Universidade do Minho
Departamento de Informática

Nuno Filipe Solinho de Azevedo

Service and Auto-configuration Framework for Secure Ad-hoc Environments in Android and Linux

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob a orientação do
Professor Doutor António Luís Duarte Costa

Outubro de 2011

Abstract

Ad-hoc networks can be useful in many contexts because they can be spontaneously created and do not require any sort of infrastructure. They can be useful for small groups when no other network is accessible. They can also be used in wider areas as a low cost replacement for wireless infrastructure networks with multiple dedicated access points. Despite this, ad-hoc networks are not a very popular option for most users.

Unfortunately, ad-hoc networks are not as user friendly as infrastructure networks. The latter ones usually provide standardized mechanisms that perform the essential configurations for the correct functioning of the network. Ad-hoc networks do not have standardized mechanisms adapted to them. Each wireless network manager supports a different set of configuration mechanisms. There is usually no problem when every machine uses the same operating system but when different ones are used, users may need to manually perform the required configurations. Another cause for this low popularity is the lack of useful and easy to use applications. These applications are usually hosted on the Internet, as it provides a larger variety of business models.

To tackle these problems, new forms of automatically configuring machines and providing services should be explored. These services must be easy to develop, in order to attract the developers that would develop them. The designed solutions must also be adapted to ad-hoc environments. Another important aspect that must be addressed is security. In some contexts, such as public and corporate environments, security can be essential to provide authentication and even to allow the correct functioning of the network.

In this work, a functional framework that addresses these problems is designed and implemented. The framework is capable of automatically making the necessary configurations to provide a functional and secure ad-hoc environment. It is also capable of hosting specialized and easy to develop services, over the spontaneously created secure environment. The developed framework is easy to use and was throughly tested in Android and Linux, but it can be easily extended to work in many other operating systems.

Resumo

As redes ad-hoc podem ser úteis em muitos contextos visto poderem ser criadas espontaneamente e não necessitarem de qualquer tipo de infraestrutura. Elas podem ser úteis para grupos pequenos quando nenhuma outra rede pode ser acedida. Também podem ser usadas em áreas amplas como uma alternativa de baixo custo para redes sem fios infraestruturadas com vários pontos de acesso dedicados. Ainda assim, as redes ad-hoc não são uma opção muito popular para a maioria dos utilizadores.

Infelizmente, as redes ad-hoc não são tão simples de utilizar como as redes de infraestrutura. Estas últimas normalmente utilizam mecanismos *standardizados* para efectuar as configurações que são essenciais para o correcto funcionamento da rede. As redes ad-hoc não têm mecanismos *standardizados* adaptados a elas. Cada aplicação gestora de redes sem fios suporta um conjunto diferente de mecanismos de configuração. Normalmente não há problemas quando todas as máquinas usam o mesmo sistema operativo mas quando são usados diversos, os utilizadores podem ter que configurar manualmente as máquinas. Outra causa para esta baixa popularidade é a falta de aplicações úteis e de fácil utilização. Estas aplicações são normalmente utilizadas na Internet, visto que esta fornece uma maior variedade de modelos de negócio.

Para abordar estes problemas, novas formas de configurar máquinas automaticamente e de fornecer serviços devem ser exploradas. Estes serviços devem ser fáceis de desenvolver, de forma a atrair os *developers* que os irão desenvolver. As soluções desenhadas também devem estar adaptadas a ambientes ad-hoc. Outro aspecto importante que tem de ser focado é a segurança. Em alguns contextos, tais como ambientes públicos e empresariais, a segurança pode ser essencial para fornecer autenticação e mesmo permitir o correcto funcionamento da rede.

Neste trabalho, uma *framework* funcional que trata estes problemas é desenhada e implementada. A *framework* é capaz de efectuar automaticamente as configurações necessárias à criação de um ambiente ad-hoc funcional e seguro. É também capaz de fornecer serviços especializados e de fácil desenvolvimento, sobre o ambiente seguro criado de forma espontânea. A *framework* desenvolvida é fácil de utilizar e foi exaustivamente testada em Android e Linux, embora possa ser facilmente estendida de forma a funcionar em muitos outros sistemas operativos.

Agradecimentos

Gostaria de agradecer às pessoas que me apoiaram durante o processo de desenvolvimento deste trabalho, apoio este que foi essencial para este processo.

Queria em primeiro lugar agradecer ao meu orientador, o professor António Costa, que me forneceu apoio constante. Arranjava sempre alguma força para me ajudar e direccionar mesmo quando já estava cansado ou quando estava atulhado de trabalho.

Gostaria de agradecer também ao meu colega e amigo Rui Couto, com o qual troquei alguns conselhos e comandos de LaTeX e que me apontou soluções grátis para gestão de projectos e controlo de versões na Internet.

Queria também de agradecer aos meus pais pelo apoio incondicional e pelo orgulho que demonstraram e demonstram ter em mim, não só durante o desenvolvimento deste trabalho mas também durante todo o meu percurso, tendo a certeza que este apoio e orgulho serão para continuar.

Finalmente gostaria de agradecer a todos os meus amigos, principalmente pelos momentos em que deveria ter estado a trabalhar e não estive. Em vez disso eles forneceram-me um mais que necessário e merecido descanso entre diversas sessões de trabalho.

A todos agradeço o apoio demonstrado e deixo um sincero muito obrigado por tudo!

Contents

Acronyms

ABCDP – Authentication Based Controlled Datagram Protocol

AES – Advanced Encryption Standard

AH – Authentication Header

AKC – Asymmetric-key Cryptography

API – Application Programming Interface

BLOB – Binary Large Object

CA – Certificate Authority

CGA – Cryptographically Generated Addresses

DEX – Dalvik Executable

DHCP – Dynamic Host Configuration Protocol

DNS – Domain Name System

DoS – Denial of Service

ERP – Enterprise Resource Planning

ESP – Encapsulating Security Payload

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

HTTPS – HyperText Transfer Protocol Secure

IBC – Identity-Based Cryptography

IDE – Integrated Development Environment

IETF – Internet Engineering Task Force

IKE – Internet Key Exchange

IP – Internet Protocol

IPsec – Internet Protocol Security

IPv4 – Internet Protocol version 4

IPv6 – Internet Protocol version 6

ISAKMP – Internet Security Association and Key Management Protocol

JAR – Java Archive

JDBC – Java Database Connectivity

JDK – Java Development Kit

JNI – Java Native Interface

JSP – JavaServer Pages

JVM – Java Virtual Machine

MAC – Media Access Control

MANET – Mobile Ad-hoc Network

MDR – MANET Designated Router

MVC – Model-view-controller

NAT – Network Address Translation

ORM – Object-Relational Mapping

OSPF – Open Shortest Path First

PKC – Public-key Cryptography

PKG – Private Key Generator

PKI – Public-key Infrastructure

RSA – Rivest, Shamir and Adleman

SA – Security Association

SDK – Software Development Kit

SKC – Symmetric-key Cryptography

SOAP – Simple Object Access Protocol

SQL – Structured Query Language

SSL – Secure Socket Layer

TCP – Transmission Control Protocol

TLS – Transport Layer Security

UDP – User Datagram Protocol

URL – Uniform Resource Locator

USB – Universal Serial Bus

VPN – Virtual Private Network

WSDL – Web Services Description Language

XML – Extensible Markup Language

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, the introduction of this work is performed. First, the potential for ad-hoc networks and its contrasting low usage in spontaneous contexts is commented. After that, the context and motivation for this work are presented. Next, the objectives and main contributions of this work are described. Finally, the structure of this document is presented.

1.1 Introduction to Ad-hoc Networks

The potential of ad-hoc networks is very high for workgroups and organizations. The possibility of connecting a group of machines in a network without any additional infrastructure or configuration is useful for workgroups, particularly nowadays when the number of mobile devices is exponentially increasing. It is also cheap for an organization because no complex infrastructure needs to be bought nor maintained.

So, if this alternative exists, why is it so often ignored? There are various plausible reasons for this. The most probable reasons are the lack of application support and also the lack of commercial interest. No appellative economic model exists for these networks precisely because they are cheap to use. Therefore there is no encouragement to the usage of these networks. Large scale applications for ad-hoc environments are therefore out of question. But there can still be small and medium scale applications for workgroups and small organizations who seek no cost or low cost alternatives.

There is also no complete support for ad-hoc networks. When someone connects to a network, a minor spontaneous configuration is expected. The user expects the connection process to be plug-and-play, when in reality it is not. Many times people end up introducing Internet Protocol (IP) addresses manually, especially if their machines are not using the same operating systems. Actually, many of the smartphone operating systems do not even have their ad-hoc network support activated. Additionally, ad-hoc network research mostly targets routing protocols instead of support or possible applications for these networks.

It is easy to conclude that ad-hoc networks are forgotten by many users because their applications and support are not a main concern. If this would change, there would be more possibilities for these users. Some of them could actually make use of these new possibilities. The purpose of this work is developing a simple framework that tackles these problems, showing that it is possible to make ad-hoc networks easier to use than they are today.

1.2 Context

Creating and making the necessary configurations in an ad-hoc network is harder than a typical infra-structured network. In a small infra-structured network with a single access point (e.g. networks found in homes, cafés and restaurants), the access point can easily configure all nodes, typically using standardized protocols. Larger networks with several access points (e.g. networks found in a large building or a university campus) have a common supporting infra-structure that is able to configure the other nodes using the same protocols.

In a spontaneously created ad-hoc network this necessary configuration is not as easy to perform. Every node is an access point and there is no easy way of attributing a configuring role to one node. The techniques used in infra-structured networks are not as easy to apply in this scenario. Ad-hoc networking configuration routines can be accomplished with complex extensions of the routines used in an infra-structured network.

The main problem is the lack of standardization. Each operating system has its own way of attributing IP addresses and practically none of them uses an ad-hoc routing protocol. At least there could be a standard about the mechanisms that should be used. The access point where a new node connects could try to attribute an IP address that it has not seen in the network. At the very least each machine could attribute a random IP to themselves instead of waiting indefinitely for the configuration routine that the operating system uses. In terms of ad-hoc routing there are hundreds of protocol implementations to choose.

Some mobile operating systems such as Android are not even able to connect to an ad-hoc network. The only ways of connecting to a network are wireless infra-structured networks, 3G connections or some complicated tricks involving USB or Bluetooth. It is true that 3G networks have a very big coverage but why should it be necessary to have an Internet connection just to communicate with another machine in the same room? Creating a spontaneous local network simply synchronize files or other data between a smartphone or a personal computer without requiring more hardware or an Internet connection would be nice.

Ad-hoc networks can also have Internet access. One machine with a network interface connected to the ad-hoc network and another one connected to the Internet can share its Internet connection with the other machines. This can be achieved with a smartphone with a wireless interface that can connect to an ad-hoc network and a 3G connection to the Internet. It can also be achieved with a laptop with a wireless interface connected to an ad-hoc network and a wired interface connected to a phone line or to an infra-structured network with Internet access.

1.3 Motivation

People are often not aware of the potential of an ad-hoc network. People either do not know that they exist or they think that they are very complicated things, that they only work sometimes or that they do not allow Internet access. But ad-hoc networks can be very simple, can have Internet access and can even have some useful characteristics. An ad-hoc network has an extensible range as every node can be used an access point, which propagates the signal to a wide area. Packets can be sent to almost any point in the network as they can be relayed via several nodes. Moreover, at short distances packet delivery is theoretically more efficient as each packet is sent directly to its destination instead of having to travel through a router or a switch.

On the other hand people are right when they say that an ad-hoc network without Internet access is almost useless. In fact, not only ad-hoc networks but local networks in general have low usability for the masses. Local network services are usually very rarely used. People either use applications that do not make use of any type of network at all or use Internet services. Local network services usually require complex installation routines. It is not just double clicking an installer or dragging a folder to an icon. Many times it requires editing several complex files just to actually start the service.

It is also hard to understand how they function as they are usually running in the background and their feedback is typically a log in a console. People prefer to double click a program or just go to some website and see something. But that can also be easily achieved with local services. A local network service can have a simple and intuitive web interface, both for normal utilization and for configuration. Additionally, if each service can be packaged into a single file, installation routines can be very simple.

This would generate a new type of application, parallel to the other established two types (desktop applications and the Internet services). This new type would not compete with the established ones as the number of possible usages for it is not very large. Regular applications are easier to develop as programs instead of services for a server. Additionally, most Internet services take advantage of being available all over the world. The only interesting niche for these local services are services that would require communication between machines in a local context. This makes this type of application less appealing for developers.

This new application paradigm obviously needs interested developers in order to grow. But this area does not need to grow much, as the number of interesting services is not gigantic. It just needs some useful services that can be massively adopted. These services can be developed on the basis of personal need by part of the developer and can be distributed afterwards as a free and/or open-source project. These services can also be developed targeting donations. For more specific cases, such as companies, custom software can be developed based on specific needs.

1.4 Objectives

All referred details motivate the development of a framework that increases the usability of ad-hoc networks. The framework should provide plug-and-play support along with fast and easy development, packaging and installation of services adapted to these environments. Additionally, the framework should be usable in multiple operating systems, being at least one of them designed for mobile device such as smartphones. The main goal is therefore the development of such a framework. This main goal can be divided in several smaller goals.

The first goal is the development of plug-and-play support for spontaneous mobile ad-hoc networks. This support should contain ways to attribute IP addresses and names to hosts, discovery and advertisement of services hosted in the network and generation of routes between machines in the network. As the network can be public, these tasks should be as robust and secure as possible against network attacks.

The next goal is the development of a mechanism that encrypts the messages traded in the network. This mechanism should also provide a way to authenticate the sender and the receiver of each message. As the network can be public or belong to an organization where authentication is important, preventing eavesdropping and ensuring authentication are important goals. This mechanism can be a networking protocol, such as a transport layer protocol. Developing such a protocol requires the development of a transmission control mechanism, that can be designed specifically for a wireless environment.

Another goal is the development of a service framework that allows the simple usage of services designed specifically for a mobile ad-hoc environment. The services should be web based, so they can be used the same way as Internet services. Services should also be easy to develop, deploy and install. For more complex network services, a simple way to interact with services in other machines should be provided. These mechanism should be adapted to the characteristics of a mobile ad-hoc environment.

Finally, the framework should be multi-platform. It should at least work in Android and in a Linux operating system. Android development is made in Java, therefore that is the language that should be used. This way, it is easy to develop source code that can be used both by Android and by Linux. With small changes the code should also be able to run in almost any other operating system such as Windows, Mac OS and Linux based mobile operating systems such as WebOS and MeeGo.

1.5 Contributions

This work presents four contributions. The first contribution is a transport layer protocol that allows the transfer of authenticated, non eavesdrop-able and non fabricable messages. Local networks are some of the best places to eavesdrop data and tamper with services. The three referred characteristics are very important in an a local network environment. This protocol is also more adapted to the wireless environment than the most generally used Transmission Control Protocol (TCP).

The second contribution is a set of auto-configuration mechanisms that allow multi-platform plug-and-play support for spontaneously generated mobile ad-hoc networks. The developed mechanisms solve the biggest problem of these networks, which is the low user friendliness. The difficulties of configuring each machine in a way that allows the correct functioning of the network is the biggest obstacle that these networks have to overcome in order to increase their utilization.

The third contribution is a service framework that allows an easy development, deployment and installation of services. Along with the user unfriendliness, another problem with these network is the lack of easy to use services. The most used services are hosted on the Internet and are many times proprietary or designed to be run in a large infrastructure. An easy way to develop and distribute new services, designed for an ad-hoc environment is required to bring both users and developers.

The final and major contribution of this work is the developed framework, that combines the other three contributions. The framework autonomously allows an easy and secure utilization of an ad-hoc network using Linux and Android operating systems. Services developed for the framework can be easily installed and hosted. This combined approach is perhaps the best option to increase the usability of an ad-hoc network. Each individual contribution is important, but each one individually is unable to attract more users.

1.6 Document Structure

This document is divided in six chapters. This present first chapter contains the introduction of this work. First, the potential for ad-hoc networks and its contrasting low usage in spontaneous contexts is commented. After that, the context and motivation for this work are presented. Next, the objectives and main contributions of this work are described. Finally, the structure of this document is presented.

In the second chapter, the state of the art of the areas that are tackled in this work is presented. First, some relevant background information about networks and some of the focused areas is given. After that, the state of the art of auto-configuration in mobile ad-hoc networks is presented. Next, the state of the art of middleware for the tackled areas is presented. Finally, an analysis of the referred approaches is performed.

In the third chapter, possible solutions for each focused area are discussed and the integrated solution is designed. First, the target environment is defined, some initial thoughts are collected and the main components to design are chosen. After that, some scenarios where the designed solution can be useful are presented. Next, the main components of the solution are designed and described. Finally, a brief comment about the integration of every component is given.

In the fourth chapter, the implementation of the solution is described. First, an overview of the implemented framework is presented. After that, details about the implementation of each one of the main components are provided. Next, the differences between the implementation of the framework for Android and for Linux are described. Finally, a brief comment about the final status of the implementation is given.

In the fifth chapter, some results of this work are presented and analyzed. First, an introduction to the results is given. After that, the specifications of the used machines and some practical results are presented. Next, the performed tests are described. Their results are presented and analyzed. Finally, some conclusions obtained from the analyzed results are presented.

In the sixth and last chapter, the conclusion of this work is performed. First, a recollection of the performed work is made and some obtained conclusions are presented. After that, a critical analysis about some aspects of the performed work and the way it was performed is presented. Finally, the work that should be performed to further complete this work is enumerated.

Chapter 2

State of the Art

In this chapter, the state of the art of the areas that are tackled in this work is presented. First, some relevant background information about networks and some of the focused areas is given. After that, the state of the art of auto-configuration in mobile ad-hoc networks is presented. Next, the state of the art of middleware for the tackled areas is presented. Finally, an analysis of the referred approaches is performed.

2.1 Background Information

Before starting to focus on the state of the art itself, it can be helpful to perform a small overview about some concepts and technologies than are important for the correct understanding of this document and the performed work. These technologies are widely used in these days. They are the types of wireless networks, the types of cryptographic algorithms and some networking protocols that are used every day. This makes them a platform for the designed solution or valid comparison points, in the case of the protocols.

In the next sections, background information that is relevant for this work is presented. The first section provides an insight to the functioning of networks and the role of IP addresses. In the following sections, various types of wireless networks are described. After that, various types of cryptographic algorithms are presented along with the functioning of an authentication infrastructure based in cryptography. Finally, details about some protocols and technologies that are relevant for this work are presented.

2.1.1 Simple Introduction to Networks

A computer network can be seen as a set of nodes partially connected with each other via links (a graph). These nodes can send messages through links to their adjacent (neighboring) nodes. These messages can travel through various links to reach nodes that are not adjacent as long as there is a path between the sender and the destination nodes. To allow messages to be sent to specific destinations it is necessary to identify the nodes in some way and navigate the messages through the graph until they reach their destination. This is achieved with the IP and the Media Access Control (MAC) respectively.

Each node has at least one IP address that is used to identify it in the network. Additionally, each node also has a MAC address that allows it to be identified by its neighbors. It is easy for a node to send packets to all its neighbors, because they do not need to find a destination. Each node can therefore easily broadcast their IP and MAC addresses to every neighbor. This way each node is able to obtain information about their neighbors and send messages specifically to them.

But what if the destination of a message is not a neighbor? In that case, the message must be sent to a neighbor that can forward it to the destination node. The message may travel through several nodes that forward it to one of their neighbors until that neighbor is the destination of the message. For this, it is necessary to somehow generate routes in every node. These routes should contain information about which neighbor to send the message in order to reach the destination. If this is done correctly, the message should be delivered. Unfortunately, generating these routes is not always simple.

Nodes must somehow understand if the messages they receive are meant for them or for them to forward or not for them at all. For that, it is necessary to place information about the destination in the message. Each message is encapsulated in a packet containing headers, which hold the information about the source and the destination. To reach the destination a message is sent from the original sender in a packet with the MAC address of a neighbor and the IP address of the destination. If a neighbor that is not the destination receives the packet, it does not change the IP address of the packet but changes its MAC address to one of his neighbors and sends it. If the neighbor is in fact the destination, it just receives the packet and extracts the message.

A node in a network requires at least one IP address, one MAC address and routes to any node that one might want to communicate with. It is also highly recommended that IP address are unique throughout the network and that no two nodes with the same MAC address are neighbors. MAC address are attributed by the vendors of the network interfaces used in networking. The process of MAC attribution is controlled by these companies who try to minimize the possibility of two interfaces with the same MAC address joining the same network.

But two challenges still remain when it comes to keep a network working smoothly. It is necessary to somehow attribute unique IP addresses to each node in a network and to generate routes between nodes. There is the possibility of doing these tasks manually but it is a hassle when it can be done automatically. Depending on the structure of a network, doing this automatically can be more simple or more complicated. These tasks are more difficult when the network contains many nodes or there is no hierarchy between them.

2.1.2 Wireless Networks

Wireless networks are composed by various nodes that communicate with each other without wires within a physical range[1] [31]. The number of possible links is very high, especially if the nodes in the network are not very scattered. At least one node in the network acts as an access point, which allows new nodes to join the network. Small networks usually only have one access point, which facilitates the task of assigning IP addresses. Any joining node contacts the access point to connect to the network and within the connection process, the access point attributes an IP address to the joining node. As the access point attributes every IP, it is easy to maintain uniqueness.

In bigger networks (such as the one in figure 2.1), having a single access point

[1]Nodes in wireless networks communicate with each other by sending electromagnetic waves through the air and other materials. There is a physical range for these waves and it may vary with many factors such as physical obstacles and nearby radiation.

may not suffice. In wireless networks with various access points (sometimes connected to each other via wires), IP address allocation is not so simple. The access points are usually manually configured in order to assign a specific range of IP addresses to joining nodes. As this range is different for each access point, there is no risk of IP collisions (two nodes having the same IP address). Routes between access points can also be manually added.

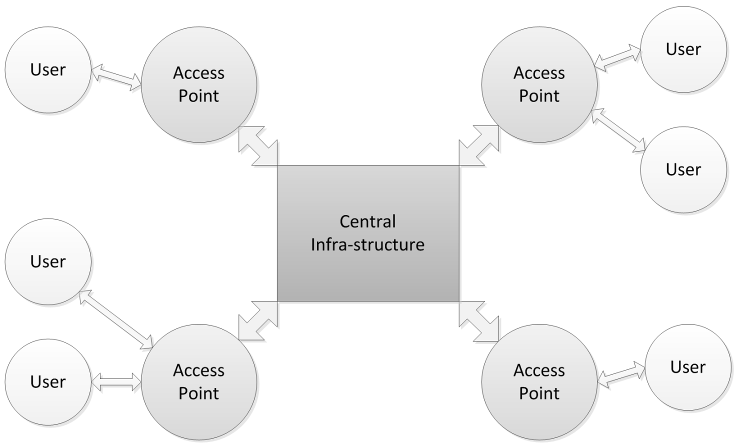


Figure 2.1: An infra-structured wireless network

Another task is the generation of routes for the joining nodes. It is realistic to consider that when connecting to an access point, that node should know best about routes, since it assigned the IP addresses. One possibility is each joining node adding a single route that sends every packet to the access point or the router specified by it. Another possibility is adding another rule that assumes each node in the network to be a neighbor and every packet with a destination IP belonging to the local network can be sent directly to the node.

2.1.3 Wireless Ad-hoc Networks

There are some more specific types of wireless networks, such as wireless ad-hoc networks [43] (figure 2.2). Networks of this type do not contain any node acting specifically as an access point. Instead, every node belonging to the network acts as an access point. This property allows a wireless ad-hoc network to be generated spontaneously by any node without any specific hardware to act as an access point. If someone wants to connect two machines in the same network and can not connect to any wireless network nearby, that someone can just create an ad-hoc network with one machine and then join the other machine to that same network.

As every node acts as an access point, the network is extensible[2]. This charac-

[2]As each node has a physical range for communication and acts as an access point, adding more and more nodes propagates the network to a wider area.

teristic makes it more likely that not all nodes are neighbors of each other, especially if a network expands to a very wide area. To make sure that communication between every node is possible, routes must be added. Additionally, as every node can act as an access point, the previous method for IP address assignment is not applicable.

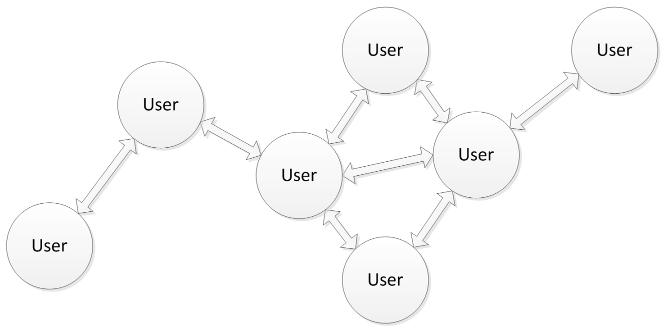


Figure 2.2: A wireless ad-hoc network

A wireless ad-hoc network is therefore a completely decentralized network. There are no inherent hierarchically superior nodes that can easily assign unique IP addresses, serve as default gateways who know best about routes to other nodes, serve as a default gateway to the outside of the network (the Internet, usually) or take care of multicast groups. Software provided in some operating systems for the management of wireless network connections provides various methods for IP assignment and route generation. Unfortunately, none of them copes with the extensibility of an ad-hoc network [30].

2.1.4 Mobile Ad-Hoc Networks

A Mobile Ad-hoc Network (MANET) [23] is a specific type of wireless ad-hoc network where nodes can physically move and therefore change the neighboring nodes of themselves and their neighbors. This characteristic complicates the generation of routes even more, because new routes can appear and old routes can disappear at anytime. But node movement causes an even bigger problem. In some cases of movement, the network can be splitted into several partitions.

Wireless communications have a limited range. Because of node mobility, new links between nodes can be formed and old ones can be broken, when nodes get in or out of range. When there is no possible path between two hosts in the same network, the network is partitioned [40]. Nodes in different partitions have no way of communicating between each other and therefore it is not possible for a node to know about every other node in the network. This makes the task of unique IP assignment even more complicated than it already was in plain wireless ad-hoc networks.

Keeping the known network with no duplicated IP addresses is not enough, as it may only be a partition. When two nodes from different partitions get in range, their two partitions merge. Suddenly, all nodes in both partitions are able to communicate.

Since IP assignment occurred independently in both partitions, it is possible that there are nodes with duplicated addresses that can now communicate. This process can be observed in figure 2.3. IP assignment in MANETs therefore requires techniques to detect and treat duplicated addresses.

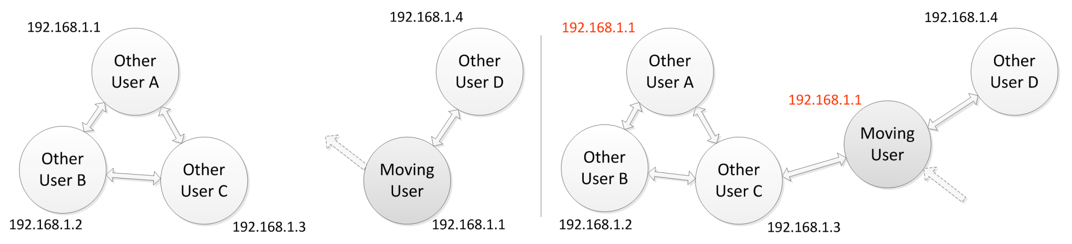


Figure 2.3: Example of a duplicated IP at a partition merge

2.1.5 Symmetric Key Cryptography

From the origins of cryptography until not too long ago, cryptography has essentially been a synonym of encryption and delivery of secret messages. Nowadays, the properties of modern cryptographic algorithms allow cryptography to be used for many other tasks such as authentication, digital signing and non-repudiation. Cryptography has evolved a lot in the last few years and is nowadays used in an almost daily basis.

Symmetric-key Cryptography (SKC) [10] is a form of cryptography where the same key is used both to encrypt and to decrypt a message. This key, usually called secret key or shared key, is used by two or more entities to exchange ciphered messages. Messages encrypted with a certain key can only be decrypted with it, therefore entities that possess the same key can trade messages in a secure way as long as no other entities know the key that they used.

There are many examples of basic SKC algorithms in classical cryptography. A simple example is the Caesar cipher. This cipher encrypts a message composed by letters and uses a number as the key. It rotates each letter in the message the number of times used as key. For example, encrypting the word EXAMPLE using the number 3 as key generates the word HADPSOH. To decrypt the message, the same number is used to rotate each letter the opposing way. Modern SKC algorithms such as Rijndael [17], Twofish [55] and Serpent [3] are obviously much more complex and are based in bits instead of letters but the Caesar cipher is a fairly simple example that gives an insight on these algorithms.

The main problem with SKC algorithms is the necessity to interchange the secret key. In a network where new nodes can join at anytime, it is impractical to distribute keys verbally or using pluggable storage hardware such as USB storage devices. The only practical way to share keys is using the network itself. But to send these keys

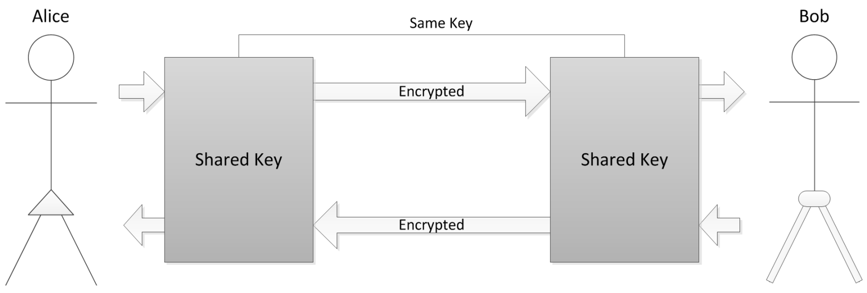


Figure 2.4: An analogy of the symmetric-key cryptography

over the network requires them to be encrypted. Using just SKC it is impossible to apply cryptography in a dynamic network environment.

2.1.6 Public Key Cryptography

Public-key Cryptography (PKC) [45] is a subtype of Asymmetric-key Cryptography (AKC). As opposed to SKC, AKC algorithms use different keys to encrypt and to decrypt messages. Keys are generated in pairs and are associated with each other. Each pair contains a public key that is used to encrypt messages and a private key that is used to decrypt messages. The private key is the only one able to decrypt messages that were encrypted using its associated public key. Additionally, the private key is not deducible from the public key. These properties allow the freely distributed of public keys throughout a network to anyone interested in sending messages to the owner of the key.

The mechanisms used in PKC and the differences between PKC and SKC are possibly easier to understand with an analogy. Alice and Bob want to trade secret messages through the mail. To make these messages secret, they send them inside a box closed with a padlock. In a SKC analogy (figure 2.4), one of them must obtain a padlock and its key. Then, it makes a copy of the key somehow gives the copied key to the other (probably in a meeting). After both of them have a key to the padlock, they are able to exchange their secret messages.

In an PKC analogy (figure 2.5), no meeting would be necessary. Each of them obtains a padlock and its key independently. When one of them wants to send a message, it requests the padlock from the one other and uses it to lock the message inside a box. The box can then be sent and only the intended recipient can open it. In this analogy, the padlock works as the public key and its key works as the private key. This type of cryptography is very important as it does not require the transfer of anything that allows the decryption of messages throughout the network.

PKC algorithms, such as Rivest, Shamir and Adleman (RSA) [54], can be used for communication between several entities. As long as each entity that wants to receive

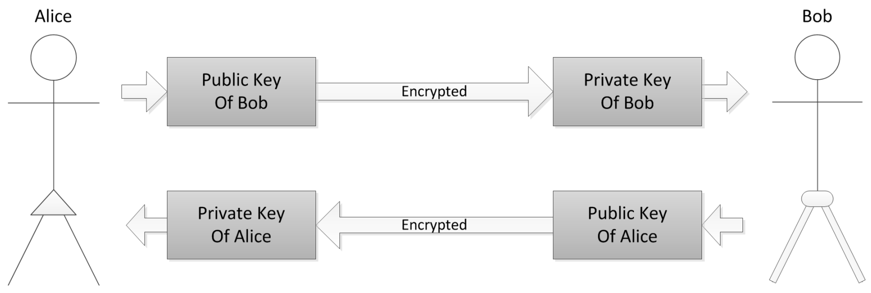


Figure 2.5: An analogy of the public-key cryptography

messages has a key pair and its public key is provided to each entity that wants to send messages, PKC can be used for data interchange. Unfortunately, PKC algorithms are slow when compared to SKC algorithms, for example. Fortunately, the advantages of both can be combined. SKC algorithms can be used for secure communication as long as the shared key can be traded and PKC algorithms can be used for the secure exchange of the shared key.

2.1.7 Public Key Infrastructure

Cryptographic algorithms by themselves are usually only used to encrypt and decrypt messages. But some properties of PKC can be used in authentication. It is not easy to impersonate the ownership of a certain public key. The purpose of having a public key is receiving messages that only the owner can decrypt. Therefore as long as the impersonator is obliged to understand the messages sent to him, he requires the associated private key to do anything useful. With this, one can acknowledge that the owner of a certain public key is virtually always the same entity. But usually this is just the first part of the authentication process.

It is hard to impersonate a public key but not the entity behind it. Someone can impersonate a specific entity using any other key pair. The necessity for a Public-key Infrastructure (PKI) is the mapping of a public key to an entity and vice-versa. A PKI [51] allows the verification that a specific entity must use a specific public key. With this second part of the authentication process, it is also not possible to impersonate an entity without having the private key associated with the public key of the entity.

This mapping can not obviously be made by the entity itself. It requires a trusted third party to acknowledge that the entity is in fact who is said to be. That third party is called a Certificate Authority (CA). A CA issues, for each registered entity, a digital certificate which binds that entity to a public key. When a second entity wants to authenticate a registered entity, the latter provides its certificate to the second entity. With the certificate, the second entity is able to authenticate the other one with the help of the CA that issued the certificate.

There is also the possibility of not attributing any entity to a public key and instead consider the public key as the entity. In this scenario, the ownership of the public key is the only required method of authentication and there is no need for the second part of the authentication process. This can be adapted to the PKI authentication model using self-signed certificates, which are certificates issued by the owner of a key instead of a CA. These certificates are considered insecure by most network service clients, which show flashy warnings when faced with one [29].

2.1.8 Hypertext Transfer Protocol

The HyperText Transfer Protocol (HTTP) [22] is the networking protocol that is typically used by web browsers and many other clients to interact with services. It allows the client to make many different types of requests such as resource obtainment and procedure invocation. The protocol is very simple because it is completely textual and not binary. Because of that it is independent from the system architecture and easily extensible. It is a request based protocol, where each request gets a response from the server.

An HTTP request has two main components: the header and the content. The header is composed by several textual lines using a specific syntax. The first line contains the request method, the Uniform Resource Locator (URL) [8] of the resource and the HTTP version to use. Each of the remainder lines contains a header field, which is a key/value pair, with the two components separated by a colon (`:`). These pairs contain information about the request that is relevant for the server such as the length of the content and the expected content type of the response. The header ends with an empty line and is proceeded by the content. The content typically contains parameters for the request. An example of a minimal HTTP request is presented in figure 2.6.

```
GET /index.html HTTP/1.1        | First line (method, URL, HTTP version)
Host: www.example.tld           | Header field (key/value pair)
                                | Empty line (end of header)
```

Figure 2.6: Example of an HTTP request

An HTTP response has a very similar format. The only difference is in the first line of the header where the method and the URL that are only relevant for a request are replaced by a response status code. This code can indicate if the request was completed successfully, if an error occurred (using different codes for different errors) and even tell the client that the request should be redirected to another URL. The content usually contains data, such as the requested resource, or a human readable error message. An example of a minimal HTTP response with content is shown in figure 2.7.

```
HTTP/1.1 200 OK                    | First line (HTTP version, status code)
Content-Length: 10                 | Header field (key/value pair)
                                   | Empty line (end of header)
1234567890                         | Content (10 bytes)
```

Figure 2.7: Example of an HTTP response

The HTTP protocol is stateless and therefore each request must contain any existing state. Alternatively, the server behind the HTTP interface may keep the state of each client identified by an ID. This state is usually called a session and is frequently used. When sessions are kept by the server, the HTTP request only needs to contain the session ID. Variables such as the session ID or other relevant information for the request are usually placed in three places.

Some variables are placed directly in the URL of the request. These are called GET variables and are useful because a user can easily interact with URLs when using a web client such as a web browser (either by writing them or clicking hyperlinks). Others are placed in the content of the request. Such variables are called POST variables and are usually big structures such as all the information from a web form. Finally, variables can also be placed in header fields, typically in a field called Cookies. These Cookies are important because they are usually managed by the web client and the web server and are used a local client state.

2.1.9 Transport Layer Security

The Transport Layer Security (TLS) [19] is a networking protocol that allows secure communications between two participants. TLS is typically used over TCP and its goal is authenticating at least one of the participants and securing the transfered data using encryption. The TLS protocol achieves this by using certificates. TLS may only authenticate the server but it can also authenticate both the client and the server. Any authenticated entity must provide its own certificate to the other entity, therefore unless the client uses a certificate only the server is authenticated. It is more common to only authenticate the server.

The process of establishing a TLS session where only the server is authenticated can explained in a simplified manner. First, both participants decide which TLS version, public key and symmetric key algorithm and corresponding key sizes are going to be used. For this, the client sends the options that it supports. The server then chooses the best ones that are supported by both of them and sends its decision. It also sends its certificate that contains a public key of the chosen public key algorithm. The client then validates the certificate by contacting the CA that issued the certificate. Finally, the client uses the public key of the server to encrypt and send a secret key of the chosen symmetric key algorithm, that both are going to use for the secure communication process.

