows to which instance a received packet containing a neighborlist belongs.

Environment Observer:

Because the PBS algorithm needs to know the current 1-hop neighbors of the node, a simple net monitor that monitors the 1-hop neighborhood has been implemented in the Environment Observer module of the framework. This monitor is used by the PBS algorithm to get the information about changes in the neighborhood.

Chapter 6

Conclusions and Outlook

This chapter gives a summary of the achievements by this Master's Thesis and give some proposals for the further development of the project.

6.1 Conclusions

New communication paradigms like mobile ad hoc networks (MANETS) can offer new ways and unique features for real-time multiplayer games and there are high expectations toward the wireless gaming market potentials. The different mobile devices need to collaborate as a self-organizing mobile ad hoc system, and therefore a service provisioning framework specially adapted for such environments are required. One functionality of such a framework, is the presented Zone Server Selection of this thesis.

It could be shown, that different to the today's game architectures, like centralized server or peer-to-peer models, a Zone-based architecture is better suited to support the demands of multiplayer games in ad hoc networks. In the zone-based model, the player nodes are divided into separated zones. In every zone, a dedicated server node handles the players belonging to the zone and synchronizes with the other Zone Servers. These server nodes should be selected in an efficient and distributed way. The main contribution of this thesis is the development of an algorithm that selects the most powerful nodes as servers. The algorithm is called PBS (Priority Based Selection) and computes in a distributed manner a Dominating Set in a given graph. The nodes in the DS will act as server. Note that there exist already several distributed algorithms that construct Dominating Sets as presented in this thesis. But most of them have been developed for the purpose of providing routing functionality in ad hoc networks and do not fulfill all the requirements for selecting servers for a Zone-based architecture.

For this selection a Node Weighted Dominating Set has to be constructed. This means every node needs an assigned weight that represents the node's capability to act as Zone Server. In addition, every node needs an ID that is unique within the network and which can be used as tie breaker by the algorithm. Independent from the network topology there has to be minimum two nodes in the DS that are acting as server for redundancy reasons. The algorithm has to run locally on every node and the time and message complexity should be kept minimal for saving resources. And as the first approach, the algorithm has to offer continuous maintenance of the computed DS even if the network graph changes dynamically due to the node's movement.

The PBS algorithm fulfills all these requirements and is well suited to provide service management support in ad hoc networks for real-time applications:
The algorithm performs in rounds and is based on exchanging neighborlists that contain the relevant information about the neighbors. In every round the algorithm sends the current neighborlist to its neighbors, receives the neighborlists from them and determines its own status. A node can be in DOMINATOR (acts as server), DOMINATEE (regular client), INT_CANDIDATE (participates in the game, but not yet determined if DOMIANTOR or DOMINATEE) or EXT_CANDIDATE (does not participate in the game, but can be chosen as DOMINATOR) status. The algorithm compares the priorities of the nodes, and chooses the highly prioritized nodes as DOMINATORs. The priority is based on the node weight, the span value, the number of DOMINATOR neighbors, and finally the node's ID.

The algorithm has been evaluated first in an analytical way and it could be shown that the performance of the algorithm can keep up with the already existing algorithm. In the simulations, the algorithm has been evaluated based on some real world scenarios. It has been shown and discussed, how much overhead is produced by the algorithm, how quick and robust PBS is, how good and stable the computed DS is, and some hints about how oscillations within the DS could be avoided.

Finally, the PBS algorithm has also been implemented in the real SIRAMON testbed for getting some practical experience and for the further development of the algorithm. Though the task of this project was mainly motivated by multiplayer games, the zone-based architecture with PBS can be used even in other real-time applications, such as collaborative working, which have similar requirements to multiplayer games.

6.2 Outlook

Based on the work of this Master’s Thesis, there exist numerous possibilities to investigate and develop the Zone Server Selection, and especially the PBS algorithm, further:

Synchronization vs. Zone Server Transferring: Some deeper investigations concerning the mechanisms that are required to support a Zone-based architecture are required. Based on the simulations, the question arose what is more expensive: Having a lot of DOMINATOR nodes acting as server or having less servers but a higher probability that due to the mobility of the nodes new Zone Servers have to be determined during the game session. This can also lead to that the clients need to change to another server. Having a lot of DOMINATOR nodes increases the complexity of the required synchronization and consistency mechanism between the servers and can slow down a game session. On the other hand, new Zone Servers needs first to transfer the game states from another Zone Server, and there is a game interruption if a client changes the server. Therefore, some investigations for different types of games about the complexity of the synchronization mechanisms, the cost of transferring the game states from one Zone Server to another, and changing server from the view of a client are needed.

Mobility Prediction Model: As shown in section 4.2, there exist an oscillation problem if a DOMINATOR node switches back to DOMINATEE if its clients are covered by other DOMINATOR nodes and it has the lowest ID among these DOMINATOR nodes. It should be investigated how mobility prediction can facilitate the stability of the computed DS and if a mobility prediction model can be developed. If there exists such a model, this could influence the node weights forcing only those nodes to join the DS with a low probability to move away.

