“FUTURE INTERNET ARCHITECTURE TO STRUCTURE AND TO MANAGE DYNAMIC AUTONOMOUS SYSTEMS, INTERNET SERVICE PROVIDERS AND CUSTOMERS”

Por

LUCIANA PEREIRA OLIVEIRA

Dissertação de Mestrado
LUCIANA PEREIRA OLIVEIRA

"FUTURE INTERNET ARCHITECTURE TO STRUCTURE AND TO MANAGE DYNAMIC AUTONOMOUS SYSTEMS, INTERNET SERVICE PROVIDERS AND CUSTOMERS"

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER IN COMPUTER SCIENCE AT THE FEDERAL UNIVERSITY OF PERNAMBUCO.

SUPERVISOR: Prof. Dr. Djamel Fawzi Hadj Sadok

RECIFE, FEVEREIRO/2008
Oliveira, Luciana Pereira

Future internet architecture to structure and to manage dynamic autonomous systems, internet service providers and customers / Luciana Pereira Oliveira. - Recife : O Autor, 2008.

xi, 120 páginas : il., fig., tab.


Inclui bibliografia e anexos.

1. Arquitetura de redes de comunicações. 2. Internet. 1. Título.
To my mother Graça, my father Vicente, my sister Ana and to my husband Ricardo.
Acknowledgments

This dissertation would not have become real if it were not for the various people that directly and indirectly contributed to it. First of all, I would like to thank my advisor, Prof. Djamel for an endless supply of suggestions and criticisms. I learned a lot from him not only how to do research, but also how to interact effectively with students. I am deeply grateful to Ramide, Eduardo James and my research group (Auristela, Igor, Marcos, Reinaldo and Tarciana) who advised me on many technical topics related to my research and for all the good times that we spent together. I was lucky to meet some great people in the Networking and Telecommunications Research Group (GPRT).

Thanks to Ericsson Research, for the sponsor of the RAN project that motivated this work.

And finally, I would like to give thanks to my family and my friends, who have given me support and encouragement over the past several years, for their patience during the times when I should have been paying attention to them, but instead I was absorbed in my studies. A special thanks to my parents Graça and Vicente who are really special. Also, thanks to my sister, Ana for her love and encouragement. Special thanks also to my husband, Ricardo, for his love, patience and encouragement.
Abstract

In recent years, the problem of dynamic networks has received a great deal of attention from the network research community. This problem consists in providing for the Internet architecture a set of mechanisms, such as addressing, information management and information forwarding, which support mobile information and mobile entities (Autonomous Systems, Internet Service Providers and Customers).

Previous work on Internet architectures proposed a way of separating the location (currently IP) and host name, due to a strong association between the IP address and a name. In general, they propose overlay routing to separate such information. Other works believe that this decoupling is not enough to solve mobile problems, since the dynamicity generates many rebinds and control messages. For this reason, they proposed new models to manage the overlay.

In this work, we propose a solution called Stable Society which adopts a role based approach. A role is a functional unit that is used to organize the communication. A distinguishing feature of our proposal from the others also based on an overlay approach is that in addition decoupling the name and the location, it also deals with the problems concerning structure and the maintenance of overlay. We define four roles: the messenger which forwards data into society; the guard seen as the most stable entity to forward inter-society messages; the worker responsible for storing information; the leader who structures and manages the overlay.

To reducing our scope of implementation during the master research, we considered the network layer as having a messenger role and the guard as having the same stability distinction. This is to study and provide a mechanism that manages the overlay routing. Therefore, as a proof of concept introduced by this proposal, we implemented the leaders and workers which acted independently of the technology access while being fundamental to solve the instability problems in storing and discovering resources in the architecture. As a result, we propose a new algorithm named Stable Society model over Distributed Hash Table (SSDHT). In addition, we compare it to another DHT solution (Chord algorithm). The results show that our proposal performs better, mainly when we increased the instability (the traffic loads, degrees of mobility, and network sizes). For example the level of success discovery was more than 90%.

Keywords: Internet Architecture, Dynamic Network, DHT Algorithms, Overlay.
Resumo

Diversos trabalhos na área de redes dinâmicas têm sido propostos na literatura com o objetivo de prover à arquitetura da Internet o suporte à mobilidade. O problema dessas redes instáveis na Internet consiste em oferecer um conjunto de mecanismo, tais como endereçamento, gerenciamento da informação e encaminhamento da informação, que suportem informação e entidade (Sistema Autônomo, Provedor de Serviços na Internet e Clientes) móveis.

Nesse contexto, alguns trabalhos para arquitetura da Internet têm proposto uma maneira de separar a localização (atualmente o IP) e o nome identificador, devido ao forte relacionamento entre o IP e o nome. Em geral, eles propõem uma abordagem de roteamento na camada overlay para separar essas informações. Outros trabalhos acreditam que este desacoplamento não é suficiente para solucionar os problemas de mobilidade, desde que a dinamicidade gera muitas mensagens de controle e atualizações do vínculo entre o IP e o nome. Por essa razão, os pesquisadores também têm proposto novos modelos para gerenciar a camada overlay.

Uma das contribuições deste trabalho é a proposta de uma solução para arquitetura da Internet denominada Stable Society que adota a abordagem de papéis. Um papel é uma unidade funcional que é utilizada para organizar a comunicação. Um importante diferencial da proposta é que além de desvincular o nome e a localização, ela também oferece soluções para os problemas relacionados a estruturação e manutenção da camada overlay. Além disso, este trabalho define quatro papéis: o mensageiro encaminha os dados dentro da sociedade; o guarda é a entidade mais estável para encaminhar mensagens entre as sociedades; o operário armazena informações; e o líder estrutura e gerencia a rede overlay.

Reduzindo o escopo para a implementação desta dissertação de mestrado, o mensageiro e o guarda foram considerados como a camada de rede sem distinção de estabilidade, desde que o fornecimento de um mecanismo de gerenciamento do overlay de roteamento foi o objetivo do trabalho. Portanto, como prova do conceito apresentado pela proposta, os líderes e operários foram implementados, porque eles agem de maneira independente de tecnologia de acesso e são fundamentais para solucionar o problema da instabilidade nos processos de armazenamento e descoberta da informação. Como resultado, um novo algoritmo denominado Stable Society model over Distributes Hash Table (SSDHT) foi proposto. Além disso, este algoritmo foi comparado com outras soluções DHT (Chord). Os resultados mostraram que o SSDHT é um bom algoritmo, principalmente quando se aumenta a instabilidade (carga do tráfego, grau de mobilidade e tamanho da rede). Por exemplo, a taxa de mensagens entregue com sucesso foi acima de 90% quando a carga de tráfego, o grau de mobilidade e o tamanho da rede foram variados.

Palavras-chave: Arquitetura da Internet, Redes Dinâmicas, Algoritmos DHT, Overlay.
## Chapter 1 – Introduction

1.1 Motivation  
1.2 Dissertation focus  
  1.2.1 Object addressing  
  1.2.2 Information management  
  1.2.3 Information forwarding  
1.3 Contributions  
1.4 Plan of thesis

## Chapter 2 – The Internet

2.1 Introduction  
2.2 Approaches for Internet architecture  
2.3 Requirements for unstable scenarios executing in future Internet  
2.4 Related work  
  2.4.1 Conceptual model for future Internet  
  2.4.2 Object addressing  
  2.4.3 Information management  
  2.4.4 Information forwarding  
2.5 Concluding remarks

## Chapter 3 – Stable Society: A Conceptual Model for Internet

3.1 Introduction  
3.2 The conceptual model for Internet  
  3.2.1 Stable Society overview  
  3.2.2 Role  
  3.2.3 Rules  
3.3 Use cases: applying the model in set of unstable scenarios  
  3.3.1. Self-organizing the Stable Society  
    3.3.1.1 Normal flow  
    3.3.1.2 Alternative flow  
  3.3.2. Communication with external endpoint  
    3.3.2.1 Normal flow  
    3.3.2.2 Alternative flow  
  3.3.3. Searching information (or the discovery of network objects)  
    3.3.3.1 Normal flow  
    3.3.3.2 Alternative flow  
  3.3.4 Exchanging the Stable Society  
    3.3.4.1 Normal flow  
    3.3.4.2 Alternative flow
Bibliography 84

Appendix A. List of Publications 87

Appendix B. Code of Stable Society Node 88
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The architecture was derived from network functionalities and requirements</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>Stable Society</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Overview of architecture based on roles</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Stable Society addressing</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Mobile scenario</td>
<td>45</td>
</tr>
<tr>
<td>4.3(a)</td>
<td>Scenario is consisting of AS, ISP and Customer entities</td>
<td>45</td>
</tr>
<tr>
<td>4.3(b)</td>
<td>Addressing for AS, ISP and Customer entities</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Society formation</td>
<td>46</td>
</tr>
<tr>
<td>4.5</td>
<td>Vertical x horizontal society graph</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>Repository for the information discovery</td>
<td>49</td>
</tr>
<tr>
<td>4.6(a)</td>
<td>The repository</td>
<td>48</td>
</tr>
<tr>
<td>4.6(b)</td>
<td>The worker leaves the society</td>
<td>48</td>
</tr>
<tr>
<td>4.6(c)</td>
<td>The leader stores W2 information after W2 leaves the society</td>
<td>49</td>
</tr>
<tr>
<td>4.6(d)</td>
<td>The worker W3 stores W2 information after W2 leaves the society</td>
<td>49</td>
</tr>
<tr>
<td>4.7</td>
<td>Stable society architecture</td>
<td>50</td>
</tr>
<tr>
<td>4.8</td>
<td>Creation and maintenance of session</td>
<td>52</td>
</tr>
<tr>
<td>4.9</td>
<td>Recovery the session</td>
<td>52</td>
</tr>
<tr>
<td>4.10</td>
<td>Discovery MOA before to forward information</td>
<td>54</td>
</tr>
<tr>
<td>4.11</td>
<td>Routing information among stable societies</td>
<td>55</td>
</tr>
<tr>
<td>4.12</td>
<td>Routing mobile information (MOA) when devices are moving between stable societies</td>
<td>55</td>
</tr>
<tr>
<td>4.13</td>
<td>Routing mobile information (MOA) when the stable society is moving</td>
<td>56</td>
</tr>
<tr>
<td>5.1</td>
<td>A logical circle of identifiers (ring) is consisting of 13 nodes and it is storing 4 keys</td>
<td>62</td>
</tr>
<tr>
<td>5.2</td>
<td>Logical ring is representing connections for message forwarding</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>SSDHT based on Chord algorithm</td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>Scenario varying number of nodes that join in network at same time</td>
<td>71</td>
</tr>
<tr>
<td>6.2</td>
<td>Node entry in Chord and SSDHT</td>
<td>73</td>
</tr>
<tr>
<td>6.2(a)</td>
<td>Time to stabilize the network</td>
<td>73</td>
</tr>
<tr>
<td>6.2(b)</td>
<td>Message rate in stabilization process</td>
<td>73</td>
</tr>
<tr>
<td>6.3</td>
<td>Network model</td>
<td>74</td>
</tr>
<tr>
<td>6.4</td>
<td>Average performance metrics for a varying offered load in a mobile network of 100 nodes</td>
<td>77</td>
</tr>
<tr>
<td>6.4(a)</td>
<td>Control plane load</td>
<td>77</td>
</tr>
<tr>
<td>6.4(b)</td>
<td>Request success ratio</td>
<td>77</td>
</tr>
<tr>
<td>6.4(c)</td>
<td>Overlay consistency</td>
<td>77</td>
</tr>
<tr>
<td>6.5</td>
<td>Average performance metrics for a varying churn interval in a network of 100 nodes</td>
<td>78</td>
</tr>
<tr>
<td>6.5(a)</td>
<td>Control plane load</td>
<td>78</td>
</tr>
<tr>
<td>6.5(b)</td>
<td>Request success ratio</td>
<td>78</td>
</tr>
<tr>
<td>6.5(c)</td>
<td>Overlay consistency</td>
<td>78</td>
</tr>
<tr>
<td>6.6</td>
<td>Average performance metrics for a varying size in mobile networks</td>
<td>79</td>
</tr>
<tr>
<td>6.6(a)</td>
<td>Control plane load</td>
<td>79</td>
</tr>
<tr>
<td>6.6(b)</td>
<td>Request success ratio</td>
<td>79</td>
</tr>
<tr>
<td>6.6(c)</td>
<td>Overlay consistency</td>
<td>79</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Today’s Internet can be considered as a structure composed of different Autonomous Systems (ASs) that are interconnected among them. These ASs are responsible for the management and routing domains. Inside an AS, one finds several Internet Service Providers (ISPs) with a large customer base. The ISP is commonly used to refer to an entity that provides Internet connectivity service, whether directly to the end user or to other service providers. The term Customer is in general used to a local network that accesses the Internet, but can be also used to an end user.

The current Internet infrastructure is based on the fact that all these entities (ASs, ISPs, and Customers) are fixed, i.e., these entities were not designed to deal with user and network mobility. This feature ensures more scalability and stability that are important to provide the efficient routing and to maintain the topology for data discovery by the Domain Name System (DNS) [Mockapetris 1987]. It provides an essential service to the Internet, mapping AS, ISP and Customer names structured to a variety of data, usually Internet Protocol (IP) addresses. Once the DNS finds the information, it returns its corresponding IP address that is then used to forward packets in the current Internet architecture.

However, recent researches indicate that many ASs, ISPs and Customers are likely to become mobile in very next future while the information already presents mobility characteristics. Multimedia content, connectivity, web services among other
advanced applications, can no longer be dependent on a location or technology. User sessions should be *migrated* across devices and network boundaries in a seamless manner. The IETF has acknowledged this with its network mobility working group [NEMO] among other efforts. Networks cannot be seen as simple pipes if they are to take us beyond good old simple connectivity. No longer physical, they need to be both transparent and virtual in order to keep users in constant touch with events at work, favorite sites and information content, entertainment material, music and movie libraries, etc. Sessions are dynamic and should not be interrupted when moving between home and the workplace. Content should seamlessly be tailored to fit a diversity of user processing and communication contexts. Productivity will benefit from these future dynamic networks. Whereas today’s networks are divided into fixed and mobile ones, future networks are implicitly mobile. The challenges that lay ahead are tremendously complex and need careful new rethinking of how we build next generation networks.

Therefore the future Internet will operate in unstable environment which means having scenarios composed of mobile information and mobile entities (AS, ISP and Customers) that dynamically join and leave their active scenes. Chief among such difficulties are object addressing, information management (including location information) and forwarding to mobile entities that also can be heterogeneous. A set of works [Stoica et al. 2002], [Clark et al. 2003], [Crowcroft et al. 2003] and [Balakrishnan et al. 2004] indicate that the first step to solve problems involving mobility suggest the separation of location identification (currently IP) and the name. Currently mobility in general may present unreachable paths due to stronger link between IP and name. Additionally, when a customer moves, its IP is modified and also the DNS may become inconsistent. This decoupling strategy may be adequate for the mobile data, but it is not enough to resolve mechanisms inherent to mobile entities that involve routing, management and storing of mobile information.

Therefore, the question is how could one redesign such mechanisms (addressing, forwarding and management) to take into consideration the unstable environment? This work looks for a solution to structure and manage the entities and information in order to avoid chaotic routing and inconsistent data structures.
1.2 Dissertation focus

The main contribution of this work is to offer a solution for the problem of how to structure and to manage devices, in order that dynamic entities (ASs, ISPs and customers) can be integrated with traditional entities, which are static, in the context of the next-generation Internet architecture. We design an Internet architecture using roles over devices in order to lead with mobility, scalability and heterogeneity.

We know that a proposal of a new architecture to structure and manage mobile entities can influence several network functionalities such as addressing, forwarding, information management, network management, performance (e.g. quality of service) and security. Each one of these provides some special and important functional capabilities for the architecture. For example, the security network functionality offers capabilities to restrict unauthorized access, usage, and visibility within networks to reduce the threat and effects of attacks. Performance gives network resources to the ability to support user requirements for capacity and delay. Network management allows the network monitoring, configuring, and troubleshooting. Information management provides a structure to find and store the information. Both addressing and forwarding offer the robust and flexible connectivity between devices [McCabe 2003].

Figure 1.1 - The architecture was derived from network functionalities and requirements.

In addition, Figure 1.1 shows the flows used in this dissertation to design our Internet architecture proposal. Among these functionalities, we choose to work with...
addressing, forwarding and information management, after we identified that there are dependencies and a stronger relation among them. Even after the network is structured, we need to know how to forward the messages in order to maintain the system working. However familiarity with the addressing mechanism is important for the forwarding of the data. It is equally important to know how to manage the information, because in dynamic environments new data keeps appearing, and hence must be processed to restructure and to maintain the network.

1.2.1 Object addressing

Network objects can be users, services, content and devices. Traditionally in the current Internet, such objects have strong and static associations among them due to a lack of mobility. For example, a device and its content were always found in the same physical network, in other words, the objects were found using unique and fixed addressing information. Current networks have embraced mobility leading to the need of associating mobile objects and heterogeneous devices to several locators. For example, a device can have one Bluetooth interface and two Ethernet interfaces (three addresses) which need to be distinguished by high level services. Although there has been some work on multi-homing, we are a long way from supporting generic object multi-addressing. Hence both device heterogeneity and object mobility remain a challenge that requires new hard work.

The evolution process of addressing began with the introduction of service naming and naming resolution schemes. Yet there is a current strong and persistent coupling among naming and physical location dependent addressing. A second step can be seen, for example, in the work by [Stoica et al. 2002] where the use of indirection through an overlay network was suggested as a means to implement layer naming using an identifier whose each packet is associated. The works [Clark et al. 2003], [Crowcroft et al. 2003] and [Balakrishnan et al. 2004] proposed other strategies and architectural changes for decoupling hosts and services in same context.

Theses approaches are important for unstable scenarios where the object can join or leave the network, because they provide an increased flexibility to an otherwise stiff Internet architecture. These works also have added new functionalities for the support of multicast and unicast, mobility of endpoints, content distribution and replication, etc.
However, the Internet architecture must specify how to bind different addressing and to extract information from theses addressing such as technologies, content, service and user in order to structure and manage mobile and heterogeneous nodes.

Therefore, we adapted the decoupling naming and redirection approach to work with three addressing levels that describe different role (technology, approximate location and any name). The Traditional Address (TA) describes the technologies such as a Bluetooth address or an IP address, because the Internet consists of heterogeneous technologies. The Route Address (RA) describes the approximate location in the physical world, because the objects can be in any location, and this information may increase the performance in routing. The Mobile Object Addressing (MOA) describes the name independently of location and of technologies. We propose a new style of addressing grouping and a different strategy to name objects according to the role of each device, user, gateway, access network worker, and content provider.

1.2.2 Information management

Future environments require the knowledge of various things such as object capabilities and location, which can be logical (e.g. geographic point is inferred from its network addressing and a close device with a known position). This data is important for objects to create and to manage communities of smart items. For example, information about location and capability can be used to improve routing. However, the management of these records in dynamic scenarios is a complex task. Additionally, the organization and maintenance with scalability is another potential problem in the Internet.

For this reason different structured mechanisms for information discovery have been propose such as Distributed hash tables (DHTs) that offer fast data access and content distribution while guarantying scalability. They also may be used in naming, addressing and DNS systems. However, existing DHT work as in [Stoica et al. 2003] assumes that nodes are stable and have equivalent processing and storage capacities. By stable it is meant that once being part of a DHT ring, the frequency of a node leaving and reentering the ring again is practically very low.
On the other hand, other researches have been designing new technologies without dealing with scalability, because they wanted to propose a mechanism to search for data in unstable scenarios with heterogeneous devices. For example, the Universal Plug and Play [UPnP] and [Jini] are not scalable mechanisms, but theses are able to advertise and search for service, to self-configure the network with little human intervention and to learn about dynamic nodes.

With the objective of proposing in this work the information management for the future Internet that will have the dynamic, heterogeneity and scalability features, we adapted the DHT mechanism in order to allow heterogeneous devices as [UPnP] and [Jini]. We applied additional addressing level(s) in the DHT with explicit support for Internet mobility and information management. Moreover, we introduced a new concept called “stable society” presented in Chapter 3 in order to overcome DHT limitations. Commonly, the word “society” refers to a group of individuals willingly living according to certain established rules while joining forces in order to reach common good [Ferreira 1988], namely, in this case network stability and seamless homogeneity. These rules are given in section three of this document through the adoption of a functional hierarchy and roles for the adopted society.

1.2.3 Information forwarding

Dynamic networks are often subject to location change events, the swapping of devices and domain boundaries crossings, resulting in new important challenges for the correct forwarding of messages with or without QoS. Ideally, user sessions should be migrated across devices and network boundaries in a seamless manner.

Currently, the majority of Internet traffic uses a simple static unicast forwarding model in which hop by hop relaying of packets is performed. Multicast traffic is supported to deal with multimedia group communications, but not all Internet routers chose to enable multicast support for fear of traffic congestion problems. Static packet forwarding is clearly unsuitable for the future Internet. It breaks and interrupts dynamic sessions when users move between home and work place for example.

Many solutions have been proposed to offer host mobility at the network level including the IETF Mobile IP, SIP and even some combinations as in [Huang et al.
This proposal looks at a different paradigm for the creation of a virtual architecture capable of self-organizing and monitoring network/node conditions and improving it. This is achieved by building resilient and scalable virtual networks as a means to take networking into the next era. We propose the use of dynamic tunneling and forwarding data according to some economic and social objectives used to determine the role of each one of the existing network devices, users, gateway, access network workers, and content providers.

Therefore, we propose how to combine Stable Society concept and our addressing approach, which describes technologies, location and name information separately, in order that dynamic network should be able to forward data. We propose how to adapt the routing mechanism to exchange information with an overlay (structure of information management) constituted of roles that controls dynamic networks.

### 1.3 Contributions

In the following, we summarize the main research issues related to this work.

- A new conceptual model is proposed, called Stable Society, to describe the roles and interaction among them in a network in order to provide an effective support for the organization of dynamic networks composed of mobile and heterogeneous nodes.
- The proposal of architecture based on Stable Society concept. This architecture presents a complete support for object addressing, information forwarding and management in unstable scenarios.
- The architecture prototype is known as the Stable Society over DHT – SSDHT as it uses both concepts and mechanisms from both the Stable Society and DHT. The goal is creating a scalable overlay to control the mobility and heterogeneity. The Stable Society concept is applied over the Chord algorithm here, a DHT algorithm, but could similarly apply to other DHTs. Moreover the Chord modification was used to validate our architecture and our concept model.
- The analysis of prototype architecture for addressing, information management and overlay routing in unstable environment using the OverSim simulator that allows to build mobile scenario with several nodes. Moreover, the Chord and
SSDHT were implemented to evaluate the impact of node entry on DHT network at same time in real scenario with restriction.

1.4 Plan of dissertation

In order to achieve the proposed objectives, the remainder of this dissertation is organized as follows:

Chapter 2 gives an overview of the Internet research, describing the main approaches for addressing, routing, and information management. This chapter also discusses the main problems faced by the current Internet architecture, including: heterogeneity, mobility and the scalability of addressing, information forwarding and information management. Finally, some related works are discussed.

Chapter 3 presents this work’s approach taken by Internet conceptual modeling. First, the Stable Society model is introduced and some unstable scenarios are described.

Chapter 4 presents the proposed approach. The target requirements and the main components of the proposed architecture are detailed. The chapter also introduces key design decisions and some examples for each mechanism presented.

Chapter 5 shows the implementation of our architecture prototype to manage and structure mobile entities and data. As our solution uses DHT algorithms, some aspects of the DHT are also discussed, specifically the Chord mechanism is presented as our reference model. It was evaluated and compared to our proposal.

Chapter 6 provides the proposal’s validation. First, the scope of validation is introduced. Next, some basic features of the evaluation and its results are discussed.