A handshake where both the client and the server authenticate is not very different. The main difference is that both participants exchange their certificates and both validate the certificate of each other. In a TLS session establishment it is also possible to use self-signed certificates. These certificates are not considered secure by the TLS protocol because they can not be used to authenticate an entity using a PKI. They can only be used to encrypted the exchanged data.

As a final note, it is important to refer that using HTTP over TLS is very common. This combination of protocols is usually referred as the HyperText Transfer Protocol Secure (HTTPS) protocol [53]. This protocol is the standard mechanism used by web clients, such as web browsers, to securely send HTTP requests and receive the their corresponding responses from the authenticated server. When the server uses a self-signed certificate, web clients display warnings referring that the connection is not secure and ask the user for permission to proceed, which is usually not very appealing for the user.

2.1.10 Internet Protocol Security

The Internet Protocol Security (IPsec) [39] is a protocol suite that allows the exchange of encrypted and authenticated IP packets. While TLS works over TCP, IPsec operates at network layer, between IP entities. It provides mechanisms to ensure integrity and confidentiality to IP packets exchanged between an authenticated IP source and an authenticated IP destination.

Security services are provided by inserting special IPsec headers into the IP packets: an Authentication Header (AH) [37] or an Encapsulating Security Payload (ESP) [38]. AH provides authentication and integrity to all payload data carried inside the IP packet including the IP header (except for some fields that may change in multi-hop transfers). It can be applied alone or combined with ESP. The ESP header can be used to provide confidentiality, but also data-origin authentication, integrity and anti-replay service. The difference between integrity and authentication services provided by AH and ESP is the extent of the coverage, since ESP does not cover the fields in the IP header.

IPsec security services can be applied between a pair of hosts, a pair of routers or between a router and a host. It can operate in two modes: tunnel mode and transport mode. In tunnel mode, IPsec security services are applied to the entire IP packet that is encapsulated inside a new one with the AH/ESP headers. The new packet is addressed between the tunnel endpoints. This mode is usually applied only between routers or firewalls. In transport mode, security services are applied only to the payload data of the original IP packet. The AH and ESP headers are included between the IP header and the transport layer header.

Both IPsec headers include a field called Security Parameter Index that identifies a Security Association (SA). That identifier is used by IPsec entities as an index in their table of active SAs. A SA is simply an encrypted connection. When receiving a packet, the receiver extracts the IPsec header and performs a lookup on the SA table to find the correct way to process that packet. The table includes information about the keys that must be used, the algorithms, the sequence numbers, etc. In a similar way, the sender also consults its SA table to see how to encrypt the packets. Entries in the SA table are created according to some specified policies that dictate which packets should be protected. These policies are described by rules entered in a Security Policy Database.

Before creating a SA for a given connection, IPsec entities must first negotiate some security parameters to use. This negotiation requires itself a master security association that has to be previously established. In the first SA, entities have to perform mutual authentication, either by using a pre-shared key or by using public key signatures. This process may use a PKI. After this first phase, further SA negotiations can be carried on in a protected way. This key negotiation is done using the Internet Security Association and Key Management Protocol (ISAKMP) and the Internet Key Exchange (IKE) [36].

2.2 IP Auto-configuration

One of the aspects that needs to be addressed is the auto-configuration of an IP address. The IP is used to uniquely identify a machine in a network, therefore each machine in a network must have an unique IP. Usually networks can do this in a spontaneous way but that is not possible in a MANET. As a MANET can split into partitions it cannot have a unique server that is accessible by every host, or even that has information of all the nodes in the network. That can cause conflicts when two or more partitions merge. Still, for the massification of MANETs a mechanism that allows spontaneous assignment of unique IPs is necessary [34, 40].

Usually IP addresses are either statically assigned by a network administrator or dynamically assigned by a Dynamic Host Configuration Protocol (DHCP) [20, 21] server. The first case is the precise opposite of the pretended auto-configuration. The use of DHCP is also not a very viable alternative. Every arriving node must be configured by a DHCP server. It is also necessary that the configuration is only made by one server or by various collaborating servers. Due to the partitioning problem this is hard to maintain. Additionally, once a DHCP server disconnects, another node needs to start a server to substitute it. Lastly, the DHCP server will probably need some configuration. Solving these problems would require additional software running on each host in the network that would constantly monitor the state of the DHCP and DHCP-relay servers. It would also require that every host in the network had a DHCP server configured exactly the same way. This is not a very practical solution. It would be best to design a solution that would do both jobs at the same time and without requiring any configuration.

Afterwards, the Internet Engineering Task Force (IETF) [32] created the Zeroconf Working Group [33], with the goal of creating a set of mechanisms that allow a zero configuration network. One of the approached areas was the IP configuration [13, 59]. The proposed mechanism was broadcasting a randomly generated IP and then waiting until another machine responded that it already has that IP. If the wait times-out, the proposed address is unused and can be auto-assigned. The process is repeated until a free address is found. This process is valid for most networks but sadly not for MANETs due to partitioning. But this initial Zeroconf alternative can be slightly adapted to satisfy the host mobility requirements of MANETs. As it is easy to implement, it can be easily extended. If a duplicate address detection system is implemented it becomes a viable alternative. The detection system can be the periodic broadcast of advertisement messages containing the IP address of the host.

The modified Zeroconf approach is a completely distributed solution. One centralized solution is the Agent Based Addressing [24]. This approach tries to continuously have one and only one centralized IP addressing agent per network partition. When a new host joins the network it chooses a temporary IP from a defined range (which is assumed to be unique) and broadcasts a request to obtain an IP. The agent responds with an unassigned IP that the new host can use. Note that the agent is centralized and therefore has access to all assigned IPs in the network. If the new host does not receive an answer after a specified time it assumes that there is no agent in the network and becomes one.

There is the possibility of a host (which can be the agent) leaving the network and also the possibility of network partitioning. To assure that there is one and only one agent in the network, the agent broadcasts a heartbeat message and every host that receives it responds to it. A host that does not answer to the agent will probably have left the network and will have its address released. If a host does not receive a heartbeat message after a while, it assumes that there is no agent in the network and becomes one. If an agent receives the heartbeat message of another agent, one of them must leave its agent post. Typically this is decided by comparing the numbers from the IPs, and the lowest or biggest will stop being the agent.

Examples of both centralized and distributed approaches have been presented. Still, distributed approaches can be subdivided in stateless and stateful. The Zeroconf approach is an example of a stateless alternative. A stateful approach usually makes use of full replication. This means that all nodes have full knowledge of the IP addressing state in the network by using consistency control protocols. An example of such an approach is the MANETconf [47]. When a new host joins the network it broadcasts a neighbor discovery message and every other host answers. If no response is received until a timeout the new node assumes that is the only node in the network and assigns an IP for itself. Otherwise, the new node (the requester) chooses one of the nodes that answered (the initiator) and asks an IP address. Every node has the list of assigned and pending IPs. The requester chooses an IP that is neither assigned nor pending, adds it to the pending list and broadcasts this IP. Every other node checks if the IP is neither assigned nor pending. If its not, the IP is added to their pending list and an affirmative reply is sent to the initiator. Otherwise, a negative reply is sent. After the initiator has collected all the replies, if at least one is negative, an abort message is broadcasted, the IP is removed from the pending lists and the process is restarted. If all the answers are affirmative, a commit message is broadcasted, the requester can use the IP which is also removed from the pending lists and added to the allocated lists. The new node can generate his list of assigned IPs from the initial neighbor discovery query responses.

Once again, detection of nodes exiting the network and network partition is necessary. Every time a new node joins the network and needs to be configured, the initiator asks permission to assign a chosen IP and waits replies from all other nodes. If one node left the network or the network got partitioned, no answers will be received from the nodes that are no longer accessible. The IPs of these nodes can be removed from the assigned lists. To detect partition merges, a partition ID is assigned to each partition. This ID is a tuple containing the lowest IP in the network and a unique ID proposed by that node. Partition merging can be detected by routing table analysis. When two nodes establish a link they exchange their partition IDs. If the two are different, they also exchange their allocated IPs lists. They broadcast the new list to the nodes in their partition and all nodes update their allocated lists. If an IP conflict is detected, one of the conflicting IPs must change. The node with the fewest TCP connections or with the shortest connections initiates the IP assignment routine using a non-conflicting IP node as the initiator.

Both no replication and full replication distributed IP addressing approaches have been analyzed. To complete this section, a partial replication distributed approach is presented. The Quorum Based IP Address Autoconfiguration [61] divides the network into a group of clusters. One member of the cluster is the cluster head which is the host responsible for assigning IPs to the remaining common nodes in the cluster. To belong to a cluster, a node must be at most two hops away from the cluster head. If a node joins the network and does not have any cluster head within two hops, it becomes a cluster head. Otherwise it becomes a common node. In either case, the nearest cluster head is the node that configures the joining node. Each cluster head keeps an assignable IP block, the list of assigned IPs and the list of the neighbor cluster heads.

The algorithm for IP assignment is very similar to the one from MANETconf [47] with two differences: the initiator is the nearest cluster head (not any other node) and not every node votes, just the cluster heads vote as a quorum. The algorithm for partition detection is also based on using the lowest IP in the partition as the network ID. When a node receives a message with a different network ID the nodes of the network with the biggest network ID will need to acquire new IP addresses from the other partitions.

2.3 Host Naming

Another problem with the spontaneous creation of MANETs is the attribution of unique human-readable names to machines. Users prefer to use intuitive, descriptive and easy to remind names instead of sequences of numbers or codes when referring to a resource. The most commonly used is the Domain Name System (DNS) [44], whose names are commonly part of URLs and email addresses. The problem here is very similar to the above IP addressing problem: the names must be unique although it is not possible for a machine to have access to the complete state of the network. But there are some slight differences. The name must not only be unique but also meaningful. This means that it is not only necessary to assign a unique name, the name is also supposed to be equal or similar to a wanted name.

One possible approach is the Manet DNS [1], where the existing DNS approach is modified to suit the MANET environment needs. To solve the problem of name uniqueness they use one centralized DNS server that manages all the names. This approach is very similar to the Agent Based Addressing [24] that was referred in the above subsection, but also has to guarantee that a host has a name that it wants. When a new host joins the network, it broadcasts a server solicitation message and the DNS server or any other node that knows the IP of the server unicasts a server advertisement message to the new node, which sends a name register message to the server. If the name is already taken the server notifies that a new name must be chosen. If no response is received after a while the new host assumes that there is no DNS server. In that case, it becomes one and broadcasts a server advertisement message, so all other nodes know that it is the new server. The nodes will then register their names in the new server. With the name conflict problem solved, the problem that now needs to be addressed is the existence of one and only one DNS server per partition.

If a network becomes partitioned then one partition will not have a DNS server and the other will have a set of unreachable host names registered. Also if the server leaves the network there wont be any server in the partition either. When a server tries to connect to a queried name and the machine is not accessible it notifies the DNS server that the host is no longer accessible and the name is removed from the table. In the partition that has no DNS server, a host will eventually notice that it has not been receiving server advertisement messages for a while. It then decides to become the new server and the already described process is repeated. When there are two DNS servers in one partition, one of the servers will eventually receive a server advertisement message and notice that there is more than one server. One of the servers, for example the one with the lowest IP address, will then send his name table to the other and will finish its role as a DNS server.

Another possible approach is the one provided by the already mentioned Zeroconf group [25]. This approach is the Multicast DNS (mDNS) which uses multicast to send query messages to a group of machines that has a registered name. Nodes can issue name-to-address or address-to-name queries in the multicast group and the machine with the correct name or IP address responds to the query. Each machine that wants to register a name joins a defined multicast group and multicasts its name. If a machine already has the wanted name it will respond with a conflict message and a new name must be chosen. Once again the initial Zeroconf approach was not designed for MANET and therefore partition caused conflicts can occur. The solution is the same as the one described above to maintain IP uniqueness, which is sending advertisement messages with the current name until a conflict is detected. Another problem that might exist in a MANET is the possible lack of multicast support. This approach is also possible by using broadcast instead of multicast which would actually simplify the solution. There is no need for a multicast group if all the nodes in the network use this service.

Another problem that has been referred and that has not been analyzed yet is how to generate new names when a conflicts occurs. As it has been explained in the beginning of the subsection, it is expected that one machine is able to be configured with a specified name. But what if the name is already used? A new name must be generated. Without any alternative given by the user, it is hard to generate a completely new meaningful name. Either the difference is very small (adding a number at the end of the name, for example) which can be confusing for users because they can hardly distinguish the two names or the difference is big and this usually results in names that are hard to remember.

One approach for this problem is the Dynamic Shortest Discriminating Names

(DSDN) [35] which solves this problem by forcing each user to choose not one single name but a list of various words. The generated names are a subset of the first words of this list concatenated with hyphens that separate them. The first generated name uses the first two words. When a name is generated it is broadcasted to every node. If a host already uses that name, it contacts the broadcaster and they both try to generate new non-conflicting names by adding more words. If the first n words are equal, both hosts use their first $n + 1$ words. If the list of both hosts is exactly the same, a random number is appended at the end. To solve the partitioning problem of the MANET environment, every node broadcasts its name periodically.

2.4 Routing by Name

One completely different approach is the Private Address Maps [35] that tries to eliminate IP addressing from the problem. This approach defies the standard networking approach where an IP is necessary for the identification of a machine and states that in a MANET environment, a machine should be identified by its name. In the current Internet networking model, a name is translated to an address and that address is later accessed through a route. In a MANET environment, one could broadcast a name resolution query and a route reply would come from the target machine. These packets will travel by a usable route that can be used again later. This way, it is possible to obtain a route directly from a name (routing by name) and IPs would simply become identifiers to next hops.

This approach makes an extreme use of Network Address Translation (NAT) to solve the problem. Each node has its own NAT domain where it defines an IP for itself and every host that it wants to communicate with. The address of a host in the network can be different for each node, and every node can assign the same IPs for themselves, because the IP is only useful inside each private addressing realm. When the name resolution query is issued, the query and its reply travel through various nodes. This path is a valid route and each of the nodes can save the necessary routing information. Both the source and destination nodes generate a private IP for each other and can now communicate with each other. Their NAT modules will translate the source and destination IPs when necessary.

The Private Address Maps [35] approach is very different from the approaches referred earlier in this document. Instead of trying to preserve IP uniqueness in the network, it removes this need by making an extreme use of NAT. This provides a conflict-free alternative to IP addressing, by reducing the role of IP addresses to the sole identification of forwarding paths. This approach just addresses the IP problem by implementing a “routing by name” protocol. It does not define a specific naming protocol. Any protocol can be used as long as it is based on broadcast for name querying. Still, the same document [35] presents DSDN, described in the previous subsection, as a possible naming protocol.

2.5 Spontaneous Mobile Ad-hoc Network Security

Another important matter to address in spontaneously generated MANET environments is security. Along with the problems of any local wireless network, where any node is able to listen to any transfered packet and is also able to send any packet impersonating any MAC and IP addresses, there are also problems specific to MANETs. In an environment where no central authority exists, it is not possible have any entity controlling any configuration securely. Every node must therefore collaboratively configure the network while trying to minimize the problems caused by any attackers.

As seen in some of the background information sections, cryptography can be used to solve some of these problems. Using PKC alone or in conjunction SKC, it is possible to encrypt packets, which prevents packet eavesdropping. It is also possible to use PKC for authentication, which can be used to prevent MAC and IP address impersonation. But in order to use PKC, it is necessary that every node has access to the public keys used by the other nodes. Since no entity can be trusted, it is not possible to define a trusted third party. Without one, every node must collect the public keys of the other nodes. Additionally, no PKI such as the one presented in section 2.1.7 can be used. Other ways to authenticate entities must therefore be used.

A possible approach for this is presented by SelfOrgPKM [60]. In this approach, each node defines a key pair for itself before joining the network. Using the public key and some more relevant information, such as a sequence number and an expiry date, the node generates a self-signed certificate. With a valid self-signed certificate, the node is able to join the network. While in the network, the node exchanges its certificate with its neighboring nodes for route establishment purposes. The certificate is provided to the remaining nodes on a need to know basis. This approach does not generate a mapping between public keys and entities, therefore each individual key is considered an entity.

Other approaches make use of Identity-Based Cryptography (IBC) [9, 57]. It is a type of PKC where any string can be used as a public key, typically using some unique information of a user. This type of cryptography requires the existence of a Private Key Generator (PKG), which is an entity that generates and provides the associated private keys to their owners. This node generates a master public/private key pair. The master public key is used by any node to generate any public key from its identifier. The private key is known only by the PKG, which uses it to generate the private keys. This is problematic in a MANET environment, since no node can be trusted and additionally, even if a node could be trusted, nodes may not be in the same partition as it.

But approaches such as one presented in Daza-Morillo-Ràfols [18] use techniques to distribute the role of PKG by a set of nodes. These techniques are based in distributing the master private key by various nodes, requiring more than one PKG node to generate a private key. For this task, this approach uses the concepts of secret sharing schemes [56]. These schemes basically allow the generation of a secret that is splitted in n pieces, each one privately given to a different node. The secret can be reconstructed by t of these nodes. This approach in particular uses a bivariate polynomial to split the secret key.

Another concern in MANET environment security is the secure generation of IP addresses, to prevent Denial of Service (DoS) attacks such as claiming every IP address. One possible way of securely generating addresses is using Cryptographically Generated Addresses (CGA) [4]. This technique allows the creation of IP addresses from public keys. It is possible to generate virtually unique IP addresses for every user in a network, as long as any user owns a public/private key pair. Using CGA, an attacker is unable to claim any IP address that he wants, such as IP addresses used by other nodes. To claim an address, the node must contain a key pair that generates that address and prove that it actually possesses it.

Another of the problems in a MANET is the generation of routes between nodes. Ad-hoc routing protocols typically use multicasting or broadcasting methods to advertise and/or discover neighboring nodes and the nodes that they can communicate with. With this information, there are many ways to generate routes between machines. The problem is that these multicasts or broadcasts can be generated by attackers, which can basically say that they have the best routes for every node in the network.

Since it is necessary to rely not only on the destination node but also on the nodes that are going to forward the packet, there is no way of generating routes that can not be negatively affected by attackers. Although it is not possible to completely prevent this, there are various ways to minimize this problem. One of these ways is analyzing the behaviors of other nodes to establish a trust value for each node. This trust value can be used to choose secure routes. Such an approach is used by the Trust Based Multi Path DSR Protocol [52].

But there are other possible approaches. The SRDV [16] approach uses the verification of end-to-end physical path characteristics to avoid suspicious paths. It uses characteristics such as end-to-end delays and feedback to detect possible attacks. It then uses load balancing to counter these possible attacks. Both these approaches rely on PKC for the authentication of nodes but the solutions that they use to protect the routes from attacks are very different.

2.6 Service Providing Middleware

After addressing the problems of automatic IP configuration, host naming and security in spontaneously created MANETs, it is time to address the services that will be used in those networks. There are many aspects that can be analyzed, from service organization to resource replication techniques. Several existing middleware with different characteristics will be analyzed, mainly focusing their major contributions.

Some of the first questions that can be asked are related with service organization, registry and delivery. How can services be registered and discovered? Is there any way to organize them? What parameters can a service discovery query contain? How can services be used by clients? These are the main questions that are addressed in the Konark middleware [28]. In Konark, every host has a HTTP server that can be used both as a client and as a server. Services are registered at the local HTTP server. Discovery requests can be issued via broadcast and hosts with matching services advertise them via unicast. Each server also collects information about the received advertisements and maintains a partial view of the services in the network. This view can be accessed by the user to check which services are available and where are they located.

Services are organized into a tree, similar to a file system directory tree. Each node of the tree is identified by a name and contains a collection of services and other nodes. Each leaf is also identified by a name and contains a specific service. Additionally, each specific service has a service type (printer, music) and contains a set of keywords. With all these information, service discovery queries can be very specific. Queries can be based on generic service paths, specific services names, service types and keywords.

Each service is described in a Extensible Markup Language (XML) based service description language based on Web Services Description Language (WSDL). The description contains the above information, a set of properties that describe the service in a human readable way and the set of functions that can be called in the service. When a host wants to use a service, it contacts the server that hosts the service and requests its description. With the description, which contains the description of the functions of the service, functions can be called using Simple Object Access Protocol (SOAP) [5].

Regarding network configuration, Konark assumes IP level connectivity. This way Konark can be used in any environment where IP can be implemented, such as 802.11 and Bluetooth. This was justified by stating that operating systems usually have support for ad-hoc networks.

There are some middleware that remove the burden of complete IP uniqueness. One example is the RAMP middleware [7]. In RAMP it is stated that networks should be configured in a mission-oriented way. The network is divided in several mission-oriented subnets, each with its local addressing scheme. Each subnet is generated for its “mission” (such as a group sharing some files) and is independent from other subnets. This means that the nodes from a subnet can only communicate with each other and not with the nodes of other subnets. Still, each node can belong to various subnets, and have a different IP address in each. Some nodes in different subnets can perfectly have the same IP addresses assigned, because communication is only performed inside a subnet.

In RAMP it is also stated that allowing heterogeneity of links in ad-hoc networks is important. Support for an ad-hoc network where, for example, some nodes communicate via 802.11 and others via Bluetooth is the main goal of RAMP. To achieve this goal, an application-layer routing approach that makes use of cross-layering is used. There is no direct client-to-server communication. Every communication is done via single-hops. In order to attain multi-hop communication, each packet traverses various single-hop connections, through various socket pairs. Reactive routing techniques are used: a route between hosts is generated when communication between them is wanted. The application layer acquires all the information it needs to allow the processing and forwarding of packets. Broadcast is treated as multi-unicast where packets are unicasted to every neighbor node. By limiting the number of retransmitting hops, overhead can be reduced in many situations where full retransmission is not necessary.

There is no complex service organization. Each service can only be searched by its name. Other than that, in terms of service organization, it is practically equal to Konark. Each service is registered locally and can be discovered via broadcast. When a query is issued, each node that hosts a service with the specified name advertises it via unicast. It differs from Konark because the broadcast can be limited to a certain number of hops, to find services nearby.

Another detail that needs to be addressed is replication. The REDMAN middleware [6] makes use of read-only resource replication to improve service availability and performance in dense MANETs. By definition, a node belongs to a dense MANET if the number of its neighbors is higher than a specified threshold and the node density is almost constant over time. In REDMAN, resources are replicated throughout the network, maintaining a specified replication degree. All protocols and operations are designed to be lightweight. They should require low network overhead and low energy consumption. With this in mind, there is no strict maintenance of the replication degree. It must be similar to the defined value but not exactly equal. Additionally, by only replicating read-only resources there is no need for complex reconciliation operations at concurrent replica updates.

Another goal in REDMAN is complete transparency in the replication operations. Developers just specify the metadata and replication degree of each resource. Then, in a completely transparent way, the resource is replicated throughout the network. Clients also discover and retrieve resources in a transparent way. Replication is controlled by the Replica Manager (RM). The RM is elected using a lightweight iterative heuristic that tries to detect a node near the physical center of the MANET without using any positioning system. Elections occur periodically due to extra battery usage by the RM and movements inside the MANET that may change its center. The RM maintains a Shared Resource Table (SRT). This table contains one entry per resource containing its replication degree and a weakly consistent set of hosts containing a resource replica.

When a host wants to register a resource, it contacts the RM and sends the metadata and replication degree of that resource. The RM adds the new resource to the table and calculates the total number of replicas that should be generated using the given replication degree and the number of hosts in the network. The resource owner becomes the first replica and notifies some of its neighbors that they can become replicas. The neighbors can choose to become replicas or to forward the request to some of their neighbors. The nodes that accept to become replicas contact the RM, which accepts replicas until the replication degree is reached.

Clients can discover and retrieve resources using broadcasting. The query can be limited to some hops, depending on the replication degree of the wanted resource. This is another good way of reducing unnecessary overhead. When a node detects that it is going to leave the dense MANET it warns the RM. In the warning it sends a list of the resources that are being replicated by it. The RM checks the corresponding replicas in its weakly consistent replica sets and contacts some of those replicas. The contacted replicas will then try to recruit new replicas to maintain the replication degree of the resources.

A final subject to approach in middleware for MANETs is programming. A study made by Collins and Bagrodia [14] shows methods for programming applications in MANET environments. This study considers technologies such as sockets and HTTP unsuitable for a MANET environment. The analyzed approaches do not use URLs for addressing and discovery nor sockets for communication. They focus in other mechanisms that are adapted to MANET environments, such as tuple spaces and publish/subscribe messaging patterns.

A tuple space is a model of communication between various nodes based on tuples that are shared by the nodes and that can be accessed concurrently. Publish/subscribe is a messaging pattern where the destination of the message is not defined. Instead, each message has a set of characteristics, that are defined by their publisher. Other nodes subscribe the characteristics of the messages that they would like to receive. It is possible to also subscribe all messages of a certain publisher.

Since in MANET environments nodes may have a high mobility, this environment is prone to disconnections. The analyzed approaches also use techniques to handle disconnections. Some use connectionless approaches. Some others use object proxies, for example. These are based in the proxy software design pattern. This pattern uses a proxy object that may have a reference to the real object. In the analyzed approach, if communication is possible, the proxy object communicates with the real object. If not, it allows an alternative behavior.

2.7 State of the Art Overview

There are some conclusions that can be reached when analyzing the state of the art as a unit. First of all, IP and name auto-configuration approaches are very similar. All of them are focused on multicasting or broadcasting the wanted IP address or name and use a mechanism to detect collisions and partition merges. Since the solutions for these mechanisms are so similar, these two configurations can be easily performed simultaneously.

In terms of security, the way to go for providing authentication in a MANET environment is using PKC. Each node uses a public/private key pair and a method to know the public keys of the other users. IBC does not require the distribution of public keys throughout the network but it seems much more complex than using regular PKC and distributing the public keys. Since IP and name auto-configuration routines also rely on multicasting or broadcasting, the public keys can also be added to these distribution routines. Another security problem is the secure generation of routes. Unfortunately, there is no way of completely solving this problem, there are only ways to reduce it.

Middleware for MANETs focuses in many different objectives, which is a good thing. It shows that there can be a big number of applications for MANETs due to their versatility. Unfortunately, most of this middleware is of low level (does not simplify much the task of developing software). This middleware does not allow the quick development of simple services. The presented programming approaches are excessively complex for an area that still does not have many developers. Other technologies that simplify development and are widely used, such as web development frameworks should be used. But those frameworks were not designed for MANET environments, which increases the difficulty of interactions between nodes in complex services. Still, it is not necessary to completely disregard the concepts of the widely used sockets and HTTP. These frameworks can be used as a valid base for a new service framework that is both easy to use and adapted to the MANET environment. Some examples of these frameworks are the Spring Framework [58] and the Ruby on Rails Framework [27].

Chapter 3

Architecture

In this chapter, possible solutions for each focused area are discussed and the integrated solution is designed. First, the target environment is defined, some initial thoughts are collected and the main components to design are chosen. After that, some scenarios where the designed solution can be useful are presented. Next, the main components of the solution are designed and described. Finally, a brief comment about the integration of every component is given.

3.1 Initial Considerations

Solutions for the creation of a ready to use ad-hoc network with specialized services that can be safely used in multiple environments are not common. A solution that integrates security, automatic configuration and services in ad-hoc environments seems therefore an interesting goal. It is also an ambitious goal, considering the number of tackled areas.

The main purpose of this work is the development of a working prototype that is able to fulfill a large number of objectives. The goal is not to address a specific topic and find the best approach on that area. The goal is to develop a good all-around solution that is capable of solving many problems at once. This work therefore follows one same principle in every focused area: simplify what can be simple. Every focused area is tackled with a simple approach.

In the following sections, the three first steps in the design of the solution are taken. First, the target environment for the solution is defined. Then, based on this definition, a brainstorm with the goal of collecting ideas for the design is performed. Finally, the collected ideas are used to generate a road-map for the design, containing the main goals that should be achieved in the construction of the solution.

3.1.1 Environment Assumptions

Before starting to design a solution that fulfills the proposed objectives, it is necessary to define the environment in which the designed solution should operate. That definition is based on a set of assumptions regarding the target environment.

The main assumption is that the environment may not be secure. This means that the network is composed by regular users and attackers. A regular user is a user that is utilizing the developed framework and expects it to work properly in every occasion. An attacker is a user that is trying to gain access to any information relevant to another user or degrading (maybe even blocking all together) the user experience of regular users. From the point of view of a regular user, any other user can be an attacker.

An attacker has various network related ways of achieving his goals. A local wireless network is not exactly the most private network environment. It is assumed that an attacker can eavesdrop to every packet sent by any user. He is able to obtain the packet data along with any header field, such as MAC and IP addresses, unless the data or field is encrypted in a way that can not be decrypted by the attacker. An attacker is therefore able to eavesdrop any conversation occurring in the network and collect a list of the MAC and IP addresses used in the network.

An attacker is also able to send packets at anytime and to freely control the content in those packets. He can control the packet data and any header field, including once again MAC and IP addresses. It can be concluded that an attacker can use any MAC or IP addresses, including the ones from the eavesdropped packets. If any of those two fields is used to identify a user, an attacker is able to impersonate any user in the network.

As any MAC or IP addresses can be used, the attacker is also able to use different ones in different packets. He is able to assume multiple identities at anytime, being able to impersonate or create numerous identities in an instant. It is also impossible for any user to conclude that a certain user is an attacker, because that user can not be identified by an IP or MAC address. Therefore, attackers can not be expelled from the network.

If attackers can not be expelled, the best that can be done is minimize their range of actions. To achieve this, it is necessary to develop methods that allow secure authentication of users, providing a basis for secure and authenticated packet interchange. This can be achieved using cryptography. It is assumed that the cryptographic algorithms used in this work are not themselves vulnerable. It is also necessary to reduce as most as possible any form of degradation or privation of the regular user experience (DoS attacks).

This last goal can be achieved by applying one rule to every developed mechanism: a user can only trust another with tasks that affect no other (mutual security rule). Assuming a securely authenticated scenario, two users that are communicating are unable to impersonate any regular user. Therefore, if routines are designed to only affect the relation between the two participants of a communication, other users can not be affected in any form. An attacker is only able to degrade his relation with the other participant, which has no interest for him. This last principle is only valid for mechanisms that follow the above rule.

The final assumption is that any node is capable of connecting to a network. This work does not focus the creation of ad-hoc networks, nor the process of connecting to one. The mechanisms that are designed in this work act after the node has successfully connected to a network.

3.1.2 First Thoughts

Now that the target environment is defined, it is time to collect a set of ideas about what should be focused in the design of the solution. These ideas should focus different matters such as security, auto-configuration and services. The goal after the collection of ideas is the generation of a road-map, containing the areas that should be focused and the goals that should be achieved in the design of the solution.

It is considered that the target environment can be insecure. Security is an important matter in public networks and corporate environments. Authentication is essential for a secure environment. It is the key to allow secure packet exchange, user differentiation and non-repudiation. It is important that packets traveling through the network are only legible by both the sender and the receiver. It is also important to make sure that both the sender and the receiver of the packet are exactly who they say they are.

People use infra-structured networks because they are pretty much plug-and-play. When a user connects his machine to an infra-structured network, he is able to almost instantly contact any other host in the network and maybe in the Internet (in case there is an Internet gateway in the network). This can also be achieved in an ad-hoc environment through an automatic configuration of IP addresses and routes between machines. In a local environment, it is also important to identify machines by human readable names and not only their IP addresses.

Since it is intended to secure the target environment, these auto-configuration mechanisms should also be secure. They are essential for the correct functioning of the network and therefore they are a good target for DoS attacks. The designed mechanisms should not be affected in any way by attackers that want to hinder these mechanisms from working correctly.

Some level of user authentication can be achieved by solely using public keys. Since no node can be trusted, it is not possible to attribute to a node the role of assigning identities to public keys (figure 3.1). Therefore, user authentication based solely in public keys is the best can be achieved in this environment. It is a more than valid form of authentication, as long as each public key is considered an identity. This method suppresses the need for any type of identity mapping (figure 3.2).

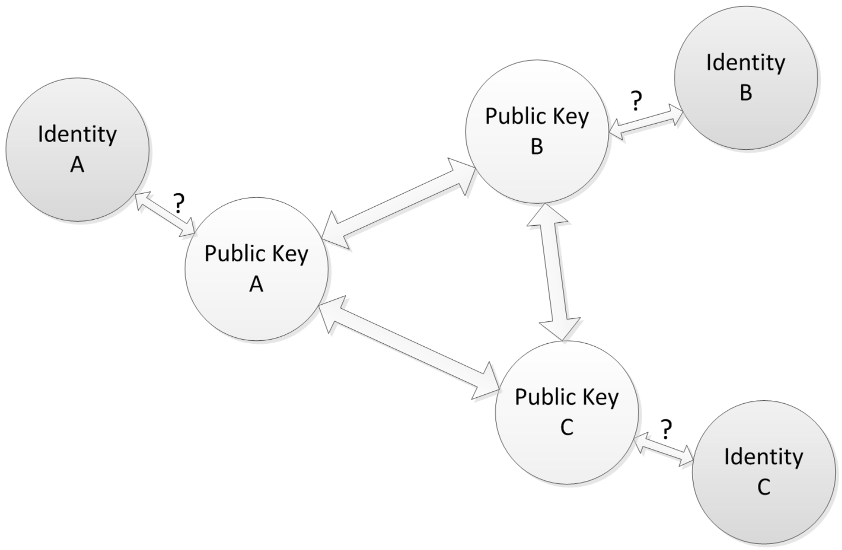


Figure 3.1: The problem of assigning identities to public keys

If no node can be trusted, no node can contain a centralized database of public keys. For this reason, each node must possess its own database. Using the mutual

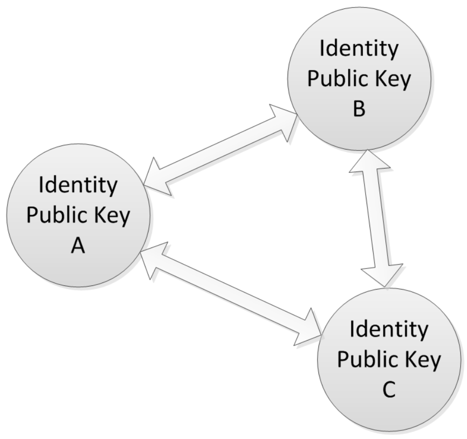


Figure 3.2: Using the public keys as the identities

security rule, each user can register his own public key in any other user. It is therefore possible for every user to securely construct a database with the public keys of every other user in the network. After the creation of this database, authenticated information can be transfered. The secure construction of these databases can also be used for the other secure auto-configuration routines.

This process of collecting the public keys every user in the network (user discovery) is essential for the correct functioning of the network. Other approaches, such as Konark [28], use service discovery routines to discover which services are running in the network. This is essential in Konark because no user discovery is performed. In fact, the concept of user or node is not relevant in that approach, it is only a mean to access the service. The approach of this work is the opposite. This approach performs user discovery and each user is considered as the owner of a set of services. Service discovery is not essential in this approach as a contacted user is able to provide a list of his services when required.

If a network has many services running in many hosts, there should be a framework to assist the development and deployment of these services. This framework should allow the development of services that are adapted to an ad-hoc environment. It should take care of many aspects, such as authentication and access to personal data. It should also allow service collaboration between equal services running in different hosts. Generically, it should ease up the whole process of service development and minimize time consumption.

Services are becoming increasingly web based. One of the main reasons for that, is that HyperText Markup Language (HTML) is a widely used markup language that is build upon standards. The only thing that is needed in an operating system is a web client (web browser) that is able to make HTTP requests and understand the content of the HTTP responses, in order to correctly display the contents of the responses.

No support of any specific programming language, virtual machine or technology is required because a browser can be developed for any platform, using any technologies.

Building services for the framework based on HTML over HTTP is a good solution for maximizing client compatibility. Virtually every operating system has at least a default browser, capable of presenting HTML content satisfactorily. Following the trend, web development is also becoming more common. Developers are having more formation and becoming more experienced with web development. Creating the framework over a web basis is also a good way of attracting developers that would develop useful services that can be widely used.

Browsers are able to establish secure and authenticated connections using HTTPS. Unfortunately, for HTTPS to consider a connection as secure, it requires the utilization of certificates signed by a CA, and each authentication process requires the CA to be contacted. This may not possible as the network may not have Internet access to confirm the requests. Additionally, signing certificates requires payment, which is something that would drive away a large set of users. HTTPS can also be used with self-signed certificates but connections using them are considered insecure and the web clients show very unpleasant warnings about that fact.

Another possibility is using IPsec to encrypt every transfered packet. A process that generates and inserts SAs for connections in the SA table, secured by the public key databases could be developed, in order to encrypt the HTTP communications. Unfortunately, in terms of IPsec, Android only provides official support for Virtual Private Networks (VPNs) with a server, which is unsuitable for this completely decentralized environment.

Another option is the implementation of a secure protocol. This protocol should not be developed over TCP because it would be vulnerable to any known TCP attacks. This protocol must therefore be a transport layer protocol. It adds a security layer by allowing authentication and encryption of the transfered data. Since it is a transport layer protocol, it must also take care of the mechanisms that guarantee successful and efficient data transfers. Since implement these mechanisms is mandatory, these mechanisms should at least be adapted to a mobile ad-hoc environment.