Real-time Multiplayer Game: A real-time multiplayer game is needed, that uses the Zone-based architecture and uses the PBS algorithm for selecting the Zone Servers. Based on the practical experiences with the algorithm, the scalability limits of the algorithm can be investigated and improvements in the performance can be done.

Appendix A

NS-2 Implementation

In this appendix, a rough overview of the implementation of the PBS algorithm in the NS-2 simulator will be given. The source of this implementation, and a detailed sourcecode documentation can be found at the project homepage [35]. This implementation has been used for the simulations as described in section 4 and is based on the Ns-allinone package release 2.28 (released Feb 3, 2005) as it was available at the NS-2 homepage [10]. This package contains all required components used for running NS-2.

A.1 About Network Simulator NS-2

NS-2 is developed and maintained by a large amount of researches and part of the Virtual InterNetwork Testbed (VINT) project [36]. It is distributed freely and open source. Versions are available for several kinds of Unix and Windows platforms.

The simulator core is written in C++. An object oriented variant of the script language Tcl, called OTcl, is used for the configuration scripts. The combination of these two languages offers a compromise between performance and ease of use of the simulator.

The simulation of a specific protocol under NS-2 consists of four steps:

1. Implementation of the protocol by C++ and OTcl code
2. Description of the simulation scenarios in OTcl
3. Running the simulation
4. Analysis of the generated trace files

For viewing network simulation traces and real word packet trace data a Tcl/TK based animation tool, called Network Animator (NAM) [37] can be used.

A.1.1 PBS Implementation in NS-2

The PBS algorithm has been implemented as agent (ZSSpbsAgent) in the core part of the NS-2 simulator that can be attached to different nodes in the simulation configuration scripts. This agent is inherited from a main Zone Server Selecting agent called ZSSAgent that provides some general functionality for selecting Zone Servers. This provides the possibility to implement modified PBS agents or completely different approaches using other Zone Server Selecting Agents by just adding a new agent that is inherited from ZSSAgent. For the communication between the agents, an own protocol (PT_ZSS) and packet format have been defined that consist of a content and a content length field, and can be used by any Zone Server Selection implementation. In Table A.1, the used directory structure of the implementation is shown. It can be seen, that all the necessary files are below the ns-2.28 main directory. Table A.2 shows the files used for the implementation in the C++ core of the simulator and a short description. The use of the PBSagent in the Tcl script files for running simulations is shown in section A.3.

Directory	**Description**
ns-2.28	The ns-2 main directory.
ns-2.28/zoneserver	The zoneserver main directory.
ns-2.28/zoneserver/utils	Contains some utils for the ZSS agents.
ns-2.28/zoneserver/doc	Contains the sourcecode documentation.
ns-2.28/zoneserver/tcl/wired	Contains the Tcl scripts for wired environments.
ns-2.28/zoneserver/tcl/wireless	Contains the Tcl scripts for wireless environments.

Table A.1: Directory Structure of the PBS Implementation in NS-2

To cope with the node's mobility and to detect changes in the topology of the network a basic net monitor has been implemented as well. In the current implementation this monitor is based on periodically sent 'alive' messages. If a node sends longer than a specified amount of time no message, it is assumed as disappeared and it will be removed from the neighborlist. New nodes can easily be detected based on newly received 'alive' messages.
Because the final goal of this project is to implement the PBS algorithm in the SIRAMON framework, that is written in Java, a way had to be found that guarantees a correct porting from the C++ code to the required Java code. Therefore, the workflow and the required actions of the algorithm has been controlled by a Finite State Machine (FSM) defined by states and trans-

File	Description
zss{.h .cc}	The main class for any ZSS implementation.
zss_pbs{.h .cc}	Implements the PBS algorithm.
timer_pbs{.h .cc}	Timer functionality used by the PBS algorithm.
neighborlist{.h .cc}	Stores and handles the relevant information about the neighbors.
node_information.h	Contains the structure of a node entry in the Neighborlist and the structure of the sent packets by the PBS algorithm.
monitor{.h .cc}	Monitors the 1-hop neighborhood.
timer_monitor{.h .cc}	Timer functionality used by the monitor.
debugger{.h .cc}	Debugger used to write outputs.
utils/fsm{.h .cc}	Implementation of a Finite State Machine (FSM).
utils/state{.h .cc}	State of the FSM.

Table A.2: Used Files of the PBS Implementation in NS-2

actions between the states, that will be executed based on incoming events. With the use of an FSM, the port from the simulator to the framework is much easier and guarantees that the simulated algorithm and its behavior is implemented in the same way in the framework.