Chapter 7 concludes the dissertation with the main research contributions and some future work.
Chapter 2

The Internet

2.1 Introduction

Networks are becoming more ubiquitous and dynamic where users are continuously changing location, swapping devices and crossing domain boundaries. Moreover, there are other new challenges to the Internet architecture: the convergence of heterogeneous machines, wireless communication, information anywhere and the ability to maintain existing communication during movement across networks. Therefore the future Internet architecture does not need to focus only on a device location, but also on the mobile content, services and users. In this context, we identified four main problems and difficulties of today’s architecture Internet:

1) Heterogeneity: new scenarios create the challenge of allowing that any device, regardless of the restrictions, to participate of networks without using all its resources, allowing the heterogeneity of equipment. Moreover, existing communication among different addressing and routing protocols is important. This allows greater flexibility and integration of the heterogeneous technologies of access.

2) Mobility: the movement of mobile nodes breaks the session connection, producing the propagation of the instability (update messages, for example) throughout all the network. The Internet architecture needs to know and manage when nodes join and leave the network and how to deal with absent nodes.
3) Adaptations according to the network context: network may be small or larger with the frequent disconnections/connections and dynamic insertion/removal services that represent its context changes. So, the architecture must deal with adaptations when changing the context (heterogeneity and mobility), discovering new physical ways for route and update the network information.

4) Scalability and robustness: the instability generated by future scenarios is a problem, because the mobile information and the mobile entities can become unavailable, increasing the control messages, decreasing the number of successful request. As a result, unstable environments may eliminate and hide several network features such as the load balance, the route organization and the growth of the network.

Analyzing the challenges presented by an unstable environment in the Internet architecture, we identified a set of requirements detailed in Section 2.3 that will influence the network functionalities presented in Section 1.2. For example, the current addressing scheme does not separate mobility information from location identities. This separation is important for the routing to support data that dynamically joins and leaves the active scenes.

In this chapter, we give an overview of the Internet architecture, describing its main approaches in Section 2.2 and detailing a set of requirements for the unstable scenarios in Section 2.3. Next in Section 2.4, we present some related works about conceptual model for future Internet and also a set of related approaches for the addressing, forwarding information, and information management that make up the network functionalities from our Internet architecture proposal. Finally Section 2.5 concludes by discussing an overview from this section’s results.

2.2 Approaches for Internet architecture

The Internet architecture is victim of its own simplicity, which ironically is the key to its success. Currently, many researchers are revealing the warring of more than 30 years evolutionary incremental Internet and its point changes. Some of them are advocating the creation of a new architecture, i.e., to start from a clean slate. Although current
performance evaluation techniques including network simulation and modeling help, they are insufficient on their own. Public funding for actual production and research network testbeds is once again needed to grasp the impact of a new architecture. As this is yet to come, researchers, operators, service providers and businesses have room to debate and exchange views. A close look at the recent proposals for new Internet architectures shows a growing concern for the need to rethink the way networks are built in order to accommodate new use and business models. But once we know how to build networks, can we really build totally different ones?

Recently proposed architectures tend to look into strategies that rely on a number of common enabling technologies and building paradigms that involve mobility and heterogeneous challenges. Related work will be shown in Section 2.3, and here we summarize the main issues only. We choose to review rapidly some of their common concepts than to describe each of the proposed architectures separately. Among these, we have identified the following:

- The use of P2P communications especially for the support of new overlay networks. This distributed processing, generally uses distributed hash tables (DHT) paradigm and has shown its capacity for the adequate and scalable handling of a number of Internet problems ranging from DNS to applications such as IP voice telephony and file sharing. The DHT mechanism is also used to naming, addressing and create additional addressing level(s) with explicit support for Internet content replication and mobility;

- New DNS designs have been proposed in order to provide more scalable naming and addressing mechanisms;

- Core for many naming proposals is their separation between location and addressing information. This is seen as an important requirement for the support of mobile computing and controlled content replication. This protocol design principle has already been used in the design of IETF protocols as in the case of mobile IP and introduces the need of dynamic resolution;

- Indirection and rendezvous mechanisms have been used as built-in design concepts, for example, seamlessly to evoke security and other middle-box packet processing during user sessions;
• Programmability of middle-boxes within networks and information (packet content) processing by these. These can be used by policy servers or administrators to adapt to changing network conditions, uploading new configurations, cooperating with others, etc.

• New architectures may be built in the form of overlays over current IP networks to provide service specific Overlay Networks (ON). ONs may introduce their own new routing protocols, packet formats and addressing schemes. Overlays are not new to the Internet as the Internet Multicast Backbone (MBONE) [Eriksson 94] and PlanetLab [Peterson et. al 2002] have been around for a long time.

• One size can no longer fit all heads. This is due to the fact that the emerging new non-IP networking contexts such as sensor networks, industrial and control networks, delay tolerant networks, etc., are inherently different A meta-architecture that interconnects emerging different networks (or contexts) arises as a new requirement. While benefiting individual networks, new interworking and end-to-end challenges are brought to the scene;

2.3 Requirements for unstable scenarios executing in future Internet

Next we identified the main challenges involving unstable scenarios with mobile information and entities with mobility in Next-Generation Internet. Theses challenges correspond to a set of new requirements to future Internet architecture. Here, we listed the requirements involving unstable scenarios that are generated by dynamic AS, ISP and Customer entities.

Internetworking: a heterogeneous network, that involves existing and future networks, must be integrated and interconnected. It is important because no single technology is adequate for all networks.

Distributed management: it is important for Internet scalability (including naming, routing and addressing). However, it may require a treatment of inconsistence and coordination of distributed management of resources;

Support for mobile objects: the new architecture must support the general notion of object mobility that includes the mobility of devices (already well-know), content,
application services, and human users. The architecture must define how naming, 
addressing, routing and resource allocation may be performed for each type of mobile 
object (information, user and device);

**Ease of attachment:** the structure design must permit easily mobile information 
and networks attachment, detachment and reattachment. The new architecture must have 
the capability of dynamically managing and preserving communication or application 
state during the movement of theses along their path;

**Mobility:** the new architecture must be defined to locate physically the object 
during movement across networks. States and resources must be managed in order to 
support rapid disconnections and reconnections, and device/user/information will be 
available for a significant period of time;

**Heterogeneity:** already presented in [Davies et al. 2006], the Internet must allow 
users’ interaction among end-point devices in a technology independent manner and 
minimize the impact of different autonomous systems, in which the internal properties of 
domains routing environment may differ radically. It is an important requirement for a 
future architecture, because for most parts of the networks participating in the Internet 
this is not really required yet, for example, the intra-domain attributes are generally 
similar across domains.

**Don’t forward instability for all networks:** a future architecture must avoid the 
propagation of instability caused for example by the upgrade of routing tables, resolution 
name, resource relocation and other mechanisms that are in constant change and could be 
a source of instability;

**Multi-Homing:** currently many devices have a set of network interfaces, but a 
final user in general may not receive and transfer data by all the available interfaces at a 
given time. In the future a user could be receiving information using all network 
interfaces of devices that are closer him. For this reason, the future architecture must 
know how to manipulate adequately a set of interfaces that involve end-points, (an end-
point is a unique network interface);

**Allow allocation of resources:** the new architecture must allocate resources 
among mobile objects. In today’s Internet, allocation occurs implicitly as a result of 
congestion control. For commercial activities, there is a desire to allocate capacities based
on willingness to pay. For operational government activities, e.g., disaster response, there is a need to allocate capacities based on the priority of task. It is not (always) the role of the network to tell the user how fast to go. The administrator should be able to ask resources to the network, and the network should be able to inform the user if it cannot meet the requests due to resource limitations;

**Dynamic adaptability:** the structure design must be able to dynamically adapt to dynamic changes in the network taking into account different environments that mobile objects are involved in so as to the user receive information with the best backgrounds and without disconnections. In other words, the architecture must identify, remap and/or reallocate path and resource according to applications/mobile object requirements needed.

**Uploading/downloading data without interruption from/to user:** a challenge to future Internet is providing service by a way that the user receives/transfers a service without interruption, even when there is connection interruption due to mobility and/or the swapping of devices by users;

**Policy and self-configuration:** already presented in [Davies et al. 2006], the Internet should provide auto-configuration of end systems and routers by supporting policies (obligation or authorization) in order to govern the choices in the networks. Obligation policies (event-condition-action rules) define what actions to carry out when specific events occur. Authorization policies define what actions are permitted or not permitted, for what or for who, and under what conditions. We need policies to provide better self-organization, and also provide competition among networks as the Internet is hosting more and more businesses..

**Scalability:** the architecture must allow the growth of the Internet. It will avoid problems with convergence, routing information propagation, and others.

**Security:** today there are many security problems such as the Distributed Denial of Service (DDoS) attacks, due to the insecurity of the current architecture or the protocols that protect the forwarders from these attacks.
2.4 Related work

In this section, we present an architectures and mechanisms focusing on the conceptual model, addressing, forwarding and information managing approaches that directly were related to the problem of mobile information and mobile entities, which are our main focus.

2.4.1 Conceptual model for future Internet

Initially, the Internet started as a small and flexible network. It was developed to support strong changes, however after 1983, the year in which Internet adopted TCP/IP protocol, this architecture passed to be less flexible. Later, with the increase and popularity of mobile and heterogeneity devices, new Internet models were developed aiming to improve the addressing of the entities, information and the structure and management of the networks, including: RBA [Braden et al. 2003] and Plutarch [Crowcroft et al. 2003].

The RBA (Role-Based Architecture) presents a role-based conceptual model for a future Internet architecture. It has no layers and the communication is not organized hierarchically but by abstract entities, called roles. A role can be, for example, "packet forwarding" and "fragmentation" and provides descriptions of network functionalities such as the forwarding and process. The roles build blocks that are well-defined, but have no fixed size, proving more heterogeneity among communication. A role is identified by a unique name called a RoleID which reflects the functionality provided.

Another conceptual model is Plutarch. It mainly deals with the heterogeneity feature in networks, introducing new notions of context and interstitial function and mapping the communication between the sets of functionalities, instead of considering a unique set of protocols for all and homogeneous networks. Plutarch focuses on inter-operation mechanism of heterogeneous networks, without global naming and does not propose a common protocol such as IPv4 and IPv6 for addressing. It depicts each node existing in an explicit context in which all names and addresses usable by the end system must be bound, using an interstitial function to map between different naming and addressing. Each function should be exposed in order to allow automatic set up, maintained and managed by network users. This proposal supports the movement of
name and addressing information, because a name or address may be passed from one such context to another which entails rebinding the referent of the name to the destination context.

2.4.2 Object addressing

The challenge of separating name and addressing is not recent. In 1991 researches discussed the Internet architecture and depicted that the architecture must be naming-based, because it implies a much larger community, and a much more dynamic (and unpredictable) operational. For this reason, many proposals for Internet architectures separate physical location (that finds the information in the network) from identity (that uniquely and persistently specifies that information), including [Balakrishnan et al. 2004], [Yang 2003] and [Balakrishnan et al. 2003].

The Layered Naming Architecture (LNA) [Balakrishnan et al. 2004] establishes a clear separation between a service identifier (or information) from its location. The objective is to allow that the content should be directly and persistently named. This accommodates mobility and multi-homing, as well as, integrates middle-box design into the architecture. This semantic decoupling between the transport and the networking layers is defined introducing one layer over current Internet which has only two global namespaces, DNS names and IP addresses. The additional layer finds endpoint identifier (abstract location and independent of IP) by service identifier (SID). The problem in this proposal is the SID approach, because LNA uses the plane model instead of hierarchical one which is more scalable.

In [Yang 2003], the New Internet Routing Architecture (NIRA) allows any user to choose domain-level routes and sequences of providers that a packet transverses. It builds a hierarchical addressing to reduce the overhead for constructing and representing a route. Each top-level provider will be allocated a globally unique address prefix and its customers receive the prefixes of top-level provider according to recursive process until a domain allocates an address prefix from one of its providers. In NIRA, a domain can have multiple prefixes whether it has multiple route segments. However this proposal does not specifies how to address and name services and other mobile objects.
In [Balakrishnan et al. 2003], an architecture called Semantic-Free Referencing (SFR) proposes that an architecture should be able to accommodate fairly rapid changes in the binding (references) between names and addresses. Moreover it proposes that it should also be able to allow the handling mobility and the replication of objects. It defines that name resolution should be a general purpose application-independent (HTTP, SMTP etc), and the references themselves should be unstructured and semantic-free (using illegible identifier for user producing by hash algorithms). The solution is a general mapping between an unstructured key offer to DHT mechanisms, the providers insert into network pairs (key and meta-data). The meta-data contains the key, real location of the information (i.e. IP and port or DNS name and port), description about application and time to live (i.e. a caching hint). However, the user does not search information using meta-data, the user only searches using an illegible identifier, hence there is a need for a general mapping between legible information (meta-data) and the key used by SFR.

2.4.3 Information management

One of the main problems in dynamic networks is how to provide a scalable mechanism for the discovery and management information according to user context. Many approaches such as [UPnP] and [Jini] provide a way for working in a small network, but do not present a scalable solution. On other hand, DHT is a scalable mechanism for fast access and management of information. However, existing DHT works assume that all the nodes are stable and have equivalent processing and storage capacities. By stable we mean that once being part of a DHT ring, the frequency of a node leaving and reentering again the ring is low. We depart from this view and do not share it as it is hardly adequate for the storage of highly dynamic information in unstable environments with mobile nodes joining and leaving networks. Under such new highly volatile scenario, the use of a DHT leads to generating a large number of ring update messages hence turning routing very unstable and the overall system less scalable [Triantafillou 2003], [Heer et al. 2006], [Cramer et al. 2005] and [Futai et al. 2004].
In an attempt to increase network scalability and robustness, a number of works proposed the clustering approach for both their physical and virtual levels of routing. Others suggested changes to the DHT algorithm in order to take into account mobility. In addition, there are those works that are concerned with heterogeneous scenarios, user context changes and instability resulting from ubiquitous computing.

Tenhunen [Tenhunen et al. 2005] put physically closed nodes into a single cluster in order to reduce the number of messages used by the routing table management process. It classified nodes into three entities: cluster-head (the most capable node to organize the node group), host (the simplest entity) and gateways (which forwarding messages and connect networks) [Baker et al. 1988].

Work in [Triantafillou 2003] applies the clustering concept to DHTs to deal with instability and scalability but fails to address dynamic environments. Similarly, the authors in [Futai et al. 2004] also use both DHTs and clustering to improve support for heterogeneous users and instable environments. Their work allows for the decomposing of clusters and node migration where their number exceeds a threshold level.

There has also been extensive research published with techniques that modified the DHT in order to offer mobility support as in [Futai et al. 2004], [Triantafillou 2003], [Heer et al. 2006] and [Cramer et al. 2005]. Except for [Heer et al. 2006], they do not consider highly dynamic scenarios. In [Cramer et al. 2005] nodes with restrictions are considered virtual nodes and do not makeup the DHT. Work in [Heer et al. 2006] integrates DSR routing with the use of the DHT mechanism but fails to deal with node heterogeneity and capacity differences often found in ad-hoc networks. The authors in [Triantafillou 2003] take a similar approach to the one in [Heer et al. 2006] where a number of problems encountered in dynamic networks are presented as well as pointing to some possible solutions to them.

2.4.4 Information forwarding

Many works have discussed issues about how to perform forwarding in dynamic environment and efficient routing. Some works considered overlay routing and how to
support heterogeneous communication technologies such as [Clark et al. 2003] and [Stoica et al. 2002] and [Huang et al. 2006].

In [Clark et al. 2003], A Forwarding directive, Association, and Rendezvous Architecture (FARA) is proposed as an abstract architecture for decoupling end-system names from network addresses. However it did not detail the technical mechanisms and design decisions. It defined a host-to-host communication as a communication between pairs of entities via logical linkages (identifier - ID) that are called associations. Nodes over network (overlay nodes) know about associations that are invisible to the routers. Each node differently can be having a particular mechanism for delivering data to appropriate addressing and routing, since the nodes use Forwarding Directive (FD) mechanism. FD contains the information needed to deliver the packet to the destination entity. In addition to the destination FD, a packet may also contain a reply FD that can be used to deliver a return packet to the source node. So, when a node wants to send a packet to one of its associations, it forwards the packet to its overlay with a header field which represents the FD. FARA decoupling of IP addressing currently executes forwarding and naming and does not have a global namespace.

Internet Indirection Infrastructure (I3) [Stoica et al. 2002] is an overlay network for general propose in order to provide easy development of services such as multicast, anycast, and mobility through a rendezvous-based communication abstraction. It associates each packet with an identifier which is used by the receiver to obtain delivery of the packet. Each host R must insert a trigger (id, R) in the i3 infrastructure to receive all packets with identifier id. This architecture supports mobility, when a host changes its address, because the host needs only to update its trigger. However, i3 does not support dynamicity. In other words, it forces a change of addressing E1 to E2 along mobility, because i3 does not treat routing issues at the IP level.

Huang [Huang et al. 2006] proposes a mechanism to treat user and network mobility considering Mobile IP (MIP) and the Session Initiation Protocol (SIP) in order to manage and forward user packets. Furthermore, it achieves the route optimization between two SIP clients without too many signaling messages over wireless links, even if the mobile network is nested.
2.5 Concluding remarks

In this chapter, we gave an overview of the Internet architecture research area. First, we introduced the main approaches already proposed for Internet architecture modeling: P2P, Indirection, rendezvous, programmability of middle-boxes, building overlays, flexible packet head size and new DNSs. Theses mechanisms are common in many proposals for future Internet architecture. However, new paradigms are important to accommodate a set of requirements that describe the challenges of the dynamic networks in order to rethink new motility and heterogeneous network support.

We discussed the main requirements faced in the next Internet architecture, including instability, heterogeneity and scalability. Since we propose a conceptual model for next generation Internet, we reviewed the literature that focuses on approaches for abstraction modeling for future Internet, information management, addressing and forwarding of data. Theses proposals have as common goal, that of offering a way to deal mobile information and scalability, but few of these works focus on heterogeneity (I3, SFR, LNA and Plutarch) and instability ([Huang et al. 2006], FARA and RBA). Among theses architectures there are some limitations: architectures based on DHT do not support mobility such as I3, due to churn problem (high number of message retransmission and table update) as depicted in Section 2.4.3.
Chapter 3

Stable Society: A Conceptual Model for Internet

3.1 Introduction

The conceptual model is important to describe a set of abstract principles for the design of protocols, algorithms and mechanisms that attend a given purpose. In this context, we propose a model, called Stable Society, to offer a stable and homogeneous view of the set of heterogeneous and mobile objects (information and network entities). One distinctive feature of the proposed architecture to structure and manage AS, ISP and Customer, entities to present a solution for the problems concerning object addressing, information forwarding and information management in dynamic environments.

The model combines features of cluster, hierarchical structures, roles and overlay approaches for supporting a set of requirements identified in Section 2.3 for the future Internet in unstable scenarios. The cluster approach provides a model to organize network and to minimize updates and other signaling messages propagating through networks. The hierarchical structure approach provides scalability and also minimizes update messages, and supports information and routing aggregation. The overlay approach provides support for device and network heterogeneity. The roles approach provides analogies mechanism (specified in this work) to structure and manage the network.

In this chapter, we present the Stable Society model and we select some scenarios to focus our work, providing more details. We also discuss the main problems in these
scenarios and show how our conceptual model proposal solves them. Finally, we conclude through comparing our model to some other related models for the next generation Internet.

3.2 The conceptual model for the Internet

In this section, we explain the overview and detail each rule, roles (guard, messenger, worker and leader) and show how each role must obey given rules.

3.2.1 Stable Society overview

The Stable Society presents features from [Tenhunen et al. 2005]. The word society refers to a group of individuals living with their own free will while obeying certain rules and joining efforts and resources for achieving some common goals [Ferreira 1988]. Tenhunen defines three types of nodes (gateway, host, and cluster head) in order to ensure better scalability and stability, since these entities reduce the number of update messages that make up the management process for routing tables.

The main society goals in our case are stability and homogeneity. The rules are established through a given functional hierarchy for decision taking and controlling behavior in the society. However, each member of a stable society may in fact belong to one or more levels of this hierarchy. It can implement one or more roles hence making it possible to have a single node in a stable society that accumulates all roles. Nonetheless, a society needs to define all roles in order to exist. An example of a stable society can be seen in Figure 3.1 that contains our four roles: leaders, workers, guards and messengers.
Comparing the work proposed by Tenhunen to our proposal, both have a hierarchal model and a set of roles to organize a grouping. However, our proposal has five differences. First, we define that each individual in a society can be staying in different levels of the hierarchy at same time because an individual can implement one or more roles. Second, instead of defining only three roles for the network layer as proposed by Tenhunen, we specified four roles: two at the virtual level (workers and leaders) and the remaining two at the physical level (guards and messengers). Third, the hierarchical organization in our proposal has internal (into society) and external (among societies) levels in order to support mobile networks and to guarantee better scalability and replication of objects. In each level, we designed roles to provide virtual communication (overlay) and physical communication (network layer). Fourth, the overlay approach adopted in our proposal allows heterogeneous technologies and decouples naming and addressing as in other proposals such as [Stoica et al. 2002], [Clark et al. 2003], [Crowcroft et al. 2003], and [Balakrishnan et al. 2004]. Fifth, we also defined rules to execute decisions, the organization and possible forwarding of messages in a hierarchical society.
3.2.2 Role

Since each device can have a set of features, we specified four important roles that represent important characteristics to ensure the stability and homogeneity of a society. The roles are: guard, messenger, worker and leader.

Guards are responsible for path discovery and forwarding messages inter-societies. They are selected among the stable entities and as such are the entry as well the exit points for a stable society. For example, they have wired interfaces as the Ethernet or DSL link technologies in order to ensure stability. They can also be seen as the connection points between different stable societies. They maintain their physical address within the same technology context.

Messengers are responsible for path discovery and forwarding messages intra-society, for example forwarding messages between two workers. They must forward messages on demand to their leaders when a worker is unable to do so and notify these that the destination worker is unreachable. They are not allocated a virtual address by their respective leaders but rather use their own physical addressing according to their technology. The major differences between messenger and guard are the translation mechanisms executed by messengers according to their capacities or the forwarding messages to other alternative messengers for translation. Messengers deal with internal routing which generally may be unstable (such as when using wireless interfaces) whereas guards deal with external routing and always have a stable connection available to them.

Workers have the tasks of virtual route maintenance and storing information related to their stable society. They represent the simplest individuals and they do not take any decisions that affect the organization of the society. According to their processing and communication capacities, they may ask messengers to let a leader know of their exit from the society or even to reduce the amount of information they requested to route or store. Workers use the same address prefix as the one of their leader along with an identifier or a selector that is unique to a given stable society. Unlike guards and messengers, workers work in virtual routing (overlay).
Leaders are responsible for the organization of the society. For instance, they can also be responsible for punishing other individuals - a leader may be forced to change its roles from leader to worker. Additionally, a leader has the prerogative to promote and demote a node to and from the status of guard, worker and leader according to their capacity state and characteristics at a given time. A leader must use such service to maintain stability and homogeneity. Additionally, leaders take the roles of routing and information storage entities when workers are not available or when it becomes impossible to perform internal routing or storage. Leaders determine the address prefix of a given stable society as well its unique identifier.

3.2.3 Rules

Values and attitudes that characterize each society should come from several sources. However, existing rules offer a powerful resource and coordinate society behavior as well as their relationships.

A policy is a statement defining which functionalities should be treated differently in the network by roles. It is defined by an administrator and specifies the rules related to handling different types of role within the network. Since in dynamic networks there are frequent changes, so the rules can be dynamic. They may be changed any time reflecting the new values and attitudes of a society such as new protocols and environment condition.

We defined seven important rules for applying in the networking concept and to guarantee a stable society design in terms of connectivity. We apply the role over device if it accepts the rules correspondent to role (guard, messenger, worker or leader). The device breaks the rule when it does not execute its functionalities according to role. For example, the forwarding rule specifies that the messenger role must forward data intra society only. If a messenger forwards data inter society, then it breaks the forwarding rule. The types of rules defined for structuring and managing the society are:

1) Addressing rule: There is a different addressing mechanism for each role. It follows a hierarchical addressing based on prefix and suffix information. The individual at the top of the hierarchy must define the first prefix and subordinated
roles must use this same prefix. Objects as content, service or other information use free addressing, but may have a specific society prefix when individuals want to store information into a society. This rule specifies that the guard role uses the addressing mechanism from their network protocol, in other words, it does not receive a virtual address from the leader. Similarly to guards, messengers use the addressing from their network protocol and do not interfere with leader’s mechanism defined for virtual addressing. This rule also specifies that workers must receive prefix address information from the leader for virtual addressing and generate a unique suffix. This rule defines that the leaders impose the prefix addressing to all workers in their mechanism for virtual addressing and generate a unique prefix. In other words, it is the prefix information that mainly distinguishes a society from another one.