3.1.3 Design Road-map

After the definition of the target environment and the brainstorm that originated the first thoughts, every condition required to enumerate a set of specific goals that should be achieved in the design of the solution is met. The goals of the solution designed in this chapter are the development of:

- A transport layer protocol:
  - Mutual user authentication;

  - Secure data transfer;
  - Support for secure network auto-configuration;
  - Optimized for wireless multi-hop data transfer;

- Support for ad-hoc networks:
  - Plug-and-play support;
  - IP and name auto-configuration;
  - User advertisement and discovery;
  - Packet routing between hosts;
- A service framework:
  - Easy service development and deployment;
  - Adapted to a mobile ad-hoc environment;
  - Web based (clients of the framework are web browsers);
  - Simple interaction between services in multiple hosts;

In the remainder of these chapter, the design of the transport layer protocol, the plug-and-play support and the service framework is studied and described. Each of them is separately analyzed and after that, an overview of the whole solution is presented. But first, a set of scenarios where a solution such as the one designed in this work can be useful is presented.

3.2 Possible Scenarios

When someone requires a small or medium sized network, their first thought is rarely associated with ad-hoc networks. An ad-hoc network could be a perfectly viable and economic option for many of these situations. Normally, this alternative is barely considered and the solution is immediately directed towards an infra-structured network.

The next sections present some situations where an ad-hoc network can be a good alternative. These situations contain small sized networks, such as networks for workgroups. Additionally, they also contain some slightly bigger and more specific networks, such as networks for emerging companies and various types of commercial areas.

These examples would be plausible if the software required for them had been previously developed. If there was specialized support for ad-hoc networks, this software would almost surely be developed. Many of this software could be very similar to the one used in some Internet services or some Enterprise Resource Planning (ERP) software. It would probably just need some small adaptations to make it fit for a mobile ad-hoc environment.

3.2.1 Workgroup

A group of college students needs to finish a project with their laptops. They all come from other towns, therefore none of them has Internet access in their residences. They usually meet in a place with an infra-structured network that has Internet access, such as the university or some café. But it is quite late and every place where they could work with Internet access is closed. The deadline for the project is obviously tomorrow. They need to complete the project and they need to be fast.

To speed up and ease up the development of projects, people started to rely on services that synchronize files is various machines automatically and in real time. This services replicate the shared files somewhere on the Internet and use this replica to synchronize the files in the various machines. The problem is that this requires Internet access, which is something the students will not have access to.

But is Internet access really necessary? Every laptop from several years ago to now is more than capable of creating an ad-hoc network. Practically everything that can be done in a network like the Internet can also be done in a local ad-hoc network. Therefore, the students can meet in one of their houses and spontaneously create a network. In that network they could make use of a local network service that allows the synchronization of files using only the replicas in their machines, without resorting to a master replica somewhere else.

Another service that would probably speed up the development of their project would be a tool that allowed the simultaneous edition of documents by all the members in the group. That way, they could edit the same documents simultaneously, viewing in real time the alterations the other members have made. The document could be any plain text file, such as LaTeX document, a program code or even a brainstorm logger.

Another possibility would be connecting one of their smartphones to the ad-hoc network. Most smartphones has not only 802.11 wireless connectivity but also 3G connectivity, that allows connecting to the Internet in almost any urban area. The smartphone could then be used as a gateway to the Internet. This way, the laptops would be able to use the services that they normally would use to speed up their project.

3.2.2 Emerging Company

Some emerging companies nowadays tend to most use free alternatives, than paid ones. A company without much money will not spend money in IT infra-structures, with several machines and licensed software. They will tend to use existing cheap or even free solutions in the web. They will use existing email servers, store information and host their webpage in cloud services.

They will also probably have small headquarters. The contents of the building may vary depending on the type of company. It may contain an office with some rooms, a small conference room, a lobby with a client greeting area or even a small warehouse. It is a small space without wide gaps between rooms, allowing laptops and smartphones to be in range of one another. In terms of IT equipment probably the personal laptops and phones of the workers will be used. In this case, a valid network solution would be an ad-hoc network.

The only other thing needed would be Internet access, to allow access to the virtual infra-structure of the company somewhere in the cloud. The 3G connectivity of a smartphone would be a solution but the payment plans probably will not compensate when compared to buying a land line. Buying a land line would provide a telephone line and an Internet connection via ADSL or fiber optics. In this case a laptop or a desktop computer (possibly the client greeting machine) could be connected directly to the Internet, and be used as a gateway to the laptops and smartphones of the other workers.

ERP software could be installed in the machines of all workers. The database used by the ERP could be also in the cloud. If it would be small enough, it could be placed in a desktop computer that would continuously be connected, such as the network gateway. If the ERP used would be a network server, it could once again be installed in a desktop machine and possibly contain the database in it.

3.2.3 Campus Building

A university typically has an infra-structure that controls all the network. This infra-structure allows a lot of things such as Internet connectivity and connectivity between any two points in the network. It also allows many other services such as the web pages of the university, where students can search for information and enroll in courses and exams. In order to be connected to the university network, a machine must somehow have connectivity with the infra-structure, which may require a big number of steps.

The department of Computer Science of a university typically has a whole lot of computers. These computers are scattered throughout classrooms, study rooms, laboratories and offices of teachers, technicians and student groups. There is at least one computer in practically every room of what can be a large building. It is quite possible that any computer in that building has point-to-point access to at least one other computer in the building. An ad-hoc network can then be used to connect all the computers in the building.

But there are more ways to connect all these machines. These computers can be connected using a cabled infra-structure. But this requires a lot of wires, switches, plugs and structures to place all of this. All these must be purchased. This structure must be also be mounted in a way that does not hinder the usability of the building and that is aesthetically pleasant. At the same time, any malfunction in any of these components needs to be fixed, therefore easy access to these components is required. This process is costly, hard to achieve and time consuming.

Another alternative is using a wireless infra-structure. This is achieved by placing signal repeaters that act as access points for the same network scattered throughout the building. This solution is better than the previous one as anyone can bring their laptops and connect to the network anywhere and not just in a free network plug that has a free cable. But this solution raises another problem. Each repeater must be connected to the infra-structure that controls the network. This is usually done using a cabled structure. The cost and complexity are much lower but the cables still need to be placed somewhere.

As it has been seen before every machine in the building can be connected in an ad-hoc network. The computer density should be sufficiently high all day long thanks to students, teachers and technicians using their machines or the ones in the various rooms. The only thing needed is that at least one machine has connectivity with the infra-structure that controls the network. This will minimize the costs and time needed for the installation, assist troubleshooting and problem solving, as well as keep the building virtually cable-free.

3.2.4 Shopping Mall

There are many reasons to have a network in a shopping mall. One of them is people going to social areas and cafés to work or to spend some time. Another is to allow shopkeepers to advertise their products and some discounts and promotions. Both shopkeepers and the mall owners have interest in maximizing the number of costumers that visit stores. Anything that can be done to attract new clients or to increase the interest of older clients ought to be a good investment.

Practically every store has at least one computer, that is used at the very least to process sales in combination with a register box. As these computers are connected almost all day long they are good access points for an ad-hoc network that will try to cover all the mall. With the exception of big open areas, there is practically always a store nearby, which makes the covered area almost all of the mall. The big open areas will typically places to sit where people bring their laptops to work or entertainment.

Shopkeepers can host in their sales computer a service distributed by the mall owners developed for advertising. The main goal will be that costumers connected to the network (typically with their phones) will have access to a catalog of stores. Additionally, for each of the stores costumers should have access to highlights, novelties, promotions and product listings with prices. Each shopkeeper should organize the information in a way they see fit to attract the biggest amount of clients to their store, and consequently to the mall.

Focusing more specific examples, a clothing store could present their new articles along with their discount clothes. A restaurant in particular could present their menu and their specialties. An computer store could advertise a clear-out sale of some articles. Some shops could even host 5 minute discount campaigns, which would probably be an incentive to paying attention at the advertisements. Anything that would make customers buy more products, even if just for a short amount of time, would be beneficial for the mall.

3.2.5 Other Commercial Areas

Similarly to the mall described above, many other commercial areas could benefit of a similar network. Stores outside malls usually have at least one computer as well. Any store can easily host the same service described above. There is no reason for a store not to benefit from free advertising. And obviously, if stores benefit from it so do many other people in town, such as the town mayor.

Many times commercial districts and city centers have a wireless network infra-structure. But this infra-structure is similar to the one that uses multiple signal repeaters presented in section 3.2.3. Similarly to the mall example, store computers and personal machines can spread connectivity over a large area. In this scenario there could be wide areas without stores nearby that would create conditions for partitioning. These areas would need signal repeaters, but the number of signal repeaters necessary could be easily reduced.

There are many areas where such a network could be widely accepted. A town that thrives with tourism would probably attract more tourists to itself and its center. A commercial district and a commercial plaza would benefit the same way a mall does. A city center has many buildings that are typically not found in many types of commercial areas but that would also take advantage of advertising.

Focusing some structures that can be found in city centers, theaters could list their current plays. Museums could describe the art exhibits they currently host. Some stores could also provide other types of services. Restaurants could have a table reservation service. Take-out restaurants could have an order service to have their food ready at the time of pickup. Museums and theaters could sell or reserve tickets to reduce queue lengths. Once again anything that would attract more customers, being them locals or tourists would be positive for the area.

3.3 Transport Protocol

One of the main goals of this work is the development of a transport layer protocol suitable for a secure mobile ad-hoc environment. The main goal of the protocol is to allow the transfer of encrypted information throughout the network, providing protection against eavesdropping. Another goal is the implicit authentication of every packet sent, providing protection against impersonation and packet fabrication. This can be used not only for secure data transfer but also for some auto-configuration routines.

A transport layer protocol also has the goal of making sure that the messages are delivered to their destinations. It is also necessary to develop an appropriate mechanism that can do this task. The chosen approach should be optimized for the MANET environment since these mechanisms have a big impact on the performance of the protocol. The most used transport layer protocol (TCP), reduces packet throughput when packets are loss. Although this reduction can have a better result than no reduction at all, too much reduction degrades performance unnecessarily. This is particularly important in a wireless multi-hop environment where radiations and physical obstacles affect packet delivery.

Java only provides support for sockets using TCP or User Datagram Protocol (UDP) over IP. These socket implementations take care of everything in the network and transport layers. To implement a transport layer protocol from scratch it would be required to extend Java using Java Native Interface (JNI) to support raw sockets, which allow the construction of whole packets, including the network and transport layers. But using raw sockets in Android can be problematic as many tasks require root permissions[1]. Additionally, raw sockets are much more complicated to use when compared with TCP or UDP sockets.

Using raw sockets does not seem the best alternative. There is no need to remake the network layer as IP is more than suitable and well supported by all devices. As for the transport layer it is possible to adapt UDP. The UDP protocol does not implement any error or flow control, nor any connection mechanism. It simply sends the packets from one local port to a port in another host. It works almost the same way as not having any protocol at all. It just adds a multiple port environment, a packet length field and a checksum. These are all useful features, therefore it is perfectly viable to build a new protocol socket over the Java UDP socket.

The new transport layer protocol is built over UDP datagrams, allows authentication at the packet level and has control over the packet flow. It can therefore be named Authentication Based Controlled Datagram Protocol (ABCDP) (figure 3.3). The core aspects of this new ABCDP protocol are detailed in the remainder of this section.

3.3.1 Securing the Protocol

One of the main goals of the protocol is being secure. Each message is sent to a specific user and only he should be able to read it. In a wireless network any machine can

[1]This problem is more detailed in section 4.5.

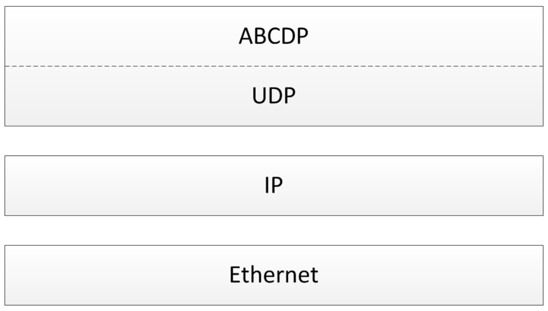


Figure 3.3: ABCDP protocol stack

receive messages sent to any user. Two machines exchanging packets through the air is similar to two people talking normally to each other: anyone near can eavesdrop the conversation. To prevent other people from eavesdropping conversations, the most secure method is talking in a way that only the two participants can understand. In a computer network there is only one way to achieve this, which is using cryptography.

As referred earlier in sections 2.1.5 and 2.1.6, there are at least two types of cryptography: Symmetric-key Cryptography (SKC) and Public-key Cryptography (PKC). The first uses a key that is shared by every participant of a conversation to both encrypt and decrypt messages. For this, the shared key must be distributed in an already secure method, which renders this type of cryptography unsuitable on its own for a MANET environment.

The latter uses different keys to encrypt and decrypt messages. Each participant uses a pair of keys, containing one public key to encrypt messages and one private key to decrypt them, and uses the public key of a user to send messages to him. Any public key can be freely distributed in an unsecured environment, since it can only be used to encrypt messages sent to its owner. The owner of a public key is the only one that possesses the associated private key, and therefore the only one able to decrypt messages sent to him.

If every user owns a key pair and some method is developed to distribute public keys to every user in the network, PKC can be used to communicate in a secure way between any two machines. Unfortunately, PKC algorithms are computationally slow when compared to SKC algorithms. Fortunately, PKC provides a secure environment to exchange shared keys. This way, connections between two machines can be established using messages encrypted using PKC, that contain a shared key. After the connection is established, messages encrypted using SKC can be securely exchanged.

So far it is possible to establish connections and exchange messages that can not be eavesdropped. But another wanted feature is authentication. IP and MAC addresses can not be used as authentication material since an attacker can freely duplicate any of them. Actually, an attacker can freely duplicate any content in the packet, therefore nothing that is not encrypted can be used as authentication material. The authentication material must then be something related with cryptography.

In fact, public keys can be used as authentication material. Since only the owner of a public key is able to understand messages encrypted with that key, he can authenticate himself by proving that he can understand a message. If he does, then he can be defined as the owner of that public key, which is something that supposedly no one else is. This way, it is possible to authenticate a user based on the public key that he uses. Considering that in this scenario no node can trust any other, it is not possible to define trusted third parties. Because of that, there is no way to computationally attribute an identity to a public key, and therefore public key authentication is the best level of authentication that is possible in this scenario.

Using this public key authentication mechanism, it is very simple to establish a mutually authenticated connection (figure 3.4). Each of the two participants choose a half of the shared key that they will use to encrypt the messages of the connection. Each of them sends their half to the other, encrypted using the public key of the other. If the participants are the actual owners of the public keys, each of them understands the half of the public key sent by the other and both have the complete shared key. Only the participants that did not lie about their public keys can complete the secret key and therefore any liar is unable to send or receive messages via the established connection. Any message sent that is exchanged via the established connection is therefore implicitly authenticated by both participants. The connection process of the protocol is more detailed in section 3.3.3.

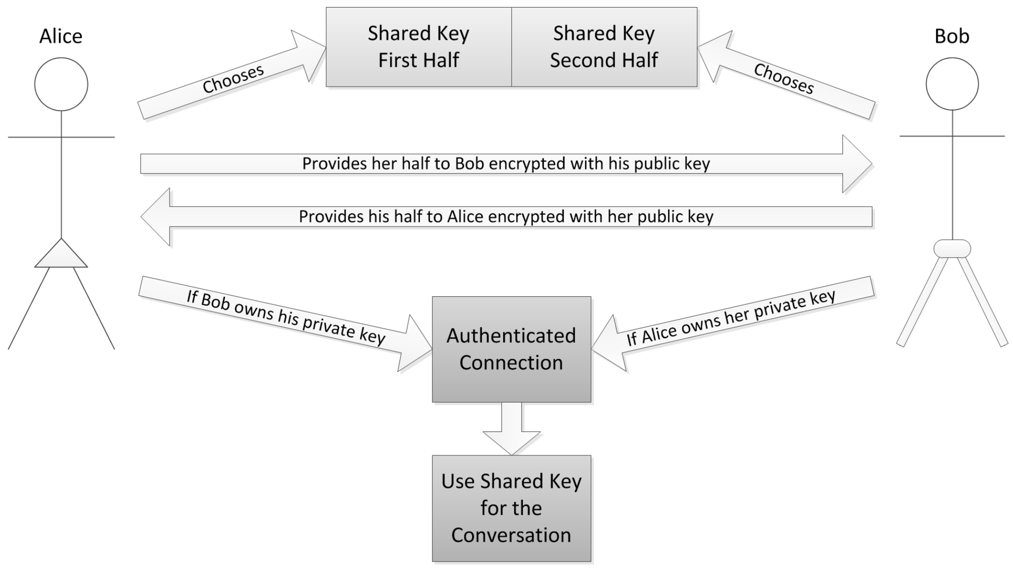

Alice
Bob
Shared Key First Half
Shared Key Second Half
Chooses
Chooses
Provides her half to Bob encrypted with his public key
Provides his half to Alice encrypted with her public key
If Bob owns his private key
If Alice owns her private key
Authenticated Connection
Use Shared Key for the Conversation

Figure 3.4: An authenticated connection

In conclusion, a simple method for the establishment of secure and authenticated connections using solely PKC and SKC was designed. The only remaining task is the selection of the cryptographic algorithms that should be used, along with their key sizes. The chosen PKC algorithm is RSA [54], because it is one of the most used public key algorithms. The chosen key size is 1024 bits. According to a study [41], this key size should be bigger, but that study focuses any possible application for the key. In a local environment, this key size should be more than enough. Also, the key size can be changed at anytime.

The chosen SKC algorithm is Rijndael [17], because it is the algorithm that won the contest that chose the Advanced Encryption Standard (AES), hosted by the National Institute of Standards and Technology of the United States of America [46]. The chosen key size is the lowest possible (128 bits). According to the same study [41], this key size is more than enough for any application. The Rijndael algorithm is referred as AES in the remainder of this document.

3.3.2 Control Mechanisms

Packets can not simply be sent to their destination without any control. It is necessary to make sure that every packet reaches its destination. This is commonly the job of a transport layer protocol. A transport protocol such as TCP has the purpose of ensuring that every packet is received and that the transmission is completed as quickly as possible. This is typically done with receiver acknowledgements, packet loss minimization and throughput maximization.

Packet loss can have various causes. TCP assumes that every packet loss is caused by its arrival at a full buffer in some network device. Packets are kept in specific buffers in memory, while waiting to be sent or received by other processes. When a packet reaches a full buffer it is discarded because there is no place for it. When this happens, TCP assumes that the packet throughput is excessively high, which is a logical conclusion when assuming that it was caused by a full buffer. To counter this, TCP reduces packet throughput drastically and then slowly increases it until a point that seems to fit.

The problem is that in a wireless environment there are other causes for packet loss [26]. The packet is sent as an electromagnetic wave and must travel through air and many times through solid obstacles. Packets can be deformed by interfering with other electromagnetic radiation[2]. Temporary packet loss can also occur due to moving obstacles. It is impractical to assume that all packet losses are caused by excessive

[2]A packet can be lost due to deformation both in a wired or wireless environment. A packet can be altered by interacting with other electromagnetic waves. A receiver may not detect a packet due to frequency, phase or amplitude alterations or severe changes to its format. Smaller alterations are detected when comparing the packet with its checksum field. These verifications occur in the network interface and the packet is discarded before it is sent to the transport layer. Deformed packets are therefore not seen by the transport layer the same way as any other lost packet, being treated the same way. This type of packet loss is much more common in a wireless environment as there are many more electromagnetic interferences in the air than inside a cable.

throughput.

A multi-hop environment also provides more conditions for error occurrence as errors can occur in any hop. The packet may have to travel through longer physical distances (traversing more obstacles and being influenced by more radiation) increasing the likelihood of packet losses. Packet reception is also dependent on the quality of signal emission and reception of all the intermediate nodes. Even if trying to always use the best path, errors will be more common than in a wireless infra-structured network, particularly in larger networks.

It is easy to conclude that TCP was not designed for wireless environments, particularly mobile multi-hop environments. TCP uses three control mechanisms: error control, flow control and congestion control. Each of these control mechanisms has a specific purpose in the task of controlling packet throughput. Any protocol with this same function must have some sort of mechanisms similar to these. The way these mechanisms are applied in the ABCDP protocol is described in the remainder of this section.

Error control allows a packet sender to know that its packets have in fact been delivered. The error control mechanism applied in ABCDP is almost equivalent to the one applied in TCP. This mechanism is based in a sequence number inside each packet. When the receiver receives a certain number of sequential packets, it sends an acknowledgment packet with that number. The sender then knows that every packet with a sequence number below that one has been received. But what if packets or acknowledgments fail their delivery and no feedback is received?

If the sender does not receive feedback after a while, it starts re-sending packets starting from the last acknowledged sequence number. If the feedback does not reach the receiver, it sends a receiver ready packet with the sequence number of the next wanted packet. The sender reads this packet as "send me packets starting from this sequence number", treating it as an acknowledgement to the prior number, and starts re-sending packets starting from that sequence number. If the receiver notices that a packet is missing, e.g. packet n and $n+2$ are received but $n+1$ is missing, it sends a receiver ready with number $n+1$ (figure 3.5). When a missing packet is received, the receiver sends a receiver ready packet requesting the next missing packet (figure 3.6).

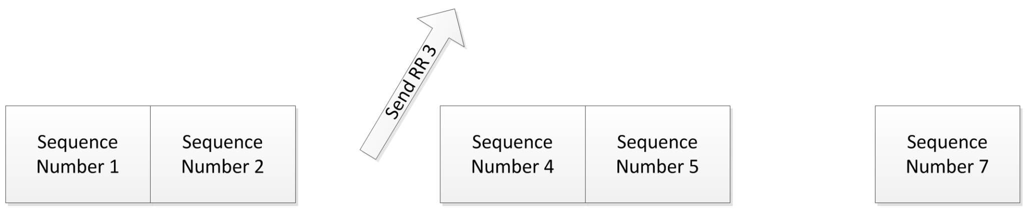


Figure 3.5: Missing packets

There is no need to implement a new error control algorithm. This mechanism

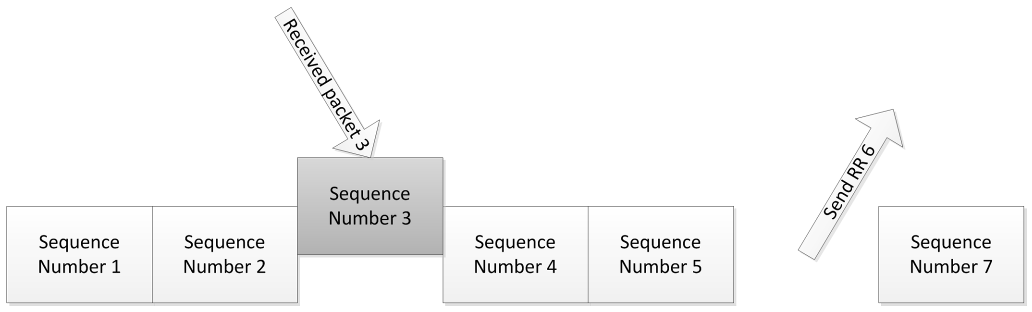


Figure 3.6: Reception of a missing packet

is also perfectly viable for a wireless environment, as it just provides feedback about packet delivery to the sender. But the methods for deciding how many packets should be received until an acknowledgement is sent and the waiting for feedback, new packets and missing packets can be different. These methods and values are controlled by the other control mechanisms.

Flow and congestion controls limit the packet throughput in order to prevent the occurrence of errors. Designing and implementing complex versions of these mechanisms requires a lot of work. This work unfortunately takes time that can not be spared at this time. But the implementation of these mechanisms is mandatory to ensure that the ABCDP protocol works smoothly. Fortunately, it is possible to develop simple control mechanisms that are adapted to a MANET environment. Therefore, the best alternative is simplifying the control mechanisms as most as possible.

In a MANET, packet losses can have many causes. Losses caused by the wireless environment are hard to predict and there is no point in reducing throughput because of them. Maintaining the throughput in these cases does not cause more errors than reducing it. Also, to prevent losses caused by full buffers, the obvious method is not allowing buffers to fill.

A simple control mechanism is using a window to limit the number of sent packets. This window is basically the difference between the last sent sequence number (sent number) and the last acknowledged sequence number (ack number). Controlling buffer sizes simply requires a fixed maximum value for the window. While the window is in the maximum value, the sender does not send any packets.

The sent and ack numbers change in various occasions. When a packet is sent, the sent number is updated to the one in the packet. When an acknowledgment is received, the ack number is updated to the one in the packet. When a receiver ready packet is received, both numbers are updated to the one in the packet. When no feedback is received, the sent number is updated to the same value as the ack number.

Another window is used by the receiver. This window defines the number of packets that the receiver should receive until it sends an acknowledgment. This is used to reduce the number of acknowledgments that is sent, reducing both CPU and bandwidth consumption. The size of both windows must be chosen. Each one of the values and the ratio between both of them are key to the performance of the protocol. The best way to choose these values is testing the protocol after it is implemented and see which values provide the best results.

The only remaining task is deciding when packet losses should be declared. As previously explained, packet losses are detected by the sender when no feedback is received after a while. Receivers detect packet losses when no new packets are received and when packets are missing after a while. The question is how much time these "whiles" are. The simplest alternative is defining a fixed value for each of them. Similarly to the window sizes, the best way to decide these values is by testing the implemented protocol.

3.3.3 Connection Handling

The UDP protocol does not use connections. A packet is simply sent from the local host through a UDP port to another host through another UDP port. A protocol like TCP is different, it is based in connections. A connection is a mechanism that allows the control mechanisms described above to be applied. Connections are the virtual entities that have sequence numbers and windows associated to them. To allow connection establishment, a server host must be listening for connections in a TCP port. A connection is established when a client host contacts a server host that is listening for connection requests in a TCP port.

The connection process is called three-way handshake because it requires three packets. A client sends a connection request from a port x to the listening port y of the server. The server responds with a packet from port y to the port x of the client, saying that it acknowledges the connection. Finally the client confirms that it received the acknowledgment once again from port x to y. This generates a connection between the hosts in ports x and y.

A connection is identified by four values. These values are the IP addresses and ports of the two connected hosts. Any packet with these same four values belongs to the same connection. Connections are important in TCP because they allow the use of values such as the sequence number. This number is necessary for the control mechanisms and to allow packet fragmentation. Packets have a maximum size and messages may need to be divided in several packets. The sequence number along with a last fragment flag can be used to rebuild the original message.

The new protocol must also be able to handle connections. Any security extension protocol used over UDP requires at least two handshakes. It requires a handshake similar to the TCP three-way handshake and a secure handshake to initialize the secure conversation. In the new protocol both handshakes can be combined in a single one, by mixing the three-way handshake with a simple version of the TLS handshake. These packets are encrypted using the public key of the other host, so that every exchanged value (such as the secret key) is not visible to an attacker.

In order to establish a connection, the client needs to know the IP address of the server. TCP and UDP sockets can use IP addresses directly or use name services to resolve a name to an IP address. The new protocol is not able to do neither of those. The framework keeps a constantly updating structure where it keeps the name, IP address and public key of each user, which will be more detailed in the auto-configuration related sections. A connection can be established with a certain user name, which is internally resolved into its IP address and its public key, both required for the handshake.

Assuming that this protocol will be used on a request based semantic[3], it is possible to simplify the handshake from a three-way to a two-way. Assuming that the client will be the one sending the first message after the connection has been established, this first message can be treated as the third handshake packet. When the client receives the connection acknowledgment packet, it knows that the connection has been established and can start sending messages. The server will wait until the first data is received. At that point it can be sure that the connection has been established.

It is possible that some of the connection related packets are lost. To tackle this problem, the client sends the connection request periodically until it receives an acknowledgment packet. The server sends the acknowledgment packet every time it receives the connection request. This mechanism is sufficient to solve the problem. After both packets have been successfully delivered the client knows that the connection is established. The server knows that if the client has not been notified, it will re-send the connection request and it just has to answer to it. If the client does not receive an acknowledgment from the server after several attempts, it assumes that the server is not accessible and stops the connection attempt.

TCP requires the binding of a random unbound local port to establish a connection with a server host. Assuming that all hosts will be running the same framework and therefore using the same server ports, it would be quite more simple if the client could use the same local port as the server port. This way, a connection attempt would just use an already bound port. The problem is that this way, several connections would need to share the same ports. It requires a mechanism to multiplex several connections in the same ports.

This mechanism can be very simple. A unique value established in the handshake by both hosts can be used as a connection ID. The value can easily be unique if

[3]The ABCDP protocol is meant to be used by the service framework and the auto-configuration routines. In both utilizations, the client always sends the first data packet containing some sort of request.

both hosts decide half of the value, each half being unique for their creator. In this handshake more values can be exchanged, such as a random initial sequence number for each host and two halves of the secret key to use. These values are encrypted using PKC which is necessary to prevent the connection from being stolen upon creation.

The first packet is always sent by a client host that wants to establish a connection. But the acknowledgment packet can be sent by an attacker. The attacker is unable to understand the contents of the request but is able to send a reply to the client host and try to steal the connection. For both hosts to be sure that the connection is being established between them, they need to prove to each other that they understood the packets that were exchanged. This is done implicitly as both hosts must understand the values sent to each other when the connection was created. Both users must know the connection ID, sequence numbers and secret key used in the connection.

Another important routine is connection closing. TCP connections should be closed to prevent connection hijacking. This consists in sending packets with the IP addresses, ports and the current sequence number of an already created connection. It is an impersonation technique and is very simple to apply in a local network where packets can be easily eavesdropped and understood if not encrypted. But in the new protocol this is not an issue. Each connection uses a different secret key to encrypt and decrypt data. Connection hijacking almost impossible because the attacker would have to know the secret key used in the connection.

Another reason why connections should be closed in TCP is port allocation. Each client connection requires the binding of a port and this bind should be released when the connection is no longer necessary. In the new protocol, a connection does not need a specific port. The only requirements are a unique connection ID for each connection and a mapping between each connection ID and its related connection data, such as the secret key and the sequence numbers. With low resource consumption and virtually no risk of hijack, there is almost no reason to close connections.

3.3.4 Packet Format

As mentioned in the previous sections, not all packets have an equal format. The connection packets are encrypted using PKC whilst the conversation packets use SKC. These packet types also require different header formats, as each connection packet requires the exchange of several values and a conversation packet requires fields for the control mechanisms.

The first problem is that these two formats must be differenced by the receiver. Conversation packets have another related problem. As each connection uses a different secret key, the connection ID must not be encrypted. Connection packets do not require a connection ID as it is one of the values generated in the handshake. The first field in both packets can then be a 32 bit integer value. This value contains a fixed zero value in connection packets and a non-zero connection ID in a conversation packet.

A connection packet requires several encrypted fields. First of all it requires a field to exchange the 32 bit connection ID. As each host decides half of the connection ID, the field can be separated in two 16 bit fields. Each host could just send its half but this way the server can immediately prove to the client that it understood the packet by sending the other half back. Another 16 bit field is necessary to trade the initial sequence number. Each host sends the sequence number that is expecting to receive in the first data packet. There is also an 8 bit field containing the initial packet window of the host[4].

There is another 8 bit field containing 1 bit flags. Only one flag is actually used[5]. It is the ACK flag that defines if the packet is a connection request or a connection acknowledgment. Finally, there is a 64 bit field for half a 128 bit secret key. The format of these packets is presented in figure 3.7.

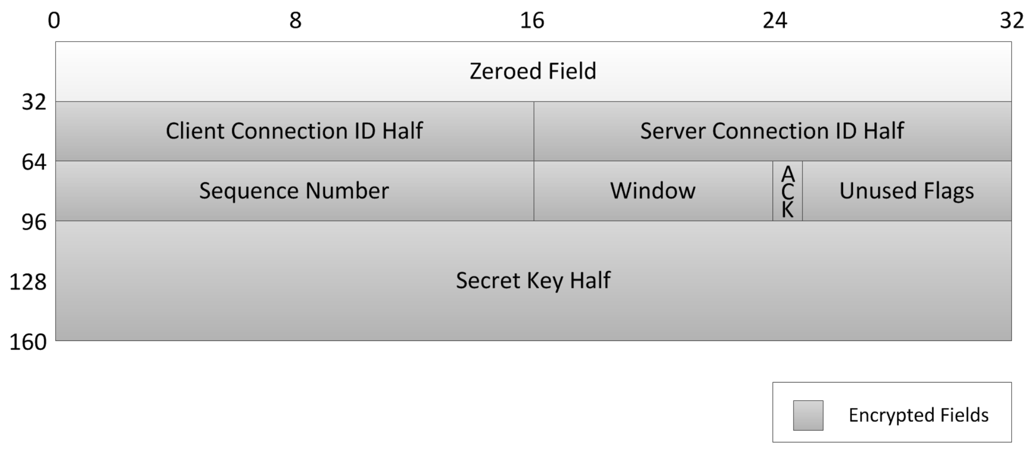


Figure 3.7: Connection packet format

Conversation packets require different fields. Along with the non-encrypted 32 bit connection ID field, a conversation packet requires several encrypted fields. It requires a 16 bit field for the sequence number[6]. It also contains an 8 bit field used to exchange window sizes[7]. There is also an 8 bit field for flags. These packets require

[4]This field is not used but is reserved for future use in case a flexible window mechanism is implemented. This way, backwards compatibility can be preserved if the mechanism is implemented.

[5]Computers are built to use values based on bytes (8 bits) and programming languages also follow the same approach. It is both easier and more efficient to use fields of multiples of 8 bits than to use the exact required size for the flags. Additionally, if this protocol is extended in the future, changes in any previously allocated but unused fields still allow backwards compatibility.

[6]At the time of implementation the chosen size for the sequence number fields was 16 bits. But these fields should be bigger, containing at least 32 bits, to prevent an attacker from inserting random data in a connection. With values this small, if many random messages are sent, some of these messages may contain the sequence number expected by the connection.

[7]Same as the window field from the connection packets.

three flags: an acknowledgment, a receiver ready and a last fragment flag. The fields of a conversation packet are displayed in figure 3.8.

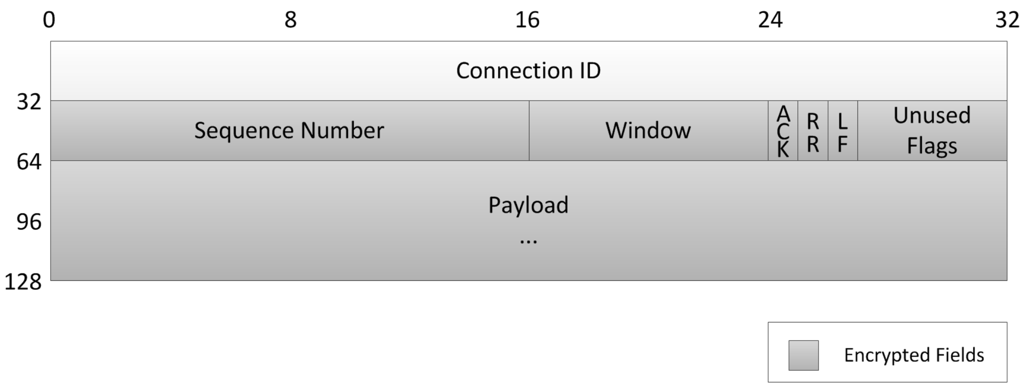


Figure 3.8: Conversation packet format

3.4 Auto-configuration Mechanisms

One of the main goals of this work is the development of mechanisms suitable for an automatic and distributed configuration of a MANET. These auto-configuration mechanisms include every routine that is required for the correct functioning of a MANET. The main focus of every designed auto-configuration mechanism is their security. These mechanisms should be as resistant as possible to any type of network attack.

As defined in section 3.3.1, each user owns a public and private key. These keys are used to establish authenticated connections between nodes. For the connections to be established, each node must know the public keys of the other nodes in the network. They also must know the IP address and name used by each one of the nodes. Users must then advertise their presence in the network. The most obvious way is using broadcasts for this.

Each advertisement broadcast can contain the information required for user discovery, IP configuration and name resolution. This results in an all-in-one routine for the three individual mechanisms. With one single routine performed by every user, each user can know the public key, IP address and name of every other user in the network, which are the basic requirements for an easy communication between named hosts in a network.

Another required auto-configuration mechanisms is the generation of routes between hosts. Although these mechanisms are not required if the two contacting hosts are neighbors, support for MANETs is not complete without route generation mechanisms. There are other useful mechanisms can also be applied to MANETs, such as service or resource replication. Each one of these mechanisms is focused individually in the following sections.

3.4.1 IP Configuration

One of the main problems that has to be addressed is the automatic attribution of unique IP addresses to every node of a MANET. As presented in the state of the art chapter, there are many ways of achieving this. Some provide the role of attributing IP addresses to one or more nodes. Others allow a node to decide their own address and posteriorly check for duplicated addresses. Any of the approaches is valid, at least until security problems are raised.

As also has been presented above, an attacker can impersonate any IP or MAC address. In fact, if IP or MAC addresses are used as identifiers, he can impersonate any node in the network and even a numerous amount of virtual entities. An attacker may also send packets with any content that he wants. It is possible to conclude that he can influence any decision concerning the selection of a node with an IP attribution role and that he can raise problems in duplicate address detection mechanisms.

Still, even being imperfect, the basis for an IP auto-configuration mechanism must be one of these. Some entity or group of entities must decide what IP addresses to attribute. Considering that each node must broadcast its own public key, it can broadcast an IP address as well. Therefore, an approach where each node chooses its own IP address and then broadcasts it seems the most fit for this distributed key environment. If a broadcast is received by a node that uses the IP address in the broadcast, it notifies the sender that it must choose another address.