A.2 General Architecture

In Fig. A.1 the basic architecture of the PBS agent in the NS-2 simulator is shown. For the communication between the agents an own packet format containing the neighborlist and using UDP has been defined. Every agent has a recv() function where the packets destined for this PBS agent arrive. This function handles the incoming packets, stores the relevant information from the packet in the own neighborlist and sends an event to the FSM. Based on the received event, the FSM performs the corresponding action.
The details and specification of the Finite State Machine can be found in Appendix C. Because the same state machine has also been used for the implementation in the SIRAMON framework.

A.3 Typical Tcl File

To run simulations in NS-2 Tcl scripts are used to build different topologies and to simulate different scenarios. A good tutorial for getting started with the Tcl script language and some basic

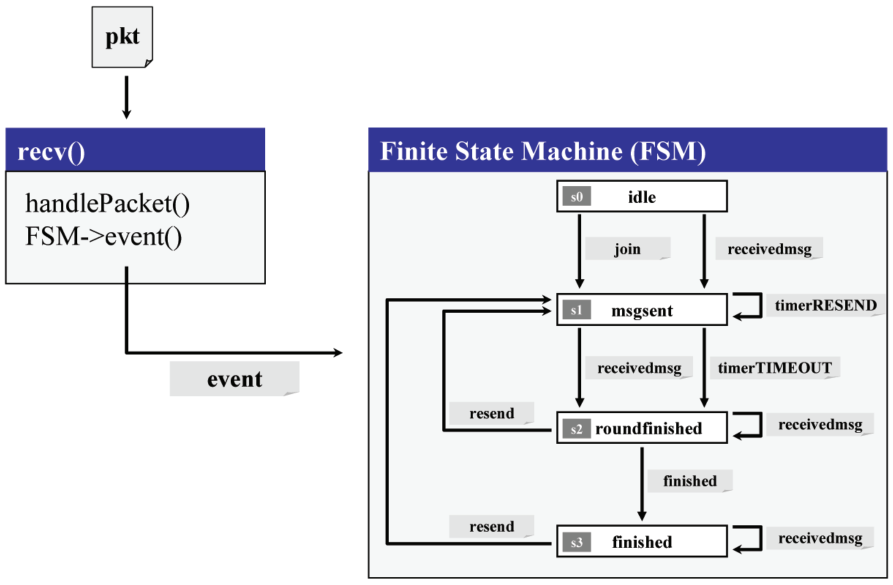


Figure A.1: Architecture of the Implementation in NS-2

examples for simple topologies in NS-2 can be found at ”Marc Greis' Tutorial for the network simulator ns” [38] for wired and wireless scenarios. In Fig. A.2, the basic Tcl commands used for attaching the PBS agent to a node is shown. Note that the shown script does not build a topology between the nodes or add some movements of the nodes. The information about that, can be found as well in Marc Greis' Tutorial.

To generate the scenarios as described in section 4, Mobigen has been used. Mobigen is a small hand made script generator, written in Java, that outputs some Tcl files that can be used for the simulations in NS-2. All of the used Tcl files and the generated scenarios can be found in the ns-2.28/zoneserver/tcl directory.

A.4 Getting Started...

For using the PBS algorithm implementation, the following steps are required:

1. Download the Ns-Allinone package from the official NS-2 homepage [10] and install it according to the instructions.

2. Copy the zoneserver directory containing the implementation into the ns-2 main directory (e.g., ns-allinone-2.28/ns-2.28).

3. Some changes in some of the ns source files are required to add the new agent, especially because a new packet format is used:

- Add to the file "`common/packet.h`" the new packet protocol ID "`PT_ZSS`" to the second last line of the `enum packet_t {}`. In the same file you have to add the line `name_[PT_ZSS]="zss";` to the `p_info()` function.
- The file "`tcl/lib/ns-default.tcl`" has to be edited, too. This is the file where all default values for the Tcl objects are defined. Insert the line `Agent/ZSS set weight_ 0` to set the default weight for Agent/ZSS.
- The new packet has also to be added in the file "`tcl/lib/ns-packet.tcl`" in the `foreach prot` loop with the entry "`ZSS`".
- The last change is a change that has to be applied to the "`Makefile`". All source-files has to be added after the `OBJ_CC=` list:
  `zoneserver/zss.o zoneserver/zss_pbs.o zoneserver/debugger.o zoneserver/timer_pbs.o zoneserver/timer_monitor.o zoneserver/monitor.o zoneserver/neighborlist.o zoneserver/utils/fsm.o zoneserver/utils/state.o`.

4. Recompile ns by using `make clean; make depend; make`.
5. After recompilation the Tcl scripts can be used to run simulations with the command `ns example.tcl`.