2) Forwarding rule: There is a supported mechanism for forwarding data and control that extracts the best in each technology and at the same time allows exchange information among different communication protocols. This is achieved due to existing functions to translate protocols, encapsulate and transmit data according to each individual capability. Moreover, it allows exchanging information among mobile nodes. The guards follow this rule, when notifying to the neighbor leaders that their society is leaving a network, or when notifying their leaders that some society has left a given set of societies, or yet when forwarding external messages and filtering out messages trying to enter their society without the correct destination. The messengers obey this policy, when notifying their leader of a worker leaving the network (unreachable path), or when notifying their leader after identifying that some worker has joined their society and when forwarding internal messages. This rule also specifies that the worker and leader must not interfere in the forwarding of messages in network routing.

3) Announcing rule: Any individual may announce given service information, however such information is distinct depending on the layers. For example, at the network layer there is information about network resources and at the overlay layer there are announcements of how to reach services, content and others. This rule specifies that guard and messenger must announce their network capabilities
(provide a description of their software and hardware capabilities) when they do not have any link with the society. Besides, the messengers must announce their different connectivity protocols. This rule also specifies that the worker and leader must announce their resource capabilities (content, service and others). However, the leaders can announce their resources capabilities for local view and/or summarize information about the society’s capabilities for public global view (to other societies if needed).

4) Discovering rule: the society provides two search levels which may be internal or external to the society. Some individual must participate at the internal level and another must execute this process at an external level. However, the search and result response for content and services must start from a participating individual in the overlay. Moreover, all roles can execute the discovery process according to their capabilities and hierarchical level. This rule defines that guard and messenger should not execute object discovery, they may only forward such messages without actually knowing their content. However, they may request some discovery services to workers. This rule also specifies that the workers must verify if the object prefix falls into the society’s interval, then whether it has the workers request leader, otherwise they request for next worker hop and use messengers for message forwarding between workers or worker and leader. Moreover, this rule defines that the leaders must only search information when it is not found into a society or when the worker responsible for the object discovery process is missing. In these cases, the leaders obey this rule contributing to the discovery process and using guards or messengers for forwarding messages between worker and leader(s).

5) Storage rule: Only individuals from the overlay may store objects, but the capability of each individual must be respected. As a result, an individual’s storage may range between 0% and more resources. The guard and messenger cannot store objects according to this rule. The workers should store information according to their capabilities in order to support the discovery process. This rule also specifies that the leaders should not store information a priori, but must store
this when there is no worker responsible for information storage or when a worker requests the leader to do so probably because it has no storage capability.

6) Management rule: Societies interactions are supported to exchange session control. A society must monitor its hosts and identify mobility to self-organize or forward a session control to another society that a mobile host will join. The mobile host also must inform its new coordinates after leaving the society and the new society that it intends to join. The guard and messenger must not interfere with this rule. This rule specifies that the workers must handshake (bind with) the old and new messengers, when they change messenger during the node mobility. Moreover, this rule defines that the leaders are responsible for session management and the support of inter-society control transfer in order to perform a handshake between the old leader and the new one when a worker leaves a society and joins a new one.

7) Decision rule: This rule corresponds to activities of society control that must be executed only by leaders. The guard, messenger and worker must not interfere in this rule. We know that the activities of control are important, but its specification is outside the scope of this work. However, in the following we specified important activities that must attend the decision mechanism in a stable society:
   - Revoking and attributing roles: The capabilities of each individual can change in time and hence it is important for the societies to monitor, identify, and reorganize a set of individuals. This self-organization is executed by revoking or attributing role from each individual, but also can be seen as a mean to punish individuals that fail to perform their activities.
   - Expulsing members: This is the highest punishment which could be applied when an individual breaks any rule or when it executes malicious activities such as denial-of-service (DoS) attacks, publishing worms and viruses. The individual must have a strong punishment, when it tries to impersonate other individuals or play other roles. However, the identification of these illegal activities is not within the current scope of our work. We are only concerned with the definition of rules and how these are applied.
o Electing a head of society: One or more individuals can be elected to manage the organization of a society. The elections can occur more than once. However, the individual chosen needs to have sufficient capabilities to exchange relatively high quantities of messages and store information when necessary as well as it needs to have a stable behavior to respond and monitor a society.

o Adding rules (Policies) – The society must support the introduction of new rules to safeguard that each specific society may have different security, QoS or other specific policies.

A summary of rules of guards can be seen in Table 3.1.

Table 3.1 Rules of roles

<table>
<thead>
<tr>
<th>Rule x Role</th>
<th>Guard</th>
<th>Messenger</th>
<th>Leader</th>
<th>Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addressing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Forwarding</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Announcing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Discovering</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓ (when there is absent worker)</td>
</tr>
<tr>
<td>Storing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓ (when there is absent worker)</td>
</tr>
<tr>
<td>Management</td>
<td></td>
<td></td>
<td>✓ (when there is absent worker)</td>
<td>✓</td>
</tr>
<tr>
<td>Decision</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

3.3 Use cases: applying the model in unstable scenarios

This section presents use cases illustrating the concept and advantages of a Stable Society Network. The Stable Society has basically four roles in order to allow communication among mobile and heterogeneous devices following a set of rules. To illustrate this wide scope, a number of different use cases are discussed. Each one involving a short scenario description, a precondition (that needs to have already occurred before this use case can start), a post condition (that will be true once the Use Case is complete), as well as, a normal flow (that describes a set of successive steps) and an alternative flow (that describes the exceptional behavior which can interrupt the normal flow).
3.3.1 Self-organizing the Stable Society

In this use case, we are going to work with the following requirements: policies and self-configuration; mobility; internetworking; heterogeneity; and distributed management.

**Scenario Description:** A tourist group is leaving their hotel carrying heterogeneous devices and plans to visit different towns within a certain region. During their bus travel, the passengers wish to share content, information, even services and others resources without losing connectivity and whenever possible they also want to use the local bus moving network. Therefore, the travel guide starts to organize the process of creating the trip network that will self-organize dynamically using the passengers own equipments with different capacities.

**Pre-Conditions:** A pattern of the information model (that contains capabilities description, services and network interfaces of each node) must be used for both request and response messages. It is important to allow the leader to define who is to become the messenger, guard and worker when it looks for a match among the request and responses from network devices.

**Post-Conditions:** When the process of discovering and choosing the roles (messenger, guard, work and leader) is finalized successfully, the leader receives an accept response message. Next, the Internet and trip network know how to exchange data and to find users. The trip network created a new region if there was no super leader higher in the hierarchy, otherwise it has just created a new sub-region (part of preexisting region) into a super leader region. The region’s (end-point identify - EID) could be determined manually, using GPS, domain information or other similar configuration.

3.3.1.1 Normal flow

The self-organization of Stable Society proceeds in two steps: internal and external. Both phases involve making decisions to organize devices within a set of rules and roles
(messenger, guard, leader or/and worker). This distribution management will provide support for user and network mobility.

During the internal self-organization, the following steps must be executed:

a. Start the creation of a network: the leader is identified to activate the creation of a society process using rule 7 through the rules 3 and 4 in order to search and find all devices capabilities. In our scenario the guide’s devices has the most power to control, trust, (re) configure and remain online until the finalization of the society. Therefore, it becomes the leader.

b. Start the definition of roles in the network: each device sends information that may be able to help the leader choosing the messenger (e.g., capability to provide interoperability between GPRS and WLAN), or on the choice of a guard (e.g., if it has the "wired network interface" enabled). However, the leader can have previous knowledge of the messenger and guard (e.g. it knew a set of stable connection points across the travel route).

c. Start the node joining and naming in the network: any time some worker (user or server) may be able to join into the society, but before this, the node needs to be aware of the policies defined by the leader that should be able to change policies dynamically using rule 7 in order to allow the meshing when possible. After agreement, each worker receives a name provided by the leader using rule 1. The previous step is executed together with this where passengers’ devices (users) are trying to join the network and to publish their software, hardware and networking capacities (e.g., VoIP application and video service).

d. Start sharing worker information: it is only at this step that the nodes can really publish, share and search for information. In last step the user was not authenticated by the leader, in other words, it did not have an EID (End-Pointer Identify or name).

e. Finalize the network setup: the internal configuration process can generate the updating of the information and the change of node roles (e.g. a node worker gets messenger role also) using rule 7. However, reconfiguration is possible at anytime as the messenger may trigger or announce that it is permanently or temporarily leaving of service and/or node.
During the external self-organization phase, the following steps are executed:

a. Start the external joining and naming process to allow access to the Internet: once the leader has finalized its internal self-organization process, it requests to join Internet as a society through the external leader and the exchange of policies. This is possible due to the guard capabilities used to reach other societies through external guards and to forward data and control message between societies according to rule 2. After the leaders agree with external policies (rule 7), the new society receives a naming by allocated to it by its higher leader in the society hierarchy (rule 1).

b. Start sharing information about the target society: next the society and its internal nodes can really start publishing, sharing and searching for information as in last step the nodes were only authenticated to exchange internal messages. This is due to the fact that their leader did have an EID (End-Pointer Identify or name) valid over the Internet scope. Its EID was valid only across the society.

c. Finalize the joining of the network in Internet: the external configuration process must minimize update traffic generated as a result of possible future change of guards. However, the process allows the updating of the aggregate information. The guard announces the leaving or joining of the societies in Internet.

3.3.1.2 Alternative flow

Should the process of discovering and choosing the role (messenger, guard, work and leader) be unsuccessful, the guide cannot start creating the trip (new network). In the normal situation (success), the guide can join other preexisting networks and afterwards invites its friends (passengers).

3.3.2 Communication with external endpoint

In this use case, we are going to consider the following requirements: mobility; internetworking; uploading/downloading data without interruption from/to user; support mobile objects; ease of attachment; and dynamic adaptability.
**Scenario Description:** During a group travel, tourists leave their hotel carrying heterogeneous devices and plan to visit different towns. The passengers go in and out from different regions, due to bus mobility. Along their travel, a user can talk to a friend in another country or region. Next, we examine how this can be achieved.

**Pre-Conditions:** The user must be made part of a stable society and before he starts communication with his friend, the user must define its stable information during the communication (e.g. the external endpoint involved in the session).

**Post-Conditions:** The user exchange data with this friend without interruption.

### 3.3.2.1 Normal flow

a. The user uses this control flow order to start communication with his friend. He transmits control messages over the path: user <-> messenger of the user <-> guard <-> friend according to rules 2 and 6.

b. The user announces his stable information (e.g. new end-point external communication) to its society leader. He transmits this control message using the path: user <-> user’s messenger <-> leader.

c. The leader stores the stable information;

d. The user and his friend transmit the data flow using the following bidirectional path: the external endpoint (friend) <-> guard <-> user’s messenger <-> user.

e. The user announces the end of the communication with his fiend, then the user requests the leader to remove stable information about that external communication using concurrently the following paths: user <-> user’s messenger <-> guard <-> friend and user <-> user’s messenger <-> leader
3.3.2.2 Alternative flow

In this scenario, there are three possible exceptional behaviors which can interrupt the normal flow: user mobility into the network, network mobility and user mobility across two or more networks.

When the user is moving into a network (stable society) the messenger must identify and signal this to the leader. Then the leader must signal this in turn to the guard that the user is now attached to a new messenger. Therefore, the external endpoint does not need to know about internal user mobility.

Another possible problem occurs when the user is moving between societies. The user announces to its previous leader about movement direction to new a society using the following path: user <-> messenger <-> leader. The leader announces to the friend about the new user’s guard using the following path: leader <-> guard <-> friend. The user receives data when it arrives at the new society over the path: friend <-> new guard <-> new messenger <-> user.

A third problem is when there is network mobility. When the leader creates a new society, the leader then associates a name to the new society (EID – endpoint identify) according to a region with a geographic limit. When there is mobility of a society, this can go out that limit, and then it gets a new name (new region and limits). In this context, the leader signals to the external endpoint the modification its EID using leader <-> guard <-> friend.

3.3.3 Searching information (or the discovery of network objects)

In this use case, we are going to work with following requirements: mobility; internetworking; distributed management; and scalability.

**Scenario Description:** The bus reaches a first city that has one of the biggest malls in this foreign place. Some passengers get off the bus in order to visit the mall and hey! It is time for some serious shopping! While walking around the mall, some are eager to find and buy given product(s) quickly, easily, fast enough and effortlessly. The shops’ own
information and marketing systems have been designed to locate and hunt for new potential customers and offer those services and deals on their products. Users entering the mall publish their profiles and add policies that may govern their objectives and characterize their targeted items. For example, a given user may want to receive sales information published by stores nearby as well as information about other lines of featuring products. The shopping mall maintains a server (search engine) for detecting new users at the mall who want to receive its advertisements and pages new ones to negotiate their approval.

**Pre-Conditions:** The user must make part of a stable society. Moreover, the user and server must use the same information model that defines the exchange and allows them to learn about each others resources.

**Post-Conditions:** The user has published his information. The server has found the user and transmitted information to him.

### 3.3.3.1 Normal flow

a. The user notifies the messenger and worker that want to leave the bus region (network) and to join a new region where the mall is located. Therefore, the messenger and worker contacted by user execute use case 3.3.4.

b. The user receives from worker the information model pattern in some description language such as XML. For example, the user receives questions about consumer preference, and then the user can indicate which products they like or not to buy, responding the worker.

c. Concurrently to steps a and b, the mall server using worker roles searches for new clients that want to receive advertising and promotional prices according to their consumer profile.

d. The worker finds the new clients and notifies the mall server.
3.3.3.2 Alternative flow

Under the normal flow, the search is performed locally by default. However, the worker (such as mall server) may want to do some research on external networks and servers. In last case, firstly the worker needs to know the EID of the region in which he wants to make the search, as well as he needs to know whether that society region supports the questions (e.g. It knows the answer to questions about the services and tourists of a specific region). Next the worker can send request messages with specific searches using the EID of the region.

There is a second alternative flow possible when the query for some service is unsuccessful within a local network. Firstly the algorithm executes the default query mode, but does not stop its execution when it fails to find the service sought. The worker needs to select the number of levels in the hierarchical network structure that may be queried before the algorithm is finalized. For example, the worker may select only one level in the execution. The algorithm executes the default query (locally) and next executes at level 1 (region of the leader of his leader) if it does not find the answer locally to its society.

3.3.4 Exchanging the Stable Society

In this use case, we are going to work with the following requirements: not to forward instability for all networks; allow allocation of resources; mobility; support mobile objects; policies; and security.

**Scenario Description:** When users move away from a source network (such as when being on a trip for example), they need to switch their connectivity domain, due to getting better connectivity from new ones. A new user to a domain accepts its policies and may even provide its own resources to this new domain. When joining a new society such as the mall one, the user benefits also from the sophisticated infrastructure of the mall and may even gain virtual money to buy at the mall’s stores.
**Pre-Conditions:** Firstly that user needs to be part of a stable society. Before the user exchanges network, the policies must be defined and executed in terms of the user and network. For example, the user and network previously know the QoS policies; the user knows his application QoS (ex. codec, frames per packet) and his preferences (ex. incoming message restrictions); the network knows the network QoS (ex. the control plane QoS supported – RSVP, MSVP, DSVP Diffserv or other), the transport QoS (ex. peak rate, network packet loss, mean delay) and its resources (ex. cache and transcoding service).

**Post-Conditions:** The user and network move to authenticate and guarantee policies.

### 3.3.4.1 Normal flow

This flow complements the one of use case 3.3.2, because it adds policies and the QoS view under both user and network mobility.

a. In terms of security, before the user joins or changes stable society (network), he needs to be authenticated by the leader in order to guarantee unique addressing/naming within the network. They transmit control messages using the path: user <-> messenger of the user <-> leader.

b. When a user joins or changes network, he needs to exchange policies with the leader through the messenger. For example, the messenger receives QoS transport and network policies that are currently supported by the society which in turn receives the user preferences and application QoS policies. The user and leader transmit control messages using respectively the following paths: user <-> messenger of the user and leader <-> messenger of the user.

### 3.3.4.2 Alternative flow

In this scenario, there are three possible exceptional behaviors which can interrupt the normal flow: user mobility into network, network mobility and user mobility across two or more networks.
When the user is moving into a network (stable society), then the messenger must transmit to the other messenger where the user will attach its policies (e.g. allocation resources to guarantee QoS application). Therefore, the external endpoint does not need to be aware of internal user mobility.

A second possible problem occurs when the user is moving between societies. The user announces to its previous leader its eminent move to a new society using the following path: user <-> messenger <-> leader. The leader tells the new leader the user’s policies using the following path: leader <-> guard <-> leader. The user receives control information when it arrives at a new society over the path: leader <-> new messenger <-> user and the messenger maps both user and network policies.

The third problem occurs when there is network mobility. When the leader creates a new society, the leader defines the society’s new policies. When there is mobility of a society, this can adapt its external control flow with the new guard’s policies. In this context, the leader signals to the external endpoint the modification of user policies over the path leader <-> guard <-> friend.

### 3.3.5 Adapting the communication

In this use case, we are going to work with following requirements: support for mobile objects; ease of attachment; dynamic adaptability; uploading/downloading data without interruption to/form a user; and multi-home.

**Scenario Description:** Before the traveling bus returning to the hotel, a user has started a conference. When arriving at the hotel room, she or he wishes to switch the video and audio input sessions to work over to his wireless Television (TV) device. However, she or he continues using her laptop’s camera for the video and audio input, while talking in the conference. The new session portion is shifted from her smart-phone to her TV device. Thus, the user can easily and freely use different devices and switch among them without service or data loss even while switching devices.
**Pre-Conditions:** Each device within the network has one EID. However, each user has one username that represents a set of final addressing (a set of devices) in order to support multi-home. In other words, each user can have one or more messengers (EIDs), but may have only one username.

**Post-Conditions:** The user receives information without interruptions.

### 3.3.5.1 Normal flow

a. When a user arrives at the hotel room, he finds a new messenger (wireless TV) according to use case for search information in Section 3.3.3.

b. The user decides to use the new messenger in order to improve QoS communication. Then he creates a message in order to announce to his friend that he will use another messenger together with the old messenger during conference. The message contains flow direction that each messenger will receive and/or send.

c. The user sends the message to his friend using the path: old_messenger <-> guard <-> friend;

d. At same time step c, the new messenger executes use case exchange the Stable Society in Section 3.3.4.

e. The user receives data by concurrently over the two paths user<->new messenger<->guard<->friend and user<->old messenger<->guard<->friend

### 3.3.5.2 Alternative flow

If the conference is with a friend located within the internal network, then the messages are exchanged directly among user, messengers and friend according to use case Communication with external endpoint in Section 3.3.2.
3.4 Concluding remarks

In this chapter, we presented our conceptual model for a future Internet and how one can apply to it a set of scenarios that involve some research areas such as self-organizing, discovering network objects, mobility and adapting communication. First, we introduced the stable society model, showing this as a set of rules and four roles: messenger, guard, worker and leader. The messenger and guard execute network activities such as forwarding packets and identifying broken communication pipes. The guards are seen as more stable individuals than messengers are responsible for translating network related protocols and hence allowing heterogeneity into a single society. The worker and leader execute overlay activities, storing and discovering information about mobile objects. Second, we developed general and main rules that both individuals must obey, but new rules can be added using dynamic policies, commonly known and used in academic literature.

In the following subsection the roles of each node were described and how their activities are executed while enforcing certain rules. Finally, the last section showed how the stable society concept will be applied in a set of scenarios that involve known networking challenges. Although this work touches onto many research areas, our main focus is on providing a solution for addressing, forwarding and information management involving heterogeneous and unstable objects in the next generation Internet.

Our approach is similar to the RBA architecture model [Braden et al. 2003], because we also used roles, however differently from RBA, our model has a fixed and small number of roles. We established four well-defined roles to deal with heterogeneous communication in overlay and network layers. Moreover, we adopt the stable society model to allow the integration and support of unstable nodes, such as mobile devices, while RBA concerns itself with only heterogeneous scenarios. Another important distinguishing feature of the proposed model is that it presents solutions for the problems related with the organization and maintenance of a set of unstable and heterogeneous nodes. While Plutarch [Crowcroft et al. 2003] looks at a homogeneous group of nodes as a context and defines an interstitial function to allow communication between different contexts, we proposed a novel approach for modeling a set of heterogeneous and unstable
nodes in order to provide homogeneous and stable group for the upper service and end-
to-end views.

In the next chapter, we will present our architecture in the context of the future Internet based on our conceptual model of the stable society concept in order to allow seamless communication for unstable scenarios. We will present our work by describing its main components.
Chapter 4

Architecture Proposal for Future Internet

4.1 Introduction

In this chapter, we are going to look for a coherent and consistent architecture for a next generation Internet in such a way that, it ensures the set of decisions and requirements depicted in Section 2.3. We chose to design and organize the network functionalities (addressing, forwarding and information management) using a role attribution mechanism according to features of each devices. This approach was used because we did not want to break the traditional Internet layering model. Therefore, the attribution of the four roles of a Stable Society would allow a better integration between our architecture and Internet.

In the following sections, firstly we are going to give a general view of the architecture. Next we are going to explain in detail the network tasks provided by the architecture (object addressing, information management, information forwarding and security) as well as how each role participates and executes its network task. This chapter also discusses how the new architecture was designed to support the separation and interrelation among addressing, naming and routing as well as to use indirection in routing in order to provide more security. Finally, Section 4.7 summarizes the chapter.
4.2 Overview of architecture

In Figure 4.1, we present the location of the Stable Society role within the OSI reference model. The messengers and guards roles execute functions which correspond to the network layer of the OSI model, motivating a definition of Messengers and Guards (MG) layer. The leaders and workers roles execute functions which correspond to transport layer and part of session and network layer, hence the definition of Leaders and Workers (LW) layer. Since the LW layer interferes with the Session layer of the OSI model, we specified the Session-SS as a modification of the Session layer. We can see that parts of the MG and LW layer execute functions of a network layer. These two layers (LW and MG) share information in order to organize topology and route information. However, the MG layer executes physical routing in the network layer, whereas LW executes virtual routing. Moreover, parts of the LW and SS-Session layers execute the management of the link states, see Section 4.5.1.

![Figure 4.1 – Overview of architecture based on roles](image)

The first layer (MG) may start forming a new society, when it identifies that there is a need for one, i.e. when there is no society with the same interest in its region. It may also perform the forwarding of messages according to a specific technology between two reachable nodes. The second layer (LW) is responsible for overlay routing which is independent of technology access. The LW layer performs the hierarchical organization of the society and also the network management. For example, this layer reorganizes the network when disconnections occur, influencing the third layer (SS-Session). Moreover, the messenger and guard must report information about the physical level and at the same
time the LW layer sends information about the logical level and uses both data in order to manage the network.

In the coming sections, we will be revisiting these layers and indicating how each entity within a Stable Society must be named or obtain an address, manage such information, route and allocate resources. Finally, we describe our security approach in this dynamic context.

4.3 Object addressing

Any future architecture must deal with the support of addressing which allows object mobility (mobile information and entities) handling, and cannot be limited to devices, while having persistent names independently of the underlying communication technologies. Further, we believe that addressing could be organized in order to induce the routing and avoid flooding in “irrelevant” areas. A nice way of achieving this, is to separate addressing and naming in order to provide scalability for growing communication systems, as well as to allow mobile objects to freely join, leave and remain in different locations.