At any point in time, new hosts can join the network or two partitions can merge. Various nodes may have chosen the same IP address in different partitions. In this situation, the address duplication can not be detected until both partitions merge. To detect these cases, the advertisement broadcast can not just be sent at the moment of the attribution. The broadcast must be sent periodically.

This method works if there are no attackers in the network. But what if there are? There are two main problems with this approach, that can be used to cause DoS attacks, as presented in figure 3.9. First, an attacker can always say that he is using the pretended IP address. This way, the arriving user is stuck in the process of choosing an IP address for himself. Still, this is easy to solve. As the odds of duplicated addresses in the Internet Protocol version 6 (IPv6) addressing space are virtually none, more than one or two sequential conflicts can be considered an attack. The node then decides to use one of its intended addresses.

The second problem lies with the other users. Each user must register every other user, in order to know their public keys, IP addresses and used names. An attacker can confuse this configuration process by simultaneously registering a new identity

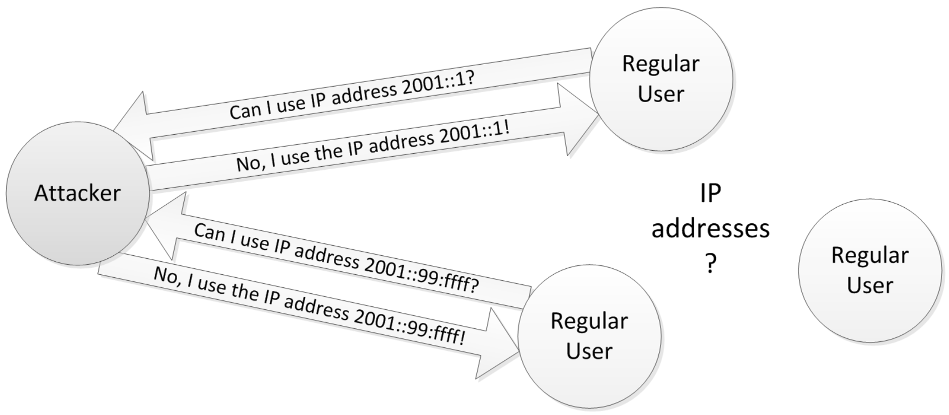

Regular User
Can I use IP address 2001::1?
No, I use the IP address 2001::1!
Attacker
IP addresses ?
Can I use IP address 2001::99:ffff?
No, I use the IP address 2001::99:ffff!
Regular User
Regular User

Figure 3.9: Denial of Service in the addressing space

that uses the same IP address. Other users detect an attempt of one machine trying to register an already chosen IP address just moments after the first one. How do they choose which one of the users to register? The requests may reach different nodes by different orders.

Typically, the first received configuration attempt is accepted by every host, as there is no reason not to accept it. When the second configuration attempt is received, at least one of them must be denied. Using IPv6, the probability of a collision in a local network is virtually none. When a collision is detected, and more importantly, when collisions are detected in every configuration attempt, there is almost certainly an attacker somewhere in the network.

But which one of the two colliding machines is the attacker and should have his attempt denied in favor of the other? There is not any real way of identifying a machine. The only identifiers existent are the IP and the MAC address and both of them can be freely altered by the attacker. An attacker can use a nearly infinite number of different identities, therefore there is no way of detecting who the attacker is.

There seems to be no good method to decide who will have his attempt denied, other than a random decision or ignoring the second attempt. Neither is good as it only needs one machine denying the correct user to oblige that user to make a new attempt. To make things worse, the attacker can simulate the machine that makes the “wrong” decision. But there can be a workaround. Both machines could try to agree who will keep the IP address using a random deciding process. But after a quick look, it is easy to conclude that the attacker could always cheat the mechanism and say that he won even when he did not.

A decision between the two machines is impracticable but what about a decision amongst every host in the network? A majority vote could solve the problem. Even if the regular host does not win at first, it will eventually win after several attempts.

But this does not solve the problem. The attacker is not limited to simulate a single user. He can simulate as many voters as he wants and win every election.

But every host can keep a list with all the hosts in their network partition, by listening to the periodic broadcasts. Using this list, they may notice that they have never heard of most of the voters. Each host may only count the votes of the hosts that it knows. But in this case the outcome may vary throughout the hosts. Maybe all hosts can reach a consensus about which hosts to count. But the majority of the votes can be held by the attacker, that can simulate as many hosts as he wants. In reality, this last case can occur even without any attacker, when two partitions merge, so it can not even be treated as an attack.

None of these methods work, therefore a different approach must be followed. If every user has some parameter that can be used as an argument for a deterministic function, every node would be able to calculate the same IP address for a specific user. This parameter must be unique for every user in the network and must not be able to be impersonated. These two characteristics seem to be very familiar. Yes, the public key can be used as this parameter.

Each user has a unique public key. This key can be used as the argument of a hash function that generates a virtually unique IP address. The level of uniqueness depends on number of possible results and the quality of the function. To maximize the number of possible results, the IPv6 addressing space is used. An IPv6 address is divided in two sections with 64 bits each. The first one is the network prefix that identifies the network and must therefore be the same for every node in the network. The other one is the host address, that identifies a node inside the network and must therefore be unique inside a network. Each node then uses a hard-coded network prefix and the result of the hash function as the host address, as represented in figure 3.10.

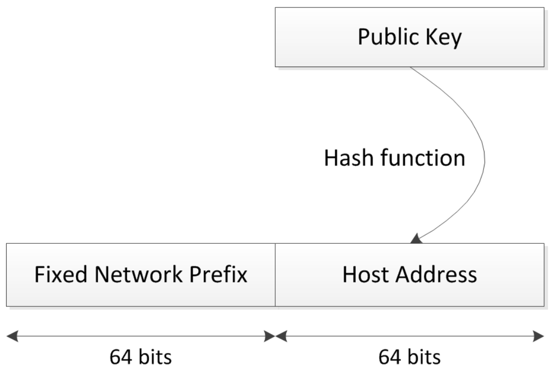


Figure 3.10: Generation of IP addresses from public keys

As an attacker is unable to register a specific public key, if the IP address is based on the public key, an attacker is also unable to register a specific IP address. This mechanism does not even require IP addresses to be sent in the broadcasts, as each host can calculate them based on the public keys. Finally, the only remaining problem is duplicated addresses. Although the odds are very low, it may still occur. When an IP address collision is detected, the solution is any of the hosts (or both to prevent any decision process) renewing its key pair. This process is detailed in section 3.4.6.

3.4.2 Name Configuration

The attribution of names to hosts is the most common way for a human to identify a machine. Even though machines identify each other using MAC addresses, IP addresses or public keys, a human hardly resorts to that. In this local network scenario, each user should be able to choose the name that he wants to be identified by. Other users can then access his machine and use his hosted services. But more than one user may want to use the same name.

The approaches studied in the state of the art typically focused the name registration processes. These processes are very similar to the ones for unique IP assignments because they consider that names should be unique. Unfortunately, they do not provide solutions to solve name collisions. Other approaches investigated these solutions. These solutions are good but obviously they do not allow the simplicity of just choosing the pretended small name (it is what causes collisions).

But do names really have to be unique? Those approaches considered the name as the primary identifier of a node. In this scenario there is already a primary identifier for a node, which is its public key. Name uniqueness is just a way to allow an indubitable resolution of every name. But name uniqueness has the same vulnerabilities as the unique IP assignment routines. In fact, the resolution for that problem is not applicable here, as a user does not want to use a name that is a hash of a public key.

It seems to be best to just forget about name uniqueness. Each user just registers the name that he wants in every other user. But then how can names be resolved to users? An attacker can register one or more identities with the same name as another user. And even if there not any attackers in the network but two users simply use the same name. How can the correct one be chosen? There is no automated way to do it. It must be done with some user input, some methods requiring it more than others.

In this local network scenario, a user typically wants to use a service from a specific user. Those users are the ones that are going to be contacted and therefore are the only names that must be correctly resolved. It is up to known users to use different names, for obvious reasons. One alternative is allowing the management of a local friend list, containing users that should have priority in the name resolution process.

Other method is allowing the local assignment of names to users. If a first user chooses a name but a second user prefers to identify him by a different name, this method allows the different name to be also resolved. Considering that these names are private, an attacker is not able to blatantly create virtual identities with that name. A more automated method is counting the number of connections established with each user and use it as a priority factor. Users that are contacted more times are more likely the ones that should be contacted.

None of these alternatives is perfect. The friend list management and local name attribution should only be performed when there are no doubts that the managed user is in fact the correct one. The automated process is useless if the correct user has never been contacted before. There is always a risk to contact the incorrect user. But forcing name uniqueness is also not a solution because there is no method that is completely immune to every form of disruption.

3.4.3 User Advertisement

This distributed public key approach requires each host to know at all times every other host in the network. This is required because connection establishment requires the public key of the other user both for packet encryption and for IP generation using a hash. Additionally, connection destinations are based on the host name. As users arrive and leave the network at any time and movement may partition or merge the network, this information must be shared many times. The easiest way is each host periodically broadcasting its own information throughout the whole network partition (flooding).

Each broadcast contains the public key and the name of the user (figure 3.11). As these are broadcasts, they are not encrypted in any way and therefore are not authenticated. This information can not be used to make any configuration directly, it can just be seen as an advertisement. Each host checks all the broadcasts it received to see if there is any new information, specifically a new public key or a different name for an already known key. If any new information is received then it must be authenticated.

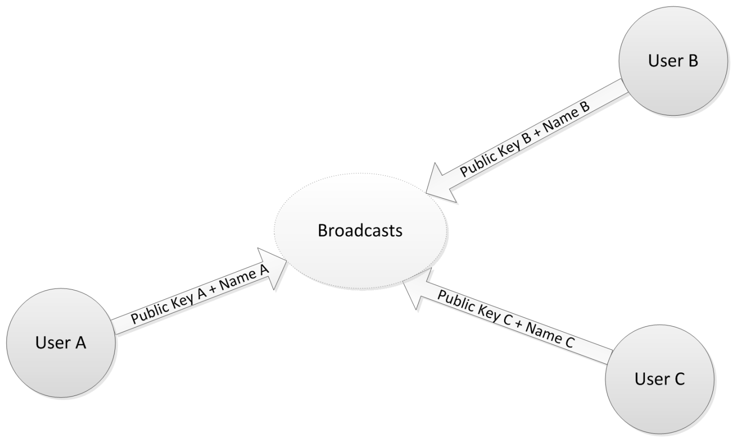


Figure 3.11: Advertisement of public keys and names

This authentication process can be achieved using an authenticated (ABCDP) connection. The broadcast receiver establishes a connection with the broadcast sender. In this connection, the information can be shared once again, but this time it is authenticated and can be used for configuration purposes. As the overhead of establishing a connection is necessary and most configurations are done when a user arrives to the network, there is no reason for just one user to send his information. Both users can send their data to each other and configure each other at one time (figure 3.12).

This authenticated configuration scheme works as every two users are just configure each other. No other users can be affected in any mutual configuration routine. This means an attacker can only mess around with the information he gives to other users about himself or with the information that he possesses about other users. The attacker has therefore no point in attacking authenticated configuration routines.

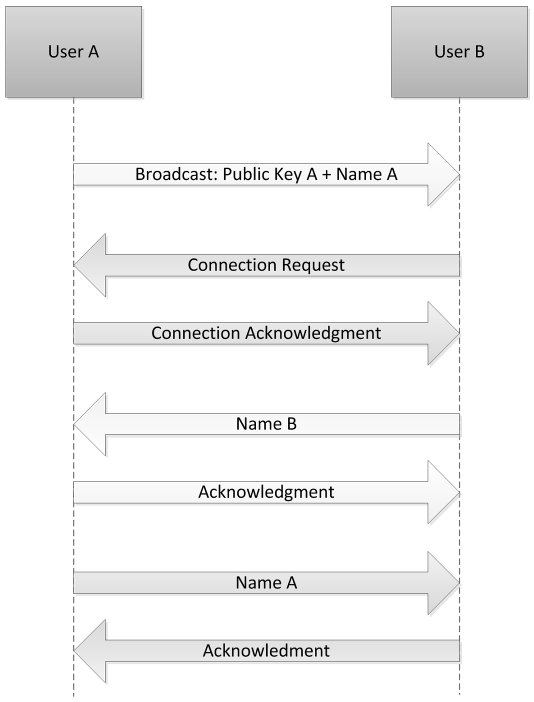


Figure 3.12: Packets exchanged in a configuration routine

This uses advertisement and configuration routines are also useful for user listings. A user or a service may want to know which users are online at the time. To keep a list that only contains online users, broadcast timing can be used. If a broadcast advertising a user is not received for a certain amount of time, the user is removed from the list. This is not a particularly good solution as any attacker can fabricate that broadcast. On the other hand it is not a serious problem. It is only a list, so the only thing that can happen is a supposedly online user not be reachable. It seems unnecessary to authenticate every broadcast simply because of this list.

3.4.4 Routing

Routing is an essential part of networking. Routing is the mechanism that allows packets to be sent between hosts, particularly if they must travel through multiple hops throughout the network. It generates paths between hosts, allowing packets to be delivered to their destinations. One of the major problems in ad-hoc networks is the generation of these paths between hosts because the shape of the network is constantly changing.

Routes consist in what path should be taken reach a destination. They work similarly to road direction signs, where in each road junction there are signs with the direction one should follow to reach a certain location. Each of these signs is a route. Each host is like a junction that contains a set of signs (a routing table with several entries). Each route entry contains a set of destination IPs and the IP that should be used as the next hop along with the network interface that should be used to reach that hop.

In the process of sending a packet, at each host (including the packet sender) the routing table is checked to see what should be the next hop and network interface to send the packet to. Afterwards, the neighbor table (which contains IP and MAC address pairs of each neighbor host) is used to obtain the MAC address of the next hop. The packet is then sent with the destination MAC of the next hop and the destination IP of the final target through the network interface in the route. If routes in every host of the path are well generated, the packet should be able to reach its final destination.

But for routes to be followed, they must first be generated. In an infrastructure network routes can be easily generated. A single route that sends all packets to the access point usually does the trick. Additionally, a route that sends all packets with local destinations directly to the target can also be created. These routes are usually automatically generated by the software used to connect to a network. When connecting to ad-hoc networks those softwares typically generate the second route, which allows packets to be sent directly to any neighbor. But this does not allow packets to be sent to any point of a network other than neighbors.

The problem with creating routes is that as traffic must be relayed through various hosts, it does not only rely in both endpoints. It also relies in every other host that relays the packets. These hosts can be attackers and suppress the traffic that they should be forwarding. Using a regular ad-hoc routing algorithm will probably allow the attacker to trick his neighboring hosts to think that he has the best routes for every destination other than their neighbors.

Ad-hoc routing algorithms are typically based on flooding. Considering that flooding is already used for the user advertisement mechanism, this mechanism could be adapted for route generation. The same authenticated configuration mechanism could be used to generate paths between hosts. The authenticated connections could be used to make sure that routes are only created between two endpoints by the same endpoints. The problem is that a route is not only dependent on both endpoints.

Each host just keeps the first hop to the target, therefore contacting with the endpoint for route generation is actually irrelevant. At most what can be done is contacting with the neighbors trying to find out the best route to the target. But these neighbors can be attackers and even using an authenticated connection they can still say that they have the best route to the target.

The simplest alternative is to use an already developed ad-hoc routing algorithm. Choose some executable code that runs in every host in the network and takes care of the whole route generation process. If packets keep failing their delivery (error control allows this), a host will try finding a new route, probably ignoring the attackers route. There are many ways to attack this, like impersonating a new virtual user, but route generation and security problems are extremely hard to tackle. They are also not a threat to data security, just to service availability.

3.4.5 Replication

Another technique that should be focused is service replication. This technique is generally replicating the same service in replicas scattered through the network, with each replica hosting the same service. This allows actions to be executed by replicas and not only by the original host. This would reduce the number of hops traveled by the packets to the service, which would reduce network overhead. Additionally, because the network may be partitioned, replication is a good way to spread the availability of a service further than one single partition.

The main difficulty in service replication is update propagation. If services have a state (which is true in this model) actions usually update the state of the service. These updates should be propagated to the other replicas and must obligatorily be propagated to the original host. But if the network is partitioned, contact between the original host and some replicas can be lost. If the original host or a replica with non-propagated updates lose contact with each other (one leaves the network or partitioning occurs), some updates may never be propagated.

Using service replication, action commitment can not be guaranteed. This would be a problem that users would greatly dislike. No user would like to reserve a table in a restaurant and receive a message saying that their table would possibly be reserved. Additionally, even if they get in contact once again, sharing a bundle of updates is quite hard, as it is necessary to know the order of the actions executed by both hosts. This would also require a lot of reconstructions in the database of the service.

Another possibility would be to only replicate services within the same partition and propagate every update before completing the request. When a replicating host is unable to propagate the update, it would stop replicating the service. The problem is that this would oblige packets to be sent between the replica and the original host in every request with state updates. There would almost no gain unless many actions do not update the state of the service. Additionally, replication would be more interesting if replicas could actually be present in different partitions, to increase the availability of a service throughout the network.

Service replication seems not to fit this model. But not only services can be replicated. Data can also be replicated. But data replication has the same problems as service replication because it also requires update propagation. But what if the only data that is replicated is data that does not receive updates (read-only data)? Some data such as large static files can be obtained by a service user. If this large file could be downloaded from different mirrors, network overhead would also be reduced.

This type of replication would in fact be interesting if it was not for the chosen security model. This model does not allow any user to blindly trust another. An attacker can easily become the host of a data replica (with the consent of the original host) and start replicating fake data. Focusing once again in service replication, even if services were distinguished between stateful and stateless and only the latter ones were replicated, the same security problems would be occur.

All these factors point towards the conclusion that replication (of either data or services) is not practical in this model. Network partitions are one of the main purposes of using replication but they are also what does not allow the successful use of replication (replica updates may not be propagated). Additionally, the chosen security model renders any type of replication impossible.

3.4.6 Key Renewal

Using the same keys for a long amount of time makes it more likely for an user to be impersonated. This time provides the opportunity for an attacker to use mechanisms to discover the private key related with the public key of an user. To prevent impersonations, key pairs should be renewed periodically. This renewal requires a mechanism that allows hosts to detect key renewals of other hosts at any time.

The problem is that in this model, public keys are the main identifiers for users. Changing the main identifier of something in a distributed environment is always complex. One way would be keeping the advertisement mechanism from section 3.4.3 unaltered and periodically checking for altered keys. This could be achieved by hosts broadcasting their older keys and other hosts detecting keys that they still have registered in old accounts. But this would require to merge both accounts and that is always a hard operation, especially with database tables.

To prevent account merging, key renewals must be detected upon the first configuration of the new key. This can be achieved by sending older keys not in periodical broadcasts but at the first configuration. Extending the mechanism in section 3.4.3, when configuring after a key renewal, at least one of the machines will configure a key that it does not know. This can either be a new user or an old user with renewed keys.

To discover which is the case, the configuring host asks for all previous public keys of that user. The unknown host starts sending its older public keys. When the configuring host receives a key that it knows, it notifies the previously unknown user about which old key it recognized. The previously unknown user then establishes an ABCDP connection to the configuring host, the latter using the old key. The previously unknown user then sends the new key via the new connection, proving that he possessed the old key and acknowledged the renewal for the new key.

These configuration routines are mutual, which means that both users configure in one another. This key renewal mechanism works even if neither of the users knows the other one i.e. if both users renewed their keys without any configuration routine between both renewals. In this case, this routine can be executed twice, once for each host. The first user starts configuring an unknown key and executes the routine. Afterwards, the second user will also do the same and both users update the accounts of each other without creating any new account.

But some problems still remain. The user may renew the key (possibly several times) and not visit networks where some services that he uses are hosted. These services will still see this user as the owner of one of his old key pairs. If an attacker obtains the private key (possibly using brute force algorithms) of one of these unused public keys, he is able to impersonate the owner of those accounts.

But obtaining the private key related with a public key usually takes months. Particularly in local wireless networks where an attacker should be someone local with interest in attacking specific individuals and ought not to possess an infrastructure with high computational power. Therefore if keys are renewed in a regular basis and a user also regularly visits the networks where the services that he uses are hosted, there should be no problem with impersonations.

3.5 Service Framework

One of the goals of this work is the development of a service framework that is easy to use both by end users and by developers, that is web based and also adapted to a MANET environment. Plug-and-play support by itself does not bring many more users closer to ad-hoc networks. For that, a set of simple and useful services is required. Obviously, these services must first be developed. For that, a simple way to develop these services is required, in order to attract their developers. The development of this framework therefore an important step for increasing the utilization of these networks.

For the framework to be web based, it is essential that the clients of the framework are web browsers. The framework must therefore communicate with web browsers using HTTP. In addition to the HTTP engine for interacting with users, an engine for communication between different machines is required for complex services. This engine can be a very simplified remote method invocation engine.

Another important feature is the adaptability to MANET environments. For a service developer, the main difference between a MANET and other networks is that, due to node movement, node accessibility can change rapidly. Links can be broken and it may take some time until the routing protocol rearranges the routes. If there is no possible route at all, access may be impossible for a long while. Nodes may even leave the network voluntarily, since the owner of the device may want to leave. This can be problematic for long routines between two nodes. Additionally, as nodes are not always present in the network, it may not be possible to perform some scheduled routines. If a node is not in the network it is obviously impossible to interact with that node.

Fortunately, there are easy ways to tackle both these problems. Because the user interacts with the framework via an HTTP engine, performing small requests with small responses, node movement is not very problematic. If a user is waiting too long for a response, the typical behavior is "clicking the thing" once again. This is repeated until either the response arrives or the user just ignores that task and moves on to another.

As for the scheduled routines, there is a more complex solution. These routines use the remote invocation engine to invoke methods in the other machine. Obviously, if the target machine is not in the same network, the methods can not be invoked. But if the invocations are postponed until a time when the target machine is also in the network, the methods can be invoked. If the invocations only matter to the invoker or to the relation between the invoker and the target, it does not matter if both machines are never in the same network again. If both meet again, then the invocations are performed.

As for easing up the utilization of the framework, there are many possible ways. It is possible to simplify the development of code along with the deployment, distribution and installation of the services. The typical way used to distribute and install any piece of software is to compress every content required for its installation in a single file. The same principle can be used in these services. But the process can be simplified.

The contents of a service can be packaged into a single archive. Considering that the framework hosts various services, the installed services can be placed in a specific directory. The framework can interpret the archives at startup and load each service from its archive. This way, installing a service is simply placing it in the correct directory. Details about the development of code are explained in the remainder of this section.

3.5.1 Web Engine

The most important component of the service framework is the HTTP engine (web engine), because it is the one that interacts with the user. This engine is based in a Model-view-controller (MVC) [12] architecture. This architecture is composed by three components, namely models, views and controllers and each of them has several other sub-components. Each of these components has its specific role in the task of responding to an HTTP request. There are several MVC frameworks for various programming languages, including Java, and each one tries to focus slightly different aspects.

This framework focuses on reducing the programming effort to the minimum by focusing mainly on what is typically necessary in a simple network service. Each of the components and sub-components of the designed MVC engine is presented in this section, along with what is done to reduce the programming effort of each. This effort should be focused in developing core routines and not in the look of the web pages. The framework makes most of the work of generating visually appealing web pages.

Views are the containers of the HTML markup that will be displayed as a web page. A view is constructed by the server in several steps. First of all, the view is initialized using a static layout that has several areas where dynamic HTML can be added. After that, the rest of the page is constructed, by filling in the dynamic HTML sections. Finally, after the view is completely formed, it is ready to be sent to a user as the content of an HTTP response and visualized as a web page. A representation of this process is presented in figure 3.13.

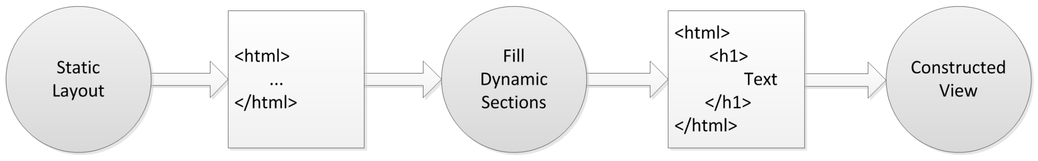


Figure 3.13: The construction of a view

A layout is the general outline of a web page. Every web page from a web site typically has the same HTML markup structure. It also contains the same meta information and uses the same stylesheets. Pages with the these characteristics have the same headers, footers, navigation bars, sidebars, shapes and colors. They have the same generic outline, the same look and feel. The differences between these pages are the contents placed inside this "layout". A layout is therefore a static HTML structure that has sections where pieces of dynamic code can be placed.

These dynamic pieces of code can be called sections. With a special markup, a programmer defines in the layout where each section should appear and how it should be named. Each section has a unique name, so that sections can be easily identified by them. The programmer then defines the contents of each section by setting or appending HTML code in them. But defining HTML structures as Java strings is not very simple for a developer. Therefore, helper classes and methods are provided to let the contents of each section be easily defined without resorting to literal HTML.

What is typically placed inside these sections? Apart from text, a service typically has structures such as forms, listings and tables. It would be nice for a programmer if he could easily create these structures easily. For this task, view templates can be used. These templates contain simple methods that allow easy generation of HTML code from objects, such as collections and multi-dimensional arrays. These templates also contain options to customize the generated structures. Being a class, a template can be extended and have overridden methods for further customization.

Web pages also contain static resources such as images, stylesheets and javascripts. These resources are typically placed in files outside of the web page and each one is pointed by a URL in the HTML code. These resources can be placed in a specific public folder, whose contents can be located by a URL. To allow programmers of simple services to focus on the development of the core routines and not on the visual aspect of their services, a default layout is provided by the framework. As stylesheets are usually developed for a specific layout, default stylesheets designed for the default layout are provided as well.

Models are entities that can be stored in a database. Each model class has a mapping to a database table and each model instance has a mapping to a database row. The Object-Relational Mapping (ORM) framework maps each publicly accessible variable (variables that are public or have the correspondent getter and setter methods) to a table column with the same name and a corresponding type. Each model class has therefore access to simple methods such as loading all models, loading by ID, conditional loading, save and delete. The programmer does not have to worry about making the object-relational mapping or any Structured Query Language (SQL) statements, he just needs to implement the class.

Each model can also be mapped to a form. Each variable can be mapped into a form field and vice-versa. For this reason, a model can generate an empty form or a filled form with the contents of each variable in each field. Analogously, a filled form can generate a model using the contents of each field. This ability provides a one line interface to directly interact between users and models.

Controllers are classes that contain the actions that can be invoked by the HTTP client. Each service contains a set of controllers and each controller contains a set of invocable actions. The actions of a controller should be related with its name. For example, a photo controller should have actions related with photos, such as their management or visualization. As actions inside a controller are related, a controller uses the same layout for every action. The programmer chooses which layout should be used by each controller.

An action is a method that can be invoked by an HTTP client. When the server receives an HTTP request, the request is parsed to decide what action should be invoked. Each action contains the code that defines what routines are performed and the contents of the output web page. These methods typically load, change and store models as well as define what should be printed in each view section, usually based on the used models. Afterwards, the content of the view is sent to the client in an HTTP response.

The action that is invoked is decided based on the requested URL. The URLs use a specific format: `http://domain.tld/service/controller/action`. This way, the server is able to map which action from which controller from and which service should be invoked. This is a convention at the service and controller levels. At the action level this convention can be changed. This subject is covered in mode detail in section 4.4.3.

A diagram of the whole web engine is presented in figure 3.14. Summing up, there is a tendency to suppress the programming effort in sub-components, such as layouts, URL to action mapping, and object-relational mapping to none if the provided default alternatives are used. View programming is also reduced to view section programming. Additionally, these sections can contain complex structures with very few lines of code. Programming effort related with views is almost none. Models also require minimum programming effort, as it is only necessary to define what variables each class contains. Code development is almost reduced to its core routines, placed inside the actions of each controller.

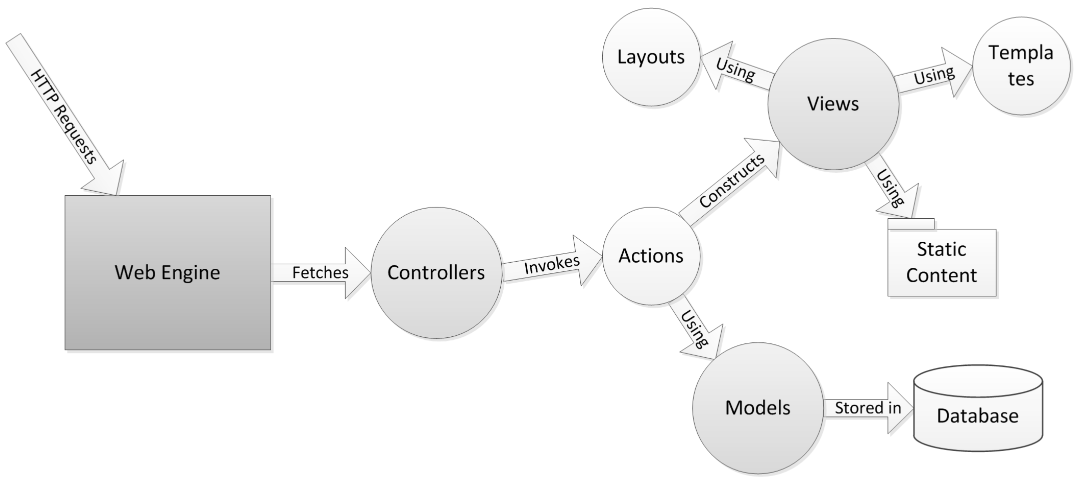


Figure 3.14: Diagram of the web engine

3.5.2 Programming Features

Developing code in a programming language or a framework becomes more simple if features that reduce the amount and complexity of the developed code are provided. Therefore, any relevant feature is welcome to the framework. Apart from the base MVC engine and its helper features, such as the view templates, many features can be added.

Many services store data that is relevant to users. Those services usually store that information by making use of a user account system. Since every request that reaches the service framework is authenticated because of the ABCDP protocol[8], the creation of user accounts is not required and each user can have a direct relation with his data. One way of providing a simple interface for these implicit accounts is assigning owners to models. With this, it becomes simple to manage the objects that are relevant to each user.

Another feature that can help the development of services is grouping. Many entities can be grouped, such as users and models. Allowing the creation and management of user groups and model groups can assist many tasks. Grouping models can be seen as tagging, which makes it possible to attribute a certain quality to a model. With that, it is possible to list models with one or several of these qualities, easing up the management of the model database. Grouping users is useful for remote invocations, as explained in section 3.5.3.

User groups are completely local. Each node defines its own groups and no interface is provided to directly change the groups of other nodes. Using a specific example, for example a folder sharing service, each share can have a specific name, such as the name of the folder. One model group with that name can contain one model per file in the shared folder and one user group with the same name can contain the other users that share the folder (each node manages these groups autonomously).

As every entity of the HTTP engine is a class, every single one of them can be extended. For example, any existing controller can be extended and have overridden methods or new methods. Class extension can be used in many ways. A view template is supposed to have many methods, each of them with a tiny role in the generated HTML markup. Extending a view template can therefore be used to slightly tweak the generated markup, such as adding an ID or a class to an element of the HTML code.

This extension can also be used to create more helper classes, such as a controller that has actions that manage a whole model table. This controller can be designed using a reference for a the generic model class and contain actions that list the models, generate forms that create or modify models and that allow the deletion of a model. If

[8]The process that converts TCP based HTTP requests used by web clients to ABCDP based ones is detailed in chapter 4.

this class is abstract and contains an abstract method that should return a model class, implementing a controller that extends this class only requires the implementation of the abstract method, that should simply return the model class that should be managed.

3.5.3 Remote Invocation Engine

The main goal of the remote invocation engine is providing a way to interact between different instances of the same service, hosted in different nodes. It is an essential feature in the development of network services. The biggest challenge in the development of a remote invocation engine is making it simple to use.

The commonly used web services are easy to use due to all the automation behind them. They are also easy to use because they are not complicated. They do not have a big number of features renders makes them hard to use. A developer simply designs a remote Application Programming Interface (API) and implements its methods. Automated tools generate the service descriptions for the clients and sometimes even provide test HTML interfaces. This remote invocation engine follows the philosophy of the web services, it tries to simplify the task of developing services.

Each service has one single remote method API. This API contains the methods that can be remotely invoked by other instances of the service. These methods are invoked by the controllers or by any thread created by them. It is not necessary to generate any description for clients. Because the service is the same, the remote method API can be used to automatically generate a local remote client API. As the developer implements his service, he implements the both the remote methods and their invocations in the same source code.

The target of a remote method invocation can be a user or a group. If the target of the invocation is a group, the method is invoked in every member of the group. User and group listings can be used to obtain any user, online or not, or any group. The target user or group can also be obtained by their name. The invocation targets chosen by developers are typically a user or group chosen by the user of the service, a specific group related with an automated task or all the users in the network.

There are two types of remote invocations: instant invocations and persistent invocations. In any of the two, online target users receive the invocation request at the time of the invocation and respond to it. In an instant invocation, the users that miss the request, either by not being online or by not being accessible at that time, do not respond to the invocation. In a persistent invocation, the list of users that did not respond to the invocation is stored. When one of these users becomes accessible, the request is performed again. The goal is guaranteeing that the invocation is performed by every target.

This engine is only meant to be used between the framework, therefore no commonly used methods such as SOAP envelopes inside HTTP requests are necessary. Invocations can be performed directly over ABCDP, using a very simple binary protocol, containing the name of the invoked method and its serialized arguments. The only extra content required is the number of arguments and the size of the each of the fields in a fixed size, to allow the correct interpretation of the packet.

3.5.4 The Index Service

Every HTTP server provides a default webpage, no matter how simple it may be. This service framework should be an exception. In fact, considering that so far, there is no default way for an user view the online users or the services that are available, this is a good opportunity to provide that information. It is also a good place to allow the user to manage some basic settings, such as the name to use or any management operation related with other users.

These features can be placed in a service called "`index`", that contains controllers with the actions that implement these features. The main action of the service lists the users that are online and the services that are running in the host machine. This action can also be visited by other users. This allows any user to see the services that are being hosted in any machine. The actions related with the settings are obviously only accessible to the owner of the service.

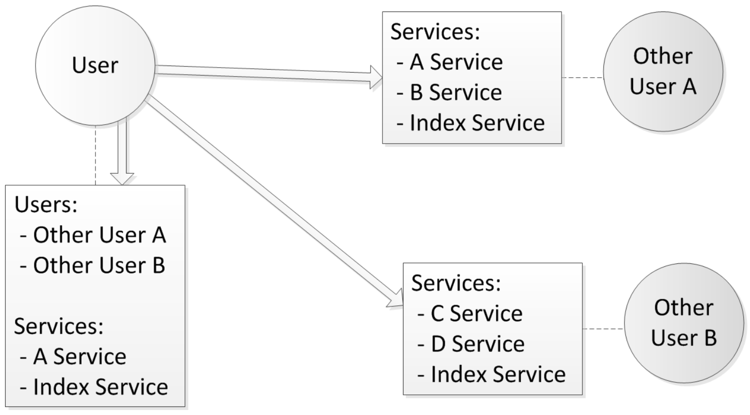


Figure 3.15: Representation of the index service

By placing hyperlinks in each of the listed users and services, it is possible to traverse the whole network and access any service just by clicking or touching hyperlinks. Although this does not provide any full view of the services in the network, it still provides user and basic service discovery functionalities, as presented in figure 3.15. More complex functionalities can be provided using the remote invocation engine to gather information about services hosted in every machine.

3.6 Final Overview

With the description of the main components, the design of the solution is complete. With the design completed, it is possible to represent the complete protocol stack. This representation is presented in figure 3.16. Each one of the three designed components is able to fulfill the goals that were set in the road-map. Each component is able to be used independently in a network. That is a good thing, but it is not the main purpose of this work.

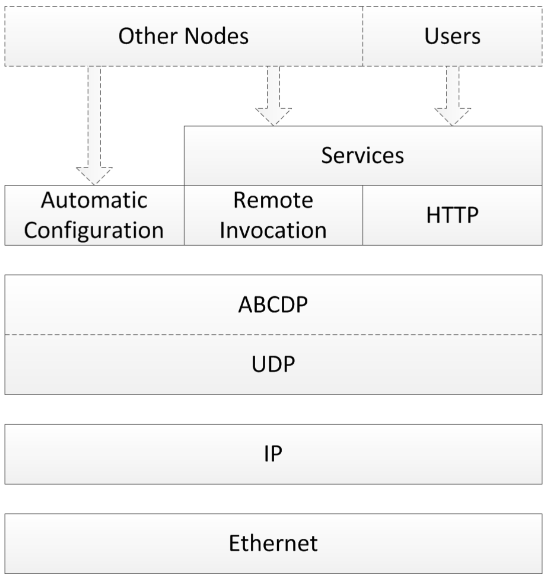


Figure 3.16: Framework protocol stack

The three main designed components are well integrated. The ABCDP protocol is used both by the auto-configuration routines and the service framework. The user information collected by the auto-configuration routines is used by the ABCDP protocol to establish authenticated connections and by the service framework to identify the user that is making the request.

This integration is the most important contribution of this work. Many approaches are able to provide better solutions for specific problems but they do not provide the level of automation that this approach does. By keeping things simple it was possible to design a simple and flexible framework that can be easily deployed in a machine. Without installing or configuring anything else, this framework is able to provide secure support and services for a useful network environment that is many times insufficiently focused.

Chapter 4

Implementation

In this chapter, the implementation of the solution is described. First, an overview of the implemented framework is presented. After that, details about the implementation of each one of the main components are provided. Next, the differences between the implementation of the framework for Android and for Linux are described. Finally, a brief comment about the final status of the implementation is given.