For viewing network simulation traces and real word packet trace data the Tcl/TK based animation tool, called Network Animator (NAM) [37] can be used. In the NAM tool, the PBS agent colors the DOMINATOR nodes red, the DOMINATEE nodes blue, the INT_CANDIDATE nodes black, and the EXT_CANDIDATE nodes gray. In Fig. A.3, a print screen of the NAM tool illustrating a School Yard scenario with 35 nodes at time 7.42s is shown. It can be seen, that at this time instance there are 3 DOMINATOR nodes (node 22, 24, and 32) determined. All other nodes are DOMINATEE or EXT_CANDIDATE nodes.

```
set ns [new Simulator]

#create 3 nodes
set node(0) [$ns node]
set node(1) [$ns node]
set node(3) [$ns node]

#create agents
set agent(0) [new Agent/ZSS/PBS]
set agent(1) [new Agent/ZSS/PBS]
set agent(2) [new Agent/ZSS/PBS]

#attach agents to the nodes
$ns attach-agent node(0) agent(0)
$ns attach-agent node(1) agent(1)
$ns attach-agent node(2) agent(2)

#set node weights
$agent(0) set weight_ 10
$agent(1) set weight_ 20
$agent(2) set weight_ 30

#Schedule events
$ns at 0.00001 "$agent(0) init_zss"
$ns at 0.00001 "$agent(1) init_zss"
$ns at 0.00001 "$agent(2) init_zss"
$ns at 0.1 "$agent(0) join"
$ns at 0.1 "$agent(1) join"
$ns at 0.1 "$agent(2) join"
$ns at 900.0 "finish"
```

Figure A.2: Tcl Commands for PBS Agent

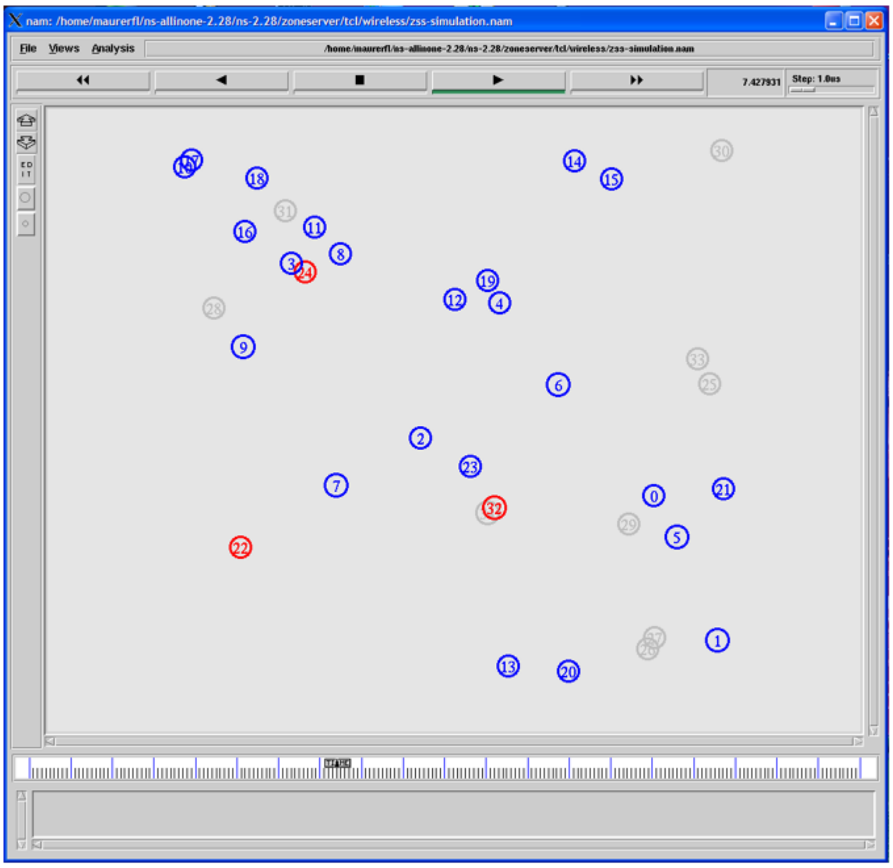

nam: /home/maurerfl/ns-allinone-2.28/ns-2.28/zoneserver/tcl/wireless/zss-simulation.nam
File
Views
Analysis
/home/maurerfl/ns-allinone-2.28/ns-2.28/zoneserver/tcl/wireless/zss-simulation.nam
7.427931
Step: 1.0us

Figure A.3: School Yard Scenario Shown in NAM

Appendix B

SIRAMON Implementation

In this appendix more detailed information about implementation of the PBS algorithm in the SIRAMON framework [11] is given. The sourcecode of this implementation and the corresponding sourcecode documentation can be found at the project homepage [35].