Therefore our view is that there must be three independent addressing forms all part of the future Internet as can be seen in Figure 4.2. The Traditional Addressing (TA) will continue to be comprehended by traditional technologies. Additionally, the devices will get new routing addresses, when joining a new region, reflecting their locations. Hence this dynamic routing and addressing (RA) could induce the routes to forward messages to some region or other location. The mobile object will have persistent addressing, called a Mobile Object Address (MOA) that is both location and technology independent. MOA is a globally unique name like luciana, cin.ufpe.br or alice, but the specification of unique mechanism is not within the scope of this work.

| Human user or content or service = luciana | Mobile object address |
| Worker = Brazil PE 1 | Leader = Brazil PE 0 |
| messenger or guard = 192.168.0.14 | Routing address |
|                          | Traditional address |

Figure 4.2 - Stable Society addressing
These three addressing information consist of the unique Endpoint IDentifier (EID) for a client, RA and TA. The EID is sufficient to identify a route in the stable society, aiming to reduce the routing state information, messages and router table size, hence tackling one of the greatest challenges in the current Internet, namely, table route growth.

With regard to RA, we suggest that the leader (often seen as the most stable device in some region) must generate the RA hierarchically. The hierarchy and stability provided by leaders can give important improvements to Internet routing. In other words, under this scheme, a router can forward the packets according to the stability of other routers, using a dynamic small path length and proximity information.

In order to take into account heterogeneity, we propose that only LW (leaders and workers) nodes have RA and that the MG layer should forward data using specific technology dependent addressing. So the guards and messengers do not receive a virtual address (RA), nonetheless they still have their own physical address.

Figure 4.3 (a) presents an example consisting of ASs, ISPs and mobile Customers and Figure 4.3 (b) shows a possible address configuration for mobile entities using hierarchy based on prefix and postfix, geographical proximity through the names of regions in Brazil and roles from the stable society concept. For example, the society locality in Piauí is represented by a leader with PI address value. Further Figure 4.3 shows that an entity can be active in more than a single addressing level, receiving more than one index. For example, AS4 is the leader for the Pernambuco region, then it has PE as its address value. The PE leader is also identified as a worker from PE with identifier Recife as a more specific address value and decreasing one level in the naming hierarchy. Additionally, the AS4 is also the leader for Recife and has Recife address value.

Moreover, the RA addressing reflects the mobility among different regions. Figure 4.3 (a) shows the entities and how they move between networks. Figure 4.3 (b) presents the definition of the Stable Society over entities using cities’ names. For example, the group1 society in Figure 4.3(b) is an isolated society consisting of two workers and one leader with only one hierarchy level. Moreover both figures show group1 leaving the Caruaru society and going to join in JP society. When group1 society left Caruaru, it also releases the PE.Caruaru prefix and was identified only by postfix.
When group1 will join in the JP society, it is going to be identified as PB.JP.group1. However another society with group1 name value could already be in the PB.JP society. For this reason we will define some negotiation in the future to obtain the unambiguous addressing when group1 moves between societies.

(a) Scenario is consisting of AS, ISP and Customer entities  
(b) Addressing for AS, ISP and Customer entities

Figure 4.3 - Mobile scenario

Therefore, the proposed addressing supports simple naming and routing based on location and provides better performance among them. Further it ensures the decoupling among the three addressing (MOA, RA and TA). Additional MOA addressing can be represent naming for content, service and other information.

4.4 Information management

In this section, we are going to explain the network tasks that involve the organization of a society trough policies and information. Since information is important to maintain the network, we also describe our store and a discovery mechanism.
4.4.1 Self-configuration and policies

A self-configurable society is one that is run by its own leaders, workers, messengers and guards. Policies are rules which can help in this self-configuration. We defined three policies for such purpose: formation policy, maintenance policy and finalization policy.

The formation policy is the decision rule capable of influencing the choice of messengers, guards, leaders and workers aiming to create the society. This policy is executed by a node to attribute the role according to the node’s location. An example is shown in Figure 4.4, where a set of nodes create a society. First, the MG layer identifies if there is or not a society around the region. This layer can execute a routing protocol in order to discover and exchange messages with the neighbors. Second, if no neighbor is found and the node has the capacity to be a leader, then the new society is created. The leader chooses the nodes which have access to other societies to be guards. The nodes which have storage capacity may become workers and the others messengers. After selecting the roles, the leader provides addressing for all workers. A worker gets a unique RA that is built by the prefix’s leader and a unique postfix into the stable society. The leader has the authority to determine addressing, in order to avoid that a malicious node causes failure, illegal or incorrect addressing.

![Figure 4.4 - Society formation](image)

The maintenance policy aims to update the society over time, whenever there are changes in this set of nodes. This rule determines that whenever the leader intends to leave a society, it must choose a node with sufficient management capability in order to take leader’s role. The leader must execute a worker’s role before leaving the society. Further, the leader must provide an RA and a role for any new node joining the society.
The finalization policy is executed by a society when the leader leaves a society and there is no other node to take up this role. Then each node must release the RA and their roles.

These policies allow self-configuration that may be executed in vertical and horizontal society scales as illustrated in Figure 4.5. The vertical increase in the hierarchy organization of a stable society allows more specific location. For example, SS3 has more general location than SS3.2.1 which has more local location information. On the other hand, increasing horizontal levels in an organization allows a high number of neighbors to be close at this location, hence allowing unstable nodes to be managed by the same leader. In a stable society, as shown in Figure 4.5, a node can be a worker inside some society and a leader in another one. Moreover, the hierarchical topology offers better resilience against problems such as network splitting and mergers, because of its use of an adaptable addressing space.

![Figure 4.5 - Vertical x horizontal society graph](image)

### 4.4.2 Information discovery

We propose a discovery process using the Stable Society architecture in order to store and to find information storing MOA such as naming service. So, this process can be seen as the different resolution name process from a DNS mechanism present in today’s Internet, because the proposal does not make use of a central location for the management, location and name search. Additionally, the information discovery proposal provides a scalable mechanism consisting of mobile/fix devices with heterogeneous
capabilities, aiming to support mobile information and entities. Moreover, this process does not link name to the IP network address.

Firstly, we create an information discovery topology that consists of the RA hierarchy, interlacing several prefixes, where the low level has the prefix from a higher level society in addition to using a new prefix. This way, we guarantee that simple name space management uses one that reflects a group location due to a set of regions addressing.

According to the stable society architecture in Section 2.1, each entity gets different roles to implement information discovery (resolution naming): the worker stores the information, the leader manages the topology, the guards and messengers discover reachable paths to an RA already resolved by worker or leader. However, if a messenger does not know a reachable path, then it must create the unreachable path event and forward the notification to the leader of its stable society in order for this leader to execute self-configuration of society presented in Section 4.4.1.

The worker must store tuples (MOA:RA) in a repository that is managed by the leader as can be seen in Figure 4.6. When the worker intends to leave a society in Figure 4.6(b), it can send a message to a messenger notifying the leader about this. A worker can also request reducing its task according to its available resources and processing and communication capacities. A leader organizes the society and worker capacities in order to obtain good performance by taking into consideration workers with lower capacities (e.g., lower memory and processing). If the request is denied, the leader as shown in Figure 4.6(c) or even some worker as shown in Figure 4.6(d), will store naming information when the worker W2 is offline or when some worker has no capacity to store the information.
4.5 Information forwarding

In this section, we are going to explain our proposal of dynamic binding, and we present how our roles deal with dynamic routing.

4.5.1 Addressing relationship to route information

Before we explain our proposal of information forwarding, we will revisit the stable society architecture, in order to present how to bind dynamically each addressing, to control the communication and to allocate the resources for better communication services. Care must be taken to avoid mobility pitfalls in order to be capable of allocating and disconnecting resources without causing unnecessary high delays.

The [Ismailov et al.2006] work analyses three possible generic architectures for mobile information and entities. It discusses state management and the location of handover point to support mobility. Further, it associates the communication and application state with an endpoint, a communication flow and a communication session. The communication state contains all the necessary information required for restoring the
flow/session in the case of interruption; and the application state contains information about the status of the application specific to process the data.

Similarly to the approach in [Ismailov et al.2006], we associated two states with an endpoint, flow and session. However, our architecture is different, because we also used four roles: messenger, guard, worker and leader to handle mobility, heterogeneity and management that are introduced to control these states.

Figure 4.7 shows our proposed architecture based on the layered communication stack model and we present the relationship among addressing. Here, a logical communicating object can be dynamically remapped to a different network interface. The name for a mobile object or even the endpoint is constructed by the tuple: [Local Port#, ID] and ID and does not change even when there is mobility. In other words, it is the static addressing where the ID is MOA. The dynamic part of addressing are numbers 4 and 5 in Figure 4.7 that consist of a care-of-address (CoA – an address assigned to a particular interface at the current time) and the prefix of RA. Hence the re-mapping mechanism can be applied when the LW layer changes its location, leading to changes of the prefix; or when there are changes in the CoA and a rebinding between LW and MG is made.

Figure 4.7 – Stable society architecture
The communication state (if present) is entirely controlled by a session protocol such as Session Initiation Protocol (SIP) which does not need to worry about mobility and stabilities during object mobility. The application state is controlled by the application itself. An analysis of the handover points of this architecture (numbers 4 and 5 in Figure 4.7) shows the following capabilities:

- Handover can be performed within any network technologies and anywhere with or without mobility, because an ID is independent of network location and the network interface being used. In our example, the rebinding occurs at number 4 and the exchange of network interface at number 5.

- Handover between various interfaces can be performed simultaneously for all data flows. Separate data flows may coexist on different interfaces. This is shown at number 5 with the MG layer rebinding. These handovers could be subject to policies (number 5).

Yet in Figure 4.7, the worker and leader must implement the session layer to receive the delegation for storing application and communication states. However, the leader performs the storage only when the worker is absent from the society. The motivation for this state organization is to provide handover management per data flow and allowing handover between mobile objects.

A worker may leave a stable society and may notify it or not. When there is no such notification or when the worker intends to return back to its society, then the leaders and messenger must maintain states about worker communication and of network and application for a configurable duration.

In following, we present how the proposed roles of stable society are used in management of session. During a communication between two users such as Renata and Luciana, we propose three steps: discovery of information/user (MOA) in a virtual overlay that returns the RA value of Luciana; creation and maintenance of session and keeping the session; and recovery of the session when there are communication problems such as disconnections generated by mobility. In the information discovery step, the LW layer returns the virtual addressing according to a “geographical location” approach (RA). For example, we can see from Figure 4.3 (a) and (b) and assuming that Luciana’s
user (MOA) is at PE.Recife/group1 and Renata is at PB.JP, so the information discovery will return an RA of PE.Recife/group1 indicating the location of Luciana.

Yet in the same previous scenario, Figure 4.8 shows the creation and maintenance of a session. Both Luciana and Renata must request to each other the RA that represents the society location and the interface addressing of their guard. Next, they exchange a keep alive message in order to identify when MOA moves and changes its society addressing (RA).

![Figure 4.8 - Creation and maintenance of session](image)

Figure 4.8 shows the users trying to recover the session when someone or both change society, due to mobility of MOA. The MOA that changes RA must notify its session about the new RA and consequently the new guard network interface. After, both users must exchange keep alive messages in order to maintain the session.

![Figure 4.9 - Recovery the session](image)

Figure 4.9 shows the users trying to recover the session when someone or both change society, due to mobility of MOA. The MOA that changes RA must notify its session about the new RA and consequently the new guard network interface. After, both users must exchange keep alive messages in order to maintain the session.

Therefore, we can see in Figure 4.8 and 4.9 that only the mobile information affects the session control due to changing MOA, RA and TA and provide architecture based on roles presented in Figure 4.7. However, the next Section (4.5.1) is going to
show that our proposal supports mobile information and also mobile networks (AS, ISP and Customer).

### 4.5.2 Routing

The hierarchical topology based on roles provides many benefits to our routing. It is seamless for an end-point to join/leave the network as this structure avoids network-wide flooding and limits the control plane overhead. Further the architecture allows route aggregation executed by leaders and guards that have the control between external societies and can aggregate information before forwarding this to a society.

Although our approach provides these benefits, it does not imply in modifying existing routing protocols. Instead a layer (message encapsulation) is added. Consequently, it is possible for the messengers and the guards to explicitly provide the multi-path property to RA (addressing based on location) of workers and leader. This legacy physical routing helps the routing protocol to handle the transient presence of nodes while creating clusters using RA. Further, dynamic address allocation allows heterogeneous and also transient devices to work together in the network.

Three routing approaches are considered: legacy routing made by messenger and guard, routing based on location and routing based on names performed by workers and leader. The guard and the messenger have the task of forwarding messages in answer to requests and when they are using a proactive protocol. They need to discover reachable paths towards a given target node. They differ however in terms of scope: a messenger handles a local/internal routing and a guard has a global/external routing view. When the messenger is unable to find a reachable path to a target worker, it must forward messages to the leader of the stable society and notify the leader that it failed in finding a path itself.

A worker is required to maintain the virtual routes, route messages to the messenger and/or store relevant information related to a society. It may send a message to a messenger asking this to notify a leader in its behalf that it is exiting form a society, or request a task or storage reduction according to its current capacity state. Moreover, workers need to rely on the concept of indirection routing.
The leader must be connected to both a messenger and a guard. This is because the leader is responsible for the overall organization and bootstrap of a stable society. It may attribute and revoke guard, worker and leader roles according to its policies and device capabilities for each node. A priori, the leader only executes information routing and storage when the worker is offline or when there is no way to route and store the information by present participating member workers.

In the following, we present how the proposal roles of stable society are used in routing and two scenarios with mobility. The MOA movement could break the communication, but we are going to show in Figures 4.10 – 4.13 that the roles exchange information in order to support dynamic network, mobile devices or mobile information.

Firstly, Figure 4.10 shows that the MG layer consists of the messenger, guards and devices that exchange information with virtual overlay, where there are leaders and workers from LW layer. So, before user Luciana starts communication with some MOA service, she must request the LW layer to ask what is the RA that corresponds MOA.

![Diagram of MOA discovery](image)

**Figure 4.10 - Discovery MOA before to forward information**

Once Luciana knows the RA value returned by the LW layer, she can forward information as can be seen in Figure 4.11, because RA represents the location of MOA service. During the routing, both RA and MOA are always used, because when there is mobility of MOA, the LW layer can discover the new RA rebinding MOA, and when the network (society) moves the leader can reorganize the new location of the RA. Additionally, Figure 4.11 shows the data being forwarded into the society through messengers that route the information between heterogeneous technologies until the data
go to the guard of society (SS-Group1). When the data reaches the guard, it forwards
information among guards until the guard of the final society (SS-PB) that routes data
into the society using the messengers in order to send data to final MOA.

Figure 4.11 - Routing information among stable societies

Figure 4.12 shows a case involving mobile information, where both device and
information move. In this scenario, the mobility of device modifies the RA of MOA,
because MOA leaves SS-PB society and joins SS-PE. Firstly, Luciana tries to forward
data to MOA that was into the SS-PB society, but messengers and guards of SS-PB are
unsuccessful. In a second step, the SS-PB guard notifies the leader that the MOA left the
society and requests the new RA. In a third step, the LW layer returns the new RA to
rebind the MOA and finally the SS-PB guard sends data to the MOA that is in SS-PE
society.

Figure 4.12 - Routing mobile information (MOA) when devices are moving between
stable societies
Figure 4.13 shows a case involving mobile network, where the SS-PB society moves, but the information does not move into the society, so we do not need to rebind RA and MOA. However, we must update the RA routing table. In this scenario, Luciana tries to send data to MOA, but the routing table of SS-PE must be updated. In the second step the leader of SS-PE is notified and returns the new value for the RA, so the guard of SS-PE updates the routing table. Finally in a fifth step, the guard of SS-PE can forward the information to MOA.

Therefore, Luciana can route data even when there is network mobility as in Figure 4.12 and when there is mobility of the service or of the device that stores the service as in Figure 4.13.

**4.6 Security**

There are currently many types of threats in the Internet. One known problem is the denial-of-service (DoS) attack. It must be identified and dealt with in order to protect network stability. Security is traditionally important in any network and is crucial in many mobile networks mainly due to their vulnerability. Potentially any mobile object can access information through networks anywhere and at any time as well as the rapid dissemination of malicious actions – hence turning the network unstable.
A DoS attack may cause problems due to bad intended traffic leading to inappropriate resource allocation and consequently taking down important network services. In a DDoS attack, an attacker creates a zombie group that executes his commands creating a coordinated attack of a target. However, both threats can be countered by tracing back and filtering malicious traffic. The tracing back allows the identification of the potential locations of the attackers and the filtering allows the elimination of the attacking traffic. Existing methods for tracing back and filtering traffic such as those in [SNOEREN et al. 2001] and [KORKMAZ et al. 2007] could be installed into guards seen as the gateway to a stable society.

Although our proposal does not focus on security, it adopts a simple approach towards dealing with DDoS attacks. This is supported by the use of an indirection mechanism. When the guard is under attack, the leader can simply remove this guard and nominate another one, hence maintaining a robust packet forwarding mechanism. If some worker is attacked, then the leader can dynamically change the address of this worker in order to relieve it from unwanted traffic. If some leader is attacked, then another leader of same society can dynamically remove the leader that was attacked. However, if the society has only one leader, then an attack of leader may destroy the society.

4.7 Concluding remarks

In this chapter, we presented our architecture for a future dynamic Internet. We described the components of the architecture and the interrelationships between them. We discussed issues for object addressing (including naming), information management (involving self-configuration and information discovery), information forwarding (involving routing and mobility) and security. We structured the architecture based on stable society model that includes four roles. The architecture can be seen as a model where the leader exchanges messages with all roles in order to provide: an adaptive logical addressing scheme; a distributed and efficient location; the design of management structure; and the definition of a robust packet forwarding mechanism for heterogeneous with or without mobility.

Our object addressing approach is similar to LNA [Balakrishnan et al. 2004] as we defined a separation between service name and location, but differently our goal was
to allow mobility. We suggested a naming using RA for partial addressing. We adopted the use of prefixes and postfixes for hierarchical addressing as in NIRA [Yang 2003]. Unlike NIRA, we specified how to associate addresses and names to mobile objects as well as we offered a design mechanism for their resolution. The aim here is not to define a model using meta-data for characterizing mobile objects as in the SFR proposal [Balakrishnan et al. 2003], but to allow mobile and heterogeneous nodes to dynamically join and share information.
Chapter 5

Implementation

5.1 Introduction

We see the instability problems as mobility and dynamicity problems that generate many rebinds and control messages, because there is a stronger link between IP and name in today’s Internet. For this reason, the works [Stoica et al. 2002], [Clark et al. 2003], [Crowcroft et al. 2003] and [Balakrishnan et al. 2004] have been proposed the creation of overlay using Distributed Hash Table (DHT) to separate addressing IP (location) and name of content. However, they did not concern overlay constituted of mobile node that also is responsible for instability problem, blocking the joining of new nodes (scalability) and quality of communication, because the mobile nodes exchanges more control messages about disconnections than message of data.

Observing the instability problem for Internet architecture in previous chapters we presented a solution in Chapter 4 based on four roles. The proposal is composed of SS-Session, LW and MG layers that were designed to manage and to structure the future Internet in term of the addressing, forwarding and information management using theses roles.

Since other workers does not concern instability problem in overlay constituted mobile nodes, we reduced our scope for master research to manipulate this. We choose to implement the LW which is responsible for creating and management of the overlay, the structure of the addressing and the structure of the message forwarding. SS-Session was
not implemented, because we did not simulate the communication between users. The MG layer was simplified to forward only the messages, since we did not implement the negotiation to support heterogeneous devices through the announcing of capabilities.

The LW layer is responsible for overlay and it was planned to act independent from technology access. For this reason, we have planned to use DHT in the implementation of the LW layer framework. DHT algorithms are important for the information discovery process depicted in Section 4.4.2. This process consists of the storing and the searching mechanisms that are manageable, distributed and hierarchical, because they are based on stable society concept. Moreover DHTs are important for the creation of overlay networks, providing rapid information access and scalability for architecture.

This chapter describes our DHT proposal for implementation of LW layer. The implementation was based on Chord [Stoica et al. 2003] that is a DHT algorithm. Our proposal was called Stable Society over DHT (SSDHT). The Section 5.2 describes the Chord algorithm. The Section 5.3 describes SSDHT algorithm. Finally, we summarize the chapter with some concluding remarks.

5.2. Chord algorithm

Chord [Stoica et al. 2003] is a DHT algorithm that was designed to build overlay networks (virtual network) and to provide scalable lookup information for large-scale applications over that. All nodes in the virtual network have unique and location-independent overlay addresses. In general, the network address is mapped to a number of \( m \) bits through a hash function, as SHA-1 [FIPS 180-1], which distributes the object storage and retrieval in order to maintain the load balance. Theses identifiers shape a ring topology that consists of a linear space with less than \( 2^m \) nodes.

In this algorithm, the nodes take into account the wrap-around according to their unique addressing. Each node maintains a virtual link to its successor, which is the node directly following, proving the set of nodes ordered. With this structure, any node can route messages to any other node simply by each intermediate node that forwards the message to its successor until the destination is reached. This results path lengths of \( O(n) \),
in others words the routing is not scalable. However, Chord maintains additional information about others nodes in order to become the scalable routing. Theses data are stored in a routing table, called finger table, of $m$ entries. Each $i$ node has a finger table where the index $l$ of the entry $\in \{0,1, \ldots ,m-1\}$. Moreover, each entry is a virtual link to the first node in the address range $R_l(i) = [i+2l, i+2l+1] \mod m$.

As a result, each forwarding of message maintains $O(\log_2 n)$ state information in order to reduce to a half the resting virtual hops until destination. Additionally the average number of routing is $\log_2 n$. Therefore, Chord has two important characteristic. First, each node stores the information of little number of nodes, further the majority of them are on close nodes. Second, the finger table does not have enough information to directly route the message, because some nodes may forward to successor’s nodes.

However, the Chord must not be implemented directly in LW layer, because this algorithm does not provide rapid information access, neither is scalable in unstable environment with mobile nodes joining and leaving networks. Under such new highly volatile scenario, the use of a DHT leads to the generation of a large number of ring update messages hence turning routing very unstable and the overall system less scalable [Triantafillou 2003], [Heer et al. 2006], [Cramer et al. 2005] and [Futai et al. 2004]. To overcoming these problems, we proposed a new algorithm called SSDHT.

5.3. Stable Society over DHT

Stable Society over DHT (SSDHT) simplifies the design of an architecture that is compatible with instability generated by ubiquitous systems. Successfully we adapted Chord algorithm to the concepts of stable society in order to manage the addition and removal of nodes. Through the adequate combination of DHT algorithm and the stable society architecture we offered support to routing, addressing, and information storage in unstable and heterogeneous environments.

However, the implementation of SSDHT does not present the MG layer that is responsible for physical paths, because we are concerned with verifying overlay management in terms of instability generated by mobility and disconnections. Hence we presented only LW layer where the workers and leader build and manage virtual paths.
In following, we detailed the SSDHT algorithm in terms of addressing, topology, routing and searching information as well as management. The code of Stable Society Node is in Appendix B. This algorithm was implemented in Java to execute the measurement of some metrics specified in Chapter 6. Further the algorithm was implemented in C++ to be simulated in OverSim [Baumgart et al. 2007], because the measurement of scalability was not possible.

**Addressing**

SSDHT, similar at Chord, maps a given name (Mobile Object Address - MOA) or node (Route Addressing - RA) to one unique identifier (ID), which is location-independent due to overlay addresses. All \( n \) nodes build a virtual network, that shape a topology differently Chord. SSDHT has a hierarchical space shaped by regions with less than \( 2^m \) nodes which are considered heterogeneous in terms of capabilities, addressing and routing.

However, SSDHT manipulate RA and MOA differently. MOA is created equally in both algorithms, because we use hash function of Chord such as SHA-1 that generate a virtual address for each information name. In other hand, RA in SSDHT is different from Chord, because SSDHT needs to know the type of node (worker or leader) before it maps a given node to an RA. Moreover the RAs are divided into two parts: prefix and suffix. Each part has m-bit, totalizing a size of 2m-bit.