4.1 Implementation Overview

The goal of this work is designing and implementing a framework capable of automatically creating a secure MANET environment, over which web based services can be hosted and used. The methods for achieving this goal have been analyzed in the previous chapter and it is now time to implement a prototype that makes use of the designed methods.

The first issue that must be addressed is the programming language that should be used. The main programming language used to develop applications and services for Android is Java. It is possible to use native binaries but they should only be used if the pretended goal can not be achieved using Java because native code may crash the application without exception treatment. Since Java [42, 49] is a multi-platform language and also allows a high level of productivity due to its simplicity, libraries and complex Integrated Development Environments (IDEs), it seems the logical choice for both Linux and Android implementations.

After that, the implementation must be organized. Similarly to the design of the framework, its implementation can also be divided in the same three components: the ABCDP protocol, the auto-configuration mechanisms and the service framework. But even though they are three separated components, they all share a specific resource. This resource is the configured user information. The auto-configuration routines manage this information, the ABCDP protocol uses it to establish authenticated connections and the service framework can use it to personalize the response to a request. Additionally, the ABCDP protocol is used both by the auto-configuration routines and the service framework.

Since every component must have access to a shared resource, the three components should be implemented as a unit. All the components should be built in a single program, a single Java SE application and a single Android app. The configured user information must provide an API that allows the management and retrieval of user data, in order to be used by the other components. For the same reason, the ABCDP protocol must also provide an API. The other components can be implemented with any structure, since the other components do not try to interact with them.

This should complete the generic overview of the framework, but there is a still a missing detail. Web clients such as web browsers send their HTTP requests over TCP, but this framework does not intend that. Every HTTP traffic should be transfered over ABCDP. Since the commonly used web clients can not be modified to use ABCDP, the TCP messages of the web client must be converted to ABCDP.

The way that was chosen for this is the utilization of a proxy server. A web browser can be configured to use a proxy server. When it is configured to use one, it sends every request to that selected server instead of the final destination and expects the selected server to redirect the request to the final destination. The developed proxy server can perfectly accept local TCP connections and establish ABCDP connections with the final destination. This way, every HTTP traffic that travels through the network is encrypted and authenticated.

But forcing a user to manually set the used proxy in the browser settings is a bad policy. First, the user may simply not to do it, either by ignoring, forgetting or not knowing how to do it. Additionally, it goes against the automatic configuration policies followed in this work. It would be much better if this proxy step could be transparent. It can be so, but not using very clean methods. Still, it is better than asking a user to manually set a proxy when using a plug-and-play support framework.

To allow transparency, every HTTP request must be directed to the local host. A relatively clean solution would be using the NAT table of the `iptables`[1] program to redirect every TCP traffic sent to port 80 (HTTP port) to the local host. Unfortunately, the `iptables` version for IPv6 does not contain the NAT table and therefore this method is impracticable.

The chosen solution is using a local name resolution service to resolve every name to the local host. For this, a simple DNS server that responds local host to every query was implemented. Every connection with destination defined by a name is now directed to the local host. This is not problematic because the traffic generated by the framework has its destination controlled directly by IP addresses and every user requested traffic should be generated by web clients, because every service in the network is hosted by the service framework of some user.

The only thing left to do is implement the proxy server in TCP port 80, because that is still the destination of every traffic generated by the web clients. This proxy server can not use the default name resolution methods, because they always resolve to the local host. This server resolves every name internally in the framework, using the information of the configured users.

The full design of the framework is now completed. Each instance of the framework requires the use of many ports. First, it requires the TCP port 80 for the proxy server and the UDP port 53 for the related DNS server. It also requires two UDP ports for the ABCDP connections of the service framework: one for the HTTP engine and another for the remote invocation engine. Finally, it also requires two UDP ports for the auto-configuration routines: one for sending and receiving advertisement broadcasts and another for the confirmation ABCDP connections. These UDP ports can be any port with a number higher than 1024, but the chosen ones ranged from

[1]The `iptables` program contains a group of tables with rules to process and change packets inside the kernel before they are sent to a network interface or delivered to the correct socket interface. This rules are managed by the user and can do many tasks such as drop packets or change fields in the IP headers. The NAT table is used to change the source and destination IP addresses and ports of packets.

8080 to 8083.

This completes the generic design of the implementation of the framework, which is presented in figure 4.1. In the remainder of this chapter, the three main components of the implementation are described in more detail. Additionally, the differences between the implementation for Linux and the implementation for Android are also presented in detail.

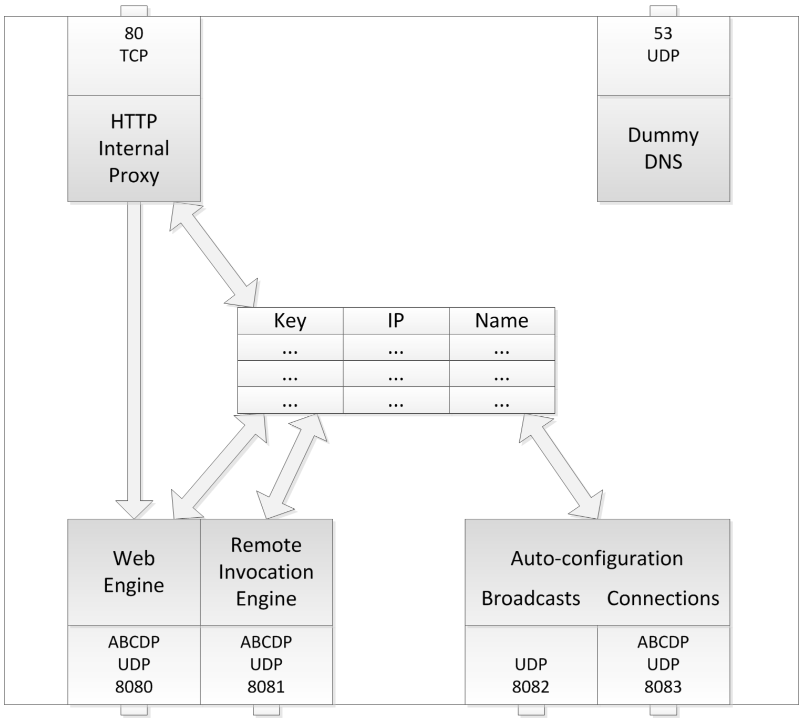


Figure 4.1: Overview of the framework

4.2 Transport Protocol

One of the goals of the framework is the design and implementation of a secure transport layer protocol. This protocol must be able to establish authenticated connections and allow secure data transfer using a distributed public key structure. The protocol should provide an interface designed in a way that allows developers to quickly get used to it. The easiest way is to make the interface as similar as possible to something already existent and widely used.

Considering that the framework is based on Java, the ABCDP socket API should be almost identical to the standard Java TCP socket API. This TCP socket API consists in two main classes with some associated helper classes. It contains a server socket class, that is basically a socket that listens for connections in a specific port and a regular socket class. The latter uses its constructor to establish a connection with a given endpoint and allows reading and writing messages via two standard Java streams (one input stream and one output stream).

Although the API should be the same, there are some details in the protocol design that do not resemble with the TCP design. For example, each connection is not established between a pair of ports. Instead, the same two ports are used by both hosts. If the ports were used to distinguish connections, only one connection would be possible in the same port. To counter this, each port can have multiple connections based on a connection ID field in the header of each packet.

This difference and along with some other minor ones require a different implementation, but the external API can perfectly be the same. The TCP socket API does not require the selection of a source port for the connection, only the destination port is chosen. In this case, the destination port is used for both. Additionally, even though several connections can use the same port, that is controlled by the implementation. The external API is based on the socket objects and each of them represents one specific connection, even if several of them use the same port. The server socket mechanism can also be equal to the TCP one.

If the API can be the same, then there is no reason for not to be the same. The main classes for the new protocol API should then be the same two socket classes and the reading and writing Java streams (figure 4.2). But the internal implementation of a protocol such as this one is quite complex. Along with the API classes, mechanisms for establishing connections, sending and receiving packets, encrypting and decrypting messages, handling acknowledgments and timeouts, and more are required. To develop this protocol, many classes have to be implemented, with methods that make use of UDP sockets and manage ports, threads and ABCDP connections.

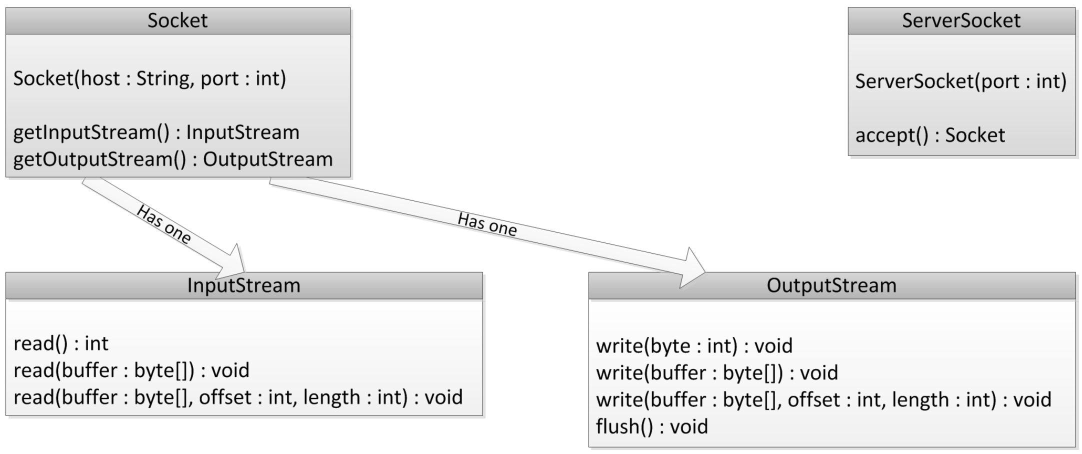

Socket
Socket(host : String, port : int)
getInputStream() : InputStream
getOutputStream() : OutputStream
ServerSocket
ServerSocket(port : int)
accept() : Socket
Has one
Has one
InputStream
read() : int
read(buffer : byte[]) : void
read(buffer : byte[], offset : int, length : int) : void
OutputStream
write(byte : int) : void
write(buffer : byte[]) : void
write(buffer : byte[], offset : int, length : int) : void
flush() : void

Figure 4.2: Basic API of the ABCDP protocol

4.2.1 Implementation Structure

Java provides a UDP socket API that allows the creation of a UDP socket bound to a port, socket closing and message reading and writing. The reading method blocks until a message is read. For this reason each port has its specific thread to read messages. For writing processes the thread that writes could be the one that invokes the write method. Unfortunately if many threads are used when writing to several connections in the same port, the reader thread would not have enough CPU time to cope with acknowledgment reading. This would cause unnecessary delays and packet retransmissions, therefore one specific thread is used by each port to write messages.

The ports are the centerpiece of the protocol implementation. Each port has an associated UDP socket, two threads and can have multiple connections, along with a connection listening socket. All information relative to each port and its connections is managed by the port manager. The port manager is a class with static variables and methods that create port objects and retrieve port objects by port number. When a socket is created, either being a server socket that listens for connections or a regular socket, the information relative to the port is retrieved from the port manager. If there is no information associated with that port, it means that it has not been used yet. In this case a new UDP socket is bound and the a new port information object is generated and added to the port manager.

A port object simply has an association to its reader and writer threads. These threads are the ones that actually do all the work. The port manager and its port objects are just a form of managing the creation of these threads and providing to each newly created socket their respective threads. Each one of these threads maintains a mapping from connection IDs to the associated ABCDP socket objects using their port. The socket object contains all the relevant information about its connection, such as the state of acknowledging, the currently sending message and the secret key to use. The next sections explain in a lightweight form, how these threads function.

4.2.2 Reader Threads

Reader threads receive packets and treat them depending on their type. There are three types of received packets. There are connection packets and two types of conversation packets: message packets and feedback packets. Each one of these types of packet has a specific purpose and must therefore be treated in a different manner.

Each reader thread has a loop that continuously reads packets. When a packet is received, its content is parsed to discover its type. As most of the packet is encrypted, the thread must know what cryptographic algorithm and key to use. For that, the first 32 bits of the packet are used. If they contain the 0 value then the packet is a connection packet and the local private key is used to decrypt the rest of the packet.

Otherwise the value is the connection ID and the packet is a conversation packet. In this case, the connection ID is used to obtain the related ABCDP socket and its secret key is used to decrypt the rest of packet. Connection packets are focused later in section 4.2.4.

Conversation packets can be subdivided in message and feedback packets. As the names specify, a message packet contains a message fragment and feedback packet contains feedback related with the sent packets. The goal of these packets is to reliably exchange messages inside a connection by tackling problems such as packet losses and maximum packet sizes. When all the fragments of a message are received, the message is added to the input stream of the corresponding socket. This occurs when a series of packets with sequential sequence numbers starting from the first packet of a message[2] to a packet with the last fragment flag are received. When a message is added to the input stream, it is ready to be read using the external API, completing its trip through the protocol implementation.

When a message is being received, its fragments are kept in a waiting list until all have been received. An acknowledgment with the sequence number of a received packet is sent when the reception window is filled with sequential packets. If there are missing packets and the received packet is the first of them, an acknowledgment is also sent. This acknowledgment has the sequence number of the packet before the new first missing packet or the one of the most recent message packet (if there are no packets missing now).

Reader threads also receive feedback packets of the sent massages but have no interest in them. These packets are interesting for the associated writer thread to know when it should send new packets or re-send older ones. When a feedback packet is received, the acknowledgment related structure in the ABCDP socket is updated. When new feedback is received, the writer thread is in condition to either send more packets (because there is space in the sending window) or to confirm that the write request is complete. If the feedback is irrelevant, e.g. a duplicated acknowledgment, there is no information to add to the structure. This structure is used by the associated writer thread, which will be explained later.

The final step of the main loop is the error feedback routine. This routine iterates between each connection looking for situations which require sending error feedback. These situations occur when connections that are receiving a message have not received any new fragments or have late fragments for specific amounts of time. This can be done easily because each connection stores the time of reception of the latest fragment. When these situations occur, a receiver ready packet is sent. The sequence number in the packet is the one after the most recent fragment or the first late packet (if there

[2]The first packet of a message is the packet with the sequence number immediately next to the one of the latest packet received that is marked as a last fragment. When no other message has yet been received, the first packet is the first message packet of the connection.

are late packets).

4.2.3 Writer Threads

Writer threads have the task of sending message packets at specific times (feedback packets are sent by the reader threads). Message packets are sent when threads invoke write requests i.e. invoke the flush method of an output stream. These threads then block until the write request is complete. To prevent reader thread starvation, message packets are sent by a specialized writer thread. This thread fragments the messages, constructs all the required message packets and sends them when appropriate.

As previously referred, packets have a maximum number of bytes, limited by the maximum transfer unit of a network interface[3]. This limitation does not allow sending big messages in a single packet. To tackle this problem, messages are divided in multiple fragments. The maximum size of each fragment can be calculated using the maximum data that fits in a UDP datagram[4], the ABCDP header sizes and the size of each encrypted data chunk[5]. With this number, the message can be divided in fragments and each fragment can be placed in a conversation packet.

Message packets should be sent in two occasions. The first one is when there is enough window to send a packet. The other one is when feedback has not been received for a while. In this latter case, the window is reseted and the next sequence number becomes the one after the last acknowledged one. Without using more than one thread, there is no way other than polling to discover which write requests have not received feedback for a specific time.

Each writer thread therefore iterates between connections that have active write requests. As said before, a write request may have enough window to send packets or not. These requests are treated with higher priority because they surely have packets to be sent (the reader thread adds these priorities when feedback is received). But the remainder of the requests may have not received feedback for a while. To detect requests with no feedback, when there are no requests with window to send packets, a loop between every request selects which is the next one to be treated. If one of these requests actually had no feedback for a while, the window is reseted and packets since the last acknowledged one are sent once again.

A write request is complete when feedback acknowledging the reception of the last fragment has been received. When this occurs, write request is removed from the list of active requests. Additionally, the thread that invoked the flush method and is blocked in the write request is woken up and is able to proceed with its own routine.

[3]The maximum transfer unit of an interface is considered to be 1500 bytes.

[4]The maximum data that fits in a UDP datagram is the maximum transfer unit less the size of the IP and UDP headers.

[5]RSA 1024: encrypts 0 to 117 bytes chunks in 128 bytes chunks; AES 128: encrypts 0 to 16 bytes chunks in 16 bytes chunks.

To prevent excessive CPU usage, each polling iteration is preceded by a sleep time. This sleep time is calculated based on the number of active requests. The goal is that a complete request loop should last approximately a specified time and therefore the sleep time is the division between that value and the number of requests. Furthermore, if there are no active requests, the writer thread sleeps until a new request arrives.

4.2.4 Connection Establishment

Connection operations are handled by connection packets. These packets can either be connection requests or acknowledgments. A connection establishment routine starts when the ABCDP socket constructor is invoked in a client host. The thread that makes this invocation fetches the public key of the target host (required to send connection packets) and randomly chooses its connection ID half and sequence number (a list with all chosen connection ID halves is kept to ensure uniqueness[6]). Then it starts sending connection requests periodically until a specific maximum number of attempts is reached.

When the reader thread of the target host receives the connection request, it starts creating its own ABCDP socket object using the values in the packet. It also finds the public key of the requester and randomly chooses its unique connection ID half, a sequence number and a secret key. The newly created socket is added to the socket list and a connection acknowledgment packet is sent. As each connection request from the same host must have a different connection ID half, if a repeated one is received then the requester did not receive the acknowledgment. In this case, the already created socket object is obtained from the list and the associated connection acknowledgment packet is re-sent.

After the socket object has been created in the server host, it is also added to the ABCDP server socket backlog. This backlog contains the sockets that have yet not been returned by the accept method of the server socket. If a thread is blocked in the accept method, that thread is woken up and the method returns the newly created socket. Otherwise, when the accept method is invoked, a socket is immediately returned.

When the reader thread of the requesting host receives the acknowledgment, it updates the socket object with the new values and adds it to the socket list. Afterwards, it notifies the thread that started the connection request that the connection has been established. That thread wakes up and completes the ABCDP socket constructor, proceeding with its own routine. If the thread is not woken up this way until the maximum number of attempts is reached, an exception is thrown. The same happens if the public key of the destination host is not found by either one of the

[6]Connection ID uniqueness is ensured by keeping the locally chosen half unique. If half of the bits of a number is unique then the whole number is also unique.

connection endpoints.

4.2.5 Proxy for Protocol Switching

Most HTTP based service clients, such as web browsers, use the HTTP protocol over TCP. The framework is expecting communications to be performed over the ABCDP protocol, to allow secure and authenticated message interchange. It is not possible to change a commonly used browser to use ABCDP instead of TCP, therefore it is necessary to pick the TCP connection up at some point in the local host and convert it to an ABCDP connection.

There is no simple nor clean way of converting TCP connections to ABCDP connections. The chosen method was to transparently direct every connection from the HTTP clients to a local proxy. This proxy accepts TCP connections and establishes ABCDP connections with the wanted endpoints. Afterwards, it sends any messages received from each connection through the associated one.

The proxy server listens for TCP connections in port 80, the default HTTP port. Any connection that is not established from the local host is ignored right after being accepted. Every connection to port 80 is therefore established from the local host to the local host, but the wanted destination may be a different one. To discover the actual destination, the first HTTP request in the received connection is parsed. The value of the `Host` header field contains the name of the server of the URL requested by the web client. This name can be resolved using the internal user structure of the framework that contains the data of every configured user.

If the actual endpoint is in fact the local host, the local TCP socket can be directly used by the local HTTP engine. Otherwise, an ABCDP connection is established with the remote endpoint, and its HTTP engine treats the request. In the latter case, the contents received by each socket must be sent to the other one, in order to allow the communication between the web client and the HTTP engine. This is accomplished with pairs of threads.

The proxy server uses a pair of threads per received connection. The primary thread is the one that parses the request from the TCP connection and establishes the ABCDP connection. After that, this thread enters a loop that reads content from the TCP socket and writes it to the ABCDP socket. The secondary thread receives references for both connections and enters a loop that reads content from the ABCDP socket and writes it to the TCP socket. This way, the web client and the HTTP engine can transparently communicate with each other in a secure and authenticated manner.

4.3 Auto-configuration Mechanisms

A set of auto-configuration routines is required to provide plug-and-play support for ad-hoc networks. Since there are no “special” nodes to take care of the configuration of the network, every node in the network must use methods that allow a correct configuration of each node. These methods should be as resistant as possible to any type network attack, mainly impersonation attacks.

It was concluded that routines for the generation of routes could not be securely performed and that therefore there was no point in actually implementing one. An existing routing can be used in this scenario. An example of a possible protocol is the MANET extension of the Open Shortest Path First (OSPF) routing protocol [15] using MANET Designated Routers (MDRs), called OSPF-MDR [48]. This is the ad-hoc routing protocol used by the `CORE` emulator [2] that is used for testing in the next chapter, more precisely in section 5.6.

Due to the available time only the essential mechanisms were implemented. Mechanisms for key renewal and resolution of a name to the most fit user from a set of users with the same name were not implemented. Therefore, only an all-in-one solution that manages the user information was implemented. This solution contains routines for user advertisement, IP auto-configuration and name resolution.

The name resolution routines are basic, they are simply performed with Java collection lookup methods. The other routines are more complex. The IP auto-configuration involves the routines that control the network interfaces and the generation of IP addresses from public keys. Finally, the user advertisement and mutual configuration routines are the centerpiece of the implementation, along with the management of the user information. These complex methods are detailed in the following sections.

4.3.1 Extending Java

Auto-configuration routines require changes in network related values such as the IP addresses attributed to network interfaces. Unfortunately, many kernel operations related with socket I/O are not natively supported in Java. It is not even possible to simply obtain the Internet Protocol version 4 (IPv4) address of a certain network interface. Fortunately, there is a way to extend the capabilities of the Java language.

Java can be extended using the JNI. This native interface allows the Java Virtual Machine (JVM) to load and use native libraries. These libraries contain native binary code and therefore can execute any kind of operations. These obviously include kernel operations such as the ones required to obtain and manage network interface related data. The `C` Unix libraries provide the `ioctl` (I/O control) function. This function provides a massive number of I/O related operations which include socket

I/O operations.

The `ioctl` function has three arguments. The first one is an open file descriptor that is used by the function to perform the operation. For socket I/O operations, this file descriptor must be a raw socket. The second argument is an integer value that specifies the request. A set of constants is provided by the `ioctl` headers, with every operation supported by the operating system. This type of request can be for example `SIOCGIFADDR`, which means "Socket I/O Control Get InterFace ADDRess".

The last argument is a pointer to a structure that is used by the function. An operation such as getting an address from a interface (`SIOCGIFADDR`), uses as the last argument a pointer to an interface address structure. This structure has fields such as a pointer to the device name and a byte array with the address. This request fetches the IP address of the given device name and writes the fetched IP address in the correct field. The operation for adding an address to an interface (`SIOCSIFADDR`), uses the two values of the same structure to assign the given IP address to the given interface.

The function returns an integer value that contains error codes. If the returned value is zero, the request was completed successfully. Otherwise, the returned value is -1 and the cause for the error can be checked using the `errno` value.

The `ioctl` function allows many more complex operations. Using it, methods for managing network routes were also developed in the eventuality of the development of an ad-hoc routing protocol. Methods for retrieval and modification of wireless network interface modes were also developed to allow the differentiation of the behavior of the framework depending on the interface being in ad-hoc or an infrastructure mode.

4.3.2 Generating Addresses from Keys

The implemented algorithm for generating IP addresses from public keys is quite simple. An IPv6 address is composed by a network prefix and a host address. The network prefix is used to identify the network and therefore a fixed value of 64 bits with local network scope is used. The host address is the identifier of a host inside the network with the specified prefix (also has 64 bits). This value is generated by passing the public key by a hash function.

The hash function simply applies the "exclusive or" (`xor`) operation to certain bytes of the public key. As the first 29 bytes of the key are always identical they are not used in the key generation process. Each of the 8 bytes is generated by sequentially applying `xor` to the next 8 bytes of the key. For example, the first byte of the host address is obtained by applying `xor` to the bytes 30 to 37 of the key. The second value is obtained by using the same process applied to the bytes 38 to 45.

This algorithm is simple and allows easy generation of public keys that generate specific addresses. This could be used to cause DoS attacks based on duplicated IP addresses. But it is not problematic as every user configuration must be authenticated. This means that any attempt to specify that a user possesses a certain public key and the associated IP address requires that user to know the private key associated with that public key. User configurations are covered in more detail in the next section.

4.3.3 Configuration Routines

The designed model requires every user to know the public key, IP address and chosen name of every other user in the network. For that, each user sends broadcasts advertising his presence in the network and the receivers of these broadcasts use authenticated connections to confirm the advertised data in mutual configuration routines. These processes require two different types of sockets. Each host uses a UDP socket to send and receive advertisement broadcasts and ABCDP sockets to establish authenticated connections.

Configuration routines are achieved with three steps. Firstly, each host advertises its presence in the network by periodically broadcasting its own information. Afterwards, other hosts read its advertisement and verify if the advertising host has already been configured and has not changed its information. If that is not the case, the final step is executed. The last step consists in the host establishing an authenticated connection with the advertising host, both sharing their information via the authenticated connection and finally updating the received information.

This is achieved with three threads, each one for each of the steps. One thread, the broadcaster, takes care of the periodically broadcasts. Another one, the first listener receives the broadcasts, analyses them and establishes the authenticated connections. The last one, the second listener, is the authenticated connection recipient. The use of three different threads facilitates socket read and write operations. The first listener is usually blocked in a UDP socket read operation and the second listener in an ABCDP socket accept operation.

During the initialization of this component, the messages that are sent in the advertisement and in the authenticated configuration packets are built and cached. As the information that is sent in those packets rarely changes, caching these messages reduces CPU usage. Still, the information can be updated. The component provides methods for updating the user information. These methods not only update the variables in memory and in the persistent storage but also update the packet messages.

The broadcaster thread has a very simple routine. It cyclically sends a broadcast advertisement with the pre-built message and sleeps for a specific amount of time. This amount of time increases linearly as the number of configured hosts increases. To allow flooding without using a multicast routing protocol, each node also forwards the received broadcasts. Every time a broadcast is received, this thread is woken up. It then checks if the last broadcast advertising that user has been received shortly before. If not, the broadcast is forwarded. The thread that receives the broadcasts and wakes up this thread is the first listener thread.

The first listener thread has a more complex cycle. It blocks until a broadcast is received by the corresponding UDP socket. As broadcasts are also received by the sender host, the thread checks if the source IP is one of its own addresses and ignores this packet if it is so. Afterwards, it sends the packet to the broadcaster thread to allow its forwarding. After that, it checks if the user with the public key in the broadcast has already been mutually configured in this session. If so and if his information is the same as the one in the broadcast, then the packet is also ignored.

If the packet has not been ignored, then an authenticated connection is established with the broadcast sender. This connection is made between the first listener thread of the broadcast receiver and the second listener thread of the broadcast sender that is accepting authenticated connection. But establishing this connection has a problem. Although the establisher knows the public key of the broadcaster (it is present in the broadcast) and its IP address (generated from the key), the broadcaster may not know the public key of the establisher.

For such an eventuality, the ABCDP protocol provides a data field in the connection packets. This field can be used to send the public key of the establisher in the connection request packet. This way, a connection can be established between two hosts when only one knows the public key of the other. After the connection is established both hosts send their prepared message in turns (the establisher sends its information first). Both users then update the information of each other with the data sent through the connection. The way this information is stored and managed is covered in section 4.3.4.

4.3.4 User Management

In this approach, each user performs mutual configurations with other users. The exchanged information must be stored in order to be used when the configured user is not present in the network. The information about each user is placed in an object. While a user is in the same network, the information is used in memory. But when a user is not in the network, the information should still be accessible. For that, the information should be stored in a database.

The framework uses an automated ORM mechanism to directly map model classes to database table rows. Models and the automated ORM system are covered in more detail in section 4.4.5. To facilitate user data persistence, the class that contains the user information is a model class. As any model class, the object can be used in memory and be easily stored and retrieved from a database.

The user class contains all the configuration related information related to a user. Each user object and associated database table row contain the information related to one specific user. This information is composed by his public key, chosen name and IP address (generated from the public key). The database table stores all this information with the exception of the IP address which can be quickly generated and therefore does not need to be cached.

Apart from the database table, a structure containing a list of the currently online users is kept in memory. This list is built using the mutual authenticated configuration routines explained in section 4.3.3. This list is used mainly for lookups between names, IP addresses and public keys in the processes of establishing and accepting connections. It is also used to list the currently online users, as only these can be part of the list (only online users can perform mutual configurations). If no configuration broadcasts from a user are received for a while, that user is removed from the online user list.

When information about a user is updated, the corresponding user object is updated. If the object is not in the memory structure, it is retrieved from the database and added to the list. The information in the object is then updated and the object is saved in the database. If the user is new (if the public key of the user is not known), a new object is created, added to the list and stored in the database.

4.4 Service Framework

An important component of this work is the service framework. This framework allows an easy platform for the development, deployment and installation of services. This framework relies mainly on an HTTP engine based in an MVC architecture. The main goal of the framework is to reduce the amount of effort required to develop a service. The framework autonomously takes care of almost every aspect other than the development of core routines, such as the generation of visually appealing web pages and the invoking of the correct action requested by a client.

Most of the automation of the service framework is achieved using the Java reflection API. This API allows classes and their contents to be inspected at runtime. Class objects can be obtained by their full name. From them, components such as fields, constructors and methods can be obtained by their names. Alternatively, it is also possible to obtain lists of them. The field objects can be used to get or set the contents of the fields that they represent (as long as the fields are accessible). The constructor objects can be used to create a new instance of the class. The method objects can be used to invoke the methods that they represent.

Unfortunately, due to the available time, just slightly more than the essential features of the framework were implemented. The features that were not implemented include the remote invocation engine, model ownership, user and model groups and view templates for tables and forms. Still, the whole HTTP engine was implemented, along with many other features. The most important aspects of the implemented features are detailed through the remainder of this section.

4.4.1 Service Packaging and Installation

A service that is hosted by the service framework requires various components. The most important one is the Java classes that contain all the code that is executed by the framework. These classes contain the controllers, models, layouts, core service routines, etc. Another important component is a directory containing every static content used by the service, such as stylesheets, images, media or any other web resource that is referenced in the HTML code returned to a client. The final component is a properties file that contains relevant information about the service (its ID, name, description and main page URL).

One of the main differences between running Java code on a computer operating system such as Linux and running it on Android is the JVM used by them. The JVM used by Android (the Dalvik Virtual Machine) uses a different bytecode than the one running in other operating systems. Therefore, a typical Java Archive (JAR) archive with classes that is easily generated by the Java Development Kit (JDK) tools (such as `javac` and `jar`) or by an IDE is useless for an Android device. The Java classes must be compiled to the format of the bytecode used by the JVM in Android. More details about the differences between the implementation of the framework in Linux and Android are presented in section 4.5.

The Android Software Development Kit (SDK) provides a tool (`dx`) that is able to compile a directory of regular Java bytecode classes to a Dalvik Executable (DEX) file, the type of file used by the Android JVM. Therefore, using an IDE to compile the classes of a service to regular Java bytecode and using the JDK and Android SDK tools it is possible to generate both a JAR and a DEX file with the classes of the service.

To facilitate service distribution and installation, the most sensible way is to place all the service components in an archive. The JAR and DEX files, along with the directory with static content and the properties file, can all be placed in one same ZIP archive (figure 4.3). This archive contains every necessary component to run a service and can be easily distributed as it is only a single file. To facilitate service installation, the best alternative seems to be simply placing the service archives in one directory. The service framework then does all the work of extracting the files and using them correctly.

This archive should also be automatically generated by a tool. This is not hard to do if the service project has a specific structure. An IDE such as NetBeans organizes its files in a specific format. Assuming that the service is being developed as a NetBeans project, the compiled Java bytecode is placed inside the “`build/classes`” directory. It can also be defined that the static content to place in the archive is the directory

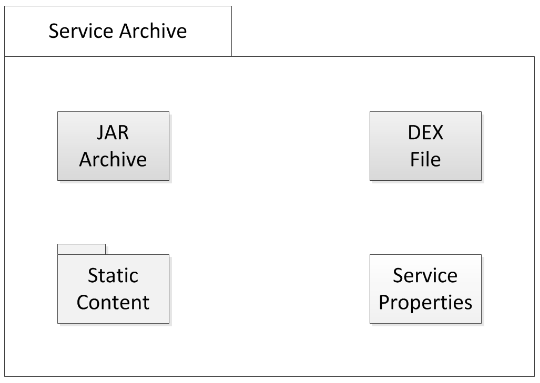


Figure 4.3: Files in a service archive

“`public`” and the properties file for the service is a file called “`service.properties`”. In this situation it is possible to generate a bash script that compiles the JAR and DEX files using the classes in the classes directory and that creates a ZIP archive with them along with the public folder and the properties file.

It is also not hard to install the archive directly in the service framework for testing. Assuming that the machine has a static link to the framework initializer in some directory of the PATH, it is possible to quickly discover where the service framework is. Running the command “`ls -l`” provides information about the static links in the directory. Doing it for every directory in the PATH provides the location of the framework initializer and from it, it is possible to easily obtain the directory that holds the services. It is also possible to manually install a service by placing its archive in the services directory.

Finally, it is not hard to make this assumptions true. If the service framework is installed with a script such as a makefile, it is easy to generate the static link to the initializer. Also, by providing a NetBeans project with a template service, it is possible to guarantee that the files are placed in the specific places. The project contains a `bin` directory with the packaging and installation scripts, a public folder and a properties file with sample content. Additionally, the public folder contains some default stylesheets and source code directory contains an example controller and model. The stylesheets are necessary for the default layout (referred in section 4.4.4) and the example code is useful for new developers. Finally, a JAR library with the service framework classes is provided. This library is imported by the NetBeans project in order to allow classes from the framework, such as the controller and model classes, to be referenced in the project without causing errors.

4.4.2 Service Initialization

The service framework contains a specific folder where all service archives are placed. The initialization process must performed for every individual service archive and is composed by various steps. Each of them must be extracted and analyzed by the framework to allow the service it holds inside to be hosted by the framework.

The information about each service is placed in one directory called "`data`". This directory contains a set of subdirectories, one for each service. Each of these subdirectories is named after the service ID. The first step in the initialization of the service is obtaining its ID. That ID is one of the fields in the properties file of the service. The properties file is extracted and parsed in order to obtain the service ID. The new directory that contains the contents of the service can now be created.

The next step is extracting the whole archive to the folder. With the files extracted it is now easy to access the files of the public directory and the rest of of the fields in the properties file. With the JAR and DEX files it is also possible to obtain the classes inside the file. Although the classes used for class loading are different in both cases, the method and its difficulties are the same. They both use a class loader object to generate class objects with a given full name (package name and class name) from the JAR or DEX files.

There is no problem if a loaded class uses other classes from the same file that were not loaded. The JVM takes care of loading those classes automatically. The only classes that must be manually loaded are the classes that are directly used by the framework. Those classes are the controller classes (more details about controller classes in section 4.4.3). The last step of the service framework initialization is therefore loading all the controller classes from the JAR or DEX file.

The class loader APIs does not provide methods to list every class in the classes file. Therefore it is necessary to take another approach for listing the controllers inside the file. A JAR file is basically a ZIP file. Java provides methods to list the files inside a ZIP file. Using the directory tree and the names of the class files inside them, it is possible to generate the full name of each class in the JAR file. For the DEX file, there is a class that interprets the contents of the file, and provides a method that lists the full name of all the classes.

With the list of the full class names, it is possible to load any class from the file. Each class is inspected to determine if it is a controller or not. A controller class has a name that ends with "Controller" and extends the generic controller class. Therefore each class with a name ending with "Controller" is loaded and Java reflection is used to determine if the class extends the generic controller class. For each class that is a controller, an instance of it is created using the Java reflection API. That instance is then placed in a collection that maps class simple names (without package name) to their respective controller object.

The initialization of each service generates an object containing the information of the service. This structure holds the values obtained from the properties file and the name to controller map. Each of these objects can also be placed in an object that holds the information of all services. This object contains a map between each service ID and the related service object. This object provides an API that contains methods to obtain the name, description and main page of a service from its service ID and to obtain a controller object from its service ID and simple class name.

4.4.3 Controllers and Action Processing

Controllers and actions are the components of a service that contain the routines that are invoked via the HTTP interface. A controller is a class that extends the generic controller class. It contains a set of methods called actions that can be invoked via HTTP requests. Additionally, this class contains one abstract method that must be implemented. This method receives no argument and returns a class that extends layout. This method is used to define what layout should be used as the initial view for the HTML response.

When an HTTP request is received, its contents are parsed and an HTTP request object containing the parsed information is created. This object contains information such as the HTTP method, the path section of the URL, the header fields and the variables (taken either from the URL or from the request content). As a request invokes an action from a controller that belongs to a service, the next step is discovering these three values.

The path section of the URL is used for that. This section is a string that can contain several "`/`" characters. In a URL this characters are typically used to describe a path in a tree, such as a file directory tree. For example, the section `/library/science/physics/list` can be easily seen as a file or a directory, as each name tends to get more specific. But these principles can also be followed to discover what action should be invoked.