B.1 PBS Implementation in SIRAMON

As already mentioned, the Zone Server functionality and the PBS algorithm has been implemented in the Service Management module of the SIRAMON framework. Similar to the implementation in the NS-2 simulator, the PBS algorithm is implemented in a class ZSS_PBS that is inherited from a basic class ZSS that provides some general functionality for selecting Zone Servers. This provides the possibility to implement modified algorithms or completely different approaches for Zone Server Selection for future projects. The Service Management module is implemented in an own java package (Siramon.Management) inside the framework. For the Zone Server Selection two subpackages, Siramon.Management.zss and Siramon.Managament.zss.utils, have been added. A short description of these new packages can be found in Table B.1. Table B.2 shows the files and a short description that are contained in the packages.

Package	**Description**
Siramon.Management.zss	Contains the main classes needed by the ZSS functionality.
Siramon.Management.zss.utils	Contains some utils for the ZSS functionality like a basic debugger and the Finite State Machine (FSM)

Table B.1: New Packages Containing ZSS Functionality

File	Description
ZSS.java	The main class for any ZSS implementation.
ZSS_PBS.java	Implements the PBS algorithm.
ZSS_PBSfsm.java	Implements the FSM used by the PBS algorithm.
ZSS_PBSpacket.java	Used packet format to be sent in the content part of the SIRAMON packet.
TimerPBS.java	Timer functionality used by the PBS algorithm.
Neighborlist.java	Stores and handles the relevant information about the neighbors.
NodeInformation	Used by the Neighborlist to store the information about a neighbor.
utils/Debugger.java	Debugger used to write outputs.
utils/FSM.java	Basic implementation of a Finite State Machine (FSM).
utils/State.java	State of the FSM.

Table B.2: Used Files of the PBS Implementation in SIRAMON

B.2 Packet Format

For the communication between the different nodes, the existing infrastructure provided by the framework has been used. The neighborlists are embedded in the message part of the defined SIRAMON packets. In order to be able to distinguish between different running instances of a ZSS algorithm supporting different services, a unique Session ID is assigned to every instance. This ID needs to be included as well in the header of a SIRAMON packet. Based on this information, the network receiver thread of the framework is able to detect to which instance a packet belongs. In addition, also the information about the server component that needs to be started, has to be transmitted inside the packet. Because an EXT_CANDIDATE node, that receives a packet from a neighbor needs to know which Server Component needs to be started in case it becomes a DOMINATOR node. The used fields of a SIRAMON packet are shown in Table B.3. The most important information is in the Service Identifier and the Message fields. The Service Identifier contains beside the identifier value ”zss” also the Session ID and the Server Component information separated by a semicolon. In the Message field the neighborlist containing the information about the neighbors is stored.

The SIRAMON packets are sent within UDP packets and according to the algorithm, the packets

Field	**Description**
Prefix	”Siramon”
Source address	Contains the address of the sending node.
Flooding Flag	Not used
Timestamp	Not used
Service Identifier	”zss:<SessionID>:<ServerComponent>”
Message	Contains the neighborlist.

Table B.3: Used Packet Fields of a SIRAMON Packet

should be sent to the 1-hop neighbor of a node. For saving resources, the packets are not sent using unicast to these neighbors, but it is enough to send it once to the broadcast address with the Time-to-live (TTL) value set to one. Due to the restriction of Java, that the TTL field can only be set to multicast sockets, every node joins to a specified multicast address and sends the packet containing the neighborlist to this address with TTL set to one.

B.3 Net Monitor

Because the PBS algorithm needs to know the 1-hop neighbors a simple net monitor that detects the neighbors is implemented in the Siramon.Network package of the SIRAMON framework. The used classes are shown in Table B.4. The monitor sends periodically broadcast messages with TTL set to 1 into the network and receives the messages from the neighbors. If a neighbor sent longer than a specified amount of time no message, the monitor assumes that the neighbor is disappeared. If a class wants to be informed by the Net Monitor about changes in the neighborhood, it has to implement the NetMonitor_Callback interface and add itself to the listeners of the monitor.

File	**Description**
NetMonitor.java	The monitor that sends and receives packages to detect the 1-hop neighbors.
NetMonitor_Callback.java	Callback interface used by other classes to be informed about changes in the neighborhood.

Table B.4: Used Files of the Net Monitor Implementation in SIRAMON

Appendix C

Finite State Machine (FSM)

In this chapter, the used Finite State Machine (FSM) [39] as used in the implementation of the PBS algorithm in the NS-2 simulator and in the SIRAMON framework is described. A FSM is a model of behavior composed of states, transitions and actions. Between different states of the machine some transitions are defined that based on incoming events perform the defined actions. The specification of the used FSM in the PBS algorithm implementation is shown in Fig. C.1.

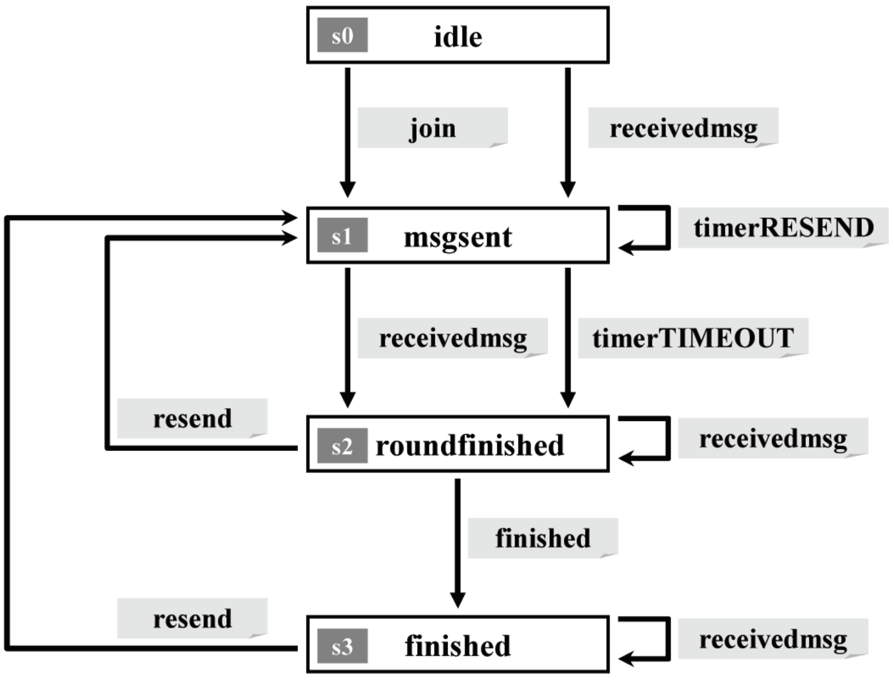


Figure C.1: Finite State Machine (FSM)

C.1 The States

The FSM consists of 4 states: ”idle”, ”msgsent”, ”roundfinished”, and ”finished”. The ”idle” state is the initial state of the FSM. Once the PBS algorithm is started, the algorithm performs in rounds. In every round it sends the Neighborlist to its neighbors and waits in the ”msgsent” state for the neighborlists from the neighbors. If all required neighborlists are arrived or the timeout timer has expired, it goes to the ”roundfinished” state and determines the own status. If there are still INT_CANDIDATE neighbors, the neighborlist is resent and the FSM is again waiting in the ”msgsent” state. If all INT_CANDIDATE neighbors have switched to DOMINATEE or DOMINATOR status, the FSM goes to the ”finished” state. It is waiting in the ”finished” state unless there are some changes in the network topology detected or there are again some new INT_CANDIDATE neighbors. If this is the case, a new round of the PBS algorithm will be started and the FSM turns again to the ”msgsent” state.

C.2 The Transitions and Actions

In Table C.1, all transitions, the corresponding events, and a short description of the action are listed:

From State	To State	Event	Action
idle	msgsent	join	The node joins the Game Session and starts sending out neighborlists.
idle	msgsent	receivedmsg	The node is still waiting in the idle state, but received a neighborlist. It starts now sending out neighborlists as well.
msgsent	msgsent	timerRESEND	The resend timer expired and not all neighbors sent a neighborlist back. Therefore, the already sent neighborlist will be resent.
msgsent	roundfinished	receivedmsg	All required neighborlists have arrived. The node determines its own status.
msgsent	roundfinished	timerTIMEOUT	Not all required neighborlists arrived, but the timeout timer expired and the node determines its own status.
roundfinished	msgsent	resend	There are still INT_CANDIDATE neighbors and a new round of the PBS algorithm needs to be started.
roundfinished	finished	finished	All nodes determined their status. There are no INT_CANDIDATE neighbors left.
finished	finished	receivedmsg	Handles incoming neighborlist even the node is already in the finished state. If required it sends a neighborlist back.
finished	msgsent	resend	Changes in the network and/or some INT_CANDIDATE neighbors have been detected. A new round of the PBS algorithm needs to be started.

Table C.1: The Transitions of the Finite State Machine (FSM) Implemented in the PBS Agent

Appendix D

Used Abbreviations

ad hoc *for this* (latin); for a particular case without any form of centralized administration.

CDS *Connected Dominating Set*; If all nodes of a Dominating Set (DS) induce a connected subgraph, the DS is connected.

DS *Dominating Set*; A dominating set of a graph, is a subset of nodes such that all nodes are neighbor of a node in the DS or are itself in the DS.

EUI-48 *Extended Unique Identifier*; Better known as MAC address. In computer networking a MAC address is a unique identifier attached to most forms of networking equipment. Most layer 2 network protocols use one of three numbering spaces managed by the IEEE: MAC-48, EUI-48, and EUI-64, which are designed to be globally unique.

FSM *Finite State Machine*; A FSM is a model of behavior composed of states, transitions and actions.

GPRS *General Packet Radio Service*; The General Packet Radio Service (GPRS) is a new non-voice value added service that allows information to be sent and received across a mobile telephone network. It supplements today's Circuit Switched Data and Short Message Service.

GSM *Game Scalability Model*; A model that provides a possibility to compare the scalability of network topologies [8].

IETF *Internet Engineering Task Force*; The IETF is the protocol engineering and development arm of the Internet.

MAC *Medium Access Control*; Handles access to a shared medium.

MANET *Mobile Ad hoc Network*; Temporary network in which devices want to communicate with each other, with a continuously changing network topology and without any form of centralized administration. MANET is also the name of an IETF working group, that is working in the field of ad hoc networks.

MDS *Minimum Dominating Set*; A Dominating Set (DS) is called a MDS if the number of nodes in the DS is minimal.

NAM *Network Animator*; Tool to view network simulation traces and real word packet trace data from the NS-2 simulator.

NS-2 *Network Simulator 2*; NS-2 is a discrete event driven simulator to support networking research.

NWDS *Node Weigted Dominating Set*; If the nodes of a graph have weights, the NWDS is the smallest weighted subset of the nodes forming a Dominating Set (DS).

OSI Model *Open System Interconnection Model*; Networking framework for implementing protocols in seven layers.

OTcl *MIT Object Tcl*; An extension to Tcl/Tk for object-oriented programming.

P2P *Peer-to-peer*; Network that does not have fixed clients and servers, but a number of peer nodes that function as both clients and servers to the other nodes on the network. Any node is able to initiate or complete any supported transaction.

PBS *Priority Based Selection*; An algorithm that selects the Zone Servers supporting a zone-based architecture based on comparing priorities of the nodes.

SIRAMON *Service Provisioning Framework for Mobile Ad-hoc Networks*; A proposal of a generic, decentralized service provisioning framework for mobile ad hoc networks [11].

SLP *Service Location Protocol*; Scalable framework for the discovery and selection of network services [40].

TCP *Transmission Control Protocol*; Highly reliable host-to-host protocol between hosts in packet-switched computer communication networks.

UDP *User Datagram Protocol*; Unreliable host-to-host protocol between hosts in packet-switched computer communication networks.

UDG *Unit Disk Graph*; A geometric graph in which there is an edge between two nodes if and only if their distance is at most one [19].

UMTS *Universal Mobile Telecommunications Service*; The name for the third generation mobile telephone standard in Europe.

URI *Uniform Resource Identifier*; Internet standard that consists of a compact string of characters for identifying an abstract or physical resource.

URL *Uniform Resource Locator*; Internet standard that formalizes information for location and access of resources via the Internet.

WLAN *Wireless Local Area Network*; In a WLAN the device (e.g. a laptop, PDA, etc.) communicates via a wireless connection with a WLAN Access Point which is connected (just like a normal computer) via a cable to the Internet or the local network. As the devices are not wired up the users are mobile. This is the advantage of a WLAN. The indoor range depends on structural factors and is considerably lower than outdoors, where WLAN connections are possible over more than 200 metres.

XML *Extensible Markup Language*; Simple, very flexible text format for electronic publishing and data exchanging, standardised by the World Wide Web Consortium (W3C).

ZSS *Zone Server Selection*; The selection procedure for choosing the Zone Servers supporting a zone-based architecture.

Bibliography

[1] Department of information technology and electrical engineering. http://www.ee.ethz.ch/.

[2] Swiss federal institute of technology zurich. http://www.ethz.ch/.

[3] ETH Zurich. Computer engineering and networks laboratory. http://www.tik.ee.ethz.ch.

[4] Game Developers Conference. San Francisco, California, USA, March 2005. http://www.gdconf.com/conference/gdcmobile.htm.

[5] K. Farkas, L. Ruf, M. May, and B. Plattner. Real-Time Service Provisioning in Spontaneous Mobile Networks. In *Proceedings of the Students Workshop of The 24th Annual Conference on Computer Communications and Networking, (IEEE INFOCOM 2005)*, Miami, Florida, USA, March 2005.

[6] T. Henderson and S. Bhatti. Networked games: a QoS-sensitive application for QoS-insensitive users. In *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, Karlsruhe, Germany, Aug. 2003.

[7] R. Steinmetz. Human perception of jitter and media synchronization. *IEEE Journal on Selected Areas in Communicatons*, 14(1):61–72, 1996.

[8] J. Muller and S. Gorlatch. GSM: A Game Scalability Model for Multiplayer Real-time Games. In *Proceedings of The 24th Annual Conference on Computer Communications and Networking, (IEEE INFOCOM 2005)*, Miami, Florida, USA, March 2005.

[9] S.M. Riera, O. Wellnitz, and L. Wolf. A Zone-based Gaming Architecture for Ad-Hoc Networks. In *Proceedings of the Workshop on Network and System Support for Games (NetGames2003)*, Redwood City, USA, May 2003.

[10] Information Sciences Institute ISI. The Network Simulator ns-2, February 2005. http://www.isi.edu/nsnam/ns/.

[11] K. Farkas. Siramon - service provisioning framework for self-organized networks. ETH Zurich, January 2005. http://www.csg.ethz.ch/research/projects/siramon/.

[12] R. Grueninger. Service provisioning in mobile ad hoc networks. Master's thesis, ETH Zurich, Computer Engineering and Networks Laboratory, 17th September 2004. MA-2004-12.

[13] GNU.org. The gnu general public license. http://www.gnu.org/licenses/licenses.html#TOCGPL.

[14] E. Cronin, B. Filstrup, and A. Kurc. A distributed multiplayer game server system. Um eecs589 course project report, University of Michigan, May 2001. http://warriors.eecs.umich.edu/games/papers/quakefinal.pdf.

[15] M. Mauve, S. Fischer, and J. Widmer. A generic proxy system for networked computer games. In *NetGames 2002 Proceedings*, pages 25–28, Braunschweig, Germany, April 2002.

[16] S. Helal, N. Desai, V. Verma, and C. Lee. Konark A Service Discovery and Delivery Protocol for Ad-Hoc Networks. In *Proceedings of the Third IEEE Conference on Wireless Communication Networks (WCNC)*, New Orleans, March 2003.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, USA, 1983.

[18] R. M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of the Symposium on the Complexity of Computer Computations*, pages 85–103, New York, New York, USA, 1972.

[19] B. N. Clark, C. J. Colbourn, and D. S. Johnson. Unit Disk Graphs. *Discrete Mathematics*, 86:165–177, 1990.

[20] R. Wattenhofer. Chapter 8 - Dominating Sets. Course material mobile computing, Distributed Computing Group, ETH Zurich, 2004. http://dcg.ethz.ch/lectures/ss04/mobicomp/lecture/8/Chapter8DominatingSets4Slides.pdf.

[21] L. Jia, R. Rajaraman, and T. Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Computing*, 15(4):193–205, December 2002.

[22] J. Wu and H. Li. *Handbook of wireless networks and mobile computing*, chapter A Dominating-Set-Based Routing Scheme in Ad Hoc Wireless Networks, pages 425–450. 2003. ISBN:0-471-41902-8.

[23] F. Kuhn and R. Wattenhofer. Constant-Time Distributed Dominating Set Approximation. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'03)*, pages 25–32, Boston, Massachusetts, USA, July 2003.

[24] K. M. Alzoubi, P-J. Wan, and O. Frieder. Message-optimal Connected Dominating Sets in Mobile Ad Hoc Networks. In *Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking & Computing 2002*, pages 157–164, Lausanne, Switzerland, June 2002.

[25] D. Dubhashi, A. Mei, A. Panconesi, J. Radhakrishnan, and A. Srinivasan. Fast Distributed Algorithms for (Weakly) Connected Dominating Sets and Linear-Size Skeletons. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms 2003*, pages 717–724, Baltimore, Maryland, USA, January 2003.

[26] P. Klein and R. Ravi. A nearly best-possible approximation algorithm for node-weighted steiner trees. *Journal of Algorithms*, 19:104–115, 1995.

[27] S. Guha and S. Khuller. Improved methods for approximating node weighted steiner trees and connected dominating sets. *Information and Computation*, 150:57–74, 1999.

[28] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998.

[29] Y. Wu, Y. Xu, G. Chen, and K. Wang. On the construction of virtual multicast backbone for wireless ad hoc networks. In *1st IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 294–303, Florida, USA, 2004.

[30] 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 ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 85–97, October 1998.

[31] T. S. Rappaport. *Wireless Communications, Principles and Practice*. PRENTICE HALL INTERNATIONAL, 1996. ISBN: 0-13-042232-0.

[32] IEEE-SA Standards Boards. Part11-wireless lan medium access control (mac) and physical layer (phy) specifications. 12 June 2003. http://standards.ieee.org/getieee802/802.11.html.

[33] World Wide Web Consortium (W3C). Xml information set. http://www.w3.org/TR/xml-infoset/.

[34] World Wide Web Consortium (W3C). Extensible markup language (xml). http://www.w3.org/XML/.

[35] Florian Maurer. Master's thesis of florian maurer, August 2005. http://www.siramon.org.

[36] Information Sciences Institute ISI. Virtual InterNetwork Testbed (VINT) project, October 1997. http://www.isi.edu/nsnam/vint/index.html.

[37] Information Sciences Institute ISI. Nam: Network Animator, July 2003. http://www.isi.edu/nsnam/nam/.

[38] Marc Greis. Tutorial for the Network Simulator ns. http://www.isi.edu/nsnam/ns/tutorial/index.html.

[39] Wikipedia. Finite state machines. http://en.wikipedia.org/wiki/Finite_state_machine.

[40] IETF. Openslp homepage. http://www.openslp.org.