The node classified as a leader has zero in m-bit as suffix value and it determines the prefix of all the virtual addresses in stable society. The worker determines only its suffix and has the prefix of leader. In other words, the SSDHT addressing follows a hierarchy where the leaders determine the prefix. Further, all nodes have unique virtual addressing and the RAs are ordered within a ring as shown in Figure 5.1, where we exemplified four societies. Also in Figure 5.1 the N6.0 value is one example of the leader’s RA and the worker’s RA are N6.3 and N6.4. Moreover, MOA was exemplified for N6.1, N6.2, N5.9 and N2.1.
Figure 5.1 A logical circle of identifiers (ring) is consisting of 13 nodes and it is storing 4 keys

**Topology**

Similarly to Chord, a given key $k$ (MOA) is allocated to the first node whose identifier is equal or successor to $k$ in the ring of RAs space known as $successor(k)$. The regions or sub-rings are uniquely identified and the maximum number of nodes is $4^m$. However, the successor of a worker is necessarily a worker and the successor of a leader is always a leader.

Figure 5.2 illustrates the routing topology already established. The structure has two levels: internal and external sub-rings called Stable Societies. Although the connections are executed by guards in external level and by messengers in internal level, they are not in Figure 5.2, because both are below overlay of the workers and the leaders.
Figure 5.2 - Logical ring is representing connections for message forwarding

Figure 5.1 and Figure 5.2 depict the SSDHT with \( m = 3 \), where the ring has 13 nodes and stores 4 keys. The successor of node N6.4 is another worker whose RA is N6.3 and both are connected to leader N6.0. Following the Chord concept about successors, the node N6.3 stores N6.1 and N6.2, and the successor of N5.9 is the node N6.0. Similarly, the successor of N2.1 is N6.2.

The topology proposal was projected to avoid the propagation of structure updating when the nodes join and leave the network. However, the Chord algorithm was preserved when the leader join or leave the network, since it is essential to allow the existence of stable society. Therefore, when a society \( s \) joins the ring, \( s \) receives a set of MOAs values that precede \( s \). When the society \( s \) leaves the network, all MOAs are stored in successor of \( s \).

In other hand, the Chord algorithm was not preserved when workers join and leave the network, because there is the negotiation between the worker and the leader in order to avoid the global instability. The leader assumes worker’s role when some worker is not present and the worker requests its MOAs to leader, when it rejoins the society. For this reason, the SSDHT ensures more global stability, even when there are point instabilities in society.
Routing and searching information

Similarly to Chord, SSDHT maintains additional routing information called finger table to accelerate the routing and its information search. Although the finger table is a routing table with fixed value entries for leaders and workers, we modified it, because the node uses part of identifier to manipulate the tables. The leaders handle prefixes and the workers use the suffixes. Moreover, the leaders and workers use different parts of the key for routing. Leaders use prefix to route whereas the workers use suffix to route.

The finger table of the node \( n \) has \( m \) entries and values \((n+2^{k-1}) \mod 2^m\), where \(1\leq k \leq \)prefix value, for each entry index \( k \). If the node is a leader, then \( m \) is the number of bits used to represent the node’s prefix. If the node is a worker, then \( m \) is the number of bits used to represent the node’s suffix. For example, in Figure 5.3, each node that is classified as a leader or a worker, maintains \( m \) entries into its finger table. The \( i^{th} \) entry at node \( n \) has the RA of the first node that is the successor of \( n \) by at least \( 2^{i-1} \) in the society.

![Figure 5.3 SSDHT based on Chord algorithm](image)

The leaders use the finger table in order to perform the routing among stable societies, since they store the prefix of RA in this table. The workers use the finger table to execute the routing into the stable society, since they store the suffix of RA.

Other difference between worker and leader: the leaders handle another table called virtual_finger that has \( m.s \) entries, where \( s \) represents the number of unavailable workers. The \( i^{th} \) entry on absent worker \( n \) contains the RA of the first node successor of \( n \) by at least \( 2^{i-1} \) within the society.

This routing structure presents two important characteristics. First, the routing tables are small and each node stores lesser information than the other nodes. It also
knows more on closer nodes. Second, the node finger table does not have enough information to determine directly the successor of a given key (RA or MOA). A great advantage of the SSDHT over Chord is that ts (the size of SSDHT finger table) will always be smaller than tc (the size of traditional table of Chord), because the tc-ts value is zero only when the number of societies equals the number of nodes (unique ring).

In SSDHT, we also implemented the find_successor that use finger table to route and search information. However, we modified this operation differently when node n is a leader or a worker. When the node is a worker, it checks whether the id belongs to a given Stable Society - (prefix.0, prefix.2^m], because if the id is not within such interval, then a request is forwarded to the society’s leader. If the id is equal to n or within the interval [n, successor], then the find_successor operation is finalized. Otherwise, node n performs a lookup in finger table for a node whose identifier n’ immediately precedes id and the messenger forwards the message to the successor of n’.

When the node is the leader, its find_successor checks whether the id belongs to the stable society interval. Otherwise the leader performs an external routing. In other words, the leader searches the table finger for the leader n’ whose id immediately precedes n. Otherwise, the leader performs a lookup in virtual_finger for a node n’, whose identifier immediately precedes id. However, if n’ does not exist, then the leader is responsible for messages. We detailed the find_successor operation in term of routing, but the searching information follows the same steps.

Management
The creation of an SSDHT initiates when the leader invokes the create_stable_society operation that defines the guards and messengers for the new stable society and initializes the finger table with predecessor null.

The join operation is used by any node n to go in some society. This operation submits the requested message to immediate success of n and member of society. If the node is a leader, it configures the predecessor to null and initializes the finger table, notifying the guards and messengers about new node in the network. Moreover, the new leader asks for its successor that is the leader in other society. If the node is a worker, it
performs similar steps of leader. However, the worker does not find its success in external society, it finds its success within the society and the success is a worker.

The SSDHT has three operations (stabilize, check_predecessor and fix_finger) that are executed periodically for each node. Theses ensure the searched information and routing efficiency, because they maintain the society structure: each node and entry of table (finger and virtual_table) point to correct success even when the network is modified by new nodes (running join operation) and new societies (executing create_stable_society operation).

The stabilize operation allows the nodes to learn about new nodes that entered or left the network. The node \( n \) uses messengers or guards to know the predecessor of \( n.successor \) through the request message. The node \( n.successor \) responses a message which corresponds to the node \( x \). After the node \( n \) verify if \( x \) belongs to \( (n, n.successor] \) interval. Moreover, this notifies the node \( n \) to node \( n.successor \). It allows node \( n.successor \) changes its success to \( n \), only if the predecessor of node \( n.successor \) has null value.

The fix_finger operation executes find_successor(\( k \)) operation to each \( k \) entry of table (finger and virtual_table) in order to certify whether the success of \( k \) is correct.

The check_predecessor operation configures the predecessor of node to null value, when the node receives a notification from guard or messenger that identifies failures in connection with predecessor of node.

5.4 Concluding remarks

In this chapter, we presented a possible algorithm to implement the LW layer of Stable Society architecture. The new algorithm was called SSDHT that was based on Chord algorithm and Stable Society model. For this reason, we showed the basic concept of the Chord which is a scalable algorithm to route and search key. Although the Chord is inefficient for unstable environments, we modified it to support mobile nodes.

In addition, we applied a hierarchical model and different node functions defined by Stable Society model over Chord. Theses modifications resulted in the SSDHT
algorithm. Although we presented the SSDHT based on Chord, the Stable Society concept may be extended to build others DHT systems like Pastry [Druschel et al.]
Chapter 6

Analysis of the SSDHT Prototype

6.1 Introduction

In last chapter, we showed how the adaptation of the Chord algorithm. This adaptation was called Stable Society over DHT that structures and manages a set of devices and information based on the Stable Society concept. Since SSDHT has a set of functionality of our architecture proposed, sometimes, it is called prototype in this chapter. Furthermore, SSDHT was designed to consider a fundamental issue for the future Internet: how to lead with instability generated by mobility, and how to ensure the scalability of network mechanisms in unstable environments?

For this reason, the analysis of the SSDHT in terms of scalability in unstable scenarios is important and this chapter presents some relevant results, as well as discussing the evaluation of Chord and SSDHT. We verified Chord scalability in unstable environments and evaluated how much stability the Stable Society model over these DHTs improves Chord.

Among analytical modeling, simulation and measurement techniques for the analysis of systems, we chose to use measurement and simulation. We started measuring SSDHT and Chord over only one machine that allowed executed the creation of overlay network with until twelve nodes joining at same time. Therefore, the scalability could not be analyzed easily in measurement technique, due to several factors such the unavailability of physical resources and experiment time needed to evaluate several load
and environments scenarios. For more sophisticated scenarios, we carried out simulation-based experiments, where we simulated node mobility in networks with a large number of nodes. The simulator allows executing scenarios with until 1000 nodes. We did not execute scenarios with more nodes, because the simulator generated many events and each execution last more than one week.

The remainder of this chapter is organized as follows. In Section 6.2 we show the general and specific objectives of the evaluation. In Sections 6.3 (scenario 1), we executed SSDHT over the TCP stack of a single host. In Section 6.4 (scenario 2) we simulated cases with more nodes and different stability conditions. Finally, in Section 6.5, we summarize the chapter with some concluding remarks.

6.2 Objective

The general objective of this evaluation is to check if SSDHT is suitable for implementing the LW layer of the Stable Society architecture proposal. SSDHT needs to be scalable even when under high mobility and frequent node disconnection, which are characteristics present in unstable environments. We expected that SSDHT obtained better results than Chord, as the latter is unsuitable for unstable environments as confirmed by [Triantafillou 2003], [Heer et al. 2006], [Cramer et al. 2005] and [Futai et al. 2004]. Next, we enumerate specific objectives of the evaluation of SSDHT and Chord.

- Impact of node entry on a DHT network: what is the number of dropped messages, due to node entry on DHT? What is the resulting number of control messages (the control traffic rate)?
- Scalability related to size, mobility and traffic: we want to verify whether our designed prototype is suitable for use in a large wireless mobile network environment. What is the delivery ratio in DHT network when the number of mobile nodes that are communicating increases? How do changing mobility and traffic patterns to low, moderate and high affect such delivery rate?
- Overlay consistency: this is measured by looking at dropped messages. In unstable scenarios, there are churn ratios that result from nodes joining and leaving with or without mobility. Both behaviors generate disconnection leading
to an increase in the number of dropped messages (and even inconsistency in the DHT network).

- Analyze the quality and the adaptation of the routes: what is the request success ratio, in other words, the number of received packets over the number of sent ones?

### 6.3 Scenario 1 (Measurement)

The results reported in this scenario are from actual system measurements using the Chord and SSDHT algorithms in a PC. Although we know that a larger number of repetitions is important to have a high confidence level and a low variance coefficient, each experiment was run four times, because we had restrictions of equipment (test-bed) use and time.

In this scenario, the goal is to analyze the entry of 12 nodes (due to these restriction) and the impact caused on the network with respect to the stabilization time and number of control messages until the overlay becomes stable. The scenario with more number of nodes was executed by simulation that allows the execution of 1000 nodes at most. The network is stable when all virtual route tables have correct values, after inserting new nodes into the overlay. In our experiment, we inserted new nodes into the overlay and waited for the network to stabilize, next we repeated these steps until the network had 12 nodes. For the measurement, we choose to implement Chord in all the implementation domain using Java. Moreover we modified the Chord algorithm by adding the support for the Stable Society concept and the roles of leaders and workers. Next, the evaluation environment where the experiments were executed is described.

### 6.3.1. Evaluation environment

The evaluation environment for the first scenario was restricted to a single Personal Computer (PC) with the following configuration: operational system Windows XP, processor AMD Athlon (TM) XP 2000, 512MB RAM memory and an Ethernet network interface. We executed over a unique PC, due to the fact that the main objective was the
measurement of the control messages that are important to maintain the DHT structure and the distribution of nodes over more PCs made little difference in the results.

The measurement station executed a set of nodes simultaneously, which implemented only one overlay algorithm at a time (either Chord or SSDHT). The measurement backend was responsible for collecting data from these nodes and for creating a log containing the metrics shown in Section 6.3.2. Each node was executed in its own thread/process, using the TCP protocol to connect bidirectional point-to-point links in order to shape the overlay topology according to the algorithm being used.

The Figure 6.1 presents the topologies evaluated and steps used to analysis. We executed eight steps separately to measure each algorithm also independently and one by one. For measurement of Chord that has a ring topology, we execute two nodes joining at same time, after 4 nodes joining at same time, after 6 nodes and so on. For measurement of SSDHT that has a hierarchical and roles, we measured one leader and one worker joining at same time, after two leader and two workers, after three leaders and three workers, after four leaders and four workers, after four leader and six workers, after four leaders and eight workers and so on. Therefore, four societies with a total of 12 nodes were created at the end of the evaluation in the SSDHT test. The Chord test was formed by 12 homogenous nodes.

Figure 6.1 - Scenario varying number of nodes that join the network at the same time
6.3.2. Metrics and parameters

The following metrics are defined for the evaluation:

- The number of control messages (from all nodes) to maintain the DHT network. This metric is determined by analyzing the message counter log that was generated for both Chord and SSDHT algorithms.

- The time (from all nodes) to finalize the exchange of messages that maintain the DHT network. This metric is determined by a time counter log that was executed for both Chord and SSDHT algorithms. The time counter registers the time when the stabilization is identified, in other words, when all DHT nodes have completed and corrected their routing tables after all nodes join into the network.

The parameter used was the number of nodes that join in DHT network. It was used to each one theses metrics.

6.3.3 Results

In this section, we show the performance of Chord and SSDHT in terms of the impact of node entry on routing.

Figure 6.2 illustrates the performance of both SSDHT and Chord according to the number of nodes joining in the network. The messages, that were considered here, are responsible for maintaining routing table consistence against network changes. The number of messages until reaching network stability was plotted against the time for the network to become stable. We observe that SSDHT had smaller stability time and used fewer messages. In contrast, a network using the Chord algorithm suffers from high control traffic and time taken to reorganize itself after an instability caused by node entry. These results demonstrate the benefit of incorporating the stable society concept within DHT nodes, since SSDHT enables less stability time and less control data in network mainly when submitted to extreme instability. This fact is due to SSDHT adopting a hierarchical routing that limits the propagation of control messages (update message) to local nodes in the hierarchy without affecting global routing.
6.4 Scenario 2 (Simulation)

The results reported in this scenario are from a simulation system using the Chord and SSDHT approaches. Theses algorithms were simulated in OverSim [Baumgart et al. 2007] which is a framework for simulation of overlay networks. OverSim was chosen because it provides support for several structured and unstructured P2P protocols. Among them, the Chord protocol was already implemented. Moreover, OverSim provides an environment to evaluate mobility, churn, complex network and scalability (many nodes).

Each set of the parameters in Section 6.4.2 was ran 30 times to provide meaningful averaged values of the evaluation metrics, where the confidence level was 95%. We simulated mobility, scalability and churn properties. As Chord already was implemented in OverSim, we only modified Chord according by introducing the Stable Society concepts to create the SSDHT that were detailed in Section 5.3.

6.4.1 Evaluation environment

Network model: we used the Simple Underling network model [Baumgart et al. 2007] of OverSim, because this is a simplified network model for large scale analysis in the simulator. In the Simple Underling model, the nodes use a global routing table to forward data directly among overlay nodes and the underlay forwarding is simplified. For example, the simulator simplifies the delay measurement using a delay model presented
in Figure 6.3 instead to simulate forwarding events. Therefore, the end-to-end packet delay of an IP packet with length $P$ between overlay nodes $A$ and $B$ is calculated from the node’s distance where $c$ is constant. In addition each node has a logical access network characterized by bandwidth $b_n$, access delay $d_n$ and packet loss $l_p$, so that heterogeneous access networks can be simulated.

\[ d_e = d_1 + \frac{l_P}{b_1} + c \cdot \| A - B \|_2 + d_2 + \frac{l_P}{b_2} \]

Figure 6.3 – Delay model

This simplification is important to execute more number of nodes in viable time, since the simulator generates less number of event messages. Moreover, we studied three network sizes (small, medium and big) in order to measure our prototype in terms of the scalability. We model the small network as an infrastructure with 100 nodes; medium network was constituted of 500 nodes; and big network has 1000 nodes (maximum number of nodes for OverSim simulator).

Traffic model: we used an application that relied on key-based routing and that periodically sent messages to random overlay keys. Moreover, we studied four traffic (low, little moderate, moderate and high traffic loads) to understand the scalability in terms of messages forwarding. We model the low traffic when each node requests at an average rate of $1/20$ Hz (27MB); little moderate at an average rate of $1/10$ Hz (54 MB); moderate traffic at $1/5$ Hz (108 MB); and high at an average rate if 1 Hz (540 MB).

Mobility model: we changes the location coordinates and the IP address of a node, because these changes can generate mobility simulation event. For example changes of the nodes coordinates may increase or decrease the delay as we can observe in Figure 6.3. Moreover, we studied three mobile behaviors (slow, medium and rapid mobility), where the nodes change theirs coordinates and a there is a probability of 10% to generate disconnections or connections between two times. We model the slow mobility when the modification of coordinate may occur between 180 and 1200; medium mobility when connection or disconnections may exist between 60 and 600 seconds; and rapid mobility when this occurs between 20 and 2s.
6.4.2 Metrics and parameters

The following metrics are defined for the evaluation:

- Delivery ratio: we used UDP flows that do not retransmit dropped packets, so this metric is important to provide an indication of the quality of the routes defined and the protocol’s ability to deliver packets and consistently to resolve lookups. It is the number of received packets over the number of sent packet.

- Number of drop messages: it calculates the number of unsuccessful requests. It can show the overlay inconsistency, because each unsuccessful request (drop message) represents incorrect set of unsuccessful find.

- Number of control messages: this refers to the number of exchanged control messages used to organize the overlay. So it can illustrate in each algorithm how mobility may generate control messages on the network, due to associated maintenance overhead.

The following parameters were used for each metric: request interval time, network size and churn rate. This last parameter represents the interval time to fail and arrive at a node in the network. Our basic scenario consists of a small network (100 nodes uniformly distributed on coordinates x and y). All nodes may change theirs coordinates as well as arrive and fail according to the rapid mobility or high churn (connections or disconnections may occurs between 20 and 2s). Each node issues requests uniformly distributed on each 20s in others words low traffic.

We analyze the respective influence of parameters request interval, churn and network size on protocol performance. When varying the value of a parameter, the values of the other parameters remain fixed using their basic values.

Each data point represents an average over 10 independent simulation runs and is shown together with 95% confidence interval approximated. A run simulates 60 minutes of time (3600s), including the bootstrapping phase.
6.4.3 Results

We evaluated the prototype varying the network, traffic and mobility model. In section 6.4.3.1, we presented the results of evaluation for different traffic model. In section 6.4.3.2, we showed the results for three mobility model (also known as churn model, because there are connections and disconnections). Finally in section 6.4.3.3, we presented the results of different network models.

6.4.3.1 Varying offered load

In the first part of the experiments, we used the small network model with a high churn model and a varying traffic load. For increasing the traffic load, the inter-arrival time of requests was reduced. Requests were sent at an average per-node rate of 1/20 Hz (27MB), 1/10 Hz (54 MB), 1/5 Hz (108 MB) and 1 Hz (540 MB), representing respectively low, little moderate, moderate and high traffic loads. We aim to observe the reaction of overlay algorithms to the varying communication load.

The number of control messages for the deployments of Chord and SSDHT is constant, because we don’t vary the mobility. However, in Figure 6.4 (a), SSDHT presents lower number of control messages than Chord during the simulations in which both algorithms had the same request emission frequency. This last result is due to the clustered nature of the virtual structure from SSDHT that is used to limit the propagation of control messages.

The request success ratio of SSDHT is the highest (see Figure 6.4 (b)). It is approximately 100% for all traffic load and Chord had a 20% success ratio. This and the high number of signal messages in Chord can be explained by the cluster argument previously presented and by the trend in its consistency degree (see Figure 6.4 (c)).

The consistency overlay is the ability to correctly resolve key lookups without dropping messages depicted in Figure 6.4 (c) where we can see that the number of dropped messages in the case of a SSDHT network is lower. In contrast, the increase of application traffic injected into the network has a severe impact on Chord consistency that increases exponentially with a little interval between two request such as 1 second.
6.4.3.2 Varying churn

In the second experiment series, we used the small network, low traffic model and a varying churn model. The minimum churn rate was between 180 and 1200 seconds for a low instability, between 60 and 600 seconds for a network of moderate instability, and between 20 and 2 seconds for a network of high instability. We can see that the SSDHT algorithm is robust against network mobility. It has approximately 100% of the requests succeeding (Figure 6.5 (b)). When varying the mobility, Chord exhibits the same inconsistency problems as in the load test (see Figure 6.5 (a)–2(c)), and the success ratio drops to 20% for the highest mobility case. In Chord, the number of control messages and the number of dropped messages increase with higher mobility. Hence, SSDHT also provides better behavior, when there are intense mobility and high traffic load.
6.4.3.3 Varying network size

In the last set of experiments, we used the low traffic model, high churn model and a varying network model (small with 100 nodes, medium with 500 nodes and big network with 1000 nodes).

With an increased network size, the success ratio of SSDHT (see Figure 6.6(b)) was yet larger than Chord, but SSDHT showed a little decrease in performance (see Figure 6.6(a)-(c)) and Chord presented a slight improvement of its success ratio during the increase of network size. We observe that the decreasing performance of SSDHT is a consequence of its simplified decision mechanism implemented to execute the simulation. The decision mechanism implemented in SSDHT is based on proximity radius of nodes that was a fixed value for any size of network. So, when the network
increases SSDHT presents lower society control and a lower success ratio. However the SSDHT always presents better performance than Chord, because the SSDHT has a decision mechanism, although using a simple method.

On other hand, Chord presents a growth line when the network is increasing, because we maintain the same mobility model during the experiment. In other words, we model the simulation using a normal function that sends mobility event according to a 10% probability for a node to move at random. So, when the network increases Chord presents a higher number of nodes that do not move, hence explaining the growth of performance line in Chord. In other words, in these experiments there is a number of mobile nodes that practically remain fix.

Figure 6.6 - Average performance metrics for a varying size in mobile networks

6.5 Concluding remarks

We investigated the SSDHT and Chord algorithms in several unstable scenarios: several nodes that join the network at the same time, varying the mobile node ratio, increasing
the offered load and varying the network size. For all investigated environments, the SSDHT approach presented better performance, mainly when we increased the instability. For example the percentage of succeed discovery operations was more than 90% when we varied the traffic loads, degrees of mobility, and network sizes.

On other hand, Chord presented a decrease in the number of successfully resolved lookups and high number of control messages that are signs of possible congestion collapse. They are the result of overlay inconsistencies caused by packet loss due to mobility or transmission errors that was identified in results of Section 6.3.3 and 6.4.3.

Moreover, we conclude about the importance to study and to develop the decision mechanism in the future in order to improve the stable society concept and the proposed architecture. Our simplified mechanism is based on the use of a static number for the proximity radius among nodes, further the creation of society proceeds when the node verifies whether there is some society near; if there isn’t any society then the node creates a new society. Thus, SSDHT obtained a small decreasing performance when the network increases. However, for all the experiments the SSDHT does not exhibit significantly signs of congestion.
Chapter 7

Conclusions and Future Directions

7.1 Research contributions

In this work, we extended the state of the art in three main directions. Firstly, we proposed a new paradigm, named Stable Society, to attribute roles over devices according to the features of a network in terms of connectivity. Secondly, we proposed the use of our paradigm to address, manage and forward messages in Internet. We showed how our roles over devices can provide more scalability and flexibility for networks, mainly when there are connections intermittent, that are generated by mobility. Our third contribution is the architecture prototype that presented how to apply the roles for addressing, forwarding and management of the overlay network. We also analyze an overlay network in terms of the mobility and scalability, when the network has or not our roles.

The main goal of this work was to stabilize the connectivity infrastructure of dynamic network. We designed a new network model that contains roles to structure the connections and rules to define how to apply the roles over devices. We specified four roles: messenger, leader, worker and guard. Messenger allows more heterogeneity in network, due to translate capabilities. Guard allows more stability, due to be fix point. The worker is able to store information and the leader manages all roles. The rules were specified to inform how to address, forward, store and announce information that can be a device, service, content or user. Although the decision mechanism is not our focus, we
identified in section 3.2.2 a set of important actions that could be designed and used by a leader to improve the management of the network.