The first component of the URL is the service ID. The last one is the action. The components in between can be joined to form the simple name of the controller class. It seems more logical to call a controller class "`SciencePhysicsController`" other that just "`SciencePhysics`". For this reason, the word "`Controller`" is appended at the end of class name. A path such as the one above therefore references the service with the ID "`library`", the controller class "`SciencePhysicsController`" and the action named "`list`".

The provided path may not contain at least three components, which are required for the routine explained above. In this case, the component "`index`" is appended until the path contains three components. A path with less than three components, such as `/library/science` is therefore extended to `/library/science/index`. The root path "`/`" is extended to `/index/index/index`. This follows what most HTTP servers default, which is considering components omitted in the URL as "`index`".

Actions can be invoked using the Java reflection API. For that, it is necessary to obtain the controller object that contains the action. Due to the class loading routines in the initialization of the framework, it also easy to obtain instances of the controller classes that are stored in the service archives (more details in section 4.4.2). With both the service ID and the simple name of controller class, the previously created controller object can be obtained from the object that holds the information of all services.

With the controller object in hand, it is now possible to invoke the action. For that, a method from the generic controller class is invoked. This method returns the method object that will be invoked and receives an HTTP request object as argument. This method uses the reflection API to find methods with one of two names. To facilitate the coexistence of actions with the same name and different behaviors depending on the HTTP method, this method tries to mix the two. Considering that the request is a `GET` request and that the action is called "`list`", the method tries first to find a method named "`getList`" and afterwards a method named "`list`". If it does not find any of them, it throws an exception which results in a "`404 Not Found`" HTTP response. If the developer dislikes this method, it is able to override it in any controller class.

An action is a method that receives both an HTTP request and an HTTP response object as arguments. The HTTP response object is also generated after the controller is obtained and contains methods to build the HTTP response. It uses the layout chosen in the controller to initialize the view for the HTML response. The methods of this class allow the modification of the status code, the header fields and the HTML response content. When the execution of the action ends, the HTTP response can be given. The information contained in the HTTP response object can be used to write the response.

There is another type of request that is more easy to process. Requests for static content almost only require one file to be opened and copied in the contents of the HTTP response. These requests are the ones where the second component of the URL path is "`public`". In this case it is not necessary to fetch a controller and everything after the second component is the location of the file to obtain. A request with the path `/library/public/images/picture.pic`, gets as response the contents of the file `/images/picture.pic` under the public folder of the service with ID "`library`".

4.4.4 Views, Layouts and Templates

Views, layouts and templates are the interfaces chosen to build the HTTP response content. The three combined try to make it easy and fast for a developer to create a sober and minimally eye-pleasing HTML interface with the minimum amount of code lines. Views are used to construct the response of the HTTP request. Layouts provide the initial views for the requests. View templates provide helper methods that generate complex HTML structures. Using the default layouts, their related stylesheets and some view templates, programming effort in HTML interfaces can be greatly reduced.

A layout is a class that extends the generic layout class that just contains one abstract method. This method receives an HTTP request as argument and returns a view. As it is abstract, it must be implemented when developing a new layout. This method generates the initial view for the response of an HTTP request. The contents of it can be based on some information of the HTTP request, being that the reason why it is passed as an argument. When implementing this method, the developer should define the contents of every static section and place the location of every dynamic named section.

A view is a fully implemented class and provides methods for managing the sections inside it. A view contains a numbered list of sections (strings) and a structure that maps names to section indexes. When adding a static section, the new section is simply appended to the end of the list. When adding a dynamic section, a section with no content is added at the end of the list and a new entry is added to the map. Methods for setting or append content to a named section are also provided. These methods use the name of the section to obtain the index and update the string in that index of the list. Finally, a method that orderly concatenates the strings of the list is provided to generate the final content of the HTTP response.

A view template is not based on any particular class. It is just a class or even a method that generates complex HTML markup strings to set or append in dynamic sections. View templates can be for instance a class whose constructor receives a collection and one method of it returns a string containing an unordered list `<ul>` with the contents of the collection. Alternatively a class could provide a static method that receives the collection as argument and returns the unordered list. The first choice with many small methods would be preferable as that class could easily be extended and some methods could be altered which would allow easy customizations of the template.

Although a default layout is provided with the framework, other layouts can be created as referred earlier. As many layouts can be obtained from external sources in an HTML format, a way to convert these files in layout classes can be useful. For that, a preprocessor that converts a JavaServer Pages (JSP) syntax file to a Java layout class was developed. This preprocessor simply separates the file in different parts using the `<%`, `<%=` and `<%@` opening tags along with the generic `%>` closing tag.

The contents outside any tags are literal HTML, which is placed in a static section.

The content inside `<%` tags is Java code that can be directly pasted in the layout class. The `<%=` tags contain one Java code operation, with a return value that can be placed in a static section. Finally, the `<%@` tag contains the name of a dynamic section, that will be filled during the execution of an action.

4.4.5 Models and Automated Object-Relational Mapping

An ORM framework takes care of the mapping between objects and database tables. Each object class is directly mapped to a database table, and each instance of an object can be mapped to a row in the associated database table. The main goal is a developer designing a completely normal object with some characteristic that specifies it is a model (in this case extending the generic model class) and with that the ORM framework provides APIs to easily fetch and store objects from the database without the developer having to use any SQL queries.

Generating such a mapping is not trivial and developers many times want a specific mapping, especially in large scale applications where data is distributed and replicated in various databases. Some fields may contain information such as default values, specific primary keys or simply a "not null" characteristic. It is also important to take into account database roles and privileges. But in local applications such as these services, it is easy to automatically generate a mapping.

In this framework, a model is a class that extends the generic model class. Each model contains an integer field named "`ID`" (declared in the generic model class) that is used as a primary key. This field is defined as auto incrementing in the database engine, therefore the database engine takes care of the value. Every other field from the model class can be directly mapped into a table row with a similar type. To achieve this, various steps are necessary. First of all, it is necessary to decide what database engine to use and establish the Java to database type mappings.

The database engine to use is limited by the Android platform. Android only provides off-the-shelf support for the SQLite database engine. This engine has various advantages when compared with other database engines. Most database engines are a specific process or set of processes with the job of running the engine and hosting the interface for other processes to interact with the database. Unlike them, the SQLite engine is a library that provides an API that receives SQL queries and directly manage the database. An SQLite database therefore does not require any installation, configuration or even initialization, as long as the libraries with the SQLite engine are provided by the programming language libraries or the application.

The most typical way to interact with databases using Java is the Java Database Connectivity (JDBC) interface. This interface uses JDBC drivers which are classes that implement the JDBC interface (interacting with a specific database engine) to provide a generic API that can be used for any database engine. Two JDBC engines were used, one for Linux computers and another for Android devices. The one for Linux contains native libraries with the SQLite engine and the one for Android is a wrapper that uses the SQLite libraries provided by the Android platform. Using JDBC drivers in both cases, the source code for the ORM framework can be used in both platforms.

The type mapping is not very complicated because SQLite does not support many types. Strings are mapped to the type `TEXT`. Integer values such as the types `int`, `char`, `long` and `boolean` are mapped to the type `INTEGER`. Floating point values such as `float` and `double` are mapped to the type `REAL`. Any instance of a serializable class can be converted to a byte array and can be seen as a Binary Large Object (BLOB). These objects are therefore mapped to the type `BLOB`.

To communicate with the database, SQL statements are used via the JDBC interface classes. These statements can be easily generated manually or with the help of the prepared statement class. To generate SQL statements that store an object in the database it is necessary to somehow extract the variables from the object. Similarly, to transform a database row fetched from a query into an object it is necessary to create a new object and set the variables in its fields. All these tasks can be performed using Java reflection.

A last point to focus is the creation, modification and deletion of the database tables. The table of a class is generated when an instance of that class is inserted in the database for the first time. The modifications and deletions occur at the ORM framework initialization. In a development environment, the model classes may change from one service framework run to another. Therefore, at startup, the existing tables are compared to their respective classes. If the class no longer exists, the table is deleted. If some fields have been added or deleted, the respective table columns are added or deleted.

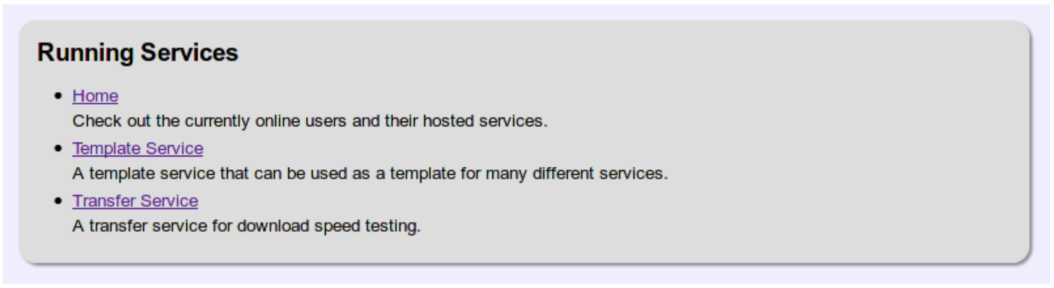
All this generates an easy to use model API. The model class possesses static methods such as `findModel`, that receives as arguments a model class and an ID and returns the respective object from the database. It also provides the static method `findAllModels` that receives as argument a model class and returns a collection with all the objects of that class. Another method is the `findAllModelsByField` that receives as argument a model class, a field and a value and returns a collection with all objects of that class with the value of that field being equal to the given value. Instances of model objects possess methods such as `save`, that stores or updates the object in the database.

4.4.6 The Index Service

An HTTP server usually contains a home page, usually referred as "`index`". This is the page that is returned by the server when it receives a request for the "`/`" resource. If this page is the home page of each user, then it is the best place to show information about the services that are running, the users that are online and change server settings. The settings and user list are only available if the client accessing the page is the owner of the service. The running service list is shown to every user.

As explained in section 4.4.3, the URL `http://localhost/` is redirected to the URL `http://localhost/index/index/index`, which specifies the action named "`index`", from the controller class called "`IndexController`", in the service with ID "`index`". Therefore there must exist such a service in the services folder every time the framework is initialized. To achieve this, a service called index is provided in a specific directory of the framework and is always copied to the services folder before service initialization[7].

This service is composed by two small controllers: the "`IndexController`" and the "`SettingsController`". The first only has the action called "`index`". This action iterates through the object that contains information of all running services (specified in section 4.4.2) and uses it to generate a list. That list contains the name of the service, its description and a hyperlink to its main page (defaults to "`/`" if not provided in the properties file). An example of this service list can be seen in figure 4.4.


Running Services

- Home
  Check out the currently online users and their hosted services.
- Template Service
  A template service that can be used as a template for many different services.
- Transfer Service
  A transfer service for download speed testing.

Figure 4.4: A running services list with 3 services (including the index service)

This action also shows the online users in the sidebar. This list contains the users that have been configured by that user and that are therefore accessible via the ABCDP protocol. Only the service owner can see this list because it may contain users that have not been configured by visited users. This list shows the names of all configured users and contains hyperlinks to their home pages, so their running services can be easily listed and accessed. An example of an online users list can be seen in figure 4.5.

The "`SettingsController`" has two actions: "`index`" and "`updatename`". Both actions are only accessible to the service owner. Any user that is not the owner is redirected to "`/`". The "`index`" action presents web forms to change settings related to the server. The only one that is implemented is changing the local user name. The

[7]This copy is used as a safeguard in case a user deletes of changes the index service archive while managing the service folder.

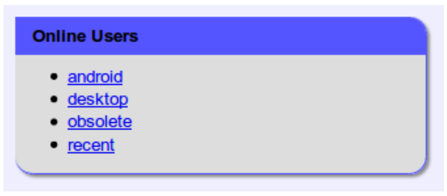


Figure 4.5: An online users list

name form can be submitted, which invokes the "`updatename`" action. This action changes the name in the framework and redirects the user back to the "`index`" action.

This service provides the functionalities of user and service discovery in a user friendly way. Furthermore, it allows easy access to any service in the network. More detailed screenshots of this service can be observed in appendix A.3. The source code of this service is provided in appendix A.1.

4.4.7 Protection Against Malicious Services

One problem that has not been addressed yet is the ability of this framework to run any sort of server side scripting. Since a user can install and host any service, malicious services that try to obtain private information or cause damage to the hosting machine can be developed. Considering that the service framework is running with root privileges, without any type of control, a malicious service can read or write to any file in the whole operating system, such as the used private key file.

It is important to develop methods that limit the possible actions of a service to a certain scope. This limitations should limit the accessible files (including the private key file), the creation of sockets, the running of external commands, the access to private user data and the invocation of non-accessible methods (which is possible using Java reflection). Due to the number of alterations required in the code, these limitations were unfortunately not implemented. But they deserve a mention due to their high importance and because some methods to implement these limitations have been designed.

One of the main problems is the root permission. The service framework must have these permissions to access the private key file and create the socket in a port lower than 1024. But this is just required in the initialization of the framework. After that, the root privileges are not required for anything else. One possibility is that during service initialization, instead of simply loading the services, a new JVM process without root privileges could be started per service. Each process takes care of loading and hosting one single service. This way, the process with root privileges does not run any sort of server side scripting. It is still this process that accepts ABCDP connections, parses requests and sends responses but instead of running the server side script routines, it redirects each request to the correct service process, possibly using local sockets.

The next problem is limiting the accessible files. Access to files should be limited to the "`data`" subdirectory of the service. Since each service is now running in a different process, Java policies can be used to limit file permissions. The Java policy [50] is a mechanism that allows the limitation of the possible actions of a JVM process, such as the files and directories that the process should be able to read or write in. With this mechanism, it is possible to limit the accessible files of each service to their corresponding "`data`" subdirectory (which obviously do not contain any private key).

Another problem is limiting socket permissions. The Java policy is also able to forbid the binding of TCP and UDP ports, along with forbidding the establishment of TCP connections with specific source or destination IP addresses and ports. If it was not for the necessity of transmitting HTTP requests and responses between the server process and the service processes, every socket operation could be forbidden. But it is possible to forbid every operation other than binding a certain port and accepting connections in it. Before running any server side script, each service process binds this port and waits for a connection from the framework server. It should only accept local connections, that supposedly are only performed by the server, since no server side script can establish connections.

It is also important to disallow running external commands because these commands are not limited by the policies. Therefore, no external command should be able to run at all. This is also achievable with the Java policy. Unfortunately, Java policies only limit the invocation of some specific methods. Equal or similar methods can be implemented using JNI code that is also not affected by the policies. Fortunately, loading JNI libraries can also be forbidden by the Java policies.

Another problem is the Java reflection API allowing the suppression of access checks. In other words, the reflection API is able to make `public` any field, method, constructor, etc. This access check suppression can also be disallowed by the Java policy. This policy should in fact be placed both in the framework process and the service processes, just for the safe side. Everything should be done to prevent the obtainment of the private key and of other personal data.

Other matter that should be addressed is the user personal data. The personal data of a user should not be accessed by other users. A malicious service may try to access the data of another service. There is no reason to allow that, so each service should have its own database, placed in its "`data`" subdirectory. This way, the data of each service is only accessible by the service itself. Another problem is accessing the data of another user. A malicious service may ask for private data to a user and provide a backdoor to obtain it. If the attacker is the owner of the service, the data is stored in his machine and there is nothing that can be done.

But the problem is different if the service is being hosted by another user that is unaware that is hosting a malicious service. In this case, something can be done because the attacker is a user like any other. A possibility is forcing the ownership of every model and allow only their owner and the root user to access them. Unfortunately, this limits the capabilities of the framework which makes it an invalid solution. Still, owned models should disallow access to other users by default and provide an option to disallow this limitation. This can sometimes prevent the obtainment of private data caused by some bugs in the developed service.

Last but not least, the policy that disallows the usage of another Java policy set should also be added. Additionally, the file that contains the policies to use should be owned by the root user and should not have write permissions for any user. There should also be a warning somewhere, stating that the policy file should not be changed nor replaced. A checksum verification of the policy file could also be placed in the initialization of the framework. As for the private key file, it should be owned by the root user and only the root user should have reading permission (no user should have writing permission).

Some of the problems of allowing uncontrolled server side scripting have been presented along with some possible solutions. There may be some more problems that have slipped by, because there numerous ways of causing unwanted behaviors with server side scripting. Allowing this type of scripting is always a risk, but it is necessary in most types of service frameworks.

4.5 Android Modifications

The implementation of the framework was at first developed to work in Linux. Since most of the implementation was done in Java, most of the code should be able to be reused for the Android version. Such assumption was correct, but still not everything could be reused. Some parts of the code had to be adapted and others had to be completely rewritten. This section presents the modifications that had to be made to the code in order to make it work in both Linux and Android.

The main difference in the development of Java code between Linux and Android is the different JVMs used by them. In computers, the Java source code is compiled to a binary format that is used by regular JVMs. Android uses a different JVM, called Dalvik. This JVM uses a different binary format, the DEX format. Some problems raised by this difference have already been referred in sections 4.4.1 and 4.4.2.

But that is not the only problem. The Dalvik JVM can only be managed by the Android operating system. At startup, the JVM is initialized by a non-root user. Considering that this JVM is the only possible way of running Java code in Android, it is not possible to run any sort of Java code that requires root privileges. Routines such as binding TCP or UDP ports under 1024 to changing the IP address of a network interface are not possible using Java.

In fact, a user should not be able to run anything with root privileges in Android. Fortunately, it is possible to do so after the device has been rooted (more details are provided in section 5.3). After the rooting operation, it is possible to run any wanted command line as the root user and Java is able to run command lines. This can be used as a workaround for the two problems referred at the end of the last paragraph.

The TCP port 80 and UDP port 53 must be bound to allow the transparent proxy to work. These ports can not be actually bound using Java code but a simple workaround is possible. Any other ports with numbers higher than 1024 are bound and the NAT table of the `iptables` is used to redirect packets. Every packet sent to the wanted ports is automatically processed by the `iptables` program and its destination port changes to the actually used ports. The `iptables` program has a command line interface that allows the management of its rules, such as the redirection rules in the NAT table.

The problem of the network interface operations is not that simple to solve. Although these operations use the JNI that imports native binaries, these binaries are still run by the Dalvik JVM and not by other processes that could have root privileges. The simplest solution is reusing the developed native `C` code and building a new program. This program is just a command line interface to invoke the functions that were previously developed for the JNI interface. It parses its the arguments and calls the requested function using the requested arguments. This programs then prints the answer in a format that can be easily parsed by the Java code.

Unfortunately, compiling this source code is not very simple. Android devices contain processors that use the ARM architecture, which uses a different format for binary code. The source code must therefore be compiled to the correct format. This was achieved using cross-compilation. A toolchain containing native libraries for ARM and programs such as a version of `gcc` compiled for the i386 architecture but that compiles code for the ARM architecture was used for the compilation.

Some more minor changes are required, such as the utilization of two different JDBC drivers. Each driver has functionalities that are not implemented in the other and for that reason, only a set of the functionalities can be actually be used. Another minor changes are required in the initialization. The Linux version uses an initialization script whilst the Android version uses an Android app to initialize the framework service.

Even though the referred modifications had to be made, most of the Java code could be reused without any modification. Still, getting started with the internals of Android, discovering the required differences along with discovering and implementing its workarounds required its fair share of time and effort. But it was worth it, since the result is a framework that works correctly in both Linux and Android.

4.6 Final Overview

This section completes the description of the implementation of the developed prototype. This prototype merged the three designed components: the ABCDP protocol, the auto-configuration routines and the service framework. Every component was developed satisfactorily and the framework is able to correctly interconnect the three components. The implemented prototype also works correctly in both Linux and Android.

The final result of the implementation is a mixture between a NetBeans and an Eclipse project. NetBeans compiles the source code to the Linux version and Eclipse compiles the Android app using exactly the same source code directory. Two more NetBeans projects have been developed, one with the Index service and another with a template service that can be used as a base for the development of new services.

Unfortunately, not every designed feature was implemented due to the limited available time. Mostly only the essential features were implemented. Still, the set of implemented features provides a valid proof of concept. The implemented prototype is, recalling one of the initial sentences of the chapter, a framework capable of automatically creating a secure MANET environment, over which web based services can be hosted and used.

Chapter 5

Results

In this chapter, some results of this work are presented and analyzed. First, an introduction to the results is given. After that, the specifications of the used machines and some practical results are presented. Next, the performed tests are described. Their results are presented and analyzed. Finally, some conclusions obtained from the analyzed results are presented.

5.1 Introduction to the Testing Process

With the process of implementing the framework completed, it is time to perform some tests to the developed code and analyze the obtained results. These results should range from functionality to performance and scalability, throughly analyzing the designed approach and the quality of the implemented framework. But first, a quick reference to the developed code should be made, since the code is by itself a result.

A large number of lines of code was written in multiple programming languages. Most of these lines were written in Java. Some of them were written in `C` for the native network interface related code, Bash for the command line scripts and HTML in conjunction with CSS for the layouts. The number of lines written in each language is presented in table 5.1. These numbers do not count empty lines nor commented lines, they only count lines that actually contain source code. As for the number of developed Java classes, each Java file contains at least one class and rarely more than one. The number of developed classes is therefore just above 100 classes.

Language	**Files**	**Lines**
Java	99	7674
`C`	4	454
Bash	8	214
CSS	4	127
HTML	3	57
Total	**118**	**8526**

Table 5.1: Lines of code written per programming language

Using this developed code, a set of tests was performed. First, the framework was installed in several machines, including one Android device. After that, it was verified that the framework was functioning correctly in each machine. Afterwards, tests that aimed for network configuration, performance and scalability were performed. This set of tests, along with their results and their respective analysis is presented in the following sections of this chapter.

5.2 Specifications of the Used Machines

The test results presented and analyzed in this chapter were obtained using 6 different machines. These machines were used for all different test types and scenarios such as functionality and performance tests performed either in a real or emulated scenarios. Initially only 5 machines were used but at a certain point a sixth machine was required. The situation is explained in more detail in the section that refers to the test that required the new machine.

One of the machines is an Android device, a Samsung Galaxy Ace GT-5830. This machine used its original operating system (Android 2.2.1), with the original kernel. This device will be referred as `Android` throughout the remainder of this chapter. The other five machines are computers running various versions of the Ubuntu Linux distributions using various kernel versions. One of the machines is a desktop computer that will be referred as `Desktop`. The other four are laptops of different generations. These machines will be referred as `Obsolete`, `Client`, `Server` and `Recent`.

Each machine used a wireless network interface with a maximum bandwidth of 54 Mbps (the most common nowadays). Every machine included this interface in its main hardware except for the `Desktop` machine that used a Universal Serial Bus (USB) wireless network interface. The remainder of the relevant specifications are shown in table 5.2. Other specifications such as storage specifications (hard drives, for example) are not relevant for the tests that were performed. The read and write speeds of the used storage devices far exceed the maximum bandwidth of the used wireless network interfaces.

Name	**CPU**				**Memory**	
	Technology	**NC**[1]	**NT**[2]	**Frequency**	**Size**	**Speed**
`Android`	ARM 11	1	1	800 MHz	278 MB	————
`Desktop`	AMD Athlon 64	1	1	1800 MHz	512 MB	400 MHz
`Obsolete`	Intel Pentium M	1	1	1600 MHz	512 MB	166 MHz
`Client`	Intel Core 2 Duo	2	2	2200 MHz	2048 MB	667 MHz
`Server`	Intel Core 2 Duo	2	2	2533 MHz	4096 MB	800 MHz
`Recent`	Intel Core i7	2	4	2700 MHz	4096 MB	1333 MHz

Table 5.2: Specifications of the machines used for testing

5.3 Initial Tests and Practical Results

To run the framework in a computer in Linux, apart from having a JVM in the system, the only required step is copying the framework files to a directory in the computer. Those files provide an initialization script that takes care of the whole initialization. An optional step is creating a static link to the initialization script in a directory belonging to the PATH. This allows initializing the framework from any directory and allows the service installation script provided in the template service project to work.

Running the framework in an Android device is more tricky. First, the device must be rooted. A device must be rooted in order to execute user requested programs with root privileges, which are essential for the framework to work. There are numerous applications that are able to root an Android phone. After that, it is necessary to

[1]NC is the number of cores.
[2]NT is the number of threads.

install BusyBox[3] for Android, because it contains some required programs. Some rooting software also installs BusyBox by default.

Afterwards, it is necessary to change the `wpa_supplicant` executable to a version that allows connecting to ad-hoc networks. This allows the device to consider an ad-hoc network as an infrastructure network. The device is able to connect to it as long as the device receives an IP address via DHCP. The device also does not function as an access point. More can be done to improve this behavior but this is sufficient for testing purposes. Finally, the framework app must be installed. Running the app provides access to the initialization of the framework.

Having installed the framework successfully in a laptop (`Server`) and in an Android device (`Android`), several basic tests were performed. The framework functioned correctly both in the laptop and in the phone. Both were able to access their index service, the information in the services list was the correct one and their hyperlinks worked correctly. After that, both devices were connected to the same ad-hoc network and both initialized the framework. The two devices were able to configure each other and the names of each other appeared in the online user list of the index service. The hyperlinks in the user and service lists worked correctly both for the owner of the service and for the visitor.

After that, the framework was installed in another laptop (`Obsolete`). This laptop also joined the network and all three initialized the framework. All three connected and everything worked correctly just like in the two machine test. In a final test, the three machines initialized the framework at different times. Several runs were performed varying the order and the time differences. The result was also satisfactory.

The development and installation of services was implicitly tested, because the index service was developed from the template service and its packaging tools. It was loaded by the framework because its service archive was present in the services directory (which is the synonym for being installed). Even so, another service was developed. Based in the template project, a controller class was developed and compiled using NetBeans. After that, using the tools provided in the project, the service archive was built and installed. After initializing the framework again, the new service was correctly listed in the service list.

An action in the developed service contains the code that had been developed a way back, in the construction of the default layout and its related stylesheets. This code can be used as showcase of the possibilities of the default layout with the help of some helper methods (some that have been developed and others that can be developed without much effort). Some screenshots of this showcase and the index service are presented in appendix A.3.

[3]Busybox embeds small versions of many UNIX utilities in one small program. The small size makes it suitable for Android devices.

5.4 Network Construction Tests

The first group of non functionality related tests that was performed measured the cost of configuring nodes in a network. The test started by connecting the machines `Android`, `Desktop`, `Obsolete`, `Server` and `Recent` to a same ad-hoc network. Afterwards, the framework was initialized in each of them, with a specific order and one at a time in one minute intervals. The framework ran in logging mode, which logs to a file every packet that was sent and received by the framework. An excerpt of such a log is presented in figure 5.1.

```
0.624040:  sent configuration:  machine-name ( 178 )
0.751051:  conn 2001::257f:49bf:ff4:15a -> * : 76ef0000 - 62c - 0 ( 260 ) | SYN | DATA
0.892215:  conn * -> 2001::257f:49bf:ff4:15a : 76ef73ed - ae1 - 0 ( 132 ) | SYN ACK |
0.952308:  pack 2001::257f:49bf:ff4:15a -> * - 76ef73ed - 62d - 0 ( 14 / 36 ) | LF | DATA
0.952593:  pack * -> 2001::257f:49bf:ff4:15a - 76ef73ed - 62d - 0 ( 0 / 20 ) | ACK LF |
0.954027:  pack * -> 2001::257f:49bf:ff4:15a - 76ef73ed - ae2 - 0 ( 14 / 36 ) | LF | DATA
0.962317:  pack 2001::257f:49bf:ff4:15a -> * - 76ef73ed - ae2 - 0 ( 0 / 20 ) | ACK LF |
```

Figure 5.1: Excerpt of a configuration in the packet log (suppressed data)

From the logs it is possible to extract the time, packets and bytes that were used in each individual configuration. With a more complex organization of the extracted information, it is possible to obtain the time, packets and bytes that were transfered for the complete configuration of a new node. By connecting one new node at a time and by a specific order, the logs of each machine can be used to obtain the values for various network sizes. For example, the log of the third node contains the information obtained from joining a new node to two that were already in the network.

This test was done various times. To automate the task, the `cron` environment was used. This environment allows commands to be executed at a specific time. It is configured with simple rules using the `crontab` program. The first node was configured to start at the minutes 0, 10, 20, 30, 40 and 50 of each hour (minutes finishing with 0). The second node started at the minutes finishing with 1 and so on. Every machine was configured to stop the framework at every minute that finishes with 8. This way, the test was executed once for each 10 minutes. In total, 9 test runs were performed.

After the data has been collected, it was analyzed to obtain the values described above. For that, a log parser was developed. This parser read each line of the log (each packet) and grouped them in connections and configuration routines. Each configuration routine possessed the time that it took (the time difference between the advertisement broadcast and the last packet of the configuration connection), the number of packets and the number of bytes. After the parser processed each log individually, their results could be combined to generate the network construction figures.

5.4.1 Configuration Figures

The average results for the configuration routines performed by each machine are presented in table 5.3. For each machine to produce the same expected values, all of them used a name with the same size (12 bytes). It is expected that each configuration contains 7 packets: an advertisement broadcast, a connection request, a connection acknowledgment, two packets (each one with one machine name) and their two acknowledgments.

Node	Time (s)	Bytes	Packets	Routines	Over expected (%)
`Android`	0.643	682.973	7.027	37	3.174%
`Server`	0.858	685.368	7.105	38	7.139%
`Recent`	0.759	683.946	7.054	37	3.571%
`Obsolete`	1.213	684.300	7.075	40	12.302%
`Desktop`	0.532	682.000	7.000	36	0.000%
All	0.809	683.745	7.053	188	5.235%

Table 5.3: Average values from the performed configuration routines

After analyzing the logs it was possible to conclude that for 12 byte name configurations, each broadcast has 178 bytes, each connection request with a public key has 260 bytes, each connection acknowledgment has 132 bytes, each message has 36 bytes and each acknowledgment has 20 bytes. These values sum up to a total of 682 expected bytes per individual configuration. These values disregard the fact that a same broadcast can be used for several configurations. At this point configurations are only being analyzed individually.

A specific number of configurations per machine is also expected. Each one makes configuration routines with 4 other machines in each of the 9 test runs. That results in 36 configurations performed by each machine. Each configuration actually counted by both machines, but once more, this analysis only focuses individual configurations. The results show that most machines perform more than 36 configuration routines. This is possible due to double configurations.

Double configurations can occur when two machines receive broadcasts from each other at approximately the same time. More specifically, they happen when a machine `A` received a broadcast from another machine `B` and started its configuration routine with `B` and `B` receives a broadcast from `A` before the configuration routine has been complete. As `B` does not have `A` configured at this point, it also initializes a configuration routine. This situation is more likely to occur with machines that take more time to complete configuration routines.

Packet losses and double configurations explain the values presented in table 5.3. It is easy to notice that machines that take more time to perform a configuration routine have more packets over the ones expected. It was already explained why that happens. But there is another noticeable performance relation. With the exception of the Android device and the desktop computer, there is a inverse relation between CPU/memory speeds and configuration times.

Analyzing some logs, such as the one in figure 5.1, it is possible to conclude that the most time consuming task is the handling of connection packets. The only major difference between those packets and the conversation packets is the algorithm used for encryption, which is more time consuming. Assuming that the implementation of the cipher engine is mostly single-threaded, this relation can be easily explained. In the case of `Android`, it is possible that the cipher engine provided with it (Bouncy Castle [11]) is more efficient.

But that still leaves the `Desktop`. In this case, no packets were lost at all, which lowers the average configuration times. Looking more closely at the logs, it was noticeable that the lowest configuration times were high when compared with the other machines (with the exception of `Obsolete`). This fits correctly with the noticed relation. But why did `Desktop` not drop any packets? The reason is possibly in the I/O architecture of the computer. Being a desktop computer, especially a not very recent one, it was not designed with a high focus in energy consumption. This probably causes the processes that handle the packets in the network devices to wake up faster in sporadic packet transfer situations, reducing wait times and clearing buffers faster.

5.4.2 Joining Node Figures

After having analyzed configuration routines in an individual basis, it was time to analyze the values related with the joining a new node. This time, the connection routines were grouped in order to obtain the time, bytes and number of packets required to join a new node to the network. The analysis process is the one described above and it shows that the order of framework initialization is important. The order was first `Android`, then `Server`, `Recent`, `Obsolete` and finally `Desktop`.

It is possible to calculate the expected values in each case. When a second machine joins, all 7 packets of the configuration routine are required. But after that, the broadcast advertisement sent by the arriving machine reaches every other node. For every extra node, only the other 6 packets are required. In other words, the expected value is one single broadcast of 178 bytes and for each “node - 1” a connection with 6 packets and 504 bytes. It is important to notice that broadcast forwards are not being counted here.

Figures 5.2 and 5.3 present the amount of bytes that were transfered during the configuration of a joining node in different network sizes. Figures 5.4 and 5.5 show the number of packets exchanged in the same situation. The shapes of the graphs are similar because there is an obvious relation between the number of packets sent and the amount of data transfered. The joining of the second and fifth machines was

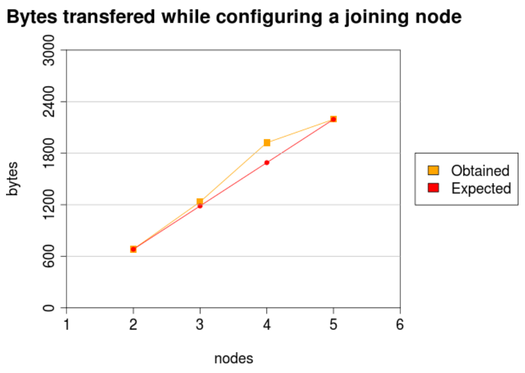


Figure 5.2: Bytes transfered while configuring a joining node versus the expected values

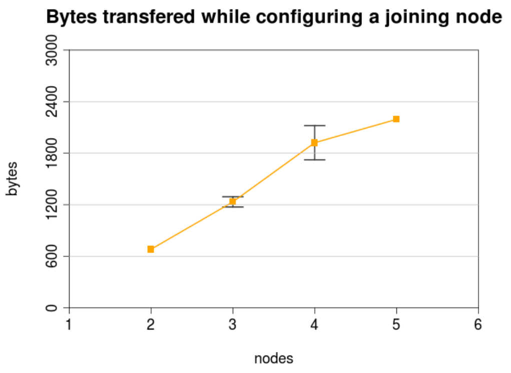


Figure 5.3: Bytes transfered while configuring a joining node with 95% confidence intervals

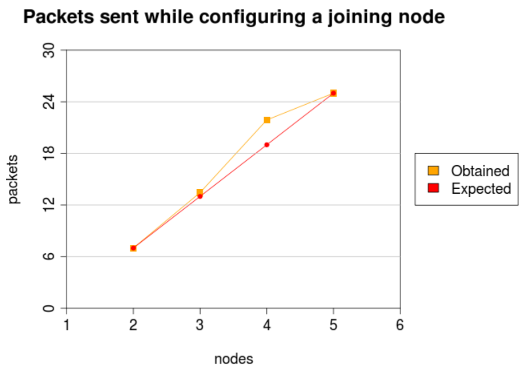


Figure 5.4: Packets sent while configuring a joining node versus the expected values

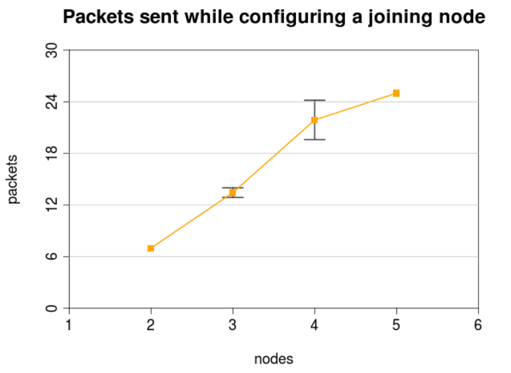


Figure 5.5: Packets sent while configuring a joining node with 95% confidence intervals

performed with the expected number of packets and bytes. On the other hand, the joining the third and fourth machines required more packets that expected.

Comparing these figures with table 5.3, it is possible to understand what happened. The fourth machine to be configured (`Obsolete`) was the one that had the worst performance. Knowing that each machine should have performed 36 configuration routines, it is possible to know how many extra routines were performed by each machine. `Obsolete` performed 4 extra configurations and the all other machines combined also performed 4 extra configurations. Considering that each routine is being counted by the two hosts that performed it, it can be concluded that every double configuration that occurred had `Obsolete` being one of the participants.

Moreover, the last machine to be configured (`Desktop`) did the expected number of configurations. This means that there were no double configurations between `Desktop` and `Obsolete`. Since it was the last machine, every `Desktop` configuration counted for the 5 node entry. This means that all 4 double configurations counted for the 4 node entry. As for the third machine, the variation between the obtained and the expected values is caused by packet losses.

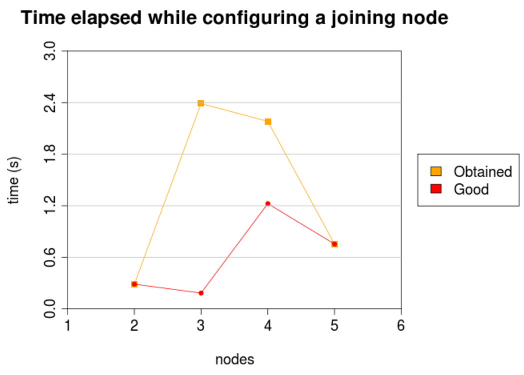


Figure 5.6: Time elapsed while configuring a joining node

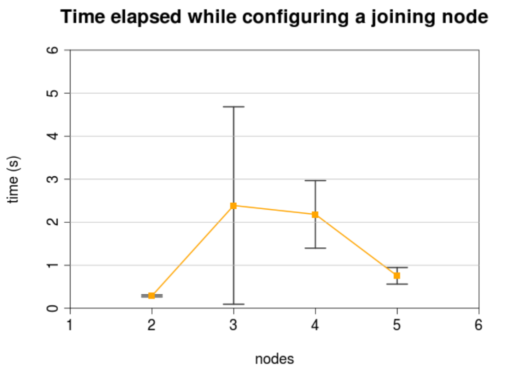


Figure 5.7: Time elapsed while configuring a joining node with confidence intervals with 80% confidence intervals

Figures 5.6 and 5.7 present the average time taken by a joining node to configure with every other node. Along with the obtained result, it presents the values from the same result set but only using the test runs where no packets other than the expected were used. These are called the “good” results.

The major conclusion that was pretended to accomplish was to observe if increasing the network size would directly affect the configuration times. The predictable answer is yes, since more nodes require more encryptions and decryptions using PKC algorithms. Each CPU core can handle one, therefore when the number of nodes of simultaneously configuring nodes exceeds the number of CPU cores, the configuration times must increase. Unfortunately, these results may not show that at all. The main reason for that is that every machine used in the test is different. The “good” results are once again inversely related with the CPU and memory speeds of the joining nodes.

Comparing the obtained results with the “good” results show the effect that packet losses and double configurations have in the configuration times. The 4 node scenario (joining of `Obsolete`) had 4 double configurations and some packet losses, therefore the excessive time is expectable. The 3 node scenario (joining of `Recent`) had some packet losses but that does not explain that excess, especially with the best “good” results. After looking at the logs, the explanation was easy to understand. In one of the runs, `Server` did not receive the advertisement broadcast sent by `Recent`. At the same time, `Recent` did not receive any broadcast from `Server`. The configuration routine only started at the second broadcast sent by `Recent`, 15 seconds after the first one. It is easy to conclude that packet losses can highly affect the configuration times of a joining node.

5.5 Data Transfer Performance Tests

One of the groups of tests that were performed had the purpose of analyzing the performance of the developed ABCDP protocol. To achieve that goal, data transfers between two machines within the wireless range of each other were performed using the ABCDP protocol. The same test was also conducted with other protocols that use encryption such as HTTPS and IPsec. To test the throughput of each protocol, files of different sizes were transfered multiple times between two machines. Three files were chosen for this task:

- A **small** file of approximately 10 KB, simulating a web page;
- A **medium** file of approximately 4.5 MB, simulating a music file;
- A **large** file of approximately 22.5 MB, simulating a video file.

This test uses the machines `Server` and `Client`, each one doing the job that is specified in their names. This test was meant to be executed between `Server` and `Recent` but the latter had problems with the program `setkey`, that will be focused later. The other laptop (`Obsolete`) does not have the same performance as the laptops used nowadays, even if they are not recent models. Because of that, another laptop (`Client`) was brought into play. Each file is obtained by `Client` using the program `curl`. The transfer using ABCDP was accomplished with the two machines running the framework, which makes `Server` act as a server. A file can be directly transfered if placed in the static content directory of a service. For the transfer using the other protocols, `Server` hosted an Apache web server was used.

The Apache server was configured in order to support TLS. For that, a self-signed certificate was generated and the Secure Socket Layer (SSL) engine was configured to use AES with 128 bit keys. For IPsec, the chosen method was `setkey`. This program uses rules to encrypt and decrypt packets transfered between specific hosts with specific keys using IPsec. AES keys with 128 bits were manually written in the configuration file. It was also defined in the configuration that every packet transfered between the two machines by the TCP port 8080 would be encrypted or decrypted using those keys. Finally, the Apache server was configured to listen in ports 80 (HTTP), 443 (HTTPS) and 8080 (HTTP over IPsec).

The program `curl` obtains the file either by HTTP or HTTPS. After the download is complete, it produces a customizable output. From this output, it is simple to produce one line per file transfer with meaningful values such as the transfer time. By obtaining the file from different ports, the different protocols were used, so it was simple to generate a bash script that automatized the file transfers.

5.5.1 Performance Between Two Laptops

The average transfer times and speeds are presented in figures 5.8 and 5.9 respectively. From these graphics it is possible to conclude several things. There does not seem to be much difference between HTTP and HTTPS. Strangely, in the large file transfers, the one with encryption is actually faster. Between the two established competitors that use encryption, IPsec is slightly slower than HTTPS.

It is also possible to conclude that TCP limits the transfer speed unnecessarily. The blue dotted line is the maximum throughput between the two machines according to `iperf`. The transfer speed using ABCDP is actually closer to the maximum throughput than it is from the transfer speed of its competitors.

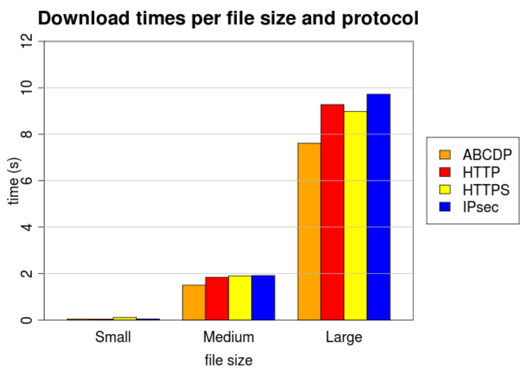


Figure 5.8: Average download time per file size and protocol

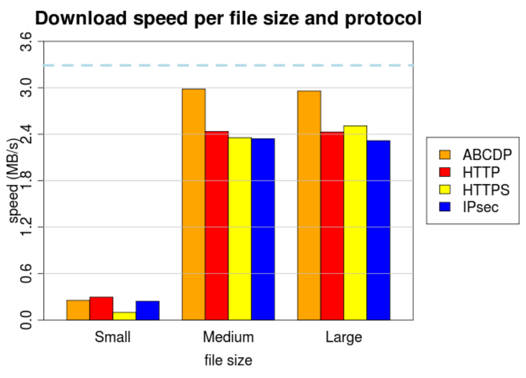


Figure 5.9: Average download speed per file size and protocol

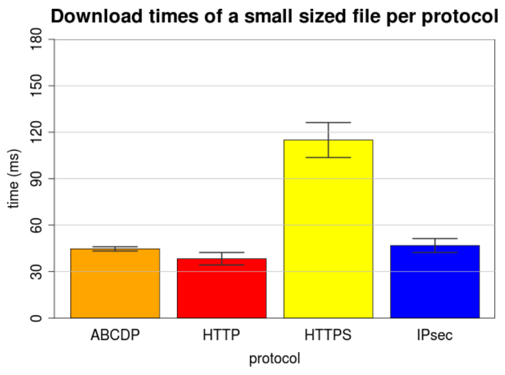


Figure 5.10: Download times of a small sized file per protocol with 95% confidence intervals

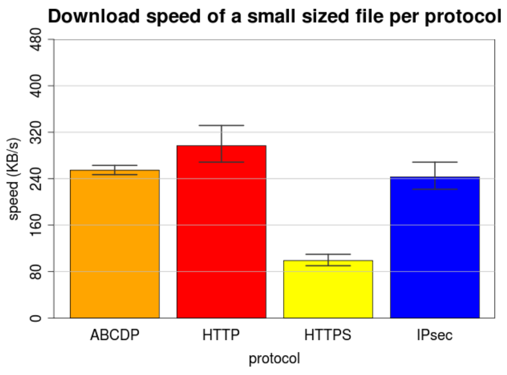


Figure 5.11: Download speed of a small sized file per protocol with 95% confidence intervals

In figures 5.10 and 5.11, the small file transfers are compared. In this scenario, the number of packets required for transferring the file is quite low, therefore the connection establishment overhead is very meaningful. The most distant of the four protocols is the HTTPS. That is caused by the massive number of packets required for the TLS handshake. The contents of some of these packets are encrypted using PKC that require a significant amount of time to be encrypted and decrypted.

The HTTP is without any surprise the fastest protocol, as the only handshake required is the TCP three-way handshake. In IPsec the situation is the same because the keys are already defined using `setkey`. The only overheads are the encryption of every individual packet and the reduction of the data that fits in each packet. Lastly, the ABCDP protocol has a result just slightly better than IPsec. The two connection packets that are encrypted using PKC slow down the connection but afterwards all the packets are sent at nearly the maximum throughput unlike the protocols build over TCP due to its slow start mechanism.

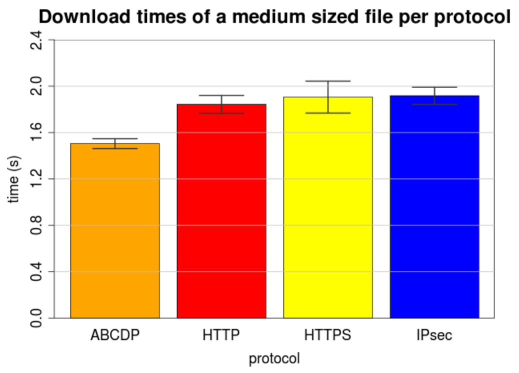


Figure 5.12: Download times of a medium sized file per protocol with 95% confidence intervals

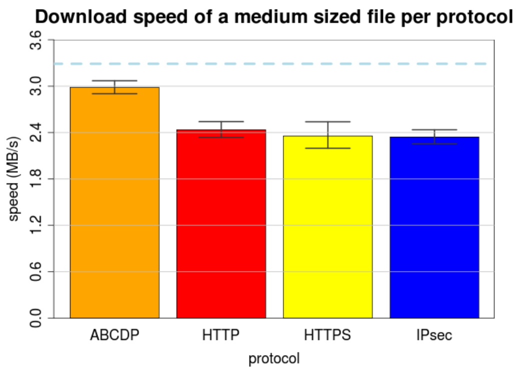


Figure 5.13: Download speed of a medium sized file per protocol with 95% confidence intervals

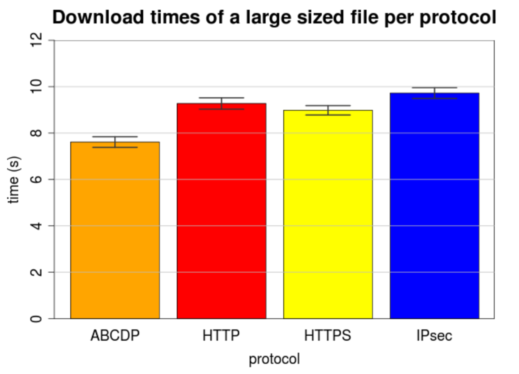


Figure 5.14: Download times of a large sized file per protocol with 95% confidence intervals

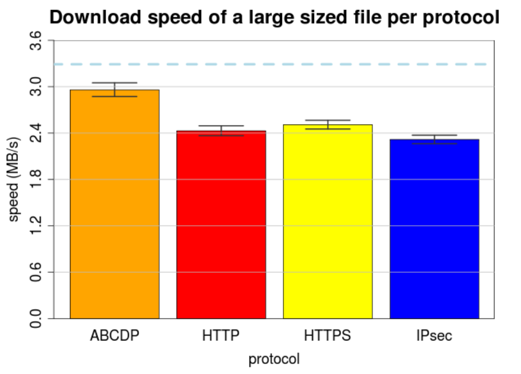


Figure 5.15: Download speed of a large sized file per protocol with 95% confidence intervals

Figures 5.12 and 5.13 compare the medium sized file transfers. Figures 5.14 and 5.15 compare the large sized file transfers. IPsec is the slowest alternative, due to the overhead caused by its large header. HTTP and HTTPS are slightly faster that IPsec. Both `curl` and `wget` showed that the download speed was higher with HTTPS than HTTP. In the medium file transfers, HTTP still beat HTTPS because the overhead caused by the TLS handshake was significant. Being HTTPS simply HTTP over an additional layer, at first sight these results do not much sense. It is possible that the implementation of the SSL engine from the Apache server results in more TCP friendly conditions in the `Server` machine.

The best result is achieved by the ABCDP protocol, because it is not build over TCP. The average value is close to the maximum throughput and it includes tests where packets were lost. The values obtained when no packets were loss are even closer to the maximum throughput (3.16 MB/s versus 3.29 MB/s). ABCDP sends packets at the maximum rate possible for most of the time. Current laptop processors have more than enough power to encrypt a message, send it, receive one encrypted message and decrypt it much faster than the rate at which a 54 Mbps wireless network interface can send two consecutive packets. Under this circumstances, it seems difficult that a network interface related buffer would be filled and cause packet losses. Therefore it seems possible to send packets with a less strict flow control.

Unfortunately, not all processors are that fast and even the fastest processors have to schedule various processes. A process that takes care of buffer copying inside the kernel may sometimes not be scheduled in time, which can lead to filled buffers and consequently packet losses. Having a flow control is always a good safeguard. If the flow control uses a window with a decent size, the packet throughput is mostly at the maximum rate. The difference between the obtained maximum results and the maximum throughput is caused by the packet headers and the fixed cipher block sizes that limit the amount of data that can be sent in a packet.

5.5.2 Performance Between a Laptop and an Android Device

To test how the protocol performs in Android the tests were also performed between the machines `Android` and `Server`. The programs and configuration used by `Server` were the same but `Android` required other different tools. It is not easy to get binary tools for an Android device. It does not only use an ARM architecture which requires specific executables, but it also lacks most libraries used by the required tools. Therefore, the dynamic link compilations for the ARM architecture do not work on Android. Static compilations are required and those are hard to find. Because of that, the tests were performed using only the tools provided with BusyBox.

BusyBox provided a version of `wget` that did not support HTTPS. For this reason, the only protocols that were tested were ABCDP and HTTP. This version of `wget` is also not very verbose, therefore there is no way of obtaining any reliable data from its output. Therefore, `wget` was used in conjunction with `time`, that outputs the amount of time that a program took to complete. This value is considered as the transfer time of the file requested using `wget`. BusyBox also provides the `iperf` program to obtain the maximum throughput conditions. The obtained results are presented in figures 5.16 and 5.17.

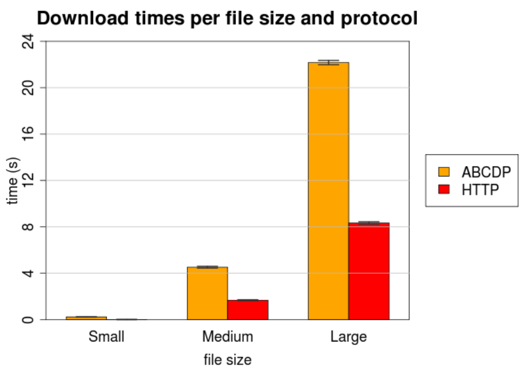


Figure 5.16: Average download time per file size and protocol using an Android device with 95% confidence intervals

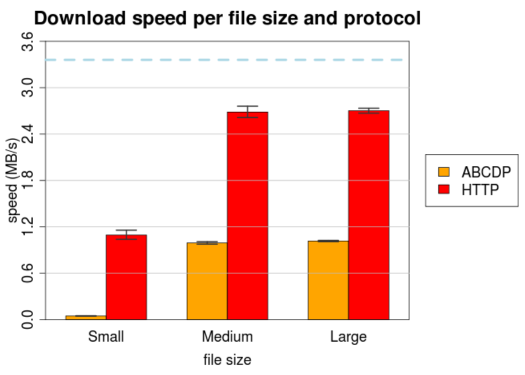


Figure 5.17: Average download speed per file size and protocol using an Android device with 95% confidence intervals

The results are very different from the ones obtained in the other scenario. Although HTTP maintained its performance, this time ABCDP had by far the worst performance. After some investigation, it was possible to conclude that the cause was the CPU usage. During file transfers, the CPU consumption of the framework process was near 90%. After the cause has been diagnosed, the protocol was optimized (the presented results are from the final version). The number of buffer copies was reduced and many objects were reused to reduce memory consumption and garbage collection effort. The performance increased but only to the point presented in the results.

Java is perhaps not the best choice of programming language to implement a transport layer protocol. These protocols should be implemented using a language that is compiled to native binaries such as `C`. At least this allowed to confirm what was referred above about the flow control. Without flow control, `Server` would have overwhelmed `Android` with packets that would obviously be lost. This would have to be detected as an error by `Android` using the error control routines which would slow the protocol down dramatically.

To check how the performance fares in laptops that are not as recent, the same tests were also done between `Obsolete` and `Server`. Using exactly the same programs and configurations as the ones used between `Client` and `Server` the results using HTTP, HTTPS and IPsec were very similar. The results using ABCDP were much better that using the Android device, but they were still the worst. The average speed was about 1.9 MB/s and CPU consumption was near 40%. The CPU consumption using any of the other three laptops was irrelevant, although still slightly higher than when using the other tested protocols.

5.6 Scalability Related Tests

Another group of tests that were performed aimed for the scalability of the framework and of the ABCDP protocol. For these tests, the CORE emulator [2] was used. This emulator is able to emulate complete networks in one single machine. It is able to emulate many types of networks, including MANETs. It creates one virtual machine for each emulated node. These virtual machines only virtualize the network stack, making it possible to emulate many nodes. This emulator is able to run real code such as the one developed, making it a good option for the scalability related tests.

In total, three scalability tests were performed using the CORE emulator. The first two of them are functionality tests. One of them was checking how the framework handles the merge of partitions. The other one consisted in constructing a somewhat large network with multiple hops and allow all nodes to configure each other. The last one was a performance test, with the purpose of seeing how the protocol fares when transferring data over multiple hops.

For the first test, two groups of connected nodes were created and both were allowed to completely configure. After that, one of the nodes was moved in order to stay in range of both partitions. The result was positive, the periodic broadcasts were propagated through all the nodes and all configurations were performed. The configurations were obviously not immediate due to the time between the periodic broadcasts, but still, the most important is that the configurations have been performed. Figures 5.18 and 5.19 show the topology of the network used for this test before and after the partitions merged.

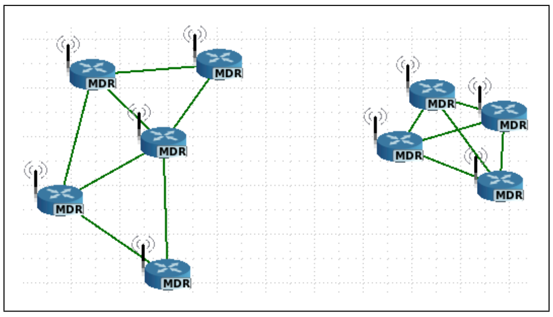


Figure 5.18: Topology of the emulated network before joining both partitions

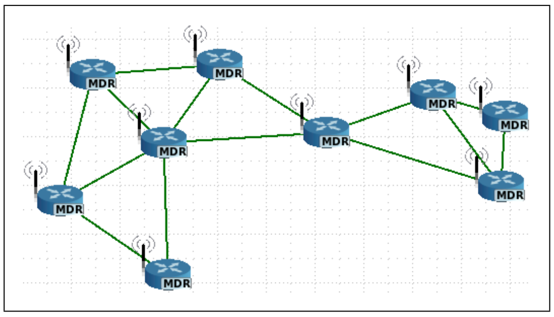


Figure 5.19: Topology of the emulated network after joining both partitions

For the second test, a group of 40 nodes was created, with a maximum distance between them of 9 hops. The center of the network was relatively dense and the outer sections were more sparse. After the network has been created, the framework was initialized in every node, one at a time. Due to the amount of connection packets (that require encryptions and decryptions using PKC) all running in just one machine with 40 JVMs running, some packets were not delivered in time which did not allow every configuration to be executed at first. Some configurations were only performed at the second wave of advertisement broadcasts, but every node was able to configure every other node. The topology of this 40 node network is presented in figure 5.20.

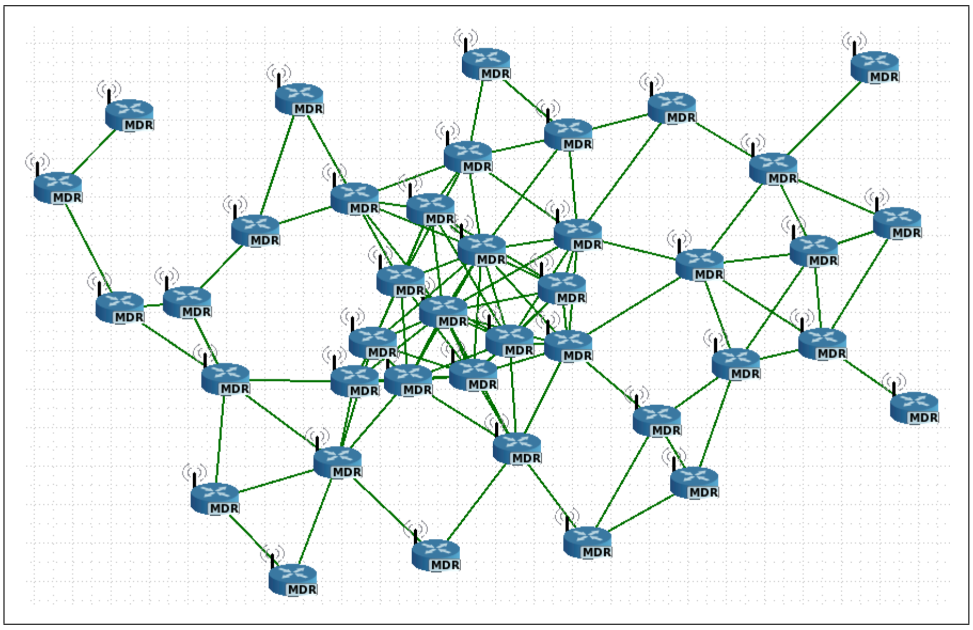


Figure 5.20: Topology of the emulated 40 node network

The last test used the same 40 node network to measure the performance of the ABCDP protocol in multi-hop data transfers. These transfers were performed multiple times between two random nodes. This time, only the medium sized file with nearly 4.5 MB was transfered. The program chosen to obtain the file was `curl`. The ABCDP tests can be performed with every node running its instance of the framework. To test other protocols further configuration is required.

Configuring IPsec and HTTPS in a way that it would be used in every node would be time consuming. Also, the performance results previously analyzed show that the difference between the three established alternatives is not very big. What is meant to compare in this scenario is how the download times and speeds are affected as the number of hops increases. As all three protocols are built over TCP, there should be no significant difference between them. Therefore, only HTTP was compared with ABCDP.

To easily run 40 HTTP servers, one per node, the program `micro httpd` was chosen. The first tests resulted in values that were very different from the ones obtained in the real scenario. The problem was that the default configuration of the emulated scenario was using the complete 54 Mbps bandwidth (this value is only the theoretical maximum of the interface) and a delay that was excessively high. Both those values were modified to a point where the maximum transfer rate obtained with `iperf` and the response time obtained with `ping` were similar to the ones from the real scenario.

The analysis that detected those problems also detected that the delay between each packet was practically constant and that there were no packet losses whatsoever. It was possible to force a packet error rate but the minimum that was allowed was 1%, which largely exceeds the normal rate. About the packet delay it was possible to define the average value (modified above) but not its variance. The obtained results are presented in figures 5.21 and 5.22. They can be analyzed independently in figures 5.23, 5.24, 5.25 and 5.26.

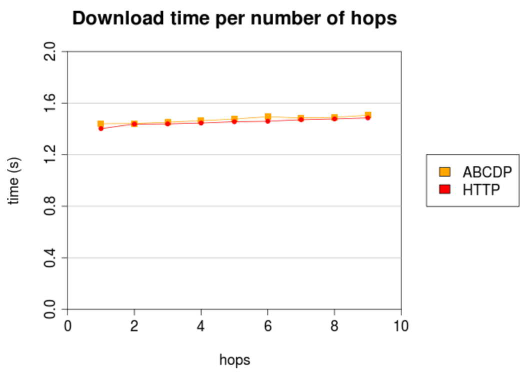


Figure 5.21: Download time per number of hops

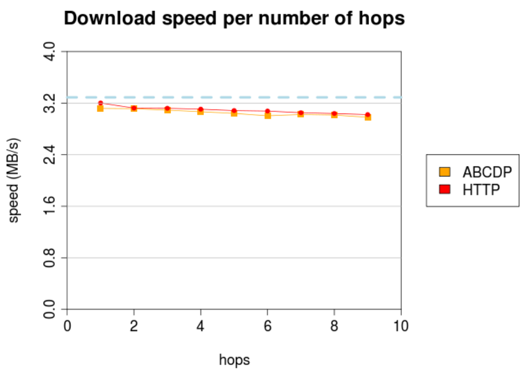


Figure 5.22: Download speed per number of hops

The performance decrement is linear for both protocols. The decrease rate is also similar for both. But the increase of the HTTP performance was unexpected. The values for ABCDP are quite similar to the ones of the real scenario. But the values for HTTP are even higher than the ones for ABCDP. This is possibly caused by some behavior of the emulator that does not occur in a real wireless network, such as the delay between packets being nearly constant. TCP must be in perfect conditions since the value for one hop is virtually the maximum throughput, represented by the blue dotted line.

When the values for the delay were too high, the ABCDP protocol was negatively affected. Although, the protocol tries to use the maximum bandwidth, it is still completely synchronous. This means that each send operation blocks until the whole message has been successfully delivered. There is always a time period per message where the flow window is not fully used. This can be reduced by sending big messages. The messages for static content in the framework have 1 MB of size, which is already a high value. Still, when sending a 4.5 MB file there are still 4 periods between messages where the transfer rate is reduced. Still, with the packet delays of the real scenario, this performance loss is irrelevant.

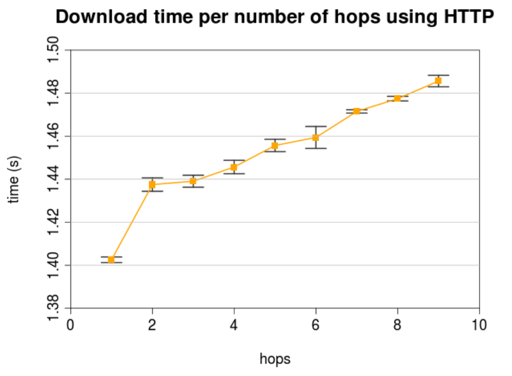


Figure 5.23: Download time per number of hops using HTTP with 99% confidence intervals

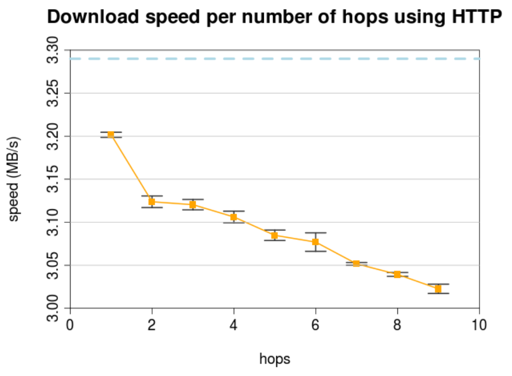


Figure 5.24: Download speed per number of hops using HTTP with 99% confidence intervals

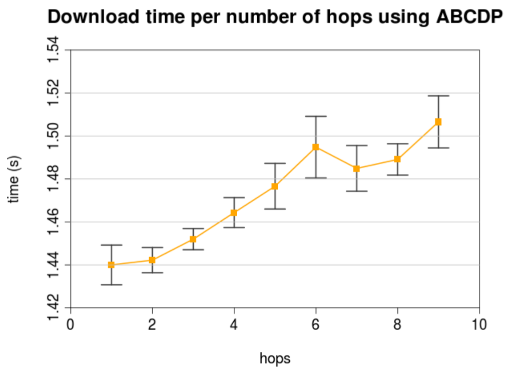


Figure 5.25: Download time per number of hops using ABCDP with 99% confidence intervals

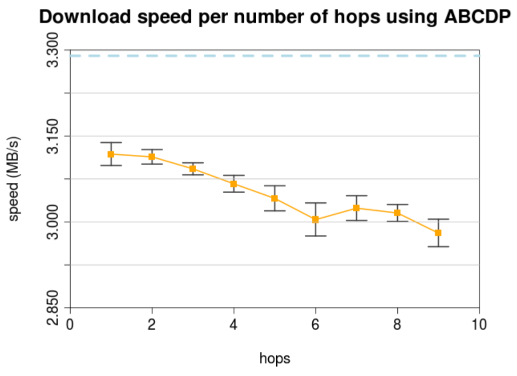


Figure 5.26: Download speed per number of hops using ABCDP with 99% confidence intervals

Tests were also performed with the minimum packet error rate of 1%. The results were also not very bright as ABCDP waits a specific time interval until it detects that a packet has been lost. TCP adapts this time and therefore behaves much better with high error rates. But it is important to notice both in this test and the one before (with increased delay), that TCP was under near perfect conditions in this emulated scenario, which does not seem realistic. Additionally, ABCDP was not designed to work in scenarios with such high packet delays and such high packet error rates. In a realistic wireless environment, even with poor signal conditions caused by physical obstacles and electromagnetic radiation, the changes to these values should be that relevant.

5.7 Overview of the Results

These previous sections described the testing process, presented the results obtained in that process and analyzed those same results. It was shown that every component of the framework functioned correctly and that all of them were well integrated. It has also been shown that the developed framework works both in Linux and Android. After that, the tests aimed for the performance and scalability of the framework.

In the real scenario, every node was able to configure every other node, usually in less than a second. It was also shown that the protocol is capable of achieving a high throughput, near the maximum throughput achievable by the wireless network interfaces. Unfortunately, this performance is only possible if the CPU of the machine allows it. The protocol could probably be more optimized and should have been developed in a language that is compilable to native binaries.

In the emulated 40 node scenario, every node was once again able to configure every other node. It was also shown that the protocol handles multi-hop packet transfers satisfactorily. Unfortunately, the other results were not as good. Still, in the emulated scenario, TCP was under near perfect conditions and some parameters of the scenario could only be set to unrealistic values. The tests performed in the real scenario are the most important ones and in that scenario, the ABCDP protocol has proved itself fit and a good alternative when compared with other alternatives built over TCP.

Chapter 6

Conclusions

In this chapter, the conclusion of this work is performed. First, a recollection of the performed work is made and some obtained conclusions are presented. After that, a critical analysis about some aspects of the performed work and the way it was performed is presented. Finally, the work that should be performed to further complete this work is enumerated.

6.1 Conclusions and Final Remarks

This work focused on ways to increase the usability of ad-hoc networks. For that, a framework based on three components was designed and implemented. One of these components is a set of auto-configuration mechanisms suitable for ad-hoc networks. Other is a transport layer protocol that allows the transfer of authenticated messages. The last one is an easy to use service framework that is also adapted to an ad-hoc environment.

Many existing protocols are not well adapted to a local wireless environment. The protocol that seamed to fit this environment was HTTPS but unfortunately it is designed for the Internet where trusted third parties can authenticate each entity. This is not possible in spontaneous local networks where every node has the same role and therefore there are no trusted nodes. HTTPS clients immediately render connections with nodes that are not authenticated by a trusted third party as insecure, flashing big warnings that are not intended in this environment.

Another alternative would be using IPsec. Generating Security Associations (SAs) using a newly developed distributed process would allow authentication and encryption of every packet, without requiring a trusted third party. Unfortunately, in terms of IPsec support for Android, there is only official support for Virtual Private Networks (VPNs) using servers. These use a client-server model, which is not pretended in this environment.

This motivated the development of a new protocol. This protocol was a transport layer protocol extending UDP and was designed specifically for this environment. Building it over UDP made it possible to increase the throughput to nearly the maximum allowed by the wireless network interfaces. A simplified handshake with two packets, compressing both the connection and the authentication handshakes reduced the time taken by the connection process. The use of Public-key Cryptography (PKC) made it possible to develop a mutual authentication mechanism without requiring any third parties.

Another of the tackled problems was the development of suitable auto-configuration routines. These routines have two main problems in Mobile Ad-hoc Network (MANET) environments. The first is the lack of protection against Denial of Service (DoS) attacks and the other is the possibility of network partitioning. As IP addresses must be unique, there must be an agreement about which addresses to use. These agreements can be easily disrupted by many ways. But PKC can be used to secure these routines. Using Cryptographically Generated Addresses (CGA), it is possible to attribute IP addresses to each node and avoid any agreement. The address of a node is calculated from its public key, using a deterministic function.

By making each node broadcast their public keys and name, every node can know the public key, IP address and name of every other node. The received public keys can be used to ensure that the node with the IP address generated from the public key actually owns the public key by being able to understand messages encrypted by that key. With each node broadcasting periodically, the problem caused by partitions is minored. When two partitions merge, nodes from the two partitions start to gradually know about the existence of each other.

Together, the developed protocol and the auto-configuration routines are secure against many types of network attacks. It is assumed that there are no vulnerabilities in the used cryptographic algorithms and that discovering the private key associated with a public key is impossible in near real time. Assuming this, it is not possible to disrupt the processes of IP attribution and name choosing, impersonate an entity, read packets sent to another entity, hijack a connection or terminate a connection established between two other entities.

A service framework for developing, deploy and host services adapted to MANET environments was also developed. This framework had three main purposes: being easy to use, allowing the easy development of services adapted to the target environment and use web based service interfaces. With that in mind, the framework was based in a Model-view-controller (MVC) model where the development of each of these three types of components is highly simplified.

Deployment, distribution and installation of services was part of the main focus. A simple way to distribute and install a service is to package it into a single archive. Installing a service is done by simply placing the service archive in a specific directory that contains the hosted services. To facilitate deployment, methods for automatic packaging are provided. Additionally, providing a template service NetBeans project, with example code and the packaging scripts, allows a developer to quickly start producing his own services in an IDE.

Unfortunately, due to the available time, only the basis of the service framework along with some essential features were developed. The service framework also contains a default index service, that allows the listing of online users and their hosted services, with hyperlinks for each of them. This service provides an user interface for the features of user and service discovery, along with the possibility of accessing any service without writing any URL.

The whole designed framework is easily portable to both a Linux system or Android. No major alterations are required in the Android device, such as new ROM installations or kernel recompilations. The framework can be installed exactly the same way as any other Android app and can be initialized using only that app. Additionally, with virtually no alterations, the framework could be used in other Linux based mobile operating systems such as MeeGo and WebOS. With some minor changes, it could be used in Windows, Mac OS, other BSD systems and virtually any operating system that is able to run a Java Virtual Machine (JVM).

The ability of working without any need for additional infrastructure and the possibility of creating one at anytime makes ad-hoc networks suitable for many scenarios, such as small workgroups and starting companies. Improving support for ad-hoc networking would be the first step to increase their usage. As their usage increases the interest on developing applications for them also increases. The presented framework is a possible approach to increase the number of applications and the user friendliness of ad-hoc networks. Any of the three individual components alone does not result in an easy to use product, but the three combined result in a solid framework that is both powerful and easy to use.

6.2 Personal Opinion and Critics

During the progress of this work, not everything was rosy. Performing tests in a real scenario is very complicated. It is necessary to find voluntaries willing to lend their machines, which is not always easy. It is also hard to collect results in the same network conditions when conditions vary so much. The number of access points in almost any urban location is high, sometimes scattered through different wireless channels. Some other times the limitations are caused by the network interface. Sometimes, by simply disconnecting and connecting it once again, it is possible to obtain much better results with any of the tested protocols.

The occurrence of unexpected situations can also be very complicated. For example, the most recent and powerful machine that was used for testing was unable to produce results for the performance tests because the IPsec suite crashed the whole operating system. Due to this, another laptop that was not very old had to be obtained, to generate performance results with using two machines with hardware resources similar to the currently used machines.

Although performing tests in a real scenario is much more complicated than in an emulated scenario, it is obvious that the results obtained in the real scenario are much more important. Emulators do not portray exactly every aspect of a real wireless network scenario. In the performed tests, the emulator obtained a much higher throughput using TCP than in the real scenario. In every performed transfer using TCP throughout the whole development of the framework, including file transfers between machines without the purpose of testing, throughput using TCP was never even close to the one obtained in the emulator.

In an attempt to approximate the emulated results to the real results, the scenario was manually parametrized. The values of bandwidth and packet delay were changed to as near as possible as the ones obtained in the real scenario. Unfortunately, without being able to parametrize the jitter, which was near 0 and with the minimum packet error rate being 1%, which is very high, it was not possible to obtain completely realistic values. The results of the developed protocol were near the real ones, but the results for TCP were better than the real ones.

Although the final result of the implementation is a framework usable in both Linux and Android, the development for both was not easy. In this case, the code was developed to work in Linux and later was adapted for Android. Most of the code can be used by the two versions but some of it is not. Some details of the inner workings of Android can cause a big headache if one is not counting with them. The impossibility of running any sort of Java code as the root user is one of them.

Android is also extremely picky with wireless networks and IP addresses attributed by processes that not the default one. It is common to have the Android device ignoring broadcasts and ignoring packets sent to the added IP address. The latter problem usually stops when the Android device sends a packet using the added IP address as the source address, finding out at that moment that it actually has that address. These problems are probably caused by some caching, used to optimize this devices, which is understandable since they have low hardware resources. These problems are usually not serious in a normal utilization of the framework but during testing they are able to cause differences in the obtained values.

6.3 Future Work

There is much that can be done to improve this work. First of all, there are many features that were designed and referred in the document but that unfortunately were not implemented due to the available time. Most of these features belong to the service framework and are not essential as a proof of concept. Still, developing them would result in a more complete framework. The first task of the future work is implementing these features. The features in question are:

- Table relations in the automated ORM framework;
- Helper methods for table and form generation;
- Groups and model ownership;
- The remote invocation engine;
- The key renewal algorithm;
- A way of selecting one user from a set of users with the same name.

After the implementation of the missing features, a complex service could be developed. A useful service for a local network would be a service similar to Dropbox[1]

[1]`https://www.dropbox.com/`

but completely distributed. This service could take advantage of groups, for the shared folders and of the remote invocation engine, for the routines that compare and synchronize the shared folders.

The minor changes to the source code required for the framework to run in many other operating systems can also be implemented. The native libraries must be compiled for the different platforms, such as Windows, Mac OS and BSD systems (maybe the last two can use the same compilation). In the case of Windows, the native code must be implemented using the specific Windows kernel and network related libraries, that are different from the ones used by UNIX systems. For other mobile operating systems, such as MeeGo and WebOS, trial and error routines would discover the specific differences of the operating that would require changes to the code. It would probably be easy, because when compared with Android, any of those systems is much closer to a regular Linux distribution.

The support for Android can also be improved. The inner workings of the operating system can be analyzed to minimize the problems explained in section 6.2. Other than that, the user-agent field of the HTTP requests can be used distinguish if the client is a mobile device or a computer. This can be used to differentiate the HTML markup that is returned to the client. Although mobile browsers are more than capable of showing layouts designed for the size of computer screens, layouts can be optimized for the small screens of smartphones.

Another task that can also be performed is more elaborate testing. Other emulators or additions to the used emulator can be used to more accurately emulate a wireless environment. Additionally, since wireless network conditions vary so often, tests in the real scenario should be done with more repetitions. The tests should also try to maximize the range of different conditions.

To finalize, another task that can be done is a more profound study of the areas that were not part of the main focus of this work. These areas include existing routing protocols and transport layer protocols, mainly their error, flow and congestion control routines. In the latter it would be possible to analyze a cross-layer solution with the datalink layer, to gather data about the quality of the signal, with the purpose of, for example, predicting packet losses.

Appendix A

Index Service Source Code

A.1 Index Service

A.1.1 Index Controller

```java
package cfs.index.index;

import java.util.ArrayList;
import java.util.List;
import org.concept.framework.helper.UnorderedList;
import org.concept.framework.http.HTTPRequest;
import org.concept.framework.http.HTTPResponse;
import org.concept.framework.layout.DefaultLayout;
import org.concept.framework.mvc.Controller;
import org.concept.framework.mvc.Layout;
import org.concept.framework.service.Service;
import org.concept.framework.service.ServiceManager;
import org.concept.user.User;
import org.concept.user.UserManager;

import static org.concept.framework.helper.HTMLHelper.*;

public class IndexController extends Controller {

    @Override
    public Class<? extends Layout> layoutClass() {
        return DefaultLayout.class;
    }

    public void index (HTTPRequest request, HTTPResponse response) {

        boolean root = request.isRequestingUserRoot();

        response.appendToComponentInView("title",
                                         root ? "Concept Home" :
                                                "Home of user: " + UserManager.getLocalName());

        response.appendToComponentInView("header",
                                         h1(root ? "Concept Home" :
                                                "Home of user: " + UserManager.getLocalName()));

        List<String> navigation = new ArrayList<String> ();
        navigation.add( a("Home", "/") );
        if (root)
            navigation.add( a("Settings", "/index/settings") );

        response.appendToComponentInView("navigation", "" + new UnorderedList(navigation));

        List<String> services = new ArrayList<String> ();
        for (Service service : ServiceManager.listServices())
            services.add(xhtml(a(service.getName(), "/" + service.getId() + service.getMainPage()),
                                p(service.getDescription())));

        response.appendToComponentInView("content",
                div(null, "block",
                    h2("Running Services"),
                    new UnorderedList(services).toString()
        ));

        if (root) {
            List<String> users = new ArrayList<String> ();
            for (User user : UserManager.listOnlineUsers())
                users.add(a(user.getName(), "http://" + user.getName() + "/"));

            response.appendToComponentInView("sidebar",
                    div(null, "block",
                        h3("Online Users"),
                        new UnorderedList(users).toString()
                    )
            );
        }

        response.appendToComponentInView("footer", p("2011 - Concept Framework"));
    }

}
```

A.1.2 Settings Controller

```
package cfs.index.settings;

import java.util.ArrayList;
import java.util.List;
import org.concept.framework.helper.TagAttributes;
import org.concept.framework.helper.UnorderedList;
import org.concept.framework.http.HTTPRequest;
import org.concept.framework.http.HTTPResponse;
import org.concept.framework.layout.DefaultLayout;
import org.concept.framework.mvc.Controller;
import org.concept.framework.mvc.Layout;
import org.concept.user.configuration.ConfigurationServer;

import static org.concept.framework.helper.HTMLHelper.*;
```

```java

public class SettingsController extends Controller {

    @Override
    public Class<? extends Layout> layoutClass() {
        return DefaultLayout.class;
    }

    private boolean checkAccess (HTTPRequest request, HTTPResponse response) {
        boolean ok = request.isRequestingUserRoot();
        if (!ok) {
            response.setStatus(HTTPResponse.MOVED_PERMANENTLY);
            response.addHeaderField("Location", "/index/index/index");
        }
        return ok;
    }

    public void index (HTTPRequest request, HTTPResponse response) {

        if (!checkAccess(request, response))
            return;

        response.appendToComponentInView("title", "Concept Settings");

        response.appendToComponentInView("header", h1("Concept Settings"));

        List<String> navigation = new ArrayList<String> ();
        navigation.add( a("Home", "/") );
        if (request.isRequestingUserRoot())
            navigation.add( a("Settings", "/index/settings") );

        response.appendToComponentInView("navigation", "" + new UnorderedList(navigation));

        response.appendToComponentInView("content",
                div(null, "block",
                    h2("Change Name"),
                    form("POST", "/index/settings/updatename",
                        ul(
                            li(TagAttributes.create("class", "label"), label("New user name:")),
                            li(inputText("name", ""))
                        ),
                        ul(TagAttributes.create("class", "buttons"),
                            li(inputSubmit("", "Submit"))
                        )
                    )
                )
        );

        response.appendToComponentInView("footer", p("2011 - Concept Framework"));
    }

    public void updatename (HTTPRequest request, HTTPResponse response) {

        if (!checkAccess(request, response))
            return;

        ConfigurationServer.changeName(request.getRequestParameter("name"));

        response.setStatus(HTTPResponse.MOVED_PERMANENTLY);
        response.addHeaderField("Location", "/index/settings/index");
    }

}
```

A.2 Default Layout

A.2.1 Default Layout Markup

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
     strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><%@ title %></title>
        <link rel="shortcut icon" href="<%= DirectoryHelper.publicWebResourcePath("favicon.ico") %>"
             />
        <link rel="stylesheet" type="text/css" href="<%= DirectoryHelper.publicWebResourcePath("
             themes/default/structure.css") %>" />
        <link rel="stylesheet" type="text/css" href="<%= DirectoryHelper.publicWebResourcePath("
             themes/default/blue.css") %>" />
    </head>
    <body>
        <div id="page-container">
            <div id="header">
                <%@ header %>
            </div>
            <div id="navigation">
                <%@ navigation %>
            </div>
            <div id="wrapper">
                <div id="sidebar">
                    <div class="padding">
                        <%@ sidebar %>
                    </div>
                </div>
                <div id="content">
                    <div class="padding">
                        <%@ content %>
                    </div>
                </div>
            </div>
            <div id="footer">
                <%@ footer %>
            </div>
        </div>
    </body>
</html>
```

A.2.2 Compiled Default Layout

```
package org.concept.framework.layout;

import org.concept.framework.http.HTTPRequest;
import org.concept.framework.mvc.Layout;
import org.concept.framework.mvc.View;
import org.concept.util.DirectoryHelper;

```

```java
public class DefaultLayout extends Layout {

    @Override
    public View generateInitialView (HTTPRequest request) {
        View view = new View();
view.addFixedComponent("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\" \"http://www.w3.
     org/TR/xhtml1/DTD/xhtml1-strict.dtd\">\r\n" +
"<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"en\" lang=\"en\">\r\n" +
"    <head>\r\n" +
"        <meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\" />\r\n" +
"        <title>\r\n");
view.addNamedComponent("title");
view.addFixedComponent("</title>\r\n" +
"        <link rel=\"shortcut icon\" href=\"\r\n");
view.addFixedComponent(DirectoryHelper.publicWebResourcePath("favicon.ico"));
view.addFixedComponent("\" />\r\n" +
"        <link rel=\"stylesheet\" type=\"text/css\" href=\"\r\n");
view.addFixedComponent(DirectoryHelper.publicWebResourcePath("themes/default/structure.css"));
view.addFixedComponent("\" />\r\n" +
"        <link rel=\"stylesheet\" type=\"text/css\" href=\"\r\n");
view.addFixedComponent(DirectoryHelper.publicWebResourcePath("themes/default/blue.css"));
view.addFixedComponent("\" />\r\n" +
"    </head>\r\n" +
"    <body>\r\n" +
"        <div id=\"page-container\">\r\n" +
"            <div id=\"header\">\r\n" +
"                \r\n");
view.addNamedComponent("header");
view.addFixedComponent("\r\n" +
"            </div>\r\n" +
"            <div id=\"navigation\">\r\n" +
"                \r\n");
view.addNamedComponent("navigation");
view.addFixedComponent("\r\n" +
"            </div>\r\n" +
"            <div id=\"wrapper\">\r\n" +
"                <div id=\"sidebar\">\r\n" +
"                    <div class=\"padding\">\r\n" +
"                        \r\n");
view.addNamedComponent("sidebar");
view.addFixedComponent("\r\n" +
"                    </div>\r\n" +
"                </div>\r\n" +
"                <div id=\"content\">\r\n" +
"                    <div class=\"padding\">\r\n" +
"                        \r\n");
view.addNamedComponent("content");
view.addFixedComponent("\r\n" +
"                    </div>\r\n" +
"                </div>\r\n" +
"            </div>\r\n" +
"            <div id=\"footer\">\r\n" +
"                \r\n");
view.addNamedComponent("footer");
view.addFixedComponent("\r\n" +
"            </div>\r\n" +
"        </div>\r\n" +
"    </body>\r\n" +
"</html>\r\n");
        return view;
    }
}
```

A.2.3 Structure Stylesheet

```

/* RESET */

html, body, div, span, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
abbr, address, cite, code,
del, dfn, em, img, ins, kbd, q, samp,
small, strong, sub, sup, var,
b, i,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, dialog, figure, footer, header,
hgroup, menu, nav, section,
time, mark, audio, video { margin: 0; padding: 0; border: 0; outline: 0; font-size: 100%;
               vertical-align: baseline; background: transparent; }



/* TYPOGRAPHY */

body            { font-family: Arial, Helvetica, Verdana, sans-serif; font-size: 0.84em; }
h1          { padding-bottom: 0.5em; color: #fff; font-size: 2em; }
h2          { padding-bottom: 0.5em; font-size: 1.4em; }
p           { padding: 0.25em 0; }



/* STRUCTURE */

#header         { min-height: 5em; padding: 1em 2em; }

#navigation     { height: 2em; padding: 0 1em; }
#navigation ul li   { list-style: none; float: left; }
#navigation ul li a { padding: 0.4em 1em; display: block; text-decoration: none; color: #000; }

#wrapper        { min-height: 20em; }

#content        { margin-right: 25%; }
#content .padding   { padding: 1em 2em; }
#content .block     { margin: 1em -1em; padding: 1em; }

#sidebar        { float: right; width: 25%; }
#sidebar .padding   { padding: 1em 2em; }
#sidebar .block     { margin: 1em -1em; padding: 1em; }
```

```css
#sidebar .block h3  { margin: -1em; padding: 0.5em 1em; margin-bottom: 0.5em; }

#footer         { clear: both; min-height: 1em; padding: 0.5em 0; text-align: center; }



/* STRUCTURE (CSS3) */

#content .block     { border-radius: 1em; box-shadow: 0.15em 0.15em 0.1em #888; }

#sidebar .block     { border-radius: 1em; box-shadow: 0.15em 0.15em 0.1em #888; }
#sidebar .block h3  { border-top-right-radius: 1em; }



/* LISTS */

#content li     { margin-left: 2em; padding: 0.05em 0; }
#content ul     { padding: 0.25em 0; }
#content ol     { padding: 0.25em 0; }
#content dl     { padding: 0.25em 0; }

#sidebar li     { margin-left: 2em; padding: 0.05em 0; }
#content ul     { padding: 0.25em 0; }
#content ol     { padding: 0.25em 0; }
#content dl     { padding: 0.25em 0; }



/* TABLES */

#content table      { width: 100%; margin: 1em 0; border-collapse: collapse; }
#content td     { padding: 0.2em 0.5em; }
#content th     { padding: 0.3em 0.5em; text-align: left; }



/* FORMS */

#content form           { margin: 1em 0; margin-right: 0.5em; }
#content form ul li     { margin-left: 30%; list-style: none; }
#content form ul li.label   { float: left; width: 25%; margin-left: 0; }

#content form ul.buttons    { margin-bottom: 2.5em; }
#content form ul.buttons li { float: left; margin-left: 0; padding-right: 1.5em; }

#content form ul li input[type=text],
#content form ul li input[type=password],
#content form ul li textarea            { width: 100%; }
#content form ul li select          { width: 40%; }



/* FORMS (TYPOGRAPHY) */

#content form ul li.legend  { font-style: italic; color: #777; }
```

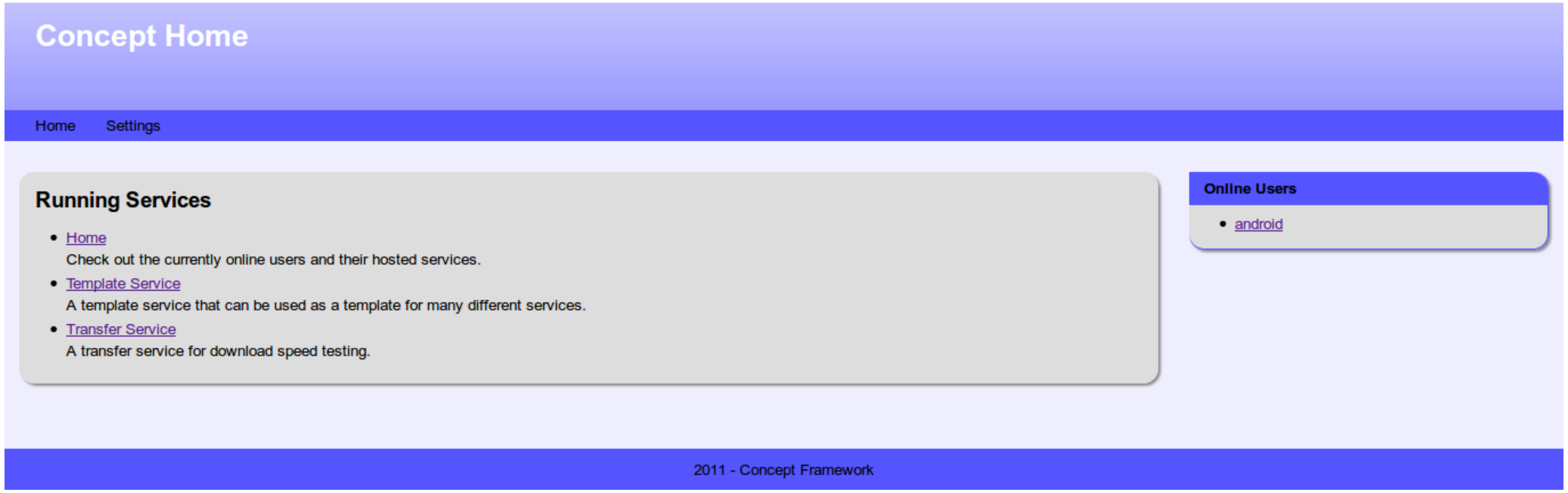
```
#content form ul li.error   { color: #f20; }
```

A.2.4 Color Stylesheet

```

/* COLORS */

#page-container     { background-color: #eef; }

#header         { background-color: #99f; }

#navigation     { background-color: #55f; }
#navigation ul li:hover { background-color: #aaf; }

#content .block     { background-color: #ddd; }

#sidebar .block     { background-color: #ddd; border-bottom: #55f solid 1px; border-right: #55f solid 1px; }
#sidebar .block h3  { background-color: #55f; }

#footer         { background-color: #55f; }



/* TABLE COLORS */

#content th     { background-color: #55f; }
#content td     { background-color: #bbb; }
#content tr.even td { background-color: #ccc; }



/* COLORS (CSS3) */

#header         { background-image: -webkit-gradient(
                        linear,
                        left bottom,
                        left top,
                        color-stop(0, rgb(153,153,255)),
                        color-stop(1, rgb(195,195,255))
              );
              background-image: -moz-linear-gradient(
                        center bottom,
                        rgb(153,153,255) 0%,
                        rgb(195,195,255) 100%
              );
}
```

A.3 Screenshots

A.3.1 Index Service Viewed From a Laptop



Running Services

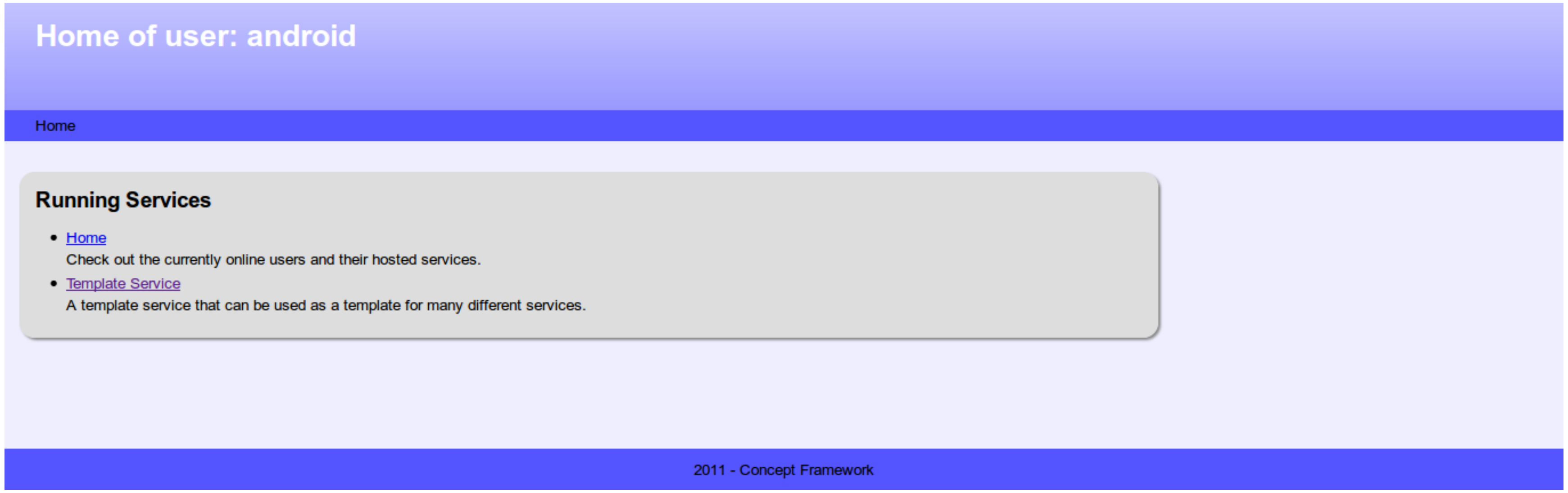
- Home
  Check out the currently online users and their hosted services.
- Template Service
  A template service that can be used as a template for many different services.
- Transfer Service
  A transfer service for download speed testing.

Online Users

- android

2011 - Concept Framework

Figure A.1: View of the local index service in a laptop


Home of user: android

Home

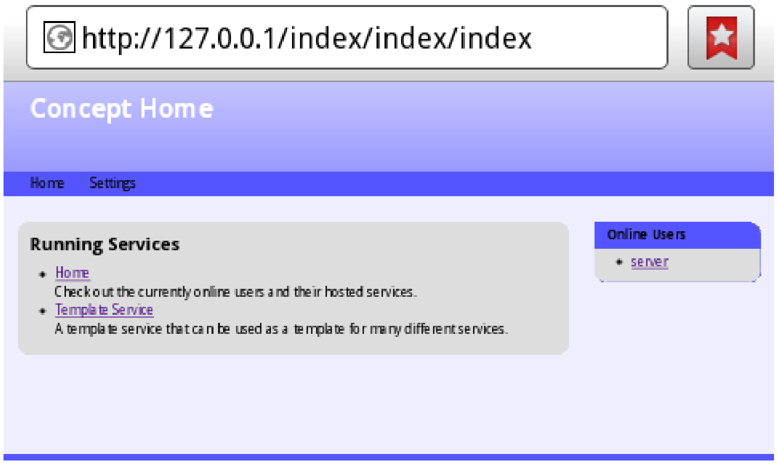
Running Services

- Home
  Check out the currently online users and their hosted services.
- Template Service
  A template service that can be used as a template for many different services.

2011 - Concept Framework

Figure A.2: View of the index service of another node in a laptop

A.3.2 Index Service Viewed From an Android Device



http://127.0.0.1/index/index/index

Concept Home

Home Settings

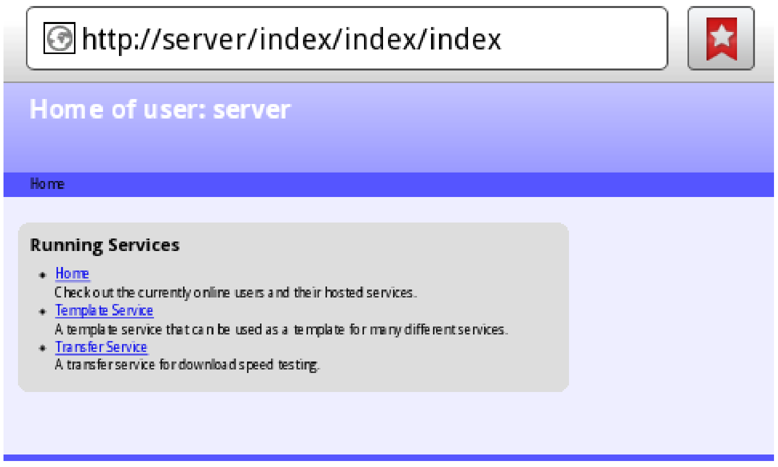
Running Services

- Home
  Check out the currently online users and their hosted services.
- Template Service
  A template service that can be used as a template for many different services.

Online Users

- server

Figure A.3: View of the local index service in an Android device



http://server/index/index/index

Home of user: server

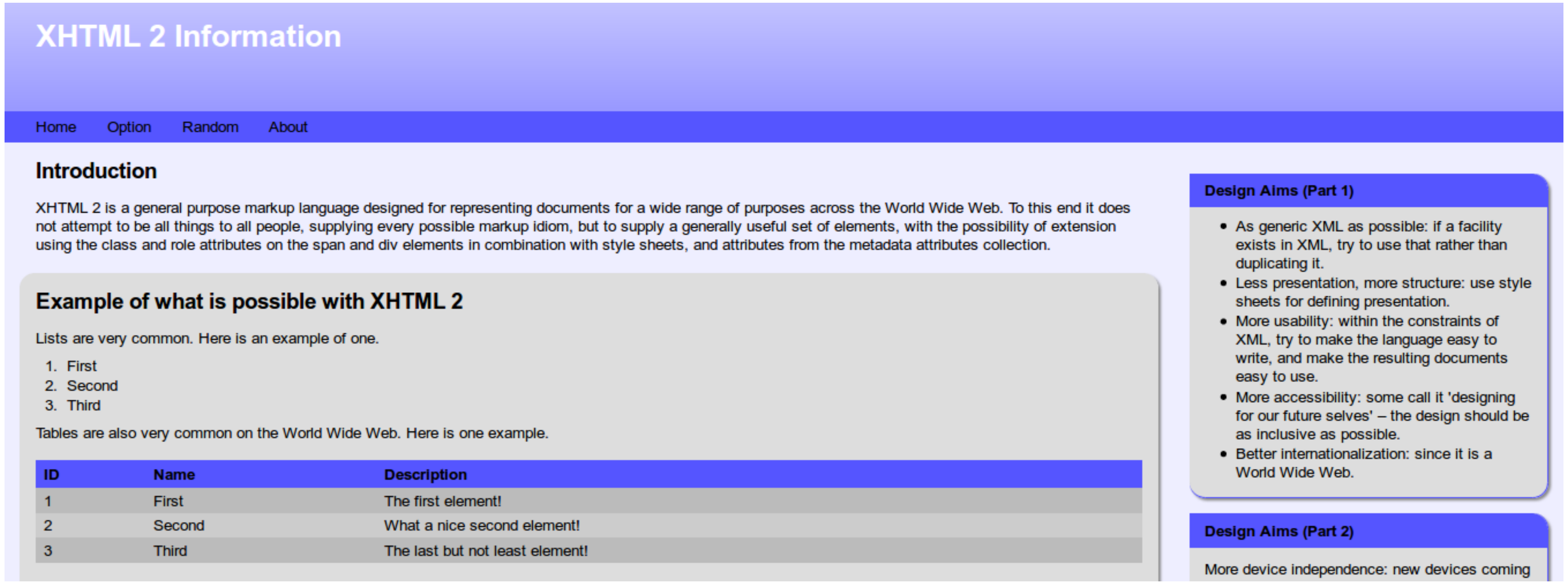
Home

Running Services

- Home
  Check out the currently online users and their hosted services.
- Template Service
  A template service that can be used as a template for many different services.
- Transfer Service
  A transfer service for download speed testing.

Figure A.4: View of the index service of another node in an Android device

A.3.3 Default Layout Showcase



XHTML 2 Information

Introduction

XHTML 2 is a general purpose markup language designed for representing documents for a wide range of purposes across the World Wide Web. To this end it does not attempt to be all things to all people, supplying every possible markup idiom, but to supply a generally useful set of elements, with the possibility of extension using the class and role attributes on the span and div elements in combination with style sheets, and attributes from the metadata attributes collection.

Example of what is possible with XHTML 2

Lists are very common. Here is an example of one.

1. First
2. Second
3. Third

Tables are also very common on the World Wide Web. Here is one example.

ID	Name	Description
1	First	The first element!
2	Second	What a nice second element!
3	Third	The last but not least element!

Design Aims (Part 1)

- As generic XML as possible: if a facility exists in XML, try to use that rather than duplicating it.
- Less presentation, more structure: use style sheets for defining presentation.
- More usability: within the constraints of XML, try to make the language easy to write, and make the resulting documents easy to use.
- More accessibility: some call it 'designing for our future selves' – the design should be as inclusive as possible.
- Better internationalization: since it is a World Wide Web.

Design Aims (Part 2)

More device independence: new devices coming

Figure A.5: Showcase of the default layout

[1]XHTML 2 information in figure A.5 was taken from `http://www.w3.org/TR/2006/WD-xhtml2-20060726/introduction.html`

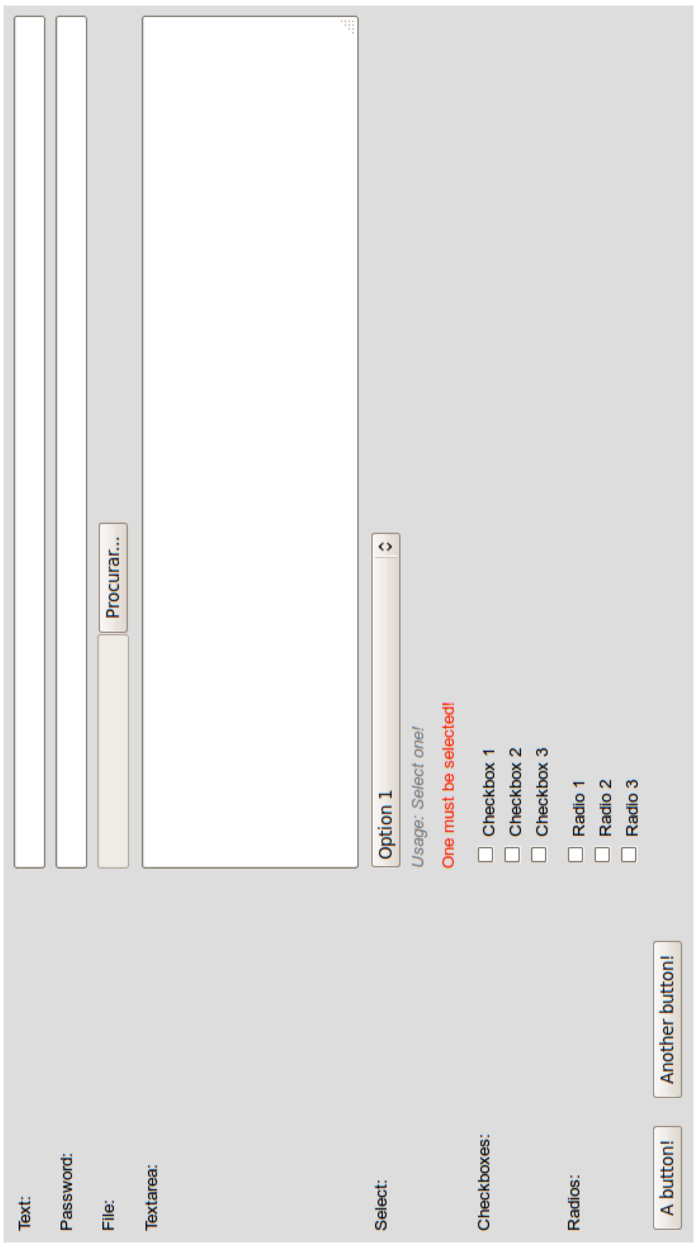


Figure A.6: Showcase of a form in the default layout

Bibliography

[1] Sanghyun Ahn and Yujin Lim. A modified centralized dns approach for the dynamic manet environment. In *Proceedings of the 9th international conference on Communications and information technologies*, ISCIT'09, pages 1506–1510, Piscataway, NJ, USA, 2009. IEEE Press.

[2] J. Ahrenholz, C. Danilov, T.R. Henderson, and J.H. Kim. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1 –7, November 2008.

[3] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A proposal for the advanced encryption standard. *NIST AES Proposal*, June 1998.

[4] T. Aura. Cryptographically Generated Addresses (CGA). RFC 3972 (Proposed Standard), March 2005. Updated by RFCs 4581, 4982.

[5] M. Baker and M. Nottingham. The "application/soap+xml" media type. RFC 3902 (Informational), September 2004.

[6] P. Bellavista, A. Corradi, and E. Magistretti. Redman: a decentralized middleware solution for cooperative replication in dense manets. In *Pervasive Computing and Communications Workshops, 2005. PerCom 2005 Workshops. Third IEEE International Conference on*, pages 158 – 162, 3 2005.

[7] Paolo Bellavista, Antonio Corradi, and Carlo Giannelli. The real ad-hoc multi-hop peer-to-peer (ramp) middleware: An easy-to-use support for spontaneous networking. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 463 –470, 6 2010.

[8] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), December 1994. Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986, 6196, 6270.

[9] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference*

on Advances in Cryptology, CRYPTO '01, pages 213–229, London, UK, 2001. Springer-Verlag.

[10] Jerome Burke, John McDonald, and Todd Austin. Architectural support for fast symmetric-key cryptography. *ACM SIGARCH Computer Architecture News*, 28:178–189, November 2000.

[11] Bouncy Castle. Official web page of the bouncy castle crypto apis. `http://www.bouncycastle.org/`, October 2011.

[12] Stefano Ceri, Florian Daniel, Maristella Matera, and Federico M. Facca. Model-driven development of context-aware web applications. *ACM Transactions on Internet Technology*, 7, February 2007.

[13] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), May 2005.

[14] Justin Collins and Rajive Bagrodia. Programming in mobile ad hoc networks. In *Proceedings of the 4th Annual International Conference on Wireless Internet*, WICON '08, pages 73:1–73:9, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[15] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. OSPF for IPv6. RFC 5340 (Proposed Standard), July 2008.

[16] S. Dabideen, B.R. Smith, and J.J. Garcia-Luna-Aceves. An end-to-end solution for secure and survivable routing in manets. In *Design of Reliable Communication Networks, 2009. DRCN 2009. 7th International Workshop on*, pages 183 –190, October 2009.

[17] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. *NIST AES Proposal*, June 1998.

[18] Vanesa Daza, Paz Morillo, and Carla Ràfols. On dynamic distribution of private keys over manets. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 171:33–41, April 2007.

[19] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[20] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFCs 3396, 4361, 5494.

[21] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315 (Proposed Standard), July 2003. Updated by RFCs 4361, 5494, 6221.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[23] Pravin Ghosekar, Girish Katkar, and Pradip Ghorpade. Mobile ad hoc networking: Imperatives and challenges. *IJCA Special Issue on MANETs*, (3):153–158, 2010. Published by Foundation of Computer Science.

[24] Mesut Gunes and Jorg Reibel. An ip address configuration algorithm for zeroconf. mobile multi-hop ad-hoc networks. In *In Proceedings of the International Workshop on Broadband Wireless Ad-Hoc Networks and Services, Sophia Antipolis*, 2002.

[25] Erik Guttman. Autoconfiguration for ip networking: Enabling local communication. *IEEE Internet Computing*, 5(3):81–86, 2001.

[26] A.M.M. Habbal and S. Hassan. Loss detection and recovery techniques for tcp in mobile ad hoc network. In *2010 Second International Conference on Network Applications Protocols and Services (NETAPPS)*, pages 48–54, September 2010.

[27] David Heinemeier Hansson. Ruby on rails. `http://rubyonrails.org/`, October 2011.

[28] S. Helal, N. Desai, V. Verma, and Choonhwa Lee. Konark - a service discovery and delivery protocol for ad-hoc networks. In *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, volume 3, pages 2107 –2113 vol.3, 3 2003.

[29] Amir Herzberg and Ahmad Jbara. Security and identification indicators for browsers against spoofing and phishing attacks. *ACM Transactions on Internet Technology*, 8:16:1–16:36, October 2008.

[30] Jin-Ok Hwang, Chuck Yoo, and Sung-Gi Min. Analysis of critical points for ip address auto-configuration in ad-hoc networks. *International Conference on Networking*, 0:162–166, 2010.

[31] IEEE Std 802.11-2007. IEEE standard for information technology — telecommunications and information exchange between systems — local and metropolitan area networks — specific requirements — part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, June 2007.

[32] IETF. The internet engineering task force. `http://www.ietf.org/`, October 2011.

[33] IETF. Zeroconf working group. `http://www.zeroconf.org/`, October 2011.

[34] J. Gnana Jayanthi, S. Albert Rabara, and A. Rex Macedo Arokiaraj. Ipv6 manet: An essential technology for future pervasive computing. *International Conference on Communication Software and Networks*, 0:466–470, 2010.

[35] Christophe Jelger and Christian Tschudin. Dynamic names and private address maps: complete self-configuration for manets. In *Proceedings of the 2006 ACM CoNEXT conference*, CoNEXT '06, pages 4:1–4:9, New York, NY, USA, 2006. ACM.

[36] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Proposed Standard), September 2010. Updated by RFC 5998.

[37] S. Kent. IP Authentication Header. RFC 4302 (Proposed Standard), December 2005.

[38] S. Kent. IP Encapsulating Security Payload (ESP). RFC 4303 (Proposed Standard), December 2005.

[39] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFC 6040.

[40] H. Kumar, R.K. Singla, and S. Malhotra. Issues & trends in autoconfiguration of ip address in manet. *International Journal of Computers Communications & Control*, ISSN 1841-9836, Volume 3:353–357, 2008.

[41] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14:255–293, 1999.

[42] F. Mário Martins. *JAVA6 e Programação Orientada Pelos Objectos.* FCA, July 2009.

[43] Sudip Misra, Isaac Woungang, and Subhas Chandra Misra. *Guide to Wireless Ad Hoc Networks.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[44] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966.

[45] Pradosh Kumar Mohapatra. Public key cryptography. *Crossroads*, 7:14–22, September 2000.

[46] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the advanced encryption standard (aes). Technical report, October 2000.

[47] Sanket Nesargi and Ravi Prakash. Manetconf: configuration of hosts in a mobile ad hoc network. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 1059 – 1068 vol.2, 2002.

[48] R. Ogier and P. Spagnolo. Mobile Ad Hoc Network (MANET) Extension of OSPF Using Connected Dominating Set (CDS) Flooding. RFC 5614 (Experimental), August 2009.

[49] Oracle. Java official web page. `http://www.oracle.com/technetwork/java/index.html`, October 2011.

[50] Oracle. Permissions in the jdk. `http://download.oracle.com/javase/1.5.0/docs/guide/security/permissions.html`, October 2011.

[51] Massimiliano Pala. A proposal for collaborative internet-scale trust infrastructures deployment: the public key system (pks). In *Proceedings of the 9th Symposium on Identity and Trust on the Internet*, IDTRUST '10, pages 108–116, New York, NY, USA, 2010. ACM.

[52] Poonam, K. Garg, and M. Misra. Trust based multi path dsr protocol. In *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, pages 204–209, February 2010.

[53] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000. Updated by RFC 5785.

[54] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, February 1978.

[55] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. *NIST AES Proposal*, June 1998.

[56] Adi Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, November 1979.

[57] Adi Shamir. Identity-based cryptosystems and signature schemes. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 47–53, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[58] SpringSource. Spring framework. `http://www.springsource.org/`, October 2011.

[59] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), September 2007.

[60] Johann van der Merwe, Dawoud Dawoud, and Stephen McDonald. Fully self-organized peer-to-peer key management for mobile ad hoc networks. In *Proceedings of the 4th ACM workshop on Wireless security*, WiSe '05, pages 21–30, New York, NY, USA, 2005. ACM.

[61] Tinghui Xu and Jie Wu. Quorum based ip address autoconfiguration in mobile ad hoc networks. In *ICDCSW '07: Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, page 1, Washington, DC, USA, 2007. IEEE Computer Society.