Using the Stable Society model, we proposed for the future Internet an architecture which is role-based in order to structure and manage the mobile AS, ISP and Customers, and also deals with the problems concerning generation of mobile information and heterogeneous devices of the future Internet. Another distinguishing feature of the proposal is that the architecture defined what roles are used to organize the network using two level: i) network level, which are constituted of messengers that translate information between routing technologies and of guard that is the most stable device inside the network, ii) virtual level, which are constituted of the worker that stores and finds the information about network and of the leader that manages all network using decision mechanism. Other key issue of our system is the use of three addressing (TA, MOA and RA) together role to decouple location, name and technology information.

We implemented the overlay of the architecture using Stable Society concept over a DHT algorithm that was called SSDHT. It provides a mechanism to disseminate and to find information, even in unstable environment constituted of mobile nodes that join and leave the DHT. The advantage of our approach is that any traditional DHT may be modified, becoming scalable in unstable environment. We showed the low performance of Chord (a traditional DHT) and the analysis of the SSDHT which presented to be scalable in mobile scenarios. This also contributed current proposals of Internet architecture, since many proposals for Internet used DHT such as [Stoica et al. 2002], [Clark et al. 2003], [Crowcroft et al. 2003] and [Balakrishnan et al. 2004], but they do not resolve dynamic problem in DHT algorithms. So these works could choice SSDHT in order to provide theirs solution in unstable scenarios.

As discussed in this master dissertation, Stable Society approach is a good model to structure and manage dynamic networks. Since, we compare it to other DHT solution (Chord algorithm) and the DHT with ours roles and rules. The results show that our proposal was better than traditional DHT, mainly when we increased the instability (the traffic loads, degrees of mobility, and network sizes). For example the percentage of success discovery was more than 90% for all unstable scenarios.
7.2 Future work

This work has opened a large spectrum for new problems to be solved and some activities that aren’t executed in this master dissertation. They are listed below:

- We support the idea that definition of roles, proposed in this work, should be extended to other advance feature of network as content adaptation and context modification.

- The decision and negotiation mechanism is an important area that must be designed to improve our architecture. The leader entities specifically need them as they were simplified since they were not the focus of this work.

- Security must be studied in our architecture and a conceptual model in the context of global computing and unstable scenarios characterized by heterogeneous nodes, uncertainty and a large number of previously unknown roaming entities that join and leave network. Moreover, the future work must concern malicious nodes that intend to break the societies, so the authentication mechanism is one among various security research that must specified and implemented in our architecture.

- We intend to implement and evaluate the messenger and guards together with entities that are implemented (leader and workers) in order to verify the performance and flexibility of our solution to scenarios with various routing protocol. However, firstly the decision and negation protocol must be implemented, because it is important to define what messengers and guards are in societies. Moreover, this mechanism must every time update the society, because we are dealing with dynamic networks.

- We considered that all identifiers are unique to simplify the specification, implementation and evaluation of the proposed solution. Future work must consider the possible repetition of identifiers in different societies.
Bibliography


Appendix A. List of Publications

This list contains all publications which were approved during master research. The following is a list of the articles published that is indirectly relation to this master:


We have an article published that is directly relation to this master:


We have a paper published as result of Augmented Reality discipline:

Appendix B. Code of Stable Society Node

/**
 * @file Node.cc
 * @author Luciana Oliveira
 */

#include <IPAddressResolver.h>
#include <IPvXAddress.h>
#include <InterfaceTable.h>
#include <IPv4InterfaceData.h>
#include <InitStages.h>
#include "Node.h"

Define_Module(Node);

void Node::initializeOverlay(int stage){
    // because of IPAddressResolver, we need to wait until interfaces are registered, address auto-assignment takes place etc.
    if(stage != MIN_STAGE_OVERLAY)
        return;

    // fetch some parameters
    useCommonAPIforward = par("useCommonAPIforward");
    successorListSize = par("successorListSize");
    joinRetry = par("joinRetry");
    stabilizeRetry = par("stabilizeRetry");
    joinDelay = par("joinDelay");
    stabilizeDelay = par("stabilizeDelay");
    fixfingersDelay = par("fixfingersDelay");
    aggressiveJoinMode = par("aggressiveJoinMode");

    role = parentModule()->parentModule()->par("role").stringValue();
    // int that indicate what role of the node
    leader_ip = parentModule()->parentModule()->par("leader_ip").stringValue();
    setID();
    keyLength = OverlayKey::getLength();
    missingPredecessorStabRequests = 0;
    missingPredecessorStabRequestsLL = 0;
    missingSuccessorStabResponses = 0;
    missingSuccessorStabResponsesLL = 0;

    // statistics
    joinCount = 0;
    stabilizeCount = 0;
ffingersCount = 0;
notifyCount = 0;
newsuccessorhintCount = 0;
joinBytesSent = 0;
stabilizeBytesSent = 0;
notifyBytesSent = 0;
ffingersBytesSent = 0;
newsuccessorhintBytesSent = 0;
reconnect = false;

// find friend modules
findFriendModules();

// add some watches
WATCH(predecessorNode);
WATCH(predecessorNodeLL);
WATCH(thisNode);
WATCH(bootstrapNode);
WATCH(joinRetry);
WATCH(missingPredecessorStabRequests);
WATCH(missingPredecessorStabRequestsLL);
WATCH(missingSuccessorStabResponses);
WATCH(missingSuccessorStabResponsesLL);
join_timer = new cMessage("join_timer");

// self-messages
if(role.compare("LEADER") == 0){
    stabilize_timer = new cMessage("stabilize_timer");
    ffingers_timer = new cMessage("ffingers_timer_under_node");
    ffingers_timer_under_node = new cMessage("ffingers_timer_under_node");
    stabilize_timer_under_node = new cMessage("stabilize_timer_under_node");
} else if(role.compare("WORKER") == 0){
    ffingers_timer_under_node = new cMessage("ffingers_timer_under_node");
    stabilize_timer_under_node = new cMessage("stabilize_timer_under_node");
}

// initialize chord protocol

Node::~Node(){
    // destroy self timer messages
    if(role.compare("LEADER") == 0){
        cancelEvent(join_timer);
        cancelEvent(stabilize_timer);
        cancelEvent(ffingers_timer);
        cancelEvent(stabilize_timer_under_node);
        cancelEvent(ffingers_timer_under_node);
        cancelEvent(virtualWORKERS_timer);
        delete ffingers_timer_under_node;
        delete stabilize_timer_under_node;
        delete virtualWORKERS_timer;
        delete join_timer;
        delete stabilize_timer;
    } else if(role.compare("WORKER") == 0){
        cancelEvent(ffingers_timer);
        cancelEvent(stabilize_timer);
        cancelEvent(virtualWORKERS_timer);
        delete stabilize_timer;
        delete virtualWORKERS_timer;
        delete stabilize_timer;
    }
delete fixfingers_timer;
} else if (role.compare("WORKER") == 0) {
    cancelEvent(join_timer);
    cancelEvent(stabilize_timer_under_node);
    cancelEvent(fixfingers_timer_under_node);
    delete fixfingers_timer_under_node;
    delete stabilize_timer_under_node;
    delete join_timer;
}

void Node::receiveChangeNotification(int category, cPolymorphic* details) {
    Enter_Method_Silent();
    if (category == NF_HOSTPOSITION_UPDATED) {
        // get new ip address
        thisNode.ip = IPAddressResolver().addressOf(
            parentModule())->get4();
        joinOverlay();
    } else if (category == NF_OVERLAY_NODE_LEAVE) {
        reconnect = true;
        // send msg to leader in order to notify the leaving
        SLeaveNodeCall* call = new SLeaveNodeCall("SLeaveNodeCall");
        call->setOperario(thisNode.key);
        call->setPredNode(predecessorNodeLL);
        call->setSucNode(successorListLL->getSuccessor(0));
        call->setLength(LEAVECALL_L(call));
        sendRpcMessage(bootstrapNode, call);
        // remove this node from the bootstrap list
        bootstrapOracle->removePeer(thisNode);
        // destroy self timer messages
    }
}

void Node::setId() {
    if (role.compare("LEADER") == 0) {
        thisNode.key =
            (OverlayKey::sha1(const_cast<char*>(thisNode.ip.str().c_str()))).prefix();
    } else if (role.compare("WORKER") == 0) {
        OverlayKey prefix = (OverlayKey::sha1(const_cast<char*>(leader_ip.c_str()))).prefix();
        OverlayKey postfix =
            (OverlayKey::sha1(const_cast<char*>(thisNode.ip.str().c_str()))).postfix();
        thisNode.key = (prefix + postfix);  
    }
}

void Node::joinOverlay() {
    changeState(INIT);
    changeState(BOOTSTRAP);
}

void Node::changeState(int toState) {
    // Defines tasks to be executed when a state change occurs.
    switch (toState) {
    case INIT:  
        state = INIT;
        break;
// remove current node handle from the bootstrap list
if(!thisNode.key.isUnspecified()) {
    bootstrapOracle->removePeer(thisNode);
}

if (thisNode.key.isUnspecified()) {
    // verify whether the node has role value equals to "LEADER"
    setID();
callUpdate(thisNode, true);
}

if(role == "LEADER"){
    // initialize predecessor pointer
    predecessorNode = NodeHandle::UNSPECIFIED_NODE;
    predecessorNodeLL = NodeHandle::UNSPECIFIED_NODE;
} else{
    // initialize predecessor pointer
    predecessorNodeLL = NodeHandle::UNSPECIFIED_NODE;
}

// initialize finger table and successor list
initializeFriendModules();
updateTooltip();

parentModule()->parentModule()->bubble("Enter INIT state.");
break;

case BOOTSTRAP:
    state = BOOTSTRAP;

    // initiate bootstrap process
    cancelEvent(join_timer);
    // workaround: prevent notificationBoard from taking
    // ownership of join_timer message
    take(join_timer);
scheduleAt(simulation.simTime(), join_timer);

    parentModule()->parentModule()->bubble("Enter BOOTSTRAP state.");

    // find a new bootstrap node and enroll to the bootstrap list
    // the bootstrap is the leader in access network
    if(role == "LEADER"){
        bootstrapNode = bootstrapOracle->getBootstrapNode();
        if (bootstrapNode.isUnspecified()) {
            // create new cord ring
            bootstrapNode = thisNode;
            changeState(READY);
            updateTooltip();
        }
    } else if(role == "WORKER"){
        bootstrapNode = NodeHandle((OverlayKey::sha1(const_cast<char*>(leader_ip.c_str()))).prefix(), IPAddress(leader_ip.c_str()), 1024);
        bootstrapNode.key.setSpecified();
    }
    break;
case READY:
    state = READY;
    bootstrapOracle->registerPeer(thisNode);

    if(role == "LEADER"){
        // initiate stabilization protocol
        cancelEvent(stabilize_timer);
        scheduleAt(simulation.simTime() + stabilizeDelay, stabilize_timer);
        // initiate finger repair protocol
        cancelEvent(fixfingers_timer);
        scheduleAt(simulation.simTime() + fixfingersDelay, fixfingers_timer);
        cancelEvent(virtualWORKERs_timer);
        scheduleAt(simulation.simTime() + stabilizeDelay, virtualWORKERs_timer);

        cancelEvent(fixfingers_timer_under_node);
        scheduleAt(simulation.simTime() + fixfingersDelay, fixfingers_timer_under_node);
        cancelEvent(stabilize_timer_under_node);
        scheduleAt(simulation.simTime() + stabilizeDelay, stabilize_timer_under_node);
    }else if (role=="WORKER"){
        cancelEvent(fixfingers_timer_under_node);
        scheduleAt(simulation.simTime() + fixfingersDelay, fixfingers_timer_under_node);
        cancelEvent(stabilize_timer_under_node);
        scheduleAt(simulation.simTime() + stabilizeDelay, stabilize_timer_under_node);
    }

    parentModule()->parentModule()->bubble("Enter READY state.");
    break;
}

setReadyIcon(state == READY);
}

void Node::handleTimerEvent(cMessage* msg){
    // catch JOIN timer
    if (msg->isName("join_timer")) {
        handleJoinTimerExpired(msg);
    }

    // catch STABILIZE timer
    else if (msg->isName("stabilize_timer")) {
        handleStabilizeTimerExpired(msg);
    }

    // catch STABILIZE timer
    else if (msg->isName("stabilize_timer_under_node")) {
        handleStabilizeTimerExpiredUnderNode(msg);
    }

    // catch FIX_FINGERS timer
    else if (msg->isName("fixfingers_timer_node")) {
        handleFixFingersTimerExpiredNode(msg);
    }
}
// catch FIX_FINGERS timer
else if (msg->isName("fixfingers_timer_under_node")) {
    handleFixFingersTimerExpiredUnderNode(msg);
}

// catch FIX_FINGERS timer
else if (msg->isName("virtualWORKERs_timer")) {
    handleVirtualWORKERsTimerExpired(msg);
}

// unknown self message
else {
    error("Node::handleTimerEvent(): received self message of "
          "unknown type!");
}
}

void Node::handleUDPMessage(BaseOverlayMessage* msg){
    SocietyMessage* chordMsg = check_and_cast<SocietyMessage*>(msg);
    switch(chordMsg->getCommand()) {
    case NEWSUCCESSORHINT:
        handleNewSuccessorHint(chordMsg);
        break;
    default:
        error("handleUDPMessage(): Unknown message type!");
        break;
    }
    delete chordMsg;
}

void Node::handleRpc(BaseCallMessage* msg){
    BaseRpcMessage* _msg = dynamic_cast<BaseRpcMessage*>(msg);
    NodeHandle srcNode = _msg->getSrcNode();
    if (state != READY && (srcNode.key.postfix() ==
                            (OverlayKey::ZERO).postfix() & role == "LEADER")
        ) {
        delete msg;
        return;
    } else if (state != READY && role == "WORKER") {
        delete msg;
        return;
    }

    // delegate messages
    RPC_SWITCH_START( msg )
    // RPC_DELEGATE( <messageName>[Call|Response], <methodToCall> )
    RPC_DELEGATE( SJoin, rpcJoin );
    RPC_DELEGATE( SNotify, rpcNotify );
    RPC_DELEGATE( SNotifyUnderNode, rpcNotifyUnderNode );
    RPC_DELEGATE( SStabilize, rpcStabilize );
    RPC_DELEGATE( SStabilizeUnderNode, rpcStabilizeUnderNode );
    RPC_DELEGATE( SFixfingersNode, rpcFixfingersNode );
    RPC_DELEGATE( SFixfingersUnderNode, rpcFixfingersUnderNode );
    RPC_DELEGATE( SLeaveNode, rpcLeaveNode );
    RPC_SWITCH_END( )
}

void Node::handleRpcResponse(BaseResponseMessage* msg, int rpcId,
                              simtime_t rtt){
    RPC_SWITCH_START(msg)
RPC_ON_RESPONSE( SJoin ) {
    handleRpcJoinResponse(_SJoinResponse);
    EV << "Join RPC Response received: id=" << rpcId
         << " msg=" << *_SJoinResponse << " rtt=" << rtt << endl;
    break;
}
RPC_ON_RESPONSE( SNotify ) {
    handleRpcNotifyResponse(_SNotifyResponse);
    EV << "Notify RPC Response received: id=" << rpcId
         << " msg=" << *_SNotifyResponse << " rtt=" << rtt << endl;
    break;
}
RPC_ON_RESPONSE( SNotifyUnderNode ) {
    handleRpcNotifyUnderNodeResponse(_SNotifyUnderNodeResponse);
    break;
}
RPC_ON_RESPONSE( SStabilize ) {
    handleRpcStabilizeResponse(_SStabilizeResponse);
    break;
}
RPC_ON_RESPONSE( SStabilizeUnderNode ) {
    handleRpcStabilizeUnderNodeResponse(_SStabilizeUnderNodeResponse);
    break;
}
RPC_ON_RESPONSE( SFixfingersNode ) {
    handleRpcFixfingersNodeResponse(_SFixfingersNodeResponse);
    break;
}
RPC_ON_RESPONSE( SFixfingersUnderNode ) {
    handleRpcFixfingersUnderNodeResponse(_SFixfingersUnderNodeResponse);
    break;
}
RPC_SWITCH_END( )

void Node::handleRpcTimeout(BaseCallMessage* msg, const NodeHandle&
    dest, int rpcId){
    RPC_SWITCH_START(msg)
    RPC_ON_CALL( FindNode ) {
        break;
    }
    RPC_ON_CALL( SJoin ) {
        break;
    }
    RPC_ON_CALL( SNotify ) {
        break;
    }
    RPC_ON_CALL( SNotifyUnderNode ) {
        break;
    }
    RPC_ON_CALL( SStabilize ) {
        break;
    }
    RPC_ON_CALL( SStabilizeUnderNode ) {
        break;
    }
}
bool Node::isResponsible(const OverlayKey& key) {
    if (key.isUnspecified()){
        error("Node::isResponsible(): key is unspecified!");
    } if (state != READY && (key.postfix() ==
        (OverlayKey::ZERO).postfix() & role == "LEADER")) { return false;
} else if (state != READY & role == "WORKER"){
    return false;
} else if(key.prefix() != thisNode.key.prefix() & role == "WORKER"){
    return false;
} else if(key.postfix() != (OverlayKey::ZERO).postfix() && role == "LEADER"&
        key.prefix() == thisNode.key.prefix()) {
        // if this is the first and only node on the ring, it is
        // responsible
        // verify if underlay of leader is the responsible by key
        if(virtual_fingerTable->isResponsable(key)){
            return true;
        } else if(predecessorNodeLL.isUnspecified()&
            successorListLL->isEmpty()){ return true;
} else if(predecessorNodeLL.isUnspecified()& !successorListLL->isEmpty()){
            return false;
        // is the message destined for this node?
        // verify if key is to leader through the look for inter layer
    } else{
        return true;
    }
} else if(role.compare("WORKER") ==0){
    if(predecessorNodeLL.isUnspecified()){if(successorListLL->isEmpty()) {
        return true;
    } else {
        return false;
    }
} // is the message destined for this node?
if(key.isBetweenR(predecessorNodeLL.key, thisNode.key)) {
    return true;
} else{
    return false;
} }
else
   // verify in each node layer, if it is the responsible by key
   if (predecessorNode.isUnspecified()) {
      if (successorList->isEmpty()) {
         return true;
      } else {
         return false;
      }
   }

   // verify if the key is to leader or worker according to its layer
   if (key.isBetweenR(predecessorNode.key, thisNode.key)) {  
      return true;
   } else {
      return false;
   }

NodeVector* Node::findNode(const OverlayKey& key, BaseOverlayMessage* msg) {
   NodeVector* nextHop = new NodeVector(1);
   if (state != READY && (key.postfix() == (OverlayKey::ZERO).postfix()) && role == "LEADER") {
      return nextHop;
   } else if (state != READY && role == "WORKER") {
      return nextHop;
   }

   // if key is unspecified, the message is for this node
   if (key.isUnspecified()) {
      nextHop->push_back(thisNode);
   } else if (role == "WORKER" && key.prefix() != thisNode.key.prefix()) {
      if (bootstrapNode.isUnspecified()) {
         bootstrapNode = NodeHandle((OverlayKey::sha1(const_cast<char*>(leader_ip.c_str()))).prefix(), IPAddress(leader_ip.c_str()), 1024);
         bootstrapNode.key.setSpecified();
         cout << "bootstrapNode.isUnspecified() " << bootstrapNode.isUnspecified() << endl;
      }
      nextHop->push_back(bootstrapNode);
   } else if (role == "LEADER" && key.postfix() != (OverlayKey::ZERO).postfix() && key.prefix() == thisNode.key.prefix()) {
      // search in list of absent nodes
      NodeHandle ret = virtual_fingerTable->findNode(key);
      if (!ret.isUnspecified()) {
         nextHop->push_back(ret);
         return nextHop;
      } else {
         if (predecessorNodeLL.isUnspecified()) {
            if (successorListLL->isEmpty()) {
               // the message destined for our successor
               nextHop->push_back(thisNode);
               return nextHop;
            }
         }
      }
   }
```cpp
if (key.isBetweenR(predecessorNodeLL.key,thisNode.key)) {
    // the message destined for our successor
    nextHop->push_back(thisNode);
    return nextHop;
}

// if there is not in list of absent node, then flooding by worker
if (key.isBetweenR(thisNode.key,successorListLL->getSuccessor().key)) {
    // the message destined for our successor
    nextHop->push_back(successorListLL->getSuccessor());
    return nextHop;
}
else {
    // find next hop with finger table
    NodeHandle tmpNode = closestPreceedingNode(key);
    if (!tmpNode.isUnspecified()) {
        nextHop->push_back(tmpNode);
    }
}
return nextHop;
}
}
```
nextHop->push_back(thisNode);
return nextHop;
}
else if (key.isBetweenR(thisNode.key, successorList->getSuccessor().key)) {
    // the message destined for our successor
    nextHop->push_back(successorList->getSuccessor());
    return nextHop;
} else {
    // find next hop with finger table
    NodeHandle tmpNode = closestPreceedingNode(key);
    if (!tmpNode.isUnspecified()) {
        nextHop->push_back(tmpNode);
    }
    return nextHop;
}
const NodeHandle& Node::closestPreceedingNode(const OverlayKey& key){
    if(role == "WORKER" || (role == "LEADER" && key.postfix() != (OverlayKey::ZERO).postfix() && key.prefix() == thisNode.key.prefix())) {
        for (int i = fingerTableLL->getSize() - 1; i >= 0; i--) { 
            if (fingerTableLL->getFinger(i).key.isBetween(thisNode.key, key)) {
                // is there a closer preceeding node in the successor list?
                for (int j = successorListLL->getSize() - 1; j >= 0; j--) {
                    if(successorListLL->getSuccessor(j).key.
                        isBetween(fingerTableLL->getFinger(i).key,
                        key)) {
                        return successorListLL->getSuccessor(j);
                    }
                }
            }
        }
        // if no, settle with the node already found
        return fingerTableLL->getFinger(i);
    }
    // if no finger is found lookup the rest of the successor list
    for (int i = successorListLL->getSize()-1; i >= 0; i--) {
        if(successorListLL->getSuccessor(i).key.isBetween(thisNode.key, key)) {
            return successorListLL->getSuccessor(i);
        }
    }
    // if this is the first and only node on the ring, it is
    responsible
    if ((predecessorNodeLL.isUnspecified()) &&
        (successorListLL->getSuccessor() == thisNode)) {
        return thisNode;
    }
} else if(role == "LEADER"){
    for (int i = fingerTable->getSize() - 1; i >= 0; i--) { 
        if (fingerTable->getFinger(i).key.isBetween(thisNode.key,
        key)) {
            // is there a closer preceeding node in the successor list?
            for (int j = successorList->getSize() - 1; j >= 0; j--) {
                if (successorList->getSuccessor(j).key.
                    isBetween(fingerTable->getFinger(i).key.
                    key).isBetween(fingerTable->getFinger(i).key,
                    key)) {
                    return successorList->getSuccessor(j);
                }
            }
        }
    }
}
isBetween(fingerTable->getFinger(i).key, key))
{
    return successorList->getSuccessor(j);
}
}

// if no, settle with the node already found
return fingerTable->getFinger(i);
}

// if no finger is found lookup the rest of the successor list
for (int i = successorList->getSize()-1; i >= 0; i--) {
    if(successorList->getSuccessor(i).key.isBetween(thisNode.key, key)) {
        return successorList->getSuccessor(i);
    }
}

// if this is the first and only node on the ring, it is
// responsible
if ((predecessorNode.isUnspecified()) &&
    (successorList->getSuccessor() == thisNode)) {
    return thisNode;
}

// if there is still no node found return
NodeHandle::UNSPECIFIED_NODE
    return NodeHandle::UNSPECIFIED_NODE;
}

void Node::recordOverlaySentStats(BaseOverlayMessage* msg){
    BaseOverlayMessage* innerMsg;
    if (msg->getType() == OVERLAYROUTE)
        innerMsg = dynamic_cast<BaseOverlayMessage*>(msg->encapsulatedMsg());
    else
        innerMsg = msg;

    switch (innerMsg->getType()) {
    case OVERLAYSIGNALING: {
        SocietyMessage* chordMsg =
            dynamic_cast<SocietyMessage*>(innerMsg);
        switch(chordMsg->getCommand()) {
            case NEWSUCCESSORHINT:
                RECORD_STATS(newsuccessorhintCount++;
                newsuccessorhintBytesSent +=
                    msg->byteLength());
                break;
            break;
        }
        break;
    }
    break;
}

    case RPC: {
        if ((dynamic_cast<SStabilizeCall*>(innerMsg) != NULL) ||
            (dynamic_cast<SStabilizeResponse*>(innerMsg) !=
                NULL)) {
            
        }

    }

}

99
RECORD_STATS(stabilizeCount++; stabilizeBytesSent += msg->byteLength());
} else if ((dynamic_cast<SStabilizeUnderNodeCall*>(innerMsg) != NULL) ||
(dynamic_cast<SStabilizeUnderNodeResponse*>(innerMsg) != NULL)) {
    RECORD_STATS(stabilizeCount++; stabilizeBytesSent += msg->byteLength());
} else if ((dynamic_cast<SNotifyCall*>(innerMsg) != NULL) ||
(dynamic_cast<SNotifyResponse*>(innerMsg) != NULL)) {
    RECORD_STATS(notifyCount++; notifyBytesSent += msg->byteLength());
} else if ((dynamic_cast<SNotifyUnderNodeCall*>(innerMsg) != NULL) ||
(dynamic_cast<SNotifyUnderNodeResponse*>(innerMsg) != NULL)) {
    RECORD_STATS(notifyCount++; notifyBytesSent += msg->byteLength());
} else if ((dynamic_cast<SFixfingersNodeCall*>(innerMsg) != NULL) ||
(dynamic_cast<SFixfingersNodeResponse*>(innerMsg) != NULL)) {
    RECORD_STATS(fixfingersCount++; fixfingersBytesSent += msg->byteLength());
} else if ((dynamic_cast<SFixfingersUnderNodeCall*>(innerMsg) != NULL) ||
(dynamic_cast<SFixfingersUnderNodeResponse*>(innerMsg) != NULL)) {
    RECORD_STATS(fixfingersCount++; fixfingersBytesSent += msg->byteLength());
} else if ((dynamic_cast<SJoinCall*>(innerMsg) != NULL) ||
(dynamic_cast<SJoinResponse*>(innerMsg) != NULL)) {
    RECORD_STATS(joinCount++; joinBytesSent += msg->byteLength());
} break;
}

void Node::finishOverlay(){
    recordScalar("Node: Sent JOIN Messages", joinCount);
    recordScalar("Node: Sent NEWSUCCESSORHINT Messages",
        newsuccessorhintCount);
    recordScalar("Node: Sent STABILIZE Messages", stabilizeCount);
    recordScalar("Node: Sent NOTIFY Messages", notifyCount);
    recordScalar("Node: Sent FIX_FINGERS Messages", fixfingersCount);
    recordScalar("Node: Sent JOIN Bytes", joinBytesSent);
    recordScalar("Node: Sent NEWSUCCESSORHINT Bytes",
        newsuccessorhintBytesSent);
    recordScalar("Node: Sent STABILIZE Bytes", stabilizeBytesSent);
    recordScalar("Node: Sent NOTIFY Bytes", notifyBytesSent);
    recordScalar("Node: Sent FIX_FINGERS Bytes", fixfingersBytesSent);
    // remove this node from the bootstrap list
bootstrapOracle->removePeer(thisNode);
}

void Node::handleVirtualWORKERsTimerExpired(cMessage* msg)
{
    if (state != READY)
        return;
    // Fazer o stabilize e notify para todos os operarios virtuais
    for(int i=0; i<virtual_fingerTable->getSize(); i++){
        VirtualNode node = virtual_fingerTable->getVirtualNode(i);
        if (node.checkPred >= stabilizeRetry) {
            // predecessor node seems to be dead
            // remove it from the predecessor / successor lists
            node.pred = NodeHandle::UNSPECIFIED_NODE;
            node.checkPred = 0;
            updateTooltip();
        }
        if (node.checkStabilize >= stabilizeRetry) {
            // successor node seems to be dead
            // remove it from the predecessor / successor list
            // if we had a ring consisting of 2 nodes and our successor
            // to be dead. Remove also predecessor because the successor
            // and predecessor are the same node
            if ((!node.pred.isUnspecified()) && node.pred == node.suc) {
                node.pred = NodeHandle::UNSPECIFIED_NODE;
            }
            node.checkStabilize = 0;
            updateTooltip();
            if (node.suc.isUnspecified()) {
                node.suc = thisNode;
                return;
            }
        }
    }
    // call STABILIZE RPC
    SStabilizeUnderNodeCall* call = new SStabilizeUnderNodeCall("SStabilizeUnderNodeCall");
    call->setSrcKey(node.operario);
    call->setDestKey(node.suc.key);
    call->setRole(role.c_str());
    call->setLength(STABILIZEUNDERNODECALLL(call));
    sendRpcMessage(node.suc, call);
    node.checkPred++;
    node.checkStabilize++;
    // schedule next stabilization process
    cancelEvent(virtualWORKERs_timer);
    scheduleAt(simulation.simTime() + stabilizeDelay, msg);
}

void Node::handleJoinTimerExpired(cMessage* msg){
    // only process timer, if node is not bootstrapped yet
    if (state == READY)
        return;
    // enter state BOOTSTRAP
    if (state != BOOTSTRAP)
        changeState(BOOTSTRAP);
change bootstrap node from time to time

joinRetry--; if (joinRetry == 0) {
  joinRetry = par("joinRetry");
  changeState(BOOTSTRAP);
  return;
}

call JOIN RPC

SJoinCall* call = new SJoinCall("SJoinCall");
call->setRole(role.c_str());
call->setLength(JOINCALL_L(call));
call->setReconnect(reconnect);
if (role == "WORKER" /*& (bootstrapNode.isUnspecified())*/) {
  bootstrapNode = NodeHandle{
    OverlayKey::sha1(const_cast<char*>(leader_ip.c_str())).prefix(), IPAddress(leader_ip.c_str()), 1024);
    bootstrapNode.key.setSpecified();
    if (role == "WORKER" /*& (bootstrapNode.isUnspecified())*/) {
      sendRpcMessage(bootstrapNode, call, NULL, -1, joinDelay);
    } else {
      sendRpcMessage(bootstrapNode, call, NULL, thisNode.key, -1, joinDelay);
    }
  } else {
    sendRpcMessage(bootstrapNode, call, NULL, thisNode.key, -1, joinDelay);
  }

// schedule next bootstrap process in the case this one fails
  cancelEvent(join_timer);
  scheduleAt(simulation.simTime() + joinDelay, msg);
}

void Node::handleStabilizeTimerExpired(cMessage* msg){
  if (state != READY) {
    return;
  }

  if (missingPredecessorStabRequests >= stabilizeRetry) {
    // predecessor node seems to be dead
    // remove it from the predecessor / successor lists
    successorList->removeSuccessor(predecessorNode);
    predecessorIsDead();
    predecessorNode = NodeHandle::UNSPECIFIED_NODE;

    missingPredecessorStabRequests = 0;
    updateTooltip();
  }

  if (missingSuccessorStabResponses >= stabilizeRetry) {
    // successor node seems to be dead
    // remove it from the predecessor / successor list
    predecessorIsDead();
    NodeHandle successor = successorList->popSuccessor();
    // if we had a ring consisting of 2 nodes and our successor
    // seems to be dead. Remove also predecessor because the successor
    // and predecessor are the same node
    if (!predecessorNode.isUnspecified() && predecessorNode == successor) {
      predecessorIsDead();
      predecessorNode = NodeHandle::UNSPECIFIED_NODE;
    } else {
      predecessorIsDead();
      predecessorNode = NodeHandle::UNSPECIFIED_NODE;
    }
  }
missingSuccessorStabResponses = 0;
updateTooltip();

if (successorList->isEmpty()) {
    changeState(INIT);
    changeState(BOOTSTRAP);
    return;
}

if (!successorList->isEmpty()) {
    // call STABILIZE RPC
    SStabilizeCall* call = new SStabilizeCall("SStabilizeCall");
    call->setRole(role.c_str());
    call->setDestKey(successorList->getSuccessor().key);
    call->setSrcKey(thisNode.key);
    call->setLength(STABILIZECALL_LEN(call));
    sendRpcMessage(successorList->getSuccessor(), call);
    missingPredecessorStabRequests++;
    missingSuccessorStabResponses++;
}

    // schedule next stabilization process
    cancelEvent(stabilize_timer);
    scheduleAt(simulation.simTime() + stabilizeDelay, msg);
}
predecessorIsDead();
    predecessorNodeLL = NodeHandle::UNSPECIFIED_NODE;
}

missingSuccessorStabResponsesLL = 0;
updateTooltip();

if (successorListLL->isEmpty()) {
    changeState(INIT);
    changeState(BOOTSTRAP);
    return;
}

if (!successorListLL->isEmpty()) {
    // call STABILIZE RPC
    SStabilizeUnderNodeCall* call = new SStabilizeUnderNodeCall("SStabilizeUnderNodeCall");
    call->setRole(role.c_str());
    call->setLength(STABILIZEUNDERNODECALL_L(call));
    call->setDestKey(successorListLL->getSuccessor().key);
    sendRpcMessage(successorListLL->getSuccessor(), call);

    missingPredecessorStabRequestsLL++;
    missingSuccessorStabResponsesLL++;
}

// schedule next stabilization process
cancelEvent(stabilize_timer_under_node);
scheduleAt(simulation.simTime() + stabilizeDelay, msg);
} else if (role == "LEADER") {
    if (missingPredecessorStabRequestsLL >= stabilizeRetry) {
        // predecessor node seems to be dead
        // remove it from the predecessor / successor lists
        successorListLL->removeSuccessor(predecessorNodeLL);
        predecessorIsDead();
        predecessorNodeLL = NodeHandle::UNSPECIFIED_NODE;

        missingPredecessorStabRequestsLL = 0;
        updateTooltip();
    }

    if (missingSuccessorStabResponsesLL >= stabilizeRetry) {
        // successor node seems to be dead
        // remove it from the predecessor / successor list
        successorIsDead();
        NodeHandle successor = successorListLL->popSuccessor();

        // if we had a ring consisting of 2 nodes and our successor
        // seems
        // to be dead. Remove also predecessor because the successor
        // and predecessor are the same node
        if (!predecessorNodeLL.isUnspecified() &&
            predecessorNodeLL == successor) {
            predecessorIsDead();
            predecessorNodeLL = NodeHandle::UNSPECIFIED_NODE;
        }
    }

    missingSuccessorStabResponsesLL = 0;
    updateTooltip();
}
if (!successorListLL->isEmpty()) {
    // call STABILIZE RPC
    SStabilizeUnderNodeCall* call = new SStabilizeUnderNodeCall("SStabilizeUnderNodeCall");
    call->setRole(role.c_str());
    call->setLength(STABILIZEUNDERNODECALL_L(call));
    call->setDestKey(successorListLL->getSuccessor().key);
    sendRpcMessage(successorListLL->getSuccessor(), call);

    missingPredecessorStabRequestsLL++;
    missingSuccessorStabResponsesLL++;
}
// schedule next stabilization process
cancelEvent(stabilize_timer_under_node);
scheduleAt(simulation.simTime() + stabilizeDelay, msg);
}
void Node::handleFixFingersTimerExpiredNode(cMessage* msg)
{
    if ((state != READY) || successorList->isEmpty())
        return;
    for (uint nextFinger = 0; nextFinger < (thisNode.key.getLength()/2);nextFinger++) {
        // calculate "n + 2^(i - 1)"
        OverlayKey offset = OverlayKey::pow2(nextFinger);
        OverlayKey lookupKey;
        offset = offset <<((offset.getLength())/2);
        lookupKey = thisNode.key.prefix() + offset;

        // send message only for non-trivial fingers
        if (offset > successorList->getSuccessor().key - thisNode.key)
        {
            // call FIXFINGER RPC
            SFixfingersNodeCall* call = new SFixfingersNodeCall("SFixfingersNodeCall");
            call->setRole(role.c_str());
            call->setFinger(nextFinger);
            call->setLength(FIXFINGERSNODECALL_L(call));
            sendRpcMessage(NodeHandle::UNSPECIFIED_NODE, call, NULL,
            lookupKey, -1, fixfingersDelay);
        } else {
            // let trivial fingers point to the successor node
            fingerTable->setFinger(nextFinger,
                successorList->getSuccessor());
        }
    }
    // schedule next finger repair process
    cancelEvent(fixfingers_timer);
    scheduleAt(simulation.simTime() + fixfingersDelay, msg);
}
void Node::handleFixFingersTimerExpiredUnderNode(cMessage* msg)
{
    if (role == "WORKER"){
        if ((state != READY) || successorListLL->isEmpty())
            return;
    }
}
for (uint nextFinger = 0; nextFinger < (thisNode.key.getLength()/2); nextFinger++) {
    // calculate "n + 2^(i - 1)"
    OverlayKey offset = OverlayKey::pow2(nextFinger);
    OverlayKey lookupKey = thisNode.key + offset;
    // send message only for non-trivial fingers
    if (offset > successorListLL->getSuccessor().key - thisNode.key) {
        // call FIXFINGER RPC
        SFixfingersUnderNodeCall* call = new SFixfingersUnderNodeCall("SFixfingersUnderNodeCall");
        call->setRole(role.c_str());
        call->setFinger(nextFinger);
        call->setLength(FIXFINGERSUNDERNODECALL_L(call));
        sendRpcMessage(NodeHandle::UNSPECIFIED_NODE, call, NULL, lookupKey, -1, fixfingersDelay);
    } else {
        // let trivial fingers point to the successor node
        fingerTableLL->setFinger(nextFinger, successorListLL->getSuccessor());
    }
}
// schedule next finger repair process
cancelEvent(fixfingers_timer_under_node);
scheduleAt(simulation.simTime() + fixfingersDelay, msg);
} else if(role == "LEADER"){
    if (successorListLL->isEmpty())
        return;
    for (uint nextFinger = 0; nextFinger < (thisNode.key.getLength()/2); nextFinger++) {
        // calculate "n + 2^(i - 1)"
        OverlayKey offset = OverlayKey::pow2(nextFinger);
        OverlayKey lookupKey = thisNode.key + offset;
        // send message only for non-trivial fingers
        if (offset > successorListLL->getSuccessor().key - thisNode.key) {
            // call FIXFINGER RPC
            SFixfingersUnderNodeCall* call = new SFixfingersUnderNodeCall("SFixfingersUnderNodeCall");
            call->setRole(role.c_str());
            call->setFinger(nextFinger);
            call->setLength(FIXFINGERSUNDERNODECALL_L(call));
            sendRpcMessage(NodeHandle::UNSPECIFIED_NODE, call, NULL, lookupKey, -1, fixfingersDelay);
        } else {
            // let trivial fingers point to the successor node
            fingerTableLL->setFinger(nextFinger, successorListLL->getSuccessor());
        }
    }
    // schedule next finger repair process
    cancelEvent(fixfingers_timer_under_node);
    scheduleAt(simulation.simTime() + fixfingersDelay, msg);
}
void Node::handleNewSuccessorHint(SocietyMessage* societyMsg)
{
    NewSSuccessorHintMessage* newSuccessorHintMsg =
        check_and_cast<NewSSuccessorHintMessage*>(societyMsg);

    // fetch the successor's predecessor
    if (role == "WORKER" || (newSuccessorHintMsg->getRole() != role &&
                                 role == "LEADER") &&
        role == "LEADER")
    {
        NodeHandle predecessorLL = newSuccessorHintMsg->getPreNode();
        // is the successor's predecessor a new successor for this node?
        if (predecessorLL.key.isBetween(thisNode.key, successorListLL->
            getSuccessor().key) || (thisNode.key == successorListLL->
                getSuccessor().key))
        {
            // add the successor's predecessor to the successor list
            successorListLL->addSuccessor(predecessorLL);
            updateTooltip();
        }
    }
    else if (role == "LEADER")  // see for apenas lider
    {
        NodeHandle predecessor = newSuccessorHintMsg->getPreNode();
        // is the successor's predecessor a new successor for this node?
        if (predecessor.key.isBetween(thisNode.key, successorList->
            getSuccessor().key) || (thisNode.key == successorList->
                getSuccessor().key))
        {
            // add the successor's predecessor to the successor list
            successorList->addSuccessor(predecessor);
            updateTooltip();
        }
    }
}

void Node::rpcJoin(SJoinCall* joinCall)
{
    NodeHandle requestor = joinCall->getSrcNode();
    // compile successor list
    SJoinResponse* joinResponse = new SJoinResponse("SJoinResponse");
    int sucNum;
    bool is_virtual_WORKER = false;
    if (role == "WORKER" || (role == "LEADER" && requestor.key.postfix() !=
                        (OverlayKey::ZERO).postfix()))
    {
        if (role == "WORKER")
        {
            sucNum = successorListLL->getSize();
            joinResponse->setSucNum(sucNum);
            joinResponse->setSucNodeArraySize(sucNum);
            for (int k = 0; k < sucNum; k++)
            {
                joinResponse->setSucNode(k, successorListLL->
                    getSuccessor(k));
            }
            // sent our predecessor as hint to the joining node
            if (predecessorNodeLL.isUnspecified() && successorListLL-
                isEmpty())
            {
                // we are the only node in the ring
                joinResponse->setPreNode(thisNode);
            } else{
                joinResponse->setPreNode(predecessorNodeLL);
            }
        } else{
            if (virtual_fingerTable->ExistVirtualNode(requestor.key)){
                sucNum = 1;
                joinResponse->setSucNodeArraySize(1);
            } else{
                sucNum = 1;
                joinResponse->setSucNodeArraySize(1);
            }
        }
    }
}
VirtualNode virtual = virtual_fingerTable->removeVirtualNode(requestor.key);
joinResponse->setSucNode(0, virtual.suc);
joinResponse->setPreNode(virtual.pred);
is_virtual_WORKER = true;
}else{
sucNum = successorListLL->getSize();
joinResponse->setSucNum(sucNum);
joinResponse->setSucNodeArraySize(sucNum);
for (int k = 0; k < sucNum; k++) {
  joinResponse->setSucNode(k, successorListLL->getSuccessor(k));
}
// sent our predecessor as hint to the joining node
if (predecessorNodeLL.isUnspecified() && successorListLL->isEmpty()) {
  // we are the only node in the ring
  joinResponse->setPreNode(thisNode);
} else{
  joinResponse->setPreNode(predecessorNodeLL);
}
}
}
else{
sucNum = successorList->getSize();
joinResponse->setSucNum(sucNum);
joinResponse->setSucNodeArraySize(sucNum);
for (int k = 0; k < sucNum; k++) {
  joinResponse->setSucNode(k, successorList->getSuccessor(k));
}
// sent our predecessor as hint to the joining node
if (predecessorNode.isUnspecified() && successorList->isEmpty()) {
  // we are the only node in the ring
  joinResponse->setPreNode(thisNode);
} else{
  joinResponse->setPreNode(predecessorNode);
}
}
joinResponse->setRole(role.c_str());
joinResponse->setLength(JOINRESPONSE_L(joinResponse));
sendRpcResponse(joinCall, joinResponse);
if (aggressiveJoinMode && !is_virtual_WORKER) {
  // aggressiveJoinMode differs from standard join operations:
  // 1. set our predecessor pointer to the joining node
  // 2. send our old predecessor as hint in JoinResponse msgs
  // 3. send a NEWSUCCESSORHINT to our old predecessor to update
  //    its successor pointer
  // send NEWSUCCESSORHINT to our old predecessor
  // verifica se eh uma resposta da camada abaixo do lider
  if (role == "WORKER" || (role == "LEADER" && requestor.key.postfix() != (OverlayKey::ZERO).postfix())) {
    if (!predecessorNodeLL.isUnspecified()) {
      NewSSuccessorHintMessage* newSuccessorHintMsg = new
      NewSSuccessorHintMessage("NEWSUCCESSORHINT");
    }
newSuccessorHintMsg->setCommand(NEWSUCCESSORHINT);
newSuccessorHintMsg->setSrcNode(thisNode);
newSuccessorHintMsg->setPreNode(requestor);
newSuccessorHintMsg->setLength(NEWSUCCESSORHINT_L(newSuccessorHintMsg));
newSuccessorHintMsg->setRole(role.c_str());
sendMessageToUDP(predecessorNodeLL, newSuccessorHintMsg);
}
// the requestor is our new predecessor
predecessorNodeLL = requestor;
} else if (role == "LEADER") {
    if (!predecessorNode.isUnspecified()) {
        NewSSuccessorHintMessage* newSuccessorHintMsg = new NewSSuccessorHintMessage("NEWSUCCESSORHINT");
        newSuccessorHintMsg->setCommand(NEWSUCCESSORHINT);
        newSuccessorHintMsg->setSrcNode(thisNode);
        newSuccessorHintMsg->setPreNode(requestor);
        newSuccessorHintMsg->setLength(NEWSUCCESSORHINT_L(newSuccessorHintMsg));
        newSuccessorHintMsg->setRole(role.c_str());
        sendMessageToUDP(predecessorNode, newSuccessorHintMsg);
    }
    // the requestor is our new predecessor
    predecessorNode = requestor;
}

// if we don't have a successor, the requestor is also our new successor
if(!is_virtual_WORKER){
    if (role == "WORKER" || (role == "LEADER" && requestor.key.postfix() != (OverlayKey::ZERO).postfix())) {
        if (successorListLL->isEmpty())
            successorListLL->addSuccessor(requestor);
    } else {
        if (successorList->isEmpty())
            successorList->addSuccessor(requestor);
    }
}
updateTooltip();
}

void Node::handleRpcJoinResponse(SJoinResponse* joinResponse) {
    // determine the number of successor nodes to add
    int sucNum = successorListSize - 1;
    if (joinResponse->getSucNum() < successorListSize - 1) {
        sucNum = joinResponse->getSucNum();
    }

    // add successor node(s)
    if (role == "WORKER" || (role == "LEADER" && (joinResponse->getSrcNode().key.postfix() != (OverlayKey::ZERO).postfix()))) {
        for (int k = 0; k < sucNum; k++) {
            NodeHandle successor = joinResponse->getSucNode(k);
            successorListLL->addSuccessor(successor);
        }
    }
// the sender of this message is our new successor
successorListLL->addSuccessor(joinResponse->getSrcNode());
// in aggressiveJoinMode: use hint in JoinResponse
// to set our new predecessor
if (aggressiveJoinMode) {
    predecessorNodeLL = joinResponse->getPreNode();
}
else{
    for (int k = 0; k < sucNum; k++) {
        NodeHandle successor = joinResponse->getSucNode(k);
        successorList->addSuccessor(successor);
    }
// the sender of this message is our new successor
successorList->addSuccessor(joinResponse->getSrcNode());
// in aggressiveJoinMode: use hint in JoinResponse
// to set our new predecessor
if (aggressiveJoinMode) {
    predecessorNode = joinResponse->getPreNode();
}
}
updateTooltip();
if (role == "LEADER" && (joinResponse->getSrcNode().key.postfix() ==
(OverlayKey::ZERO).postfix())){
    changeState(READY);
    // immediate stabilization protocol
    cancelEvent(stabilize_timer);
    scheduleAt(simulation.simTime(), stabilize_timer);
    // immediate finger repair protocol
    cancelEvent(fixfingers_timer);
    scheduleAt(simulation.simTime(), fixfingers_timer);
    cancelEvent(fixfingers_timer_under_node);
    scheduleAt(simulation.simTime() +
    fixfingersDelay,fixfingers_timer_under_node);
    cancelEvent(stabilize_timer_under_node);
    scheduleAt(simulation.simTime(), stabilize_timer_under_node);
}
else if (role == "WORKER"){
    changeState(READY);
    // immediate stabilization protocol
    cancelEvent(fixfingers_timer_under_node);
    scheduleAt(simulation.simTime() + fixfingersDelay,
    fixfingers_timer_under_node);
    cancelEvent(stabilize_timer_under_node);
    scheduleAt(simulation.simTime(), stabilize_timer_under_node);
}
else{
    // immediate stabilization protocol
    cancelEvent(fixfingers_timer_under_node);
    scheduleAt(simulation.simTime() + fixfingersDelay,
    fixfingers_timer_under_node);
    cancelEvent(stabilize_timer_under_node);
    scheduleAt(simulation.simTime(), stabilize_timer_under_node);
}
}
void Node::rpcStabilize(SSstabilizeCall* call)
{
    // reply with SSstabilizeResponse message
    SSstabilizeResponse* stabilizeResponse = new SSstabilizeResponse("SSstabilizeResponse");
    stabilizeResponse->setRole(role.c_str());
    stabilizeResponse->setDestKey(call->getSrcKey());
    stabilizeResponse->setPreNode(predecessorNode);
    // our predecessor seems to be alive
    missingPredecessorStabRequests = 0;
    stabilizeResponse->setLength(STABILIZERESPONSE_L(stabilizeResponse));
    sendRpcResponse(call, stabilizeResponse);
}

void Node::rpcStabilizeUnderNode(SSstabilizeUnderNodeCall* call)
{
    // reply with SSstabilizeResponse message
    SSstabilizeUnderNodeResponse* stabilizeResponse = new SSstabilizeUnderNodeResponse("SSstabilizeUnderNodeResponse");
    stabilizeResponse->setDestKey(call->getSrcNode().key);
    stabilizeResponse->setRole(role.c_str());
    if(role == "LEADER" /*&& (call->getDestKey().postfix() != (OverlayKey::ZERO).postfix())*/) {
        if(virtual_fingerTable->ExistVirtualNode(call->getDestKey())){
            VirtualNode node = virtual_fingerTable->removeVirtualNode(call->getDestKey());
            node.checkPred=0;
            stabilizeResponse->setPreNode(node.pred);
            virtual_fingerTable->addVirtualNode(node);
        } else {
            stabilizeResponse->setPreNode(predecessorNodeLL);
            // our predecessor seems to be alive
            missingPredecessorStabRequestsLL = 0;
        } } else if(role == "WORKER"){
            stabilizeResponse->setPreNode(predecessorNodeLL);
            // our predecessor seems to be alive
            missingPredecessorStabRequestsLL = 0;
        } stabilizeResponse->setLength(STABILIZEUNDERNODERESPONSE_L(stabilizeResponse));
        sendRpcResponse(call, stabilizeResponse);
    }

void Node::handleRpcStabilizeResponse(SSstabilizeResponse* stabilizeResponse){
    // fetch the successor's predecessor
    NodeHandle predecessor = stabilizeResponse->getPreNode();
    if(role == "LEADER" /*&& (stabilizeResponse->getDestKey().postfix() != (OverlayKey::ZERO).postfix())*/) {
        if(successorList->isEmpty() ||
        // is the successor's predecessor a new successor for this node?
        if (successorList->isEmpty() ||
            predecessor != successorList->firstNode())
            // our successor seems to be alive
            missingSuccessorStabResponses = 0;
    } else
        // our successor seems to be alive
        missingSuccessorStabResponses = 0;
}
predecessor.key.isBetween(thisNode.key, successorList->getSuccessor().key)) {
    // add the successor's predecessor to the successor list
    successorList->addSuccessor(predecessor);
    updateTooltip();
}
// compile NOTIFY RPC
SNotifyCall* notifyCall = new SNotifyCall("SNotifyCall");
notifyCall->setRole(role.c_str());
notifyCall->setLength(NOTIFYCALL_L(notifyCall));
notifyCall->setDestKey(successorList->getSuccessor().key);
sendRpcMessage(successorList->getSuccessor(), notifyCall);
}

void Node::handleRpcStabilizeUnderNodeResponse(SSynchronizeUnderNodeResponse* stabilizeResponse) {
    // fetch the successor's predecessor
    NodeHandle predecessor = stabilizeResponse->getPreNode();
    if (role == "LEADER" /*&& (stabilizeResponse->getDestKey().postfix() != (OverlayKey::ZERO).postfix()) */) {
        if (virtual_fingerTable->ExistVirtualNode(stabilizeResponse->getDestKey())) {
            VirtualNode node = virtual_fingerTable->removeVirtualNode(stabilizeResponse->getDestKey());
            node.checkStabilize = 0;
            if (node.suc.isUnspecified() || node.pred.key.isBetween(node.operario, node.suc.key)) {
                // add the successor's predecessor to the successor list
                node.suc = predecessor;
                updateTooltip();
            }
            virtual_fingerTable->addVirtualNode(node);
            // compile NOTIFY RPC
            SNotifyUnderNodeCall* notifyCall = new SNotifyUnderNodeCall("SNotifyUnderNodeCall");
            notifyCall->setRole(role.c_str());
            notifyCall->setLength(NOTIFYUNDERNODECALL_L(notifyCall));
            notifyCall->setDestKey(node.suc.key);
sendRpcMessage(node.suc, notifyCall);
        } else {
            // our successor seems to be alive
            missingSuccessorStabResponsesLL = 0;
            // is the successor's predecessor a new successor for this node?
            if (successorListLL->isEmpty() || predecessor.key.isBetween(thisNode.key, successorListLL->getSuccessor().key)) {
                // add the successor's predecessor to the successor list
                successorListLL->addSuccessor(predecessor);
                updateTooltip();
            }
            // compile NOTIFY RPC
SNotifyUnderNodeCall* notifyCall = new SNotifyUnderNodeCall("SNotifyUnderNodeCall");
notifyCall->setRole(role.c_str());
notifyCall->setLength(NOTIFYUNDERNODECALL_L(notifyCall));
notifyCall->setDestKey(successorListLL->getSuccessor().key);
sendRpcMessage(successorListLL->getSuccessor(), notifyCall);
}
}
}

} else if(role == "WORKER"){
    thisNode.key.isUnspecified() << " thisNode " << thisNode.ip << " thisNode.key " << thisNode.key << endl;
    // our successor seems to be alive
    missingSuccessorStabResponsesLL = 0;
    // is the successor's predecessor a new successor for this node?
    if (successorListLL->isEmpty() ||
        predecessor.key.isBetween(thisNode.key,
        successorListLL->getSuccessor().key)) {
        // add the successor's predecessor to the successor list
        successorListLL->addSuccessor(predecessor);
        updateTooltip();
    }
    // compile NOTIFY RPC
    SNotifyUnderNodeCall* notifyCall = new SNotifyUnderNodeCall("SNotifyUnderNodeCall");
    notifyCall->setRole(role.c_str());
    notifyCall->setLength(NOTIFYUNDERNODECALL_L(notifyCall));
    notifyCall->setDestKey(successorListLL->getSuccessor().key);
    sendRpcMessage(successorListLL->getSuccessor(), notifyCall);
}
}

void Node::rpcLeaveNode(SLeaveNodeCall* call){
    VirtualNode v;
    v.operario = call->getOperario();
    v.suc = call->getSucNode();
    v.pred = call->getPredNode();
    v.count = 0;
    v.checkPred = 0;
    v.checkStabilize = 0;
    virtual_fingerTable->addVirtualNode(v);
}

void Node::rpcNotify(SNotifyCall* call){
    NodeHandle predecessor = call->getSrcNode();
    if(role == "LEADER") {
        // our predecessor seems to be alive
        missingPredecessorStabRequests = 0;
        // is the new predecessor closer than the current one?
        if (predecessorNode.isUnspecified() ||
            predecessor.key.isBetween(predecessorNode.key,
            thisNode.key)) {
            // set up new predecessor
            predecessorNode = predecessor;
            updateTooltip();
        }
        // compile NOTIFY response
}
SNotifyResponse* notifyResponse = new SNotifyResponse("SNotifyResponse");

int sucNum = successorList->getSize();
notifyResponse->setSucNum(sucNum);
notifyResponse->setSucNodeArraySize(sucNum);
for (int k = 0; k < sucNum; k++) {
    notifyResponse->setSucNode(k, successorList->getSuccessor(k));
}
notifyResponse->setRole(role.c_str());
notifyResponse->setLength(NOTIFYRESPONSE_L(notifyResponse));
notifyResponse->setDestKey(call->getSrcNode().key);
sendRpcResponse(call, notifyResponse);

NodeHandle predecessor = call->getSrcNode();
if (call->getDestKey().postfix() != (OverlayKey::ZERO).postfix() & role == "LEADER") {
    if (virtual_fingerTable->ExistVirtualNode(call->getDestKey())) {
        VirtualNode node = virtual_fingerTable->removeVirtualNode(call->getDestKey());
        node.checkPred = 0;
        // is the new predecessor closer than the current one?
        if (node.pred.isUnspecified() || predecessor.key.isBetween(node.pred.key, node.operario)) {
            // set up new predecessor
            node.pred = predecessor;
            updateTooltip();
        }
        virtual_fingerTable->addVirtualNode(node);
        // compile NOTIFY response
        SNotifyResponse* notifyResponse = new SNotifyResponse("SNotifyResponse");
        int sucNum = successorList->getSize();
        notifyResponse->setSucNum(sucNum);
        notifyResponse->setSucNodeArraySize(sucNum);
        for (int k = 0; k < sucNum; k++) {
            notifyResponse->setSucNode(k, successorList->getSuccessor(k));
        }
        notifyResponse->setRole(role.c_str());
        notifyResponse->setLength(NOTIFYRESPONSE_L(notifyResponse));
        notifyResponse->setDestKey(call->getSrcNode().key);
        sendRpcResponse(call, notifyResponse);
    } else {
        // our predecessor seems to be alive
        missingPredecessorStabRequestsLL = 0;
        // is the new predecessor closer than the current one?
        if (predecessorNodeLL.isUnspecified() || predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
            // set up new predecessor
            predecessorNodeLL = predecessor;
            updateTooltip();
        }
        // compile NOTIFY response
        SNotifyResponse* notifyResponse = new SNotifyResponse("SNotifyResponse");
int sucNum = successorListLL->getSize();
notifyResponse->setSucNum(sucNum);
notifyResponse->setSucNodeArraySize(sucNum);
for (int k = 0; k < sucNum; k++) {
    notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
} notifyResponse->setRole(role.c_str());
notifyResponse->setLength(NOTIFYRESPONSE_L(notifyResponse));
sendRpcResponse(call, notifyResponse);
}
}
else {
    // our predecessor seems to be alive
    missingPredecessorStabRequests = 0;
    // is the new predecessor closer than the current one?
    if (predecessorNode.isUnspecified() || predecessor.key.isBetween(predecessorNode.key, thisNode.key)) {
        // set up new predecessor
        predecessorNode = predecessor;
        updateTooltip();
    }
}

void Node::rpcNotifyUnderNode(SNotifyUnderNodeCall* call){
    NodeHandle predecessor = call->getSrcNode();
    if(role == "LEADER") {
        if(virtual_fingerTable->ExistVirtualNode(call->getDestKey())){
            VirtualNode node = virtual_fingerTable->removeVirtualNode(call->getDestKey());
            node.checkPred=0;
            // is the new predecessor closer than the current one?
            if (node.pred.isUnspecified() || predecessor.key.isBetween(node.pred.key, node.operario)) {
                // set up new predecessor
                node.pred = predecessor;
                updateTooltip();
            }
            virtual_fingerTable->addVirtualNode(node);
            // compile NOTIFY response
            notifyResponse->setSucNum(1);
            notifyResponse->setSucNodeArraySize(1);
            notifyResponse->setSucNode(0, node.suc);
            notifyResponse->setRole(role.c_str());
            notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
            notifyResponse->setDestKey(call->getSrcNode().key);
            sendRpcResponse(call, notifyResponse);
        } else {
            // our predecessor seems to be alive
            missingPredecessorStabRequestsLL = 0;
            // is the new predecessor closer than the current one?
            if (predecessorNodeLL.isUnspecified() ||
predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
    // set up new predecessor
    predecessorNodeLL = predecessor;
    updateTooltip();
}

// compile NOTIFY response

int sucNum = successorListLL->getSize();
notifyResponse->setSucNum(sucNum);
notifyResponse->setSucNodeArraySize(sucNum);
for (int k = 0; k < sucNum; k++) {
    notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
}
notifyResponse->setRole(role.c_str());
notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
notifyResponse->setDestKey(call->getSrcNode().key);
sendRpcResponse(call, notifyResponse);
}

} else {
    // our predecessor seems to be alive
    missingPredecessorStabRequestsLL = 0;
    // is the new predecessor closer than the current one?
    if (predecessorNodeLL.isUnspecified() ||
        predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
        // set up new predecessor
        predecessorNodeLL = predecessor;
        updateTooltip();
    }

    // compile NOTIFY response

    int sucNum = successorListLL->getSize();
    notifyResponse->setSucNum(sucNum);
    notifyResponse->setSucNodeArraySize(sucNum);
    for (int k = 0; k < sucNum; k++) {
        notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
    }
    notifyResponse->setRole(role.c_str());
    notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
    notifyResponse->setDestKey(call->getSrcNode().key);
    sendRpcResponse(call, notifyResponse);
}

void Node::handleRpcNotifyResponse(SNotifyResponse* notifyResponse)
{
    if (successorList->getSuccessor() != notifyResponse->getSrcNode()) {
        return;
    }

    predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
        // set up new predecessor
        predecessorNodeLL = predecessor;
        updateTooltip();
    }

    // compile NOTIFY response

    int sucNum = successorListLL->getSize();
    notifyResponse->setSucNum(sucNum);
    notifyResponse->setSucNodeArraySize(sucNum);
    for (int k = 0; k < sucNum; k++) {
        notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
    }
    notifyResponse->setRole(role.c_str());
    notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
    notifyResponse->setDestKey(call->getSrcNode().key);
    sendRpcResponse(call, notifyResponse);
}

void Node::handleRpcNotifyResponse(SNotifyResponse* notifyResponse) {
    if (successorList->getSuccessor() != notifyResponse->getSrcNode()) {
        return;
    }

    predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
        // set up new predecessor
        predecessorNodeLL = predecessor;
        updateTooltip();
    }

    // compile NOTIFY response

    int sucNum = successorListLL->getSize();
    notifyResponse->setSucNum(sucNum);
    notifyResponse->setSucNodeArraySize(sucNum);
    for (int k = 0; k < sucNum; k++) {
        notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
    }
    notifyResponse->setRole(role.c_str());
    notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
    notifyResponse->setDestKey(call->getSrcNode().key);
    sendRpcResponse(call, notifyResponse);
}

void Node::handleRpcNotifyResponse(SNotifyResponse* notifyResponse) {
    if (successorList->getSuccessor() != notifyResponse->getSrcNode()) {
        return;
    }

    predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
        // set up new predecessor
        predecessorNodeLL = predecessor;
        updateTooltip();
    }

    // compile NOTIFY response

    int sucNum = successorListLL->getSize();
    notifyResponse->setSucNum(sucNum);
    notifyResponse->setSucNodeArraySize(sucNum);
    for (int k = 0; k < sucNum; k++) {
        notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
    }
    notifyResponse->setRole(role.c_str());
    notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
    notifyResponse->setDestKey(call->getSrcNode().key);
    sendRpcResponse(call, notifyResponse);
}

void Node::handleRpcNotifyResponse(SNotifyResponse* notifyResponse) {
    if (successorList->getSuccessor() != notifyResponse->getSrcNode()) {
        return;
    }

    predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
        // set up new predecessor
        predecessorNodeLL = predecessor;
        updateTooltip();
    }

    // compile NOTIFY response

    int sucNum = successorListLL->getSize();
    notifyResponse->setSucNum(sucNum);
    notifyResponse->setSucNodeArraySize(sucNum);
    for (int k = 0; k < sucNum; k++) {
        notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
    }
    notifyResponse->setRole(role.c_str());
    notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
    notifyResponse->setDestKey(call->getSrcNode().key);
    sendRpcResponse(call, notifyResponse);
}

void Node::handleRpcNotifyResponse(SNotifyResponse* notifyResponse) {
    if (successorList->getSuccessor() != notifyResponse->getSrcNode()) {
        return;
    }

    predecessor.key.isBetween(predecessorNodeLL.key, thisNode.key)) {
        // set up new predecessor
        predecessorNodeLL = predecessor;
        updateTooltip();
    }

    // compile NOTIFY response

    int sucNum = successorListLL->getSize();
    notifyResponse->setSucNum(sucNum);
    notifyResponse->setSucNodeArraySize(sucNum);
    for (int k = 0; k < sucNum; k++) {
        notifyResponse->setSucNode(k, successorListLL->getSuccessor(k));
    }
    notifyResponse->setRole(role.c_str());
    notifyResponse->setLength(NOTIFYUNDERNODERESPONSE_L(notifyResponse));
    notifyResponse->setDestKey(call->getSrcNode().key);
    sendRpcResponse(call, notifyResponse);
}
// determine number of successor nodes to add
int sucNum = successorListSize - 1;
if (notifyResponse->getSucNum() < successorListSize - 1) {
    sucNum = notifyResponse->getSucNum();
}
// replace our successor list by our successor's successor list
// and add our current successor to the list
successorList->clear();
successorList->addSuccessor(notifyResponse->getSrcNode());
for (int k = 0; k < sucNum; k++) {
    NodeHandle successor = notifyResponse->getSucNode(k);
    // don't add nodes, if this would change our successor
    if (!successor.key.isBetweenLR(thisNode.key,
        notifyResponse->getSrcNode().key)) {
        successorList->addSuccessor(successor);
    }
}
updateTooltip();
}

void Node::handleRpcNotifyUnderNodeResponse(SNotifyUnderNodeResponse* notifyResponse)
{
    if(role == "LEADER") {
        if(virtual_fingerTable->ExistVirtualNode(notifyResponse->getDestKey())){
            VirtualNode node = virtual_fingerTable->removeVirtualNode(notifyResponse->getDestKey());
            // determine number of successor nodes to add
            NodeHandle successor = notifyResponse->getSucNode(0);
            // don't add nodes, if this would change our successor
            if (!successor.key.isBetweenLR(node.operario,
                notifyResponse->getSrcNode().key)) {
                node.suc = successor;
            }
            updateTooltip();
        }else{
            // determine number of successor nodes to add
            int sucNum = successorListSize - 1;
            if (notifyResponse->getSucNum() < successorListSize - 1) {
                sucNum = notifyResponse->getSucNum();
            }
            // replace our successor list by our successor's successor list
            // and add our current successor to the list
            successorListLL->clear();
            successorListLL->addSuccessor(notifyResponse->getSrcNode());
            for (int k = 0; k < sucNum; k++) {
                NodeHandle successor = notifyResponse->getSucNode(k);
                // don't add nodes, if this would change our successor
                if (!successor.key.isBetweenLR(thisNode.key,
                    notifyResponse->getSrcNode().key)) {
                    successorListLL->addSuccessor(successor);
                }
            }
        }
    }
}
if (successorListLL->getSuccessor() != notifyResponse->getSrcNode()) {
    return;
}

// determine number of successor nodes to add
int sucNum = successorListSize - 1;
if (notifyResponse->getSucNum() < successorListSize - 1) {
    sucNum = notifyResponse->getSucNum();
}

// replace our successor list by our successor's successor list
// and add our current successor to the list
successorListLL->clear();
successorListLL->addSuccessor(notifyResponse->getSrcNode());
for (int k = 0; k < sucNum; k++) {
    NodeHandle successor = notifyResponse->getSucNode(k);
    // don't add nodes, if this would change our successor
    if (!successor.key.isBetweenLR(thisNode.key, notifyResponse->getSrcNode().key)) {
        successorListLL->addSuccessor(successor);
    }
}

updateTooltip();
}

void Node::rpcFixfingersNode(SFixfingersNodeCall* call){
    SFixfingersNodeResponse* fixfingersResponse = new SFixfingersNodeResponse("SFixfingersNodeResponse");
    fixfingersResponse->setRole(role.c_str());
    fixfingersResponse->setSucNode(thisNode);
    fixfingersResponse->setFinger(call->getFinger());
    fixfingersResponse->setLength(FIXFINGERSONODERESPONSE_L(fixfingersNodeResponse));
    sendRpcResponse(call, fixfingersResponse);
}

void Node::rpcFixfingersUnderNode(SFixfingersUnderNodeCall* call){
    fixfingersResponse->setRole(role.c_str());
    fixfingersResponse->setSucNode(thisNode);
    fixfingersResponse->setFinger(call->getFinger());
    fixfingersResponse->setLength(FIXFINGERSONODERESPONSE_L(fixfingersUnderNodeResponse));
    sendRpcResponse(call, fixfingersResponse);
}

void Node::handleRpcFixfingersNodeResponse(SFixfingersNodeResponse* fixfingersResponse){
    // set new finger pointer
NodeHandle successor = fixfingersResponse->getSucNode();
fingerTable->setFinger(fixfingersResponse->getFinger(), successor);
}

void Node::handleRpcFixfingersUnderNodeResponse(SFixfingersUnderNodeResponse * fixfingersResponse){
    // set new finger pointer
    NodeHandle successor = fixfingersResponse->getSucNode();
    fingerTableLL->setFinger(fixfingersResponse->getFinger(), successor);
}

void Node::findFriendModules(){
if(role.compare("LEADER") == 0){
    fingerTable = check_and_cast<NodeFingerTable*>(parentModule()->
    submodule("fingerTable"));
    successorList = check_and_cast<NodeSuccessorList*>(parentModule()->
    submodule("successorList"));
    fingerTableLL = check_and_cast<NodeFingerTable*>(parentModule()->
    submodule("fingerTableLL"));
    successorListLL = check_and_cast<NodeSuccessorList*>(parentModule()->
    submodule("successorListLL"));
    virtual_fingerTable = check_and_cast<VirtualFingerTable*>(parentModule()->
    submodule("virtual_fingerTable"));
}else{
    fingerTableLL = check_and_cast<NodeFingerTable*>(parentModule()->
    submodule("fingerTableLL"));
    successorListLL = check_and_cast<NodeSuccessorList*>(parentModule()->
    submodule("successorListLL"));
}
}

void Node::initializeFriendModules(){
if(role.compare("LEADER") == 0){
    virtual_fingerTable->initializeTable();
    // initialize finger table
    fingerTable->initializeTable(thisNode.key.getLength()/2,
    thisNode);
    fingerTableLL->initializeTable(thisNode.key.getLength()/2,
    thisNode);
    // initialize successor list
    successorList->initializeList(par("successorListSize"),
    thisNode);
    successorListLL->initializeList(par("successorListSize"),
    thisNode);
}else{
    fingerTableLL->initializeTable(thisNode.key.getLength()/2,
    thisNode);
    // initialize successor list
    successorListLL->initializeList(par("successorListSize"),
    thisNode);
}
void Node::updateTooltip()
{
    if (ev.isGUI() && role == "LEADER") {
        std::stringstream ttString;
        std::stringstream ttStringLL;

        // show our predecessor and successor in tooltip
        ttString << predecessorNode << endl << thisNode << endl << successorList->getSuccessor();

        ttStringLL << predecessorNodeLL << endl << thisNode << endl << successorListLL->getSuccessor();

        parentModule()->parentModule()->displayString().
            setTagArg("tt", 0, ttString.str().c_str());
        parentModule()->displayString().
            setTagArg("tt", 0, ttString.str().c_str());
        displayString().setTagArg("tt", 0, ttString.str().c_str());

        parentModule()->parentModule()->displayString().
            setTagArg("tt", 0, ttStringLL.str().c_str());
        parentModule()->displayString().
            setTagArg("tt", 0, ttStringLL.str().c_str());
        displayString().setTagArg("tt", 0, ttStringLL.str().c_str());

        // draw an arrow to our current successor
        showOverlayNeighborArrow(successorList->getSuccessor(), true,
            "m=m,50,0,50,0;o=red,1");
        showOverlayNeighborArrow(predecessorNode, false,
            "m=m,50,100,50,100;o=green,1");
    }
}
Dissertação de Mestrado apresentada por Luciana Pereira Oliveira à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “Future Internet Architecture to Structure and Manage Dynamic Autonomous Systems Internet Service Providers and Customers”, orientada pelo Prof. Djamel Fawzi Hadj Sadok e aprovada pela Banca Examinadora formada pelos professores:

Prof. Nelson Souto Rosa
Centro de Informática / UFPE

Prof. Maurício Ferreira Magalhães
Departamento de Engenharia da Computação
UNICAMP

Prof. Djamel Fawzi Hadj Sadok
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 7 de março de 2008.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco.