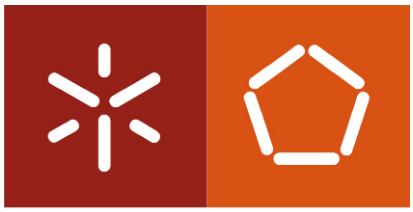


Universidade do Minho
Departamento de Informática

Nuno Miguel da Costa Abreu

On-demand Content Delivery on Spontaneous Mobile Mesh Networks

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob a orientação do
Professor Doutor António Duarte Costa

Outubro de 2011

Acknowledgements

I want to thank to my mentor, Professor António Duarte Costa for all the support provided. His assistance was crucial to guarantee that this dissertation was going on the right direction. I am very appreciated for the guidance provided throughout this work.

To my friends I demonstrate here my gratitude for being always available to help on any circumstance. In the same way, I want to thank to my family for the constant support that helped me to maintain focused on this work.

A special thanks to Iolanda, that supported me in good and bad times. I devote this work to you.

Abstract

Today there is a vast number of mobile devices. These devices allow people to access services everywhere. Typically a network infrastructure is required to support these services, like a wireless access point or a 3G connection. Sometimes such infrastructure may not exist or may not be available, making services impossible to operate. Ad-hoc networks allow infrastructure-less communication where each device can communicate with other devices from the network without depending on some infrastructure. These networks can be explored in order to provide services. For example, content delivery in case there is no infrastructure available to support the communication.

The main objective of this work is to take advantage of the potential of ad-hoc networks to provide some services related with content access. The goal is to achieve a framework that is able to explore ad-hoc networks to successfully deliver content to every interested user. Besides, it should be able to work in different devices and operating systems.

In this work, a fully functional framework prototype was implemented, requiring minimal configuration. The result is an off-the-shelf application that needs only a Java Virtual Machine (JVM) to operate. In order to successfully forward content between nodes from the server to the destination, a new routing model was developed that is exclusively based on content IDs instead of addresses. We used HTTP as presentation layer of the framework. This way we enable the customization of the interface by the server. Each user that is already familiarized with HTML pages can easily interact with our system.

Resumo

Hoje em dia existe um grande número de dispositivos móveis que permitem o acesso a serviços em qualquer lado. Para suportar esses serviços é necessária uma infra-estrutura de rede, como por exemplo, um ponto de acesso sem fios ou uma ligação 3G. Quando essa infra-estrutura é inexistente ou não está disponível, os serviços tornam-se inacessíveis. As redes ad-hoc possibilitam a comunicação independente de qualquer infra-estrutura. Estas redes podem ser exploradas por forma a fornecer serviços, tais como o acesso a conteúdo, no caso de não existir uma infra-estrutura de comunicação.

O objectivo principal é tirar partido das redes ad-hoc para fornecer serviços de acesso a conteúdo. Queremos obter uma plataforma capaz de fornecer conteúdos aos utilizadores interessados, explorando as redes ad-hoc. Para além disso, deve também ser capaz de operar em diferentes dispositivos e sistemas operativos.

Neste trabalho foi implementado um protótipo da plataforma completamente funcional. O resultado final é uma aplicação pronta a ser utilizada, que necessita apenas de uma Java Virtual Machine (JVM). Foi desenhado um novo modelo de encaminhamento baseado, exclusivamente, em IDs de conteúdo, por forma a encaminhar o tráfego entre servidor e cliente. Utilizamos o HTTP como camada de apresentação, dessa forma, o servidor pode 'desenhar' o interface. A interacção com o sistema é bastante simples no caso do utilizador estar familiarizado com as páginas HTML.

Contents

List of Figures

List of Tables

List of Acronyms

7DS Seven Degrees of Separation

ACK Acknowledgement

ADU Application Data Unit

AJAX Asynchronous Javascript and XML

AODV Ad-Hoc On-Demand Distance Vector

API Application Programming Interface

ASP Active Server Pages

ATCP Ad hoc TCP

BBS Bulletin Board System

CORE Common Open Research Emulator

CPU Central Processing Unit

CSV Comma-Separated Values

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

DREAM Distance Routing Effect Algorithm for Mobility

DSDV Destination-Sequenced Distance Vector

DSR Dynamic Source Routing

ECN Explicit Congestion Notification

ELFN Explicit Link Failure Notification

GPS Global Positioning System

GSM Global System for Mobile communication

HTML HyperText Markup Language

IARP IntrA-zone Routing Protocol

ICMP Internet Control Message Protocol

IERP IntEr-zone Routing Protocol

iMANET Internet Based Mobile Ad Hoc Network

InVANET Intelligent Vehicular Ad Hoc Network

IP Internet Protocol

JAR Java Archive

KB Kilobyte

LAR Location-Aided Routing

MAC Media Access Control

MANET Mobile Ad Hoc Network

MB Megabyte

MFR Most Forward within Radius

MID Multiple Declaration Interface

MP3 MPEG-1/2 Audio Layer 3

MPR Multipoint Relay

NFP Nearest with Forward Progress

OLSR Optimized Link State Routing

OS Operating System

OSPF-MDR Open Shortest Path First for Mobile Ad Hoc Networks

PDA Personal Digital Assistant

PHP PHP: Hypertext Preprocessor

RFN Route Failure Notification

RRN Route Reestablishment Notification

RTT Round-Trip Time

SOAP Simple Object Access Protocol

TCP Transmission Control Protocol

TCP-F TCP-Feedback

TTL Time To Live

UDP User Datagram Protocol

URL Uniform Resource Locator

UUID Universally Unique Identifier

VANET Vehicular Ad Hoc Network

WiMAX Worldwide Interoperability for Microwave Access

WSDL Web Service Definition Language

WWW World Wide Web

XML Extensible Markup Language

ZRP Zone Routing Protocol

Chapter 1

Introduction

Nowadays we are surrounded by mobile devices. Anywhere and in any circumstance, mobile devices are present. Today it is more difficult to find someone in the crowd without any mobile device than the opposite. People are becoming more and more dependent on the information accessible through mobile devices. Besides, devices are becoming more sophisticated which allows you to perform several different tasks anywhere. However, these mobile services are very dependent on some network infrastructure. Many times and in many places, a wireless access point is provided to allow access to all users. When there is no access point available, users can still access the services using WiMAX or 3G. The problem is that these technologies imply costs to users. In case we have content to share on a given event without any network infrastructure available on the event's location and in case users do not have a 3G plan or the location is not covered with 3G access, then how can users access the content? If devices are able to communicate between each other then no infrastructure is needed.

Ad-Hoc networks allow devices to communicate directly between each other. This way, the network relies on each device to route traffic to other devices in order to ensure the communication. We can share content anytime, anywhere, without depending on nothing more than the network devices. Events with a content sharing service may become more frequent as they become more easy to implement. There is no time wasted constructing the infrastructure and no costs to acquire or maintain it. Having an infrastructure-less network, we simply need a mechanism to make the contents available. We can either choose an existent solution or create a new one to take advantage of the specific characteristics of the ad-hoc network.

1.1 Objectives

The main objective of this work is to achieve a full functional framework capable of distributing content between devices over an ad-hoc network. The final solution should run on different devices with different operating systems. Besides, the framework should require minimal system configuration or dependencies on additional software. This is important as the framework is designed to deliver content on a spontaneous mobile mesh network. We can assume that on such a spontaneous network, the devices may not have all the dependencies installed and correctly configured. Therefore, we

intend to achieve an off-the-shelf solution in order to successfully work on every single device from the spontaneous network.

We intend to build a framework centered on content, because the ultimate goal is to deliver content on the network. That way, we aim to design the application in a way that the entire network cooperates in order to deliver content between its nodes. This is also known as a content-centric network.

In order to allow users to access the system, a suitable interface should be available. The interface is the ultimate frontier between user and the application. If this interface is not understood by the user, ultimately the application is useless as the user is not able to access it. However, some level of customization should be considered in order to adapt the application to the different needs from different kinds of events.

To allow the system to be used by other applications, allowing any developer to design its own interface and fulfill its own needs, the system must be easily accessible by providing a simple API.

1.2 Methodology

To successfully accomplish the objectives described, the first stage consists on studying the existing applications for ad-hoc networks. In order to understand how the ad-hoc networks work, the different types of ad-hoc networks are studied. We investigate how general applications designed for other scenarios, like wired networks or the Internet, operate over an ad-hoc network without any modification and how the nodes from the network are able to route traffic. TCP is one of the most popular transport protocols used by the most popular applications [3]. Due to that fact, we also study how this protocol performs on ad-hoc networks and how it can be improved to achieve better results. Next, the mechanisms to share content between different nodes on a network are studied. We investigate how applications are designed to share content on an ad-hoc network.

After evaluating the state of the art, we are in better conditions to make our decisions relative to the system design. First, we design the system architecture, defining which modules are going to be implemented and how they interact between each other. For each module, we propose the strategy that seems the most adequate. Next, the implementation process is initiated. The essential system modules are implemented in order to achieve the full functional framework. To ensure that we are able to finish the framework on time, optimizations are delayed in order to have the essential functionalities from each module implemented.

When a functional prototype is ready, some tests must be done in order to evaluate the global performance and to adjust the application in order to achieve the best results. To test the application, a testing scenario must be defined. First some tests are performed with the help of an emulator, which allow us to understand the application behavior. With this knowledge, we can deduce what are the causes for the results obtained and come up with new ideas and improvements to the modules developed in order to increase the global performance. We also need to perform some more tests to discover if the results from the emulated environment are replicated on a real scenario. These real scenario tests are fundamental to observe how the prototype will perform on real events. Since

the main objective is to achieve a functional framework, results from emulated environments are not enough.

1.3 Main Contributions

The main contribution from our work is a fully functional framework for delivering content on-demand. The framework is responsible for delivering content identified by an ID when it is requested. Another contribution is a new routing model based on content that seems best suited to the scenario for which our framework was developed for. Our routing module is responsible for routing files and not addresses. Although, the routing module is very versatile and can be adapted to fulfill other scenarios. We achieved a dynamic interface using HTML produced by a mini HTTP server. The server is able to write the HTML code that is going to be used to provide access to the contents. Besides, we assume that any user familiar with the World Wide Web (WWW) is capable of using our system. This enhances the usability of the system as more users are able to easily use and adapt our application interface.

From the work developed throughout this dissertation, a poster as been submitted and accepted on "11^a Conferência sobre Redes de Computadores", a national conference about computer networks. Conference to be held at the University of Coimbra, Portugal, on November 17-18, 2011.

1.4 Document Structure

The documents starts with an introduction on chapter 1. In order to understand what there is available for content distribution on ad-hoc networks, on chapter 2 we present the state of the art related with ad hoc networks. Chapter 3 presents the system architecture and some implementation decisions. The tests made and the respective results and interpretations are presented on chapter 4. The final remarks regarding the project achievements, conclusions and future work are presented on chapter 5.

Chapter 2

State of The Art

Nowadays, most of our devices already communicate using wireless technology. Therefore, we can communicate almost anywhere, at any time, provided that we have a device that supports the wireless technology. It is easy to conclude that this technology is already used by the majority of us, almost every time. We use it when sending a text message with our cell phone, or making a phone call and even in our homes, where we access the Internet through a wireless router.

Although there are many different technologies for wireless communications, such as Bluetooth, Satellite, GSM, among others, this article focuses attention on Wi-Fi (IEEE 802.11). Wireless networks represent today one of the most common means of communication between devices such as computers. Due to their flexibility, these networks have become more interesting compared to the traditionally wired networks. The technology began to be quickly adopted everywhere. When we enter a mall, usually it already provides access to a wireless connection, and also internet access through it to its visitors. One wireless access point in every home, enabling internet access to all the family is also a very common scenario.

Despite its flexibility, there is still a heavy reliance on specific equipment for supporting the communication, such as wireless routers. An infrastructure set up at a specific location, allowing access to a wireless connection, represents the most common scenario. This represents a limitation regarding users location, since each user has to position itself within reach of the access point to have connectivity. In addition, if there are multiple users concentrated in one place, they will use the same access point, and as a consequence, the access point may become overloaded. The network will eventually become saturated and the quality of communication will degrade. This work focuses on wireless ad-hoc communication which requires no infrastructure to operate, thus allowing communication to be made directly between the various network nodes. Changing the paradigm of connecting to an access point for ad-hoc communication, we can save resources because the infrastructure is no longer needed. The reach of the network will be expanded and resources will be optimized by passing the responsibility of network maintenance from the infrastructure to the nodes that form it. With wireless ad-hoc communication, we can resolve problematic scenarios where there is too much people concentrated in one place, each one wanting to access data that is available in one access point or in another node. The access point paradigm cannot handle this scenario, because the node will be overwhelmed with content

requests, and other nodes will simply be out-of-range, unable to contact the access point.

2.1 Wireless Ad-Hoc Networks

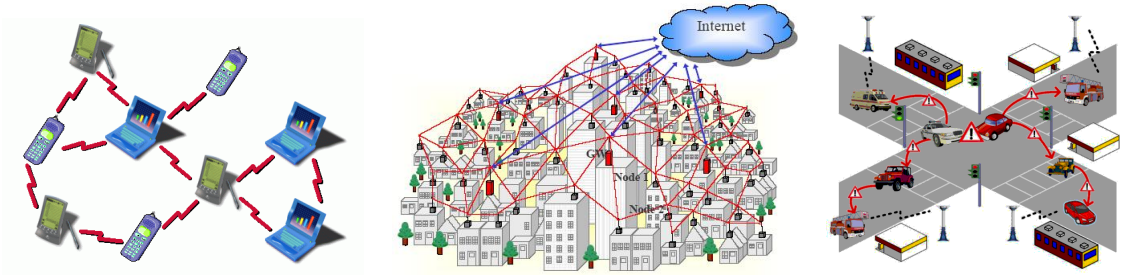
A wireless ad-hoc network is a decentralized wireless network [44]. Ad hoc is a Latin expression meaning "for this", usually meaning a specific solution to a given problem. So, it can be assumed that a wireless ad-hoc network is a special case of a wireless network where there is no network infrastructure. In these networks, the communication shall be ensured by the nodes that comprise it, instead of using routers and access points.

2.1.1 Wireless Mesh Networks

A wireless mesh network is a communications network consisting of multiple wireless nodes, organized in a mesh topology. A mesh topology consists of multiple nodes interconnected, where each node also acts as a router forwarding traffic between the various nodes. An example of such network is represented in Figure 2.1(b). In these networks, each node must be able to choose paths between the various nodes to reach a particular destination. Additionally, it should be able to choose other routes, in case of one or more nodes get disconnected. Typically these networks have a higher reliability and provide redundancy, since there is usually more than one path between a source and destination. In a mesh network, we can have mesh routers which have the job of connecting the various nodes and communicate with other mesh routers. Usually, the mesh routers have more resources compared to other network nodes and accordingly may carry more operations that require resources. These networks, in that point, differ from ad-hoc networks, since some of the nodes have less restrictions in terms of resources. Wireless mesh networks maintain good signal strength, since long distances are divided in a number of intermediate hops shorter. The wireless mesh networks can be of various types [2]:

- Infrastructure Wireless Mesh Networks
- Client Wireless Mesh Networks
- Hybrid Wireless Mesh Networks

An Infrastructure Wireless Mesh Network is very similar to a simple wireless network, where access is based on an access point. The difference is that there is no access point, but several mesh routers interconnected, which makes up the infrastructure for mesh clients. In this type of network, almost all traffic is routed to or from a gateway (router mesh). On the other hand, in Client Wireless Mesh Networks, the traffic flows between arbitrary pairs of nodes, resembling a simple ad-hoc network. The Hybrid is the result of combining the two previous approaches.



(a) Mobile Ad-Hoc Network (b) Mobile Mesh Network (c) Vehicular Ad Hoc Network

Figure 2.1: Some examples of Wireless Ad-Hoc Networks

2.1.2 Mobile Ad-Hoc Networks

A mobile ad hoc network (MANET) is a wireless multi-hop network formed by a set of mobile nodes in a self-organizing way without relying on any established infrastructure [7]. These networks are very similar to Wireless Mesh Networks, hence they are also called Mobile Mesh Networks. In these networks, each device can move freely in any direction and thus will change the connections to other devices frequently. The major difference from Wireless Mesh Networks, it is the node's mobility, as we can see from Figure 2.1(a). To establish communication between different nodes, each one must forward traffic unrelated to itself, thus functioning as a router. The main challenge on building a MANET network is to ensure that each device has, continuously, the information needed to correctly forward traffic. Mobile Ad-Hoc Networks have become increasingly important with the growing number of laptops and Wi-Fi technology. As today there are more and more devices with Wi-Fi that can be transported anywhere, such as PDAs and mobile phones, it is relevant to explore this network architecture because of the potential it represents. Some types of MANETs:

- Vehicular Ad Hoc Networks (*VANETs*)
- Intelligent Vehicular Ad Hoc Networks (*InVANETs*)
- Internet Based Mobile Ad Hoc Networks (*iMANETs*)

The VANETs are used for communication among vehicles and between vehicles and equipment placed on the road as illustrated in Figure 2.1(c). The InVANETs are also VANET networks, but use artificial intelligence in vehicles to act wisely, in case of accidents, in cases of driving under the influence of alcohol, among other situations. iMANET is a MANET network that connects mobile nodes with gateways with access to the internet. An iMANET is therefore quite similar to a wireless mesh network, which in turn, is usually architected as an iMANET.

It can be concluded that wireless ad-hoc networks are quite applicable and cover different scenarios, where a peer-to-peer communication model is needed and there is no infrastructure available.

2.2 Routing Protocols

Some routing mechanism is needed in order to forward data between peers. To accomplish that, there must be some routing protocols to enable communication between nodes from the network that are not in range with each other.

When a node needs to communicate with another node that is not in range, it must forward the packets between intermediate nodes to reach the destination. To route traffic between the nodes, each one has to cooperate to support the communication. The network needs to have a mechanism to find routes between nodes in order to connect sources and destinations. There are too many different protocols to enable routing in Ad Hoc networks. The majority of the protocols can be divided in two categories: proactive routing and reactive on-demand routing [10].

2.2.1 Proactive Routing

The proactive routing protocols focus on building information about the network. The main objective is to maintain an updated table of destinations and respective routes. Due to this fact, this is also known as Table-Driven routing. The tables are updated distributing the routing tables periodically between the nodes. Some proactive routing protocols are now presented and explained in order to understand how the proactive routing protocols work.

Optimized Link State Routing

Optimized Link State Routing (OLSR) protocol is a link-state routing protocol adapted to perform best in mobile ad hoc networks [21]. To gather information about links, the protocol includes a *Link Sensing* module. This module consist on generating and sending periodic HELLO messages. A *Neighbor Detection* module is also included that makes use of the *Link Sensing* module to discover neighbors. Each node is identified by a "main address", if the node has only one interface, the "main address" is the address of that interface. The protocol is designed in order to identify unequivocally each node. Having more than one interface, the node can be perceived as two different nodes by the neighbors. The node with multiple interfaces uses Multiple Interface Declaration (MID) messages. With this information, the neighbors will correctly identify the node.

The key concept of this protocol is the use of multipoint relays (MPRs). *MPR Selection* and *MPR Signaling* modules are responsible of selecting and announce the MPRs. MPRs are responsible for forwarding control traffic to the entire network. The objective of using MPRs is to enable each node A to select a subset of neighbors that will be responsible for retransmitting all the broadcast messages received from A in order to reach all symmetric strict 2-hop neighbors of A.

The protocol includes a *Topology Control Message Diffusion* module. This module is responsible for the diffusion of messages that contain information needed to enable each node to perform routing calculations. These messages are disseminated throughout the network with the help of the MPRs.

With the link-state information from *Link Sensing* module, the topology information received and

the MPR selection, each node stores information about:

- neighbors
- 2-hop neighbors
- MPRs
- MPR selectors

The MPR selectors of A are the nodes that selected A as MPR. With this information, each node is able to perform route calculation and to route traffic control messages.

Destination-Sequenced Distance Vector

Destination-Sequenced Distance Vector (DSDV) represents a distributed version of the shortest path problem [35]. Each node maintains a preferred neighbor for each destination. Packet header includes the destination node identifier. When a node receives a packet, it forwards to the destination's best neighbor. The forwarding process continues until it reaches destination.

To enable the packet forwarding, each node maintains a routing table. The table lists the available destinations and respective hop-number. Each entry includes a sequence number that is generated by the destination node. Nodes transmit information periodically or immediately in case there is new significant information. Besides, they broadcast their routing table. The table may vary often, so it has to be announced frequently.

For each table entry, the destination address, hop-number distance and the sequence number as defined by the destination are broadcasted. When a node receives a routing table broadcast, it forwards it again in a new broadcast after incrementing the hop-number. With the routing information received, each node is able to update its own routing table and identify the best neighbor for each destination. Routes with bigger sequence numbers are preferred on the forwarding decisions, because they represent updated routes. In the presence of two different routes to the same destination with the same sequence number, the one with the lower hop-number is preferred.

A route sequence number is usually generated by the destination. Sometimes the sequence number is changed by other nodes when a link break is detected. To distinguish sequence numbers generated by the destination from the ones generated by other nodes, the destination generates even numbers only while intermediate nodes generate only odd sequence numbers.

When a node A moves to a vicinity of different nodes, it increments the sequence number and sends update messages to its new neighbors. This is done to update the routing tables of the nodes, allowing them to access node A.

This protocol grants each node with a next-hop table for each destination. The next-hop is calculated based on hop-count.

2.2.2 Reactive On-demand Routing

With this type of routing, the route discovery process is issued on-demand. When a node wants to send a packet to another node on the network and does not have a route for it, it initiates a route discovery process. On-demand discovery process generates less overhead for maintaining routes compared with proactive approaches. The overhead generated in order to keep updated routing tables with proactive approaches in mobile scenarios can be very high. The main idea behind reactive routing is that routes must be found and maintained only when they are needed. With proactive routing, routes are discovered and updated constantly even if they are not being used.

Ad-Hoc On-Demand Distance Vector

Ad-Hoc On-Demand Distance Vector (AODV) [34] is an adaptation of DSDV to reduce the number of route broadcasts by creating routes on-demand.

Although being a reactive version of DSDV, some proactive functions remain in this protocol. In order to identify direct neighbors, each node broadcasts frequently HELLO messages. When a node A wants to send a packet to another node that is not its neighbor, it broadcasts a route request message. Upon receiving a route request, if a node B is the destination it responds with a route reply to A. Otherwise, B can respond with a route reply if it knows a fresh route to the destination. When B is not the destination and does not know a route to it, it rebroadcasts the request to its neighbors. Each request packet includes the source, destination, an unique ID and a lifespan value. The lifespan value is used to limit the scope of the request. Each time the request is rebroadcasted, the lifespan is decremented and when it reaches zero, it is discarded. The unique ID is used in order to avoid cycles. When a node receives a new request with the same ID and the same source from a previous request, it is discarded.

Similar to DSDV, each route has a sequence number in order to detect the most recent routes. Besides, each node keeps track of the next-hop for each destination instead of keeping the entire route, just like in DSDV. AODV uses route error messages when link break is detected. These messages are used to announce which nodes are now unreachable. The protocol works only on symmetric links.

Dynamic Source Routing

Dynamic Source Routing (DSR) is a protocol for multi-hop wireless ad hoc networks [22]. This protocol is entirely on-demand. There is no traffic when there is no need for routing packets.

Each packet includes on its header the full path from source to destination, this ensures that the path is loop-free. Intermediate nodes do not need to have updated routes to deliver packets. Since the full path is in the packet header, each intermediate node is able to cache the route.

The protocol assumes that each node chooses only one IP address. Although nodes can have more than one network interface with different addresses, each one has to choose one address to join the DSR protocol.

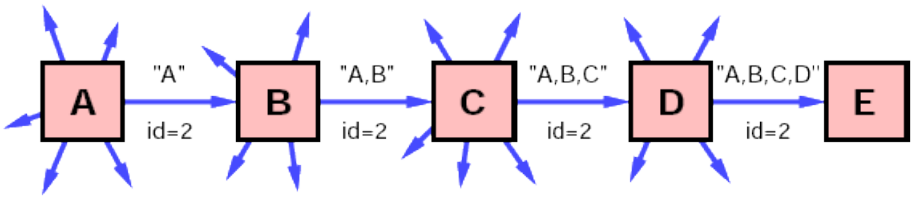


Figure 2.2: Example of routing discovery, A the initiator and E the destination

The protocol is composed by two fundamental modules:

- Route Discovery
- Route Maintenance

When there is a route to the destination on the route cache, the source node adds the route to the packet header and sends the message to the next hop. In case there is no route available, route request messages are broadcasted. The request is identified by a unique request id including information about the initiator and the target. Each individual route request contains also a list of addresses of the nodes it passes through. The list is initialized empty by the initiator. When a route request is received, it is checked if a previous request with the same ID was received. If that is true, the request is simply discarded. In Figure 2.2 you can see how the routing discovery works.

The destination can respond to the initiator if already has a route for it. Otherwise, it starts a route discovery to the initiator, but to avoid request cycles, it adds the route reply in the request. With this approach, DSR can operate in the presence of uni-directional links. Another way of sending route replies is to reverse the path on the route request.

When there are packets to send to a destination that has no route yet, they are stored in a *Send Buffer*. After some time they are discarded.

The route maintenance process is only performed when the source is sending packets. There are no other mechanism to see if the route remains valid, like in proactive protocols. The node detects that a route is invalid when it tries to use it.

To detect route errors, acknowledge packets (ACK) are used between intermediate nodes. For example, node A, sends a packet to E, passing through B,C and D similar to the scenario in Figure 2.2. If node C sends a message to D and does not receive an ACK, it sends a route error to A. Node A removes the route from the route cache and uses another route in cache or issues another route discovery.

2.2.3 Hybrid Routing

In order to take advantage of both proactive and reactive routing, there are some attempts to merge the two different approaches on a hybrid solution. That is the case of Zone Routing Protocol (ZRP) [5].

In ZRP there is a concept of zones. Each node defines a routing zone being ρ the zone radius in

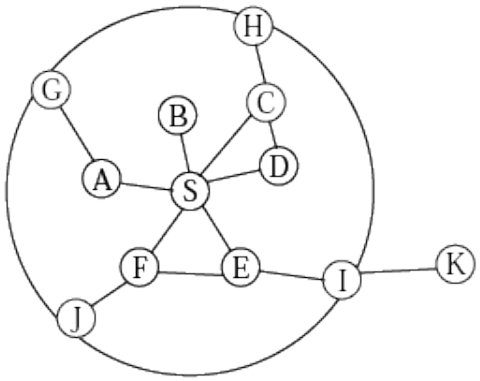


Figure 2.3: Routing zone with ρ=2

hop number. The nodes are classified as interior or peripheral nodes. The peripheral nodes are at distance = ρ.

For nodes within the zone, ZRP uses a proactive IntrA-zone Routing Protocol (IARP). For nodes that do not belong to the zone, it is used a reactive IntEr-zone Routing Protocol (IERP). IERP relies on IARP to perform route discovery and route maintenance. The two different protocols are not specified by ZRP, being the IARP a suitable proactive protocol and IERP a reactive one. IERP uses the information from IARP in order to bordercast the route request. Bordercasting is the term defined in ZRP to the process of sending requests directly to the zone border, i.e. to the peripheral nodes. When a node receives a route request, it replies if it knows the destination or bordercasts the request again. The routing information can be accumulated in the request, like in DSR, or stored as next-hop addresses like in AODV. How the routing information is stored is not defined.

To better understand the zone concept, you can look at Figure 2.3. You can see the zone of node S with ρ=2. All nodes except K are within the zone, because they are at least 2-hops away. When S wants to find a route to K, it will bordercast the route request to nodes G,H,I and J because they are peripheral nodes.

In order to reduce queries traffic, some query-control mechanisms are needed, like query detection and early termination.

Neighbor discovery can be achieved by the MAC layer or using HELLO messages.

2.2.4 Location-aware Routing

Proactive and reactive routing protocols are very tied to the topology of the network. The performance depends on the network topology and due to that fact, as the nodes mobility increases, the worse these algorithms will perform. A different approach to route traffic is to use physical position of the nodes as the base to route traffic between the network. These are called location-aware routing protocols and assume that each node is able of knowing its physical coordinates, using for example a GPS.

Distance Routing Effect Algorithm for Mobility (DREAM)

This protocol uses a *Location Table* to store information about other nodes location [4]. To accomplish this, each node announces its own location in order to be disseminated throughout the network. This table also includes for each node, the time of the last position update. "The farther two nodes are separated, the less often their location table entries need updating", this is due to the *distance effect*, that states that farther nodes appear to move slower than closer nodes. Taking this statement in consideration, two different types of packets are used. Packets have a life time based on the physical distance. Short-lived packets are sent more frequently while long-lived packets are sent less frequently to get farther in the network. The frequency of control messages sent, depends on the nodes mobility.

The main idea of DREAM is the use of the location table in order to calculate the direction where the destination node is, so the packet will be broadcasted only to the nodes that are in the direction of the destination.

Location-Aided Routing

Location-Aided Routing (LAR) [24] is a protocol similar to DREAM. LAR defines an expected zone to the destination. If the source has a previous position of the destination then it uses that information to calculate an expected zone. If the source does not know a previous destination's location, it defines the expected zone as the region that potentially comprises all the ad hoc network. In this case, the algorithm is equivalent to a basic flooding.

Besides, a request zone including the expected zone is defined. Only nodes inside the request zone forward the request. This request zone is defined by the source node. Each node uses the information about the nodes location and expected zone to route the requests only in the direction where the node is expected to be.

The major difference of LAR compared to DREAM is that DREAM uses directed flooding to deliver packets, while LAR only uses directed flooding on the route discovery and maintenance process.

2.2.5 Other Routing Protocols

There is a vast number of different routing protocols. Due to that fact, it is not possible to present all of them, but we have just presented some allowing us to perceive the different approaches to route traffic. The routing protocols presented cover a vast number of similar routing protocols based on the same ideas. Although there are some other routing protocols that worth mention.

In section 2.2.4 on page 12 we present some protocols that are based on physical nodes position. Both DREAM and LAR are protocols that use directed flooding. The principal idea is to flood packets in the direction of the destination. But there are different approaches in location-aware routing.

There are some protocols that are based on the Greedy Forwarding principle. The main idea is to forward the packet to the node that is closest to the destination on each step using only local

information. There are some protocols based on this principle like the Most Forward within Radius (MFR) [42], Nearest with Forward Progress (NFP) [20] and Compass routing [25], each one using a different notion of progress.

In the Location-aware category there is another type of routing known as Hierarchical Routing. The location-aware paradigm is used to route packets on long distances while a proactive distance vector is used when the packet is close to the destination. Grid [27] is an example of hierarchical routing.

In a large network, a vast number of different devices can be present. Due to the heterogeneity of the network, some nodes have more computational and communication capabilities [10]. These nodes are more suitable to support the network functions like routing. Cluster-based routing is based on this principle and it arranges nodes in overlapping clusters. In a cluster there is a clusterhead that is responsible of coordinating the cluster. It supports the routing process for the intra-cluster communication. Other nodes of the cluster can communicate directly with the clusterhead and gateways only. Gateways are the nodes that communicate with 2 or more clusterheads and allow inter-cluster communication. Only clusterheads and gateways participate in the propagation of control and update routing messages. This approach reduces the routing overhead and allows the adoption of existing routing protocols.

A simple algorithm to choose the clusterhead, supposing that each device has a unique ID, consists on choosing node with the most lower ID. This way, 2 clusterheads never get in contact with each other.

Every node from a cluster can communicate with the clusterhead, in that way, each node can communicate with every other node of the network with a maximum of 2-hops.

Since traffic concentrates on the clusterhead, a cluster bottleneck may occur when the clusterhead is not able to handle the tasks to support the cluster. This can be resolved by adopting a distributed cluster approach.

When the node's mobility increases, the formation of new clusters may generate excessive overhead.

There is another category of routing protocols that focus on content instead of routing between 2 specific nodes. This paradigm is known as Content-based routing. When we talk about content-based routing we are talking about the publish-subscribe paradigm [8]. Typically the content is simply an event. When a node wants to receive, for example, events regarding the company "Intel" it will create a predicate like [company = "Intel"]. This subscription must be done on its neighbors or brokers, registering the intention of receiving content that match the predicate. These nodes are the subscribers. Each node has to subscribe the content that it wishes to receive before it is produced. The publishers are the content producers that submit events in the network. When each broker receives a new event, it already knows where to forward it due to its subscription table. Content-based routing can be used, for example, on a stock market. Users intend to receive events on-the-fly related with specific companies and specific stock values. This represents a good approach to disseminate content when it is constantly being generated.

Typically, content-based routing protocols are designed for static networks. Furthermore, in [43] is proposed an adaptation that mixes content-based routing with peer-to-peer networks.

2.3 TCP on Mobile Ad Hoc Networks

TCP was initially designed to operate on wired networks [10]. In order to understand how TCP performs on mobile ad hoc networks, several studies were conducted. In face of multi-hop and mobile scenarios, TCP throughput decreases drastically. Besides, other problems are discovered. The problems are due to intrinsic characteristics of the TCP protocol. In [46, 47] three different problems were identified:

instability problem The TCP connection throughput can be very unstable, even when it is the only active connection

incompatibility problem In case there are two simultaneous TCP connections, it may happen that they cannot co-exist. When a connection develops the other shuts down.

one-hop unfairness problem With two simultaneous TCP connections, if one is one-hop and the other multi-hop, multi-hop connection is disconnected even if it started first.

Furthermore, the unstable throughput on mobile scenarios can be easily explained by the nature of the TCP protocol. Since the ratio of packets lost in wired networks is very low, TCP interprets packet loss as an indication of network congestion. In TCP, packet loss is interpreted as packets discarded by routers due to congestion. TCP triggers the congestion control mechanism, reducing the transmission windows size and retransmitting the packets. Typically, in ad-hoc networks, the packet loss is due to route failure. Having nodes changing place constantly leads to route failures that are interpreted by TCP as congestion. This leads to unnecessary retransmissions during the route reconstruction, degrading the communication throughput. Besides, the mobility can aggravate the unfairness between concurrent TCP sessions.

In [15] it is demonstrated that there is an optimal value for the size of the congestion window for a given topology and traffic pattern. The problem is that TCP operates with a much higher value resulting in low throughput.

According to [3], less attention has been devoted to UDP applications, due to the fact that the most popular applications use TCP. Besides, the interaction between 802.11 MAC protocol with TCP protocol may lead to unexpected phenomena in a multi-hop environment, like severe unfairness problems in simultaneous TCP flows and in extreme cases, channel capture by few flows. It is stated that these phenomena do not appear when UDP is used, or appear with less intensity.

2.3.1 TCP-Feedback

TCP-Feedback (TCP-F) is a scheme defined in [9]. This is a scheme that tries to introduce a feedback system to the TCP. The key idea is to use Route Failure Notification (RFN) messages to alert the

TCP sender when a route fails, preventing the congestion control mechanism from being triggered. This allows the distinction from route failure and real network congestion.

When a node detects a route disruption due to the mobility of the next-hop, it explicitly sends a RFN message to the TCP sender. Upon the reception of a RFN message, a intermediate node invalidates the route, preventing traffic from being forwarded further. If the intermediate node knows an alternative route, it uses that new route discarding the RFN message. Otherwise the RFN is propagated toward the source.

As soon as the RFN message arrives to the sender, it goes into a *snooze* state. The sender stops completely from sending packets (new or retransmissions). Marks all timers as invalid and freezes the window of packet sending. Besides, other variables are frozen like windows size or retransmission timeout. A route failure timer is started with the time for the worst case of route reestablishment. The sender keeps in snooze state until a Route Reestablishment Notification (RRN) is received.

After a RRN is received the communication is resumed using the same rate used before route failure. The main idea from this scheme is to freeze the sender, disabling TCP from issuing the congestion control mechanism.

2.3.2 ATCP

ATCP (ad hoc TCP) [28] is a layer for ad hoc TCP. The authors decided to keep TCP protocol intact and implement a new layer between IP and TCP. When the network is partitioned, the TCP sender is put on *persist mode*, stopping transmitting data, similar to the *snooze* state from TCP-F. In case packets are lost due to error, TCP sender retransmits without firing the congestion mechanism. If the network is in fact congested, TCP sender invokes the congestion control mechanism normally.

ATCP achieves this functionality by listening to network state information. It uses Explicit Congestion Notification (ECN) [37] and ICMP "destination unreachable" to interpret what is happening in the network and perform the proper actions. When ICMP "destination unreachable" messages are received, the sender is put in persist mode. The ECN messages are used to invoke the congestion control mechanism immediately instead of waiting for timeout.

In order to detect if there is a new route already, some probe packets are sent while in persist mode. When ACKs are received for these probe packets, the sender is put on normal mode.

Being the ATCP layer transparent to TCP, ATCP and non-ATCP nodes can inter-operate. Although the non-ATCP nodes will suffer from the problems of using TCP on mobile ad hoc networks.

2.3.3 Explicit Link Failure Notification

Explicit Link Failure Notification (ELFN) [19] is a mechanism similar to the previous works already presented. The route error messages from DSR were modified in order to carry a payload similar to ICMP "destination unreachable" messages. This extra information contains pertinent fields from the TCP/IP headers of the packet that instigated the notice, including the sender and receiver addresses.

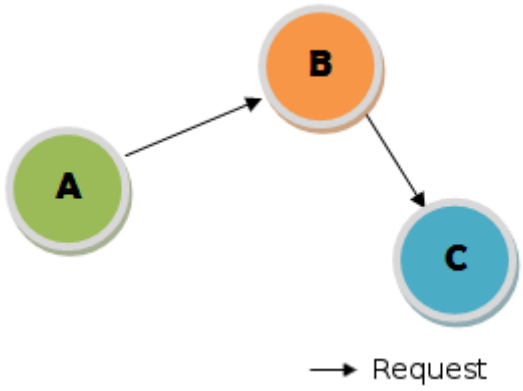


Figure 2.4: Peer-to-Peer Communication

The process when a ELFN packet is received is similar to the ATCP process when a ICMP "destination unreachable" is received. The sender enters in a *standby mode* and disables its retransmissions timers. To get into a normal state, a probe packet is periodically sent, just like in ATCP.

2.4 Content-Centric Networking

Instead of referring a specific location of the data that we are interested in, maybe it is a better approach to focus on the data we need. That's what content-centric networks are all about. We no longer need to know the specific location of the things we want to access and therefore, the data is made available without a specific absolute reference. With this approach, the data may be available in more than one location at a time, creating therefore a whole bunch of new scenarios that can be explored.

One architecture that fits the content-centric definition is peer-to-peer. Peer-to-peer is an architecture of distributed systems, where each node performs either server and client operations [40]. So the functions in the network become decentralized, unlike the usual paradigm of client-server, where we have multiple clients accessing a server. In peer-to-peer networks, we have multiple clients connecting with each other, i.e. any node can perform the functions of both server and client, in case it is serving or requesting another node. For example, in Figure 2.4, node B is a client of node C and at the same time is serving node A, performing server operations. Peer-to-peer makes possible the creation of larger and more reliable networks than other architectures, since there is no dependence on servers, avoiding single points of failure and network saturation as the number of users increases.

Nowadays, there are several peer-to-peer networks already, such as *Kademlia* [29], *BitTorrent* [12] among others. Although these systems are widely used today and allow decentralization of communications, these were designed based on Internet protocols and domain, assuming that there is a direct end-to-end connection. In turn, in ad-hoc networks, it cannot be assumed such a thing, since it is not always possible to have a direct connection between two nodes, implying routing of messages by intermediate nodes.

2.4.1 BitTorrent

BitTorrent is a peer-to-peer file sharing protocol. *BitTorrent* is one of the most used protocol for transferring big files and it is estimated to be the protocol that generates more traffic in all Internet [18]. With *BitTorrent*, we can distribute files without the need of a single source. Thus, when we want to distribute a file, we can just join the *swarm* and eventually the file will be available at all nodes interested in it. The biggest difference between this protocol and other file transfer protocols, is the way how data is delivered. In *BitTorrent*, the data is delivered by passing file parts between *peers*, as the number of *peers* rise, the more bandwidth will be available for the transfer.

For a user to upload a file, first thing to do is to create a *torrent* descriptor file. After that, the file must be sent by any conventional means, like web or email, to interested users. Then, the user makes the file available through a *BitTorrent* node that will act as a *seed*. Those who want to grab the file, just need to give the *torrent* file to their own *BitTorrent* nodes which will act as *peer* or *leecher* to download the file by connecting to *seeds* or other *peers*. When a *peer* finishes downloading the file, it becomes a *seed*.

The file is divided into segments called "pieces". When a peer receives a new piece, it becomes a source of that piece. Thus, a peer can download pieces from seeds or from peers that already have them. Seeds do not need to send all pieces to all peers, because they will be disseminated between them. The torrent descriptor file contains a cryptographic hash of every piece to prevent malicious modifications to pieces passed between peers [13]. With this approach, the distribution of files is shared by the peers who are interested in them.

The torrent file contains metadata relative to the files being shared. It contains also "tracker" related metadata. The tracker is the node that will inform about other nodes in the network from which the peer can download file pieces. So, when a user wants to access a file, first he has to have access to its torrent and pass it to his *BitTorrent* client that should contact the tracker to obtain the list of peers and seeds, connecting to them to download the file.

Although being one of the Internet's most efficient content distribution protocols, *BitTorrent* is not best suited for wireless ad-hoc networks, because of its nature. In a wireless ad-hoc network, a peer is both router and end-user. For that fact, to enable end-to-end communication, sometimes we need that intermediate peers act as routers, forwarding the traffic in the network. Beyond that, TCP performance drops seriously with the number of hops. We can extrapolate that the *BitTorrent*'s overlay, when mapped in a wireless ad-hoc network, it does not produce the best results. Being both *BitTorrent* and Wireless Ad-Hoc Networks, based on Peer-to-Peer paradigm, it seems appropriate to try to combine both concepts. In that sense, there are some investigation in how to change and adapt *BitTorrent* to these networks.

BitHoc is an enhanced variant of *BitTorrent* tuned to ad hoc networks [39]. *BitTorrent* is topology unaware by default. The TCP connections to the neighbors in the network, are established independently of their location. Thus leading to slow TCP connections due to long multi-hop paths and routing overhead. The first solution was to limit the scope of the neighborhood conducting to shorter

download times, but on the other hand, it leads to bad sharing ratios. *BitHoc* modifies the chocking algorithm and adds a new piece selection strategy. Now, the lookup process is no longer granted by a tracker as in the original *BitTorrent* protocol. In *BitHoc*, a peer wishing to find other peers in the network starts by sending *HELLO* messages with a certain *TTL*, creating a table of peers based on the responses. Then, peers are categorized as nearby or far peers, as they are up to 2 hops away or not. The idea is to concentrate the traffic in nearby peers, for example, the piece update message is only sent to nearby peers. Furthermore, a peer elects 3 best uploaders and constantly connects with them to allow a more quickly transfer. One fourth neighbor is chosen in a random way to increase diversity of pieces in the network. Besides that, there is a configurable parameter q, representing the ratio between nearby and far neighbors that the peer will serve on that fourth neighbor. When connecting to far neighbors, the piece selection follows the *absent piece stategy*, only pieces that are not in nearby neighbors are accepted. The piece selection in near nodes follows the *local rarest first* as the *BitTorrent's* standard. Although, *BitHoc* does not take in consideration users mobility.

In [36] the authors compare two different adaptations of *BitTorrent* over *MANETs*: *BTI* and *BTM*. BTI is a straightforward implementation of BitTorrent in MANETs while BTM is a cross-layer adaptation of BitTorrent for MANETs. In *BTI*, there are some adaptations made to the original *BitTorrent*. Both the tracker and seeds can respond to a torrent seeker with the torrent file. The lookup is made with the broadcast of *TREQ* messages and when a *TREP* is received, the node contacts the tracker and it gets the node list and connects to all nodes listed. Then, the client connects with server peers and starts downloading pieces that it does not already have. The unreceived pieces are delivered in a random way. When a client finishes downloading of π will decide to seed a file with probability ρ_s^π for a duration t_s^π. When the download is completed, if the node decides to seed based on ρ_s^π, then it sends a *SEED* message to the tracker. When the time t_s^π is elapsed, the node sends a *UNSEED* message to the tracker. One of the major differences is that *BTM* gathers information on the network, taking advantage of the routing layer. When a new torrent is created, it is broadcasted to neighbors that will re-broadcast it with a given probability. Information from broadcasts is stored locally at every node's cache. There is no tracker to a given torrent, so each node has to find peers by itself. If the information about torrents and/or peers is not present in the local cache, then it is used the same method like in *BTI*, using *TREQ* and *PREQ* messages. These *TREQ* and *PREQ* messages are also used to build a local cache at the nodes that receives them. Another big difference is the use of *proxy seeds*. Typically, one torrent starts with one seed and due to the nature of MANETs, the file may become inaccessible. To fix that problem, during the torrent broadcast phase, nodes which satisfy a given hash function become *proxy seeds*. *Proxy seeds* immediately connect to the initial seed and download the file, becoming seeds to that file. This ensures that there are many copies of the file in several sections of the network. At the end, authors conclude that *BTM* outperforms *BTI* in terms of performance.

2.4.2 Opportunistic Networking

In opportunistic networking no assumption is made with regard to the existence of a complete path between two nodes wishing to communicate [33]. Thus, the paradigm surrounding mobile ad hoc

networks is changed, because we no longer need to concentrate our efforts on building knowledge about the network topology. The communication is achieved by opportunistically passing data to nearby nodes as they are within reach.

In this section we present an architecture and some frameworks and middlewares designed for opportunistic networks. The architecture represents only the concept, the design of a possible system that can be adopted. A framework is a software that incorporates several functionalities in order to be used by another software to ease the design process. Various parts of the framework may be exposed through an API. A middleware is a fully functional software, operating independently, providing services to people or other applications.

An Architecture

In [41] is presented a network architecture called Haggle, designed around mobile users. Nowadays, when we want to share a file with a friend or a colleague, even if both have wireless compatible devices, most often we will just use a USB key flash drive, as pointed out by the authors, representing a *Pocket Switched Networking* scenario. *Haggle* is composed by the following modules as shown in Figure 2.5:

- User Data
- Delivery (Names)
- Protocols
- Resource Management

Now, an application running on top of a Haggle system, does not need to include network protocol functionality on its implementation. The delivery process is made using user-level names, instead of protocol specific addresses, like IP addresses. Besides that, a resource management module is included to calculate cost-benefit for a given task.

User data in *Haggle* is represented through *Application Data Units* (*ADUs*). ADUs are an encapsulation for a data item respecting to an application like a music file, a photo, a message, etc. ADUs are then sent to and from applications and also to and from other *Haggle* nodes. Normally, some user data is linked together to form complex data, like for example, a website with embedded images. Therefore, ADUs can be linked together using a *claim* attribute.

Haggle stores forwarding state in ADUs, taking advantage of their flexibility to be modified, adding or removing fields. *Forwarding ADUs* can contain information like a list of destination names and addressing hints to each destination, a list of nodes that has passed through, security information and some other relevant information. ADU's destinations are expressed using names. For each name it is checked if there is some protocol available in Protocol module, able to deliver the message. Associated to a name, there could be several protocols, like Bluetooth, 802.11, among others. Furthermore, ADUs can contain mappings to addresses, like emails, Bluetooth MAC addresses, telephone numbers, allowing the use of several different protocols from different applications.

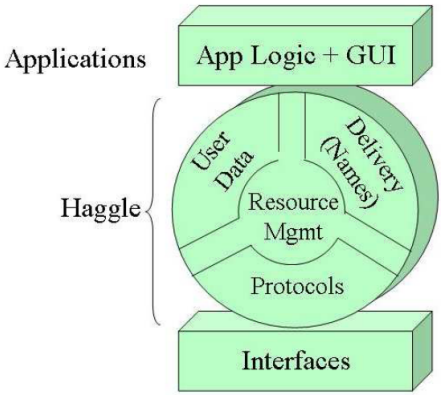


Figure 2.5: Haggle Architecture

To take advantage of connection opportunities, *Haggle* must perform neighbor discovery. One example of that, it is a Bluetooth inquiry resulting in a set of Bluetooth MAC addresses being marked as "nearby". When internet access is available through an access point, all internet domains are marked as "nearby" too. Therefore, forwarding algorithms estimate the "benefit" of transferring a given ADU (or set of linked ADUs) to decide if the ADU is forwarded in that direction or not.

The resource management module is responsible for controlling all use of resources in *Haggle*. This module performs a cost/benefit analysis on "tasks" specified by the other modules to decide what action to take next.

With *Haggle*, applications can exploit all types of data transfer without specifying code for each circumstance. Besides that, network endpoints can be specified by name schemes instead of specific network addresses. The resources are used efficiently by mobile devices by allowing users to prioritize their tasks.

Besides this architecture, it is found in the literature some other works that focus attention on the practical implementation surrounding opportunistic contacts between nodes. That is the case of both 7DS [30] and Cimbiosys [38] frameworks described below.

Frameworks

In [30], authors introduce two new applications to their earlier work, *File Sharing and Synchronization* and *News Sharing in a Bulletin Board System (BBS)*. To create these applications, 7DS has a *Discovery Module* that is built on top of *mDNS* protocol. It enables the creation of an IP network automatically. For that to be possible, *Apple Bonjour*[1] is used. *Apple Bonjour* enables automatic discovery of computers, devices and services on IP networks without the need of DHCP, DNS or directory servers. To take advantage of that system, every information is treated as a service, thus being discovered and delivered easily.

For a user to share content, it starts by defining a shared directory in which all shared objects are placed. The version controller scans the directory to find insertions or deletions to register all

[1]You can get more information online at http://www.apple.com/support/bonjour/

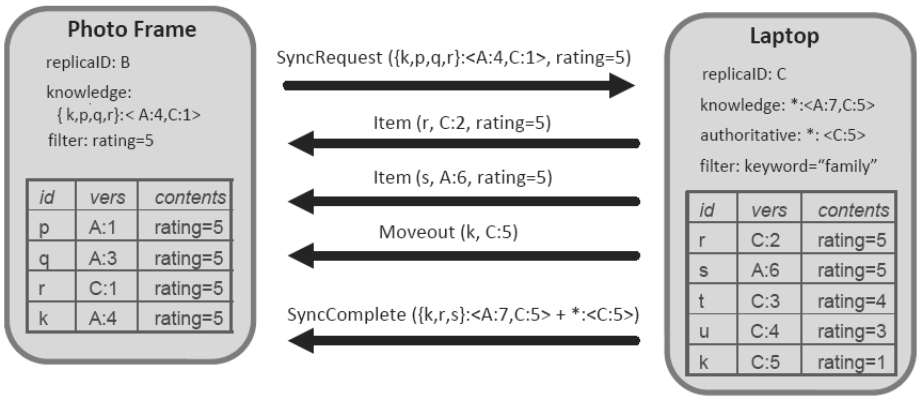


Figure 2.6: Example synchronization between a target replica, the photo frame, and a source replica, the laptop

modifications. To detect modifications, it is used the *JNotify*[2] library. Afterwards, version controller multicasts tuples of hash values and modification dates of each file, using the discovery module, reflecting the changes made. Missing or outdated files are reconciled automatically, the synchronization is transparent to user, being the synchronization request issued by the file-sync module. For the synchronization process, it is used *7DSRsync*, an adaptation of Rsync to 7DS. Unlike file-sync module, the BBS module does not exchange data automatically and transparently to the application. Metadata is exchanged when new mobile users are discovered, based on the category and type of the information, the user issues a download request.

Cimbiosys [38] is a framework for content-based partial replication. The major difference to *7DS File Sharing* is that now, the content is filtered locally, so the replication is only partial. Partial-replication implies new challenges compared to total-replication regarding filter consistency among others. Besides that, some other challenges emerge with this scenario. Authors define 5 key challenges:

effective connectivity ensuring a path between peers for every item

partial synchronization allowing incremental synchronization without wasting bandwidth with duplicate items or metadata overhead

item move-outs informing devices of items that no longer applies to their filter due to recent updates

out-of-filter updates dealing with updates that do not match the updating device's own filter

filter changes allowing a device to change its filter

The system model is designed taking in consideration the challenges presented. Cimbiosys grants two important system properties: *eventual filter consistency* and *eventual knowledge singularity*. The first one states that "Each device eventually stores precisely those items that would be returned by running its custom filter query against the full data collection", that is an important property to ensure consistency. "The state that is transmitted between devices in synchronization requests and is used to identify unknown latest versions converges to a size that is proportional to the number of replicas in

[2]Available at: http://jnotify.sourceforge.net/

the system rather than the number of stored items.", the second property grants a more economical use of bandwidth and system resources. The data replication is based on collections, each collection is composed by items. Each collection has its own access control policy that defines which operations are allowed. A replica contains copies of all or some items of a collection and each device can have various replicas of different collections. A device sharing a collection, keeps its own replica of items. Each item is composed by a *unique id*, a *version id*, a *XML* and the respective *file contents* and a *deleted* bit. Besides that, it can have additional information to deal with merging conflicts. The version id is composed with the replica id and the update count number. The XML contains information respecting to item's file. When a item is deleted it is marked as deleted using the deleted bit.

Each replica has a filter to determine the contents to be stored locally. To ensure security, versions are digitally signed by the originator and policies are used to define create/update/delete properties applied to devices. The synchronization is pull-based. Each device wishing to synchronize its content contacts other replica to perform the synchronization. To ensure consistency between replicas, each replica keeps a version vector representing versions that it knows, and this vector represents an element key in the synchronization process. For a better understanding of the synchronization process, we can look at Figure 2.6. We can see a synchronization issued from replica B to replica C. In this particular case, replica B is only interested in items with rating equal to 5. In this synchronization process, item r is updated since there is a new version *C:2*, while the version in B is *C:1*. The s item is added, since it matches filter of replica B. Item k is removed from B since it is tagged with *rating=1* on the newest version.

The *Item Store* is where all items are stored. Only the most recent known versions are stored. Besides the *Item Store*, each node has a *Push-Out Store* to keep files that are out-of-filter after an update. For example, if in a replica that filters only items tagged as "public", an item A is changed from "public" to "private", it must be stored temporarily before being discarded to ensure that the update is propagated. One of the implementations has been made in C# using Microsoft .NET Framework running on Windows. Another version was implemented in Mace [23]. Cimbiosys exports an API that enables an application to create a collection, create a local replica, add, update or delete items, run queries over items, initiate a synchronization, establish synchronization partnerships, change access permissions and change local filter. In terms of communication, Cimbiosys offers a variety of transport protocols.

Middlewares

There are various middlewares specially designed to operate in ad hoc networks. *Konark* is a middleware designed specifically for discover and delivery of device independent services in ad-hoc networks [16]. *Konark* allows the announcement and discovery of services in the network, thus each node can act as server and client. *Konark* has two main parts, *Service Discovery* and *Service Delivery*. For this to be possible, each device includes a *Konark Application* to facilitate the human interaction with the system and it also includes *SDP Managers* and *Registry* that are responsible for maintain service objects and information about services available in the network. Besides that, a micro-HTTP server is

present to handle service delivery requests. *Konark SDP Manager* is the layer responsible for finding services, register and advertise device's local services. It maintains a tree-based structure as registry, so services are represented in a tree. An example of such a tree is shown in Figure 2.7. As we move down from the root to the leaves, services become more specific. Services can be classified as "all", "generic" or "specific", for example, *(Path=RootService:Services)* means "all" services in the tree. To discover services, an *active pull* mechanism is used. On the other hand, to announce services, the mechanism used is *passive push*. *Konark* supports both mechanisms in order to enable servers and clients to advertise and discover services. To discover a certain service, clients send a message containing two parameters, the *path* to the service intended, and a *port* number. The port number represents the port where the client listens for server replies by unicast. Announcing a service, implies sending a message with the parameters *Service Name*, *Path*, *Type*, *URL* and *TTL*. When a server offers a service, it needs to have a descriptor file of the service using a language loosely based on WSDL [11]. After the service is discovered, a client may want to know more about it and for that the server must have a *service description file* containing complete information about the service properties and methods. After this, the user can now interact with the service by invoking any available function with the proper parameters. The invocation is packaged as a *SOAP* [31] request and sent to the micro-HTTP server. Konark was implemented in two different versions of Java, Personal Java 1.2 and J2ME CLDC/MIDP. The prototype is shown in Figure 2.8.

Another interesting middleware is presented in [17]. The article presents a middleware of mobile peer-to-peer content distribution. Contents are exchanged opportunistically when nodes are in range. The middleware exports a publish/subscribe API to applications. The content is organized in feeds, each feed consisting of a number of entries. Thus, a client can solicit entries from one or more feeds. A feed is represented by a unique id, a title and a timestamp. Furthermore, each feed's entry can optionally have *enclosures* that are single file attachments, typically an audio, video or text file. To efficiently deliver enclosures opportunistically, the enclosures are divided into *chunks*. If a chunk is only partially received from a peer, it is discarded. Chunks from a enclosure can be downloaded from different nodes and the download can be resumed in a different node from where the initial transfer started.

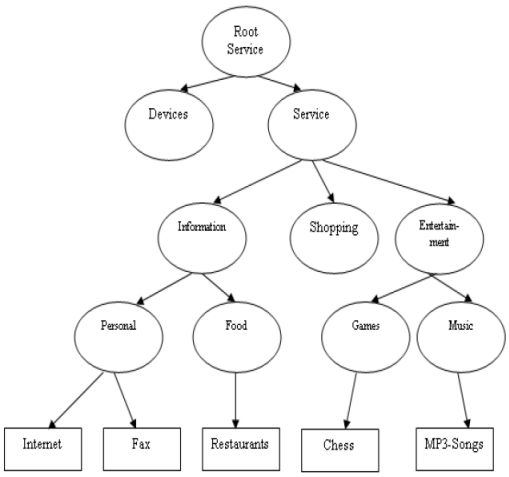


Figure 2.7: Example Service Tree

Figure 2.8: Konark Prototype

For the discovery process, the discovery module searches for neighbors running the service and decides which ones are feasible to associate with. The notification for this module, is constituted by a *node-ID* and *revision*. Thus, each node broadcasts *hello* messages containing *node-ID* and *revision*. The revision is incremented whenever new content is added to the database. This helps peers to decide if it is beneficial to re-synchronize with a certain node, avoiding contacting neighbors just to know if there is new content. Synchronization manager is responsible to mediate applications and API, prioritizing content solicitations ensuring that different applications get a fair share of network resources.

Transport module implements the request-reply protocol to download and discover node's available content. Protocol messages are in XML, being one of these: *hello*, *request*, *reply* and *reject*. When a node discovers a new peer, it sends a *request* message containing a search for a feed or metadata to discover content availability. The peer sends a *reply* message, establishing the session and responding to the query. Processing a request consists only in verifying if the content or metadata is available, and deliver them. With this approach, each node has full control on the content it downloads. The server is stateless, so all replies are independent of any other previous requests. The content solicitation is completely based by the *pull model*. Session switches between *discovery* and *download* states. A node can have multiple simultaneous sessions, operating as client (while downloading) or server (while uploading) for each session. A *request* message contains the *bloom*, *selector*, *feed*, *entry* and *chunks* elements. A *bloom filter* is populated with feed IDs and entries available in the local node. When a node receives a request with an empty bloom element, it delivers its own bloom filter in the reply message. Then, the client tests the IDs of its subscribed feeds or partially downloaded entries against the filter to discover if there is content of interest in the respective node. If the client does not have knowledge about the feeds available, or wishes to find more feeds, a *selector* can be used. When a node receives a request with a selector as top-level element, it evaluates the selector against its feeds properties. The feeds for which the selector evaluates to true are sent in a reply message. In the same way, a selector specified inside a feed will be evaluated against the entries of the feed, returning the entries that evaluate to true in the reply message. An empty selector will match all feed/entry elements.

Messages are delivered in a best-effort basis. It is not guaranteed that feed's entries are delivered in the same order in all receptors. The implementation was made in Java for the Google Android OS[3] platform based on 802.11 in ad-hoc mode.

2.4.3 Social-aware Networking

Socially-related people tend to be co-located quite regularly. This is the characteristic explored in *SocialCast* [14]. *SocialCast* is a publish-subscribe framework that explores predictions based on social interactions. The key is the assumption that users with common interest tend to meet more frequently than with other users. Routing in *SocialCast* is made in 3 phases:

- Interest Dissemination
- Carrier Selection
- Message Dissemination

On the *Interest Dissemination* phase, each node broadcasts its interests to 1-hop neighbors, along with the corresponding utility values list of the node. The utility information is stored on the neighbors routing tables and is the key element on the decisions around message forwarding. After that, in *Carrier Selection*, the utility of the local node n is recomputed to all interests i. Then to all interests i is compared $U_{n,i} > U_i + \epsilon$, being the U_i the utility value for interest i of the local node, and $U_{n,i}$ the biggest utility value announced by the neighbors to interest i that belongs to node n. The ϵ is a threshold for prevent the message to be bounced from node to node. If in fact $U_{n,i} > U_i + \epsilon$ is verified, then the node n is a better carrier to interest i than local node. Next in *Message Dissemination*, the buffer content is reevaluated against the new subscriptions and utilities and the messages are forwarded to interested nodes and/or to the best carrier. Thus, each node interested will get the messages, but that does not mean that it will store it on its own buffer. The *Message Publishing* consists in just storing the published message on the local buffer. The utility is in general a function of multiple attributes representing the different dimensions of the problem like mobility, collocation, battery level, etc. The primary utility attribute considered is the probability of a user to be collocated with another sharing the same interest.

Another similar system is *ContentPlace* [6]. *ContentPlace* makes the same assumptions as *SocialCast*, learning and exploiting information about the social behavior of the users to drive the data dissemination process. *ContentPlace* does not adopt any specific technology (802.11, Bluetooth), the connections are made pair-wise between nodes opportunistically, as adopted by *SocialCast*. Furthermore, the utility is applied to data objects and not to nodes relative to interests like in *SocialCast*. When a node contacts another one, it evaluates utility for each data object of the peer. Assuming a limited buffer, the node will select what are the data objects to be fetched, to maximize the total utility value of the data on its own buffer. Although the similarities, the authors refer some differences when compared to *SocialCast*. According to authors, *ContentPlace* uses a utility function more complete

[3]What is Android?: http://developer.android.com/guide/basics/what-is-android.html

then the one used in *SocialCast*. Besides that, authors argue that the mechanism used in *SocialCast* is more oriented to traditional forwarding than actual dissemination with respect to *ContentPlace*. Furthermore, it is pointed the fact that *SocialCast* works well when all members from a community are interested on the same type of content, but is not clear how it works in more general settings.

Apart from the differences cited before, the mechanism is similar in both works. The key idea is to take advantage of people social behavior to disseminate information in a more efficient way.

2.5 Conclusion

Throughout this chapter, several works around mobile scenarios were presented, showing there is a vast investigation on this topic.

In order to successfully forward traffic between nodes, there are many routing protocols specially designed to operate on mobile ad hoc networks. There are different approaches to route traffic on mobile scenarios. Almost every routing protocol can be classified as proactive or reactive on-demand. The location-aware bases the routing process on the physical location of the nodes, instead of basing on the network topology. In order to center the routing process on the content itself, there are some content-based routing protocols that decouple content consumers from producers. There are several different approaches to route traffic between nodes. Using one of these approaches or combining different ideas, we are able to build applications that can communicate with nodes of the network that are not in range.

Having a routing protocol to forward traffic in the network is not enough if we are interested in achieving high performance. On section 2.3 it is presented some problems of using TCP on mobile ad hoc networks. The TCP protocol was initially designed to operate on wired networks. Some problems arise on mobile, multi-hop scenarios due to TCP intrinsic characteristics. The performance and adaptation of TCP to this environment are discussed on several different works. UDP does not suffer from the same problems from TCP, but the last is more studied because the most popular applications use TCP as the transport protocol [3]. Having this knowledge in mind, we know that unmodified TCP is not adequate to build our applications. We can choose from adapt TCP or just use UDP.

Some works focus their efforts in adapting peer-to-peer protocols already available, that were designed to the Internet, like *BitTorrent*, to the mobile ad-hoc topology. Another work proposes an ambitious architecture surrounding the mobile scenario, where the addressing is made using conventional names. In Haggle architecture, a destination of a message can be defined using a name, e.g. John Smith, and then a module is responsible to choose the best protocol from the available to deliver the message.

Some of the principles presented in the Haggle architecture are also explored in other works. *7DS* introduces a file sharing and synchronization module able to share and synchronize content opportunistically between mobile nodes and presents a way of creating a network automatically without the need of *DHCP* or *DNS*. At this point it is also presented a News Sharing module that uses metadata

to classify content allowing users to choose the news of interest. To allow users to choose content of their interest, partial-replication is approached in *Cimbiosys*. Now, each node will have a filter representing the data that they are interested in. The data is now grouped in collections, and besides that, some security measures are issued through polices and digital signatures.

In order to allow the creation of new applications specially designed to the mobile scenario, some middlewares are proposed like *Konark* and [17]. In [17] the content is grouped into feeds, and files are divided into chunks, to enable the proliferation of the contents in every opportunistic contact. Chunks are similar to pieces used on the *BitTorrent* protocol. The application was built for Android phones. On the other hand, *Konark* focuses in services in exchange of data, the services are hierarchically organized in form of a tree enabling the search for a more generic or specific service going from the root to the leaves of the tree.

Normally, users with the same interest tend to be in touch more often than with other users. *SocialCast* and *ContentPlace* based their work in that and other assumptions, regarding human social behavior to opportunistically disseminate content to potential good carriers in order to reach more interested nodes.

At this point, we can conclude that is still possible to use services that were initially designed to the wired network, or the Internet, on the mobile ad hoc network. Besides that, a whole bunch of new scenarios are available to explore, that are becoming more relevant as the devices are becoming more mobile and more common.

Chapter 3

System's Architecture

The main objective of our system is to provide content to the final user. To achieve that, we decided to create a content-centric network. The network itself focuses on content and not on client addresses or other parameters.

In order to implement a file sharing system fully functional on top of an ad-hoc network, several modules must be implemented and interconnected. The architecture of the system is presented and described below.

3.1 Overall Overview

The architecture is composed by different modules represented on Figure 3.1. The Mini HTTP Server represents the presentation layer. It provides the interface to allow users to interact with the system. This module interacts with the API module to perform actions issued by the user. Besides, it also includes a visual setup module interface. That interface is used when the application runs in server mode to set the files available and the layout used to present the content. The API connects with the File Module to get and set information about the files available. It also interacts with the Transfer Module, that is responsible for the file transfer process. The Transfer Module has to interact with the Neighborhood Module and the Routing Module in order to properly route the transfer requests. Another auxiliary module was implemented in order to create logs for the application.

The global system was implemented in Java. This enables the interoperability between different devices and Operating Systems.

3.2 Solution Requirements

Our architecture fits on the definition of peer-to-peer. Each node executes both client and server functions. Although it is not similar to the BitTorrent architecture. BitTorrent protocol assumes the existence of a network layer of peers that are all interested on the same content. Besides, it assumes end-to-end connection between peers. Only the nodes interested on the content, receive and send

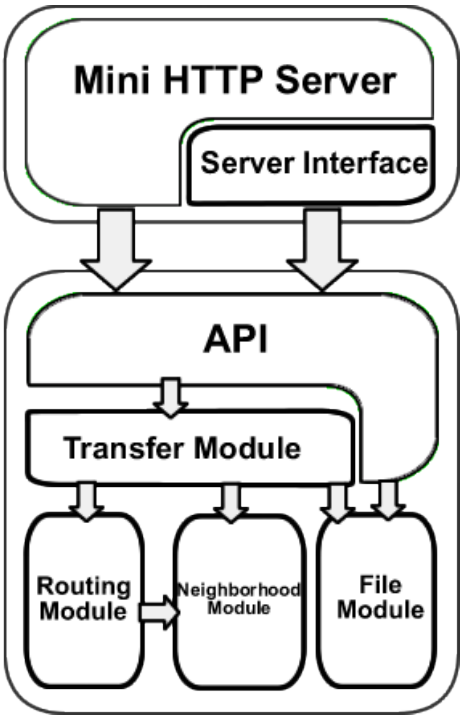


Figure 3.1: System's Architecture

data. The intermediate nodes are not considered, because end-to-end communication is assumed. This implies that intermediate nodes that are not participating on the BitTorrent protocol, are not able to cache content. The caching process would allow the node to already have data from the content. In case the node is now interested on participating on the protocol and access the data, maybe the content as already passed through it. There are some adaptations to the BitTorrent protocol to operate on ad hoc networks, like BitHoc, BTI and BTM. Even with these adaptations, the protocol is still based on TCP connections. Besides, only the nodes that are participating on the transfer, sharing or downloading the content are considered. The mobility of the ad hoc network is not considered.

The goal is to build an application where the contents are delivered on-demand. Each user is able to consult the contents that are available and choose individually each ones that it is interested in. Considering the mobility and the TCP problems, we decided to do not adopt a protocol similar to BitTorrent. Being the files requested on-demand, opportunistic networking is not enough to suffice our problems. In opportunistic networking, the communication is based on contacts between direct nodes to disseminate the content. These contacts are not enough to deliver content on-demand on our scenario. In our approach, each user explicitly defines the content that it wants to access. In case we adopt an opportunistic approach, the node has to subscribe the content and wait for another node having the content to step in the neighborhood. But we want to deliver the content to the user immediately as requested. Besides, the neighbors may have different interests and so the content becomes unaccessible, since the communication is based on direct communication. The file may not be available on the local neighborhood but is available somewhere around the network. Routing data between peers seems the best approach to deliver on-demand content. Having this in mind, we decided to exclude opportunistic networking along with the publish-subscribe paradigm from our architecture. Although, the majority of the architectures for ad hoc networks to deliver content are based on the

publish-subscribe paradigm, centering the communication only on direct one-to-one contacts between nodes, we think that this is not the best approach to deliver content on-demand.

Our system assumes there is IP connectivity. We assume that each node of the network already has a valid IP address and connectivity to other nodes. Ensuring that each node has valid IP address and connectivity to other nodes represents a totally different problem that we do not focus on this work.

Another assumption is that there is only one instance of the system running on each network. For example, if there is an ad hoc network "Conference XPTO VII", we assume that there is only one set of files available to share on this network. A way of enabling more than one instance of the application running on the same physical location is to create another ad hoc network. Like having, for example, an ad hoc network "Conference XPTO VII contents" and other "Coffee Break Funny Videos". Although the system is designed to support only one instance of the file-system, if someone attempts to introduce another one on the network, theoretically the system is able to keep working. The consequence of this action is that different nodes know different files. The set of files that each node will know depends on the data that is received on the starting process. For example, imagine that there are 2 different file-systems made available on the network, X and Y. When a node A starts the application, in the starting process it receives the set of files from a neighbor that listens to the start messages broadcasted by A. Suppose now that there is a node B knowing the set of files X and another node C knowing Y, both neighbors of A. If B answers first to A, A will know the set of files X. Otherwise, if C answers first, A will know Y instead of X. The starting process is explained on Section 3.4 on page 38. Although nodes know different files, both file-systems are available. The disadvantage is that each node is not able to choose the set of files it wants to access.

We assume that once an ad hoc network is made available with the server defining the content available, the contents are not changed over time. We assume static content from the time the system starts until it ends. Some of the reasons of this choice are further discussed on page 38.

3.3 File Module

To enable the file sharing in the network, a module must be responsible for reading and writing files. In that sense, this module is responsible for every action related with disk access, both read and write operations.

In order to enable the transfer of big files over the network, files must be splitted into parts of smaller size. This is necessary, since all the bytes must be sent over the network. If the file is too big, there is no chance of sending it without splitting. If we attempt to send a file as a continuous byte stream, for example, over TCP, some problems arise from this strategy. In case there is a connection error, file must be retransmitted from the beginning. In order to transfer the complete file with this approach, both sender and receiver device must have memory available to store the file's byte array, while sending and receiving some file. Being the transfer very long, it will imply that file will occupy the memory for too much time, slowing down the system's global performance. In case there's an

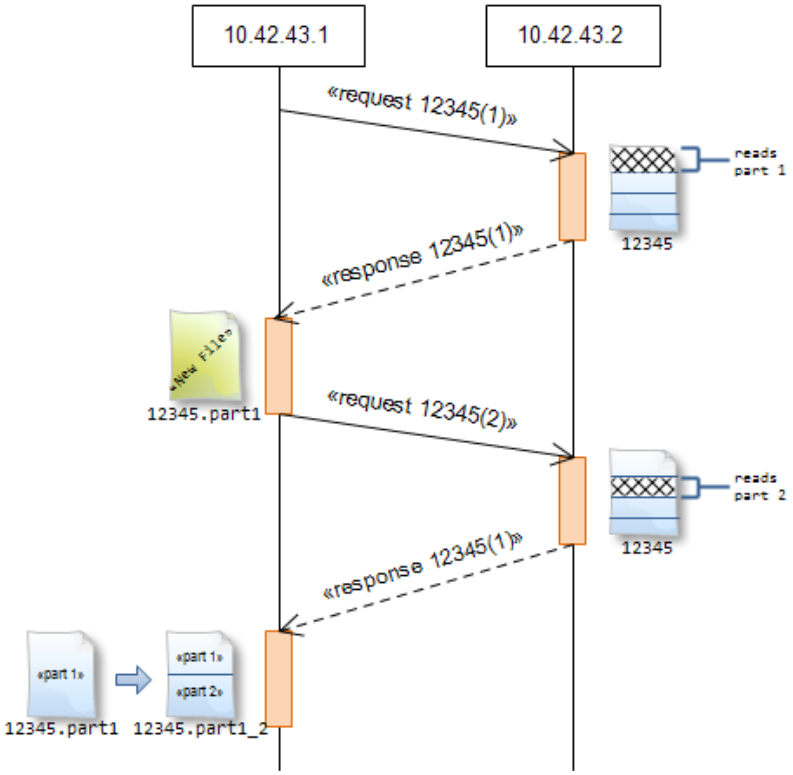


Figure 3.2: File part transfer

error and a retransmission is needed, file must remain in memory, until transfer succeeds. The issues presented make clear the need to split files into smaller parts.

When a device needs to download some given file, it will not request the file directly, but will request instead some file part. For example, if there is some file that is divided in 10 parts, the device requests file part 1, then part 2, etc. until it has all 10 file parts, merging them and saving to disk the resulting file. Figure 3.2 ilustrates the process of downloading a file, requesting file parts. The transfer process using file parts is further explained on section 3.6.

On the server's side, the process is quite simple. There is only one file, and for each part it reads 'X' bytes of data, shifting 'X*(P-1)' bytes before reading, being 'X' the size of each part, and 'P' the part number. For example, if X = 1024 bytes and device requests file part 1 (P = 1), then the server reads 1024 bytes from the file, or less case reaches end of file, shifting 1024*0 bytes. There is no shift because it is the first part of the file, so it reads 1024 bytes from the file's beginning. In case the device wishes then the second part (P = 2), server will read 1024 bytes, but shifting another 1024 bytes first.

Upon receiving a file part, the device has to write to disk the received data. The name of the output file is composed by the file id and .partX, where X represents the part number. For example, if the file id is "12345" and the downloaded part is "2", then the resulting output would be a file named "12345.part2". This is done to ensure durability on the parts already transfered and free memory space to download new file parts. These files are temporary, being merged on a single file upon transfer completion. This allows us to request and receive parts out of sequence. In order to reduce the number of file parts written, consecutive parts are aggregated in the same file. This process does not need extra processing, since the new part is appended to the end of the preceding part file.

Assuming that we already have "12345.part2" file and we have just fetched part 3 of file 12345, then byte array from part 3 is aggregated in "12345.part2" and finally the file is renamed to "12345.part2_3" as seen on Figure 3.3(a). A file "fileid.partX_Y" contains byte array from all parts between X and Y. So, when receiving a file part, the first step before writing is to scan for a preceding file part. If we receive part 4, the first step is to look for "fileid.part3". In case the part exists then part 4 is aggregated with "fileid.part3" resulting in a file named "fileid.part3_4". In case "fileid.part3" is not available, the next step is search for a file named "fileid.partX_3", being X any part number prior to 3. Part 4 is then aggregated with "fileid.partX_3" resulting in a file named "fileid.partX_4". An example of that scenario is shown on Figure 3.3(b).

The reverse process does not apply, because it is not efficient. For example, if we have part 5 and we receive part 4, then if we want to aggregate both, we need to write part 4 to disk and then read part 5 from the beginning and append to part 4. In that case, a new file part is created as shown on Figure 3.3(c). More critical scenario would be having, for example, part 5 to 10 aggregated in file "fileid.part5_10". The aggregate process here would imply the read of all parts, from 5 to 10, to aggregate with part 4.

The files are identified by a Universally Unique Identifier (UUID) [26]. When a file is added to the system, an UUID is generated. This can be easily implemented by using the **java.util.UUID** package. Being the identifier generated randomly, it can happen that 2 different files get the same UUID. To calculate the probability of generating duplicate UUIDs we can use the probability theory (Birthday Paradox) using the approximation from equation 3.1.

$$p(n) \approx 1 - e^{-\frac{n^2}{2x}} \tag{3.1}$$

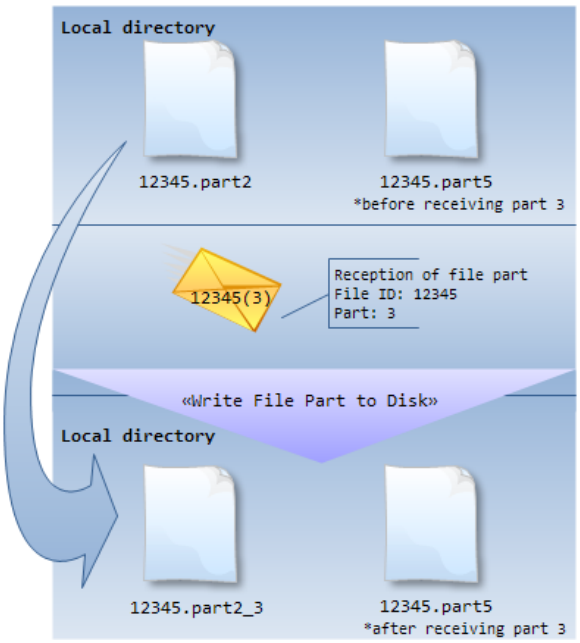
The UUID is composed by 122 random bits and more 6 bits to identify version and variant. We can now replace x with 2^{122}. Even when n is a huge number, the probability of duplicate UUIDs is almost null. So we just assume that each generated UUID is unique.

In order to best organize the files, they are separated in categories. For example, if we have a video file, it can be put on the "video" category. Similarly, an MP3 file can be put on the "audio" category. When a file is downloaded, it is saved on a folder according to the file category. With this approach, each user can navigate to the folder where the files are stored and view them without using the application interface. This way, the application can be used even without a presentation interface. Just using the API, the download can be issued and the file will be put on the proper folder.

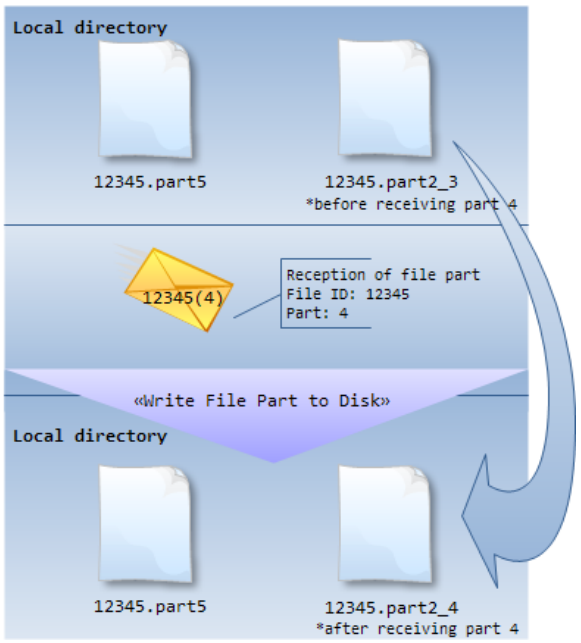
The filename is equal to the UUID of the file. This is done to easily map files location. For example, if we started the download of file "funny.mpg" with id "f646dea6-1e18-4d8c-9022-0fb1b83cc565" being of type "video", when the download is finished, the file is stored in "videos/f646dea6-1e18-4d8c-9022-0fb1b83cc565". The original filename is kept on the file descriptor object.

Besides the type id, each file descriptor includes the number of parts and file size represented in bytes. Besides, a title and a description can be set.

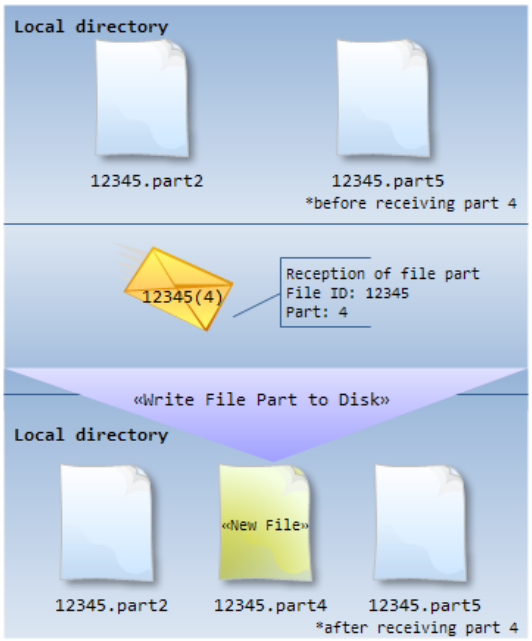
Having a file module to deal with files to transfer them over the network represents a starting point to build our application. After we have this module fully functional, we can now start building



(a) Writting file part 3



(b) Writting file part 4 first scenario



(c) Writting file part 4 second scenario

Figure 3.3: Writting file parts

a module that can communicate with other nodes in order to successfully transfer data between each other.

3.4 Neighborhood Module

In order to gather information about the neighborhood, our application includes a neighborhood module. This module is responsible of finding other nodes, announce its presence to the neighborhood and store information about its neighbors. With this module we have a proactive approach, similar to the proactive routing protocols, presented on Section 2.2.1 on page 8. The difference is that the proactive part is restricted to the direct neighbors. This approach is also similar to having a ZRP protocol [5] with $\rho = 1$. The reactive part is ensured by the route module.

Peer Discovery For peers to know each other, there must be some mechanism to announce themselves. In that sense, this module implements an HELLO mechanism, where each node, frequently broadcasts an HELLO message. These messages are used to build a local list of neighbors. HELLO messages are composed by a "database id", "update port" and "transfer port". "Database id" represents the sender's database version. Each time one node transfers or deletes a file, its local database id is incremented in order to reflect that change to local neighbors. Upon receiving an HELLO message, a node starts by fetching the address of its sender. In case the address is not already known, it is added to local neighbor list. The timestamp for the respective neighbor is updated. Then it is checked if the database id is outdated. When the database id received is greater than the local one, an update request to that neighbor is issued. That is done to update the list of available files of the neighbor in question. With this approach, each node knows which files are available at any of its local neighbors. The process of receiving an HELLO message is described on the diagram represented on Figure 3.4.

The "update port" and "transfer port", just represent the ports where the node is listening for update and transfer requests respectively. With this information being broadcasted, each node has all the information needed to interact with each of its neighbors.

Update process For the update process, each node maintains history of added or deleted files from local database. The history is maintained up to 2 latest database versions. When a node receives an update request, it starts by comparing the database version known by the update issuer with its own database version. In case the difference is only 1, then it just sends the file identifier that was added or deleted in the new version. When the difference is 2, it replies with the file identifiers that were added or deleted. If the difference is bigger than 2, it replies with all file list available. This is done to prevent the overhead of sending the file list, each time the database is updated. In Figure 3.5 is represented the diagram of the update process on the serving node.

In Figure 3.6(a) an update request is represented, issued after receiving an HELLO message with a "Database version" bigger than the latest known version for that neighbor. Node with address 10.42.43.2, issues an UPDATE request to 10.42.43.1, because it knows version 1 while node 10.42.43.1

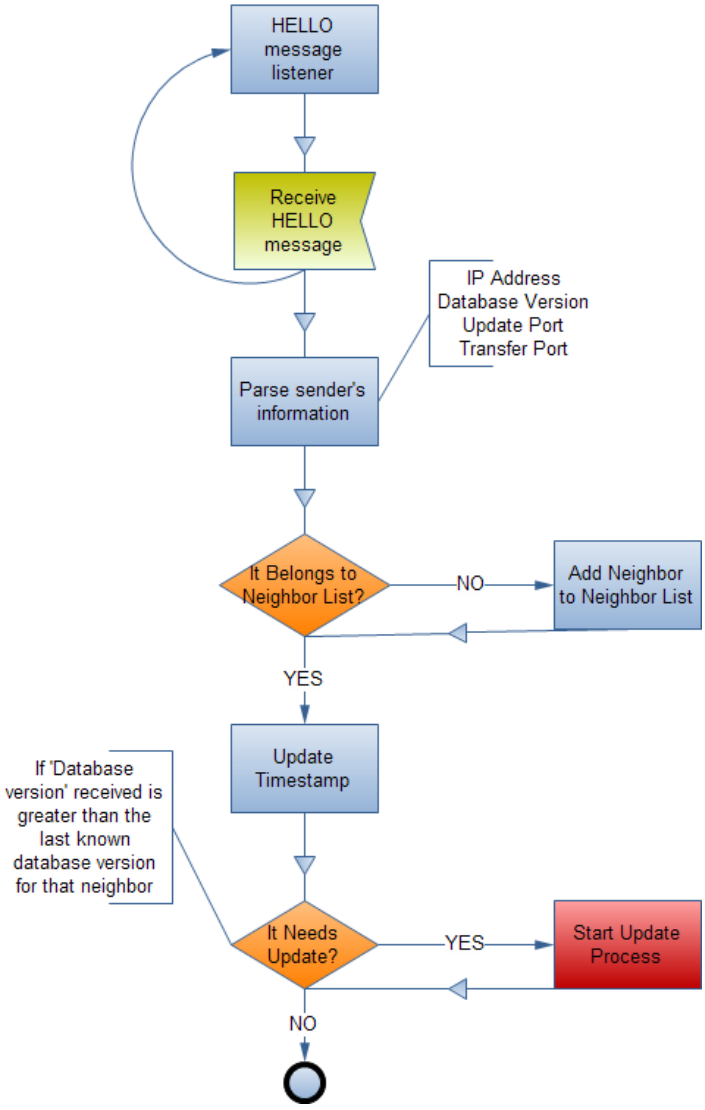


Figure 3.4: Receiving HELLO messages

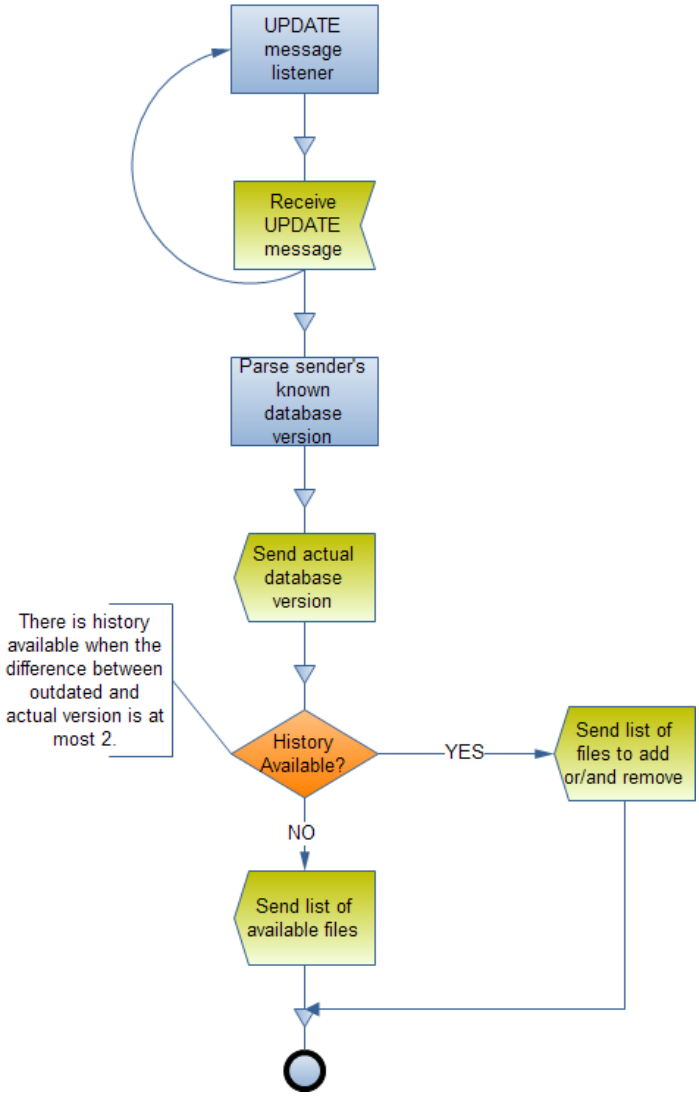


Figure 3.5: Receiving UPDATE messages

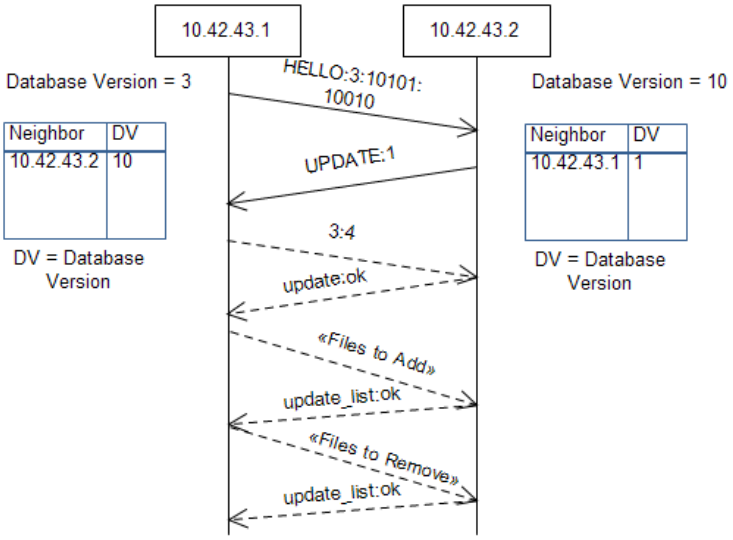
Code	Action
0	update database number only
1	update entire list
2	remove elements
3	add elements
4	add and remove elements

Table 3.1: Update control codes

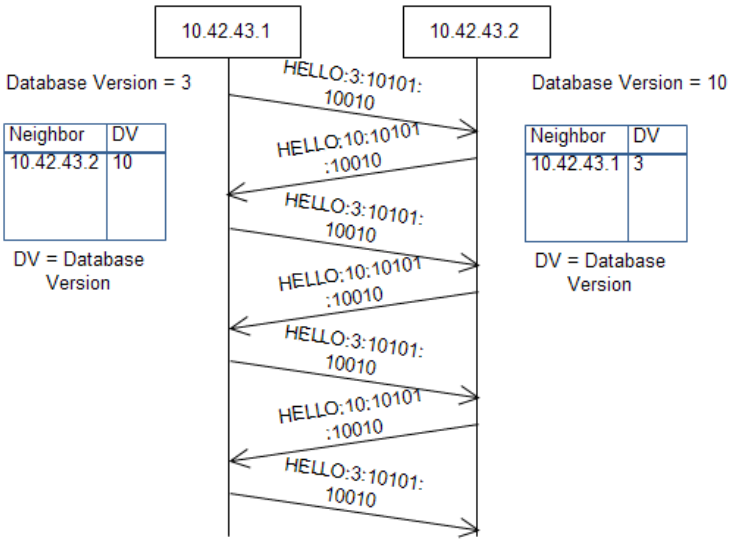
is announcing version 3. When node 10.42.43.1 receives the request, it responds with the actual "Database version" and a control code. The control code is used to announce the type of response that will be sent. The list of control codes is shown on Table 3.1. To confirm the reception of data, some confirmation messages are used, like "update:ok" and "update_list:ok", as shown in Figure 3.6(a). When the update succeeds, each node will continue to broadcast HELLO messages. This scenario is shown in Figure 3.6(b), where we can see HELLO messages being sent without receiving any response, because there is no need for an update.

Starting process Before each node starts sending HELLO messages, announcing its presence in the network, first it broadcasts START messages. START messages are used to initialize the node with information about the files available on the system. When some node receives a START message, it replies with the information needed. With this approach, each node will be part of the network only when it already has all the base information needed. This base information includes available files in the system and the layout of the webpage. The process is illustrated on Figure 3.7. These messages are used to initialize each node with the list of files that are available and respective information, including type and byte size. Another interesting information fetched is the webpage layout. The layout is only used when the HTTP module is used as the presentation layer. Although the system can be used with different applications and interfaces, it was designed having the HTTP presentation layer in mind. The layout defines how the contents are displayed on the webpage. If the system is used without the HTTP presentation layer, this information is useless.

This process is only issued one time when the application starts. We assume that the contents are static from the time the event starts until it ends. If the content is changed, the application must be restarted on all nodes. When a node finishes the starting process, it can now interact with other nodes. All nodes are treated the same way. We can say that the server node, the one that defines the layout and the set of files, is just a node having all files. When another node downloads all files, it can be said that is equivalent to a server. With this approach, when the application starts, there is no difference between server and client nodes. We just have nodes sharing content. This represents a simplistic approach to share the content. In order to allow new content to be added later, this process should be issued more than one time. To allow new content to be added when the application has already started, a bunch of new problems and questions arise. The first question is: who can submit new content? The way we designed our system, there is no server and there is no client, we just have



(a) HELLO with UPDATE sequence



(b) HELLO sequence

Figure 3.6: Examples of HELLO interactions

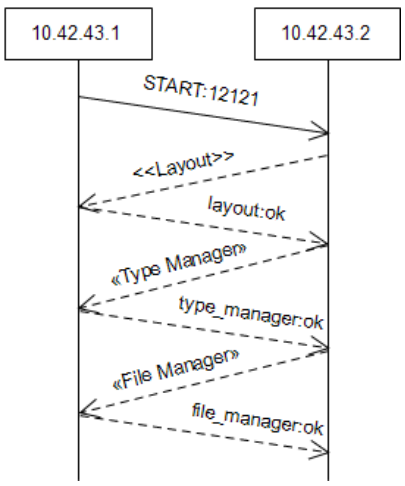


Figure 3.7: START process

nodes sharing the content between each other, this can be seen as a static shared file-system. If we allow each node to submit content, this can represent a big security issue, as the application has no control on the content that is being spread across the network. Otherwise, in case we just want to grant access to the server to submit new content, other problems arise. How to identify the server? How to avoid normal clients attempting to identify themselves as servers? Using cryptography? Even if we are able to properly identify the server, how can we guarantee that the new content is known in the network? Some consistency problems must be dealt. We need a trade-off between availability and consistency. Another possibility would be having a user database on the server defining each individual person that can insert new content. This would imply that the server is reachable by the nodes every time. Some cryptographic mechanisms would be needed, in order to authenticate each user.

Although the simplistic approach we adopted of having a static file-system, the system could be adapted in order to allow file changes, as modifications, insertions and deletions. This adaptation implies that the existing modules needs to be changed and new modules need to be created to support these changes.

With this module, we can now communicate with the local neighborhood. Now we can 'talk' with every neighbor that is in range. This is the starting point in order to download files between each other. However, this module is not sufficient to communicate with the rest of the network. It only allow us to communicate with direct neighbors, nodes that are in range. The network can be very large and include several nodes that are not in range. Although some nodes may be unaccessible directly, maybe they can be accessed by communicating with intermediate nodes. This is very important, because the data we need may not be accessible in the nodes that are in range. Concluding, we need some mechanism to interact with other nodes of the network, not just the ones that are in range.

3.5 Route Module

The file sharing process becomes very simple when all nodes are in range with each other. That may not be always the case, because the network can be very large. Each node will have only contact with a small part of the network. When a given node wants to download some file, it verifies if the file is available on its range, i.e, if there is some neighbor with the intended file. Case the file is available, the process is very simple. The node connects directly to its neighbor soliciting the respective file. For example, some device "A" with address "10.42.43.1" wants to download file with id "12345". There is a node "B" with address "10.42.43.2" in range having the file. Device "A" makes a connection to "B" requesting the file and "B" sends the file directly to "A". When the file is not available in the neighborhood, there must be some mechanism to find the file from others peers in the network. Besides finding others peers in the network, we have to discover routes to interact with them. If we find a node that has the file that we need, that is not enough, because we need to know how to contact with it.

The route discovery process is issued on-demand. When a node wants to download a file, if it is not available on the neighborhood, route request messages are spread through its neighbors to discover

routes to the file intended. The routing approach is similar to the DSR protocol. Although, the route requests messages are unicasted to each neighbor and not broadcasted. In DSR this cannot be done because the protocol is totally reactive. Having a neighborhood module, the system is able to identify direct neighbors and broadcasts are avoided by unicasting requests to the neighbors. Each node knows the files that are available at the neighborhood. This way, upon receiving a route request, the node already knows if the file is available on its neighbors or not. So the node that receives a route request, in case it knows a neighbor having the file, it responds with a route reply without forwarding the request. Typically, the server node does not receive the route request, because its neighbors respond with route replies. Another major difference between this routing protocol and the ones presented on section 2.2 on page 8 is that the protocol searches for routes to access files and not specific addresses. Content-based routing is another protocol that bases on content and not on addresses. Producers and consumers are decoupled and the content is disseminated based on subscriptions. In our system, this is not suitable, because the files are already available and are requested on-demand. That is not the case of content-based routing. Another evident difference is that there may be more than one destination. We are looking for files and not nodes, so the route request process may generate different routes to different destinations. This allows each node to adopt the best route and choose the best destination. This can be done, for example, choosing routes with smaller hop-number, resulting in closer destinations. Besides, in case there is more than one destination and in case the routes are disjunctive, i.e. the intermediate nodes are different, each node can solicit content to different destinations simultaneously. This approach also allows for an easy implementation of caching mechanisms, because the routes in cache point to content and not addresses. When an intermediate node receives a transfer request to be forwarded to the destination, it looks to the route and knows which file is requested. With this information, each node is able to pro-actively or reactively cache file parts. For example, each time a node receives a route request for a given file, it can store information about the frequency each file is requested. With this information, each node can detect which files are more solicited by the network and based on this information, pro-actively cache the content. Our routing protocol represents a new approach to route traffic that suits best our system model.

Route requests are re-transmitted upon reception to local neighborhood, unless there is a neighbor having the file. To avoid network flooding, each message has a TTL parameter that is decremented before it is broadcasted. When TTL reaches zero, the route request message stops being rebroadcasted. If there are no responses after sending a request, TTL is incremented and sent again until it finds some route or TTL reaches its maximum value. In order to avoid cycles, there is a cache of recent requests.

A node that receives a route request, first, it compares it against its local cache. If the request was already been served, it is discarded. After checking the cache, the node verifies if there is some neighbor for that file. In case it finds, it creates a route reply and returns it to the client. In order to know the route to the node and to successfully deliver the route reply, each time a request is rebroadcasted, the address of the node is added to the request message. For example, suppose that there is a device “A” with address “10.42.43.1” wishing to download some file with id “12345”. Device “A” has only one neighbor, device “B” with address “10.42.43.2”. Device “B” does not have the file “12345”. A route request message from device “A” is then sent to “10.42.43.2”. Upon receiving the request, device “B”

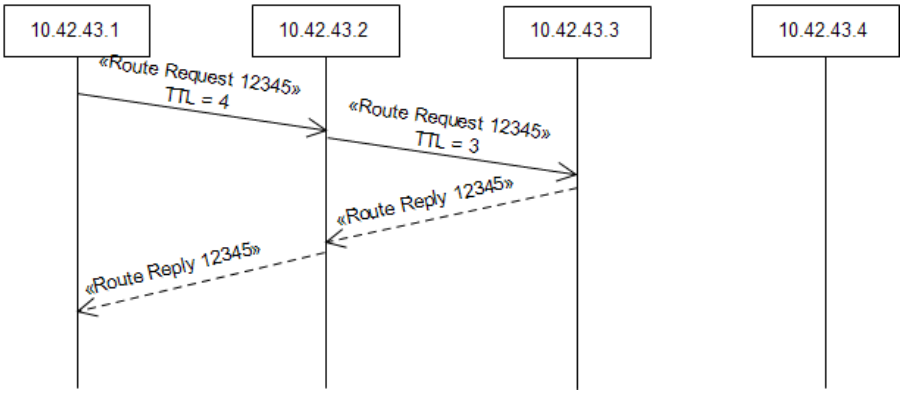


Figure 3.8: Requesting Route Example

checks if it knows some node having the file. No neighbors have the file, so "B" adds its address to the message request and sends the message to another neighbor "C" with address "10.42.43.3". This time, the device "C" knows a device "D" with address "10.42.43.4" having the file, so it adds its address and the address of "D" to the message. Then, device "C" responds with a route reply, based on the message request that already contains the addresses of the nodes it passed. So, device "C" already knows that it has to send the route reply to "10.42.43.2". Device "B" receives the reply and it knows that it has to send it to "10.42.43.1". Finally, when "A" receives the reply, it knows that the file is available at "10.42.43.4" passing trough "10.42.43.2" and then "10.42.43.3". This example is illustrated in Figure 3.8.

Upon receiving a route reply, each node (final or intermediate) stores the route for the specific file in a route cache. Each route is stored for a limited time. When some route is used, its timestamp is updated at every node it passes through. If a route is not used, it gets expired and deleted from route cache. This cache is used to allow each node to download a file without issuing a new route discovery process in case there is a route on its cache. Besides, it allows each node to reply to route discovery requests with a route already discovered before and stored in cache. The time each route remains in cache is critical and should be correctly parameterized. In order to successfully parameterize this value, further tests are needed to discover optimal values.

On the route discovery process, if a node has a route for the file on its cache, it responds with that route. That stops the flooding process and reuses valid and already discovered routes. The process described above is shown on the diagram in Figure 3.9.

The routing module operates at the application layer, this way our application works without depending on a routing protocol working at every node. For example, OSPF-MDR [32] enables routing between nodes on a MANET. To enable routing between the nodes of the network, each one needs to have this module installed and running. One of the main objectives of our system is to work off-the-shelf without needing to install or run additional software. We achieve that implementing our own routing mechanism that does not depend on other software to operate.

At this point, we can deal with files that are the base of our application using the file module. We can communicate with neighbor nodes using the neighborhood module. With this new module, we can now communicate with other nodes of the network that cannot be accessed directly because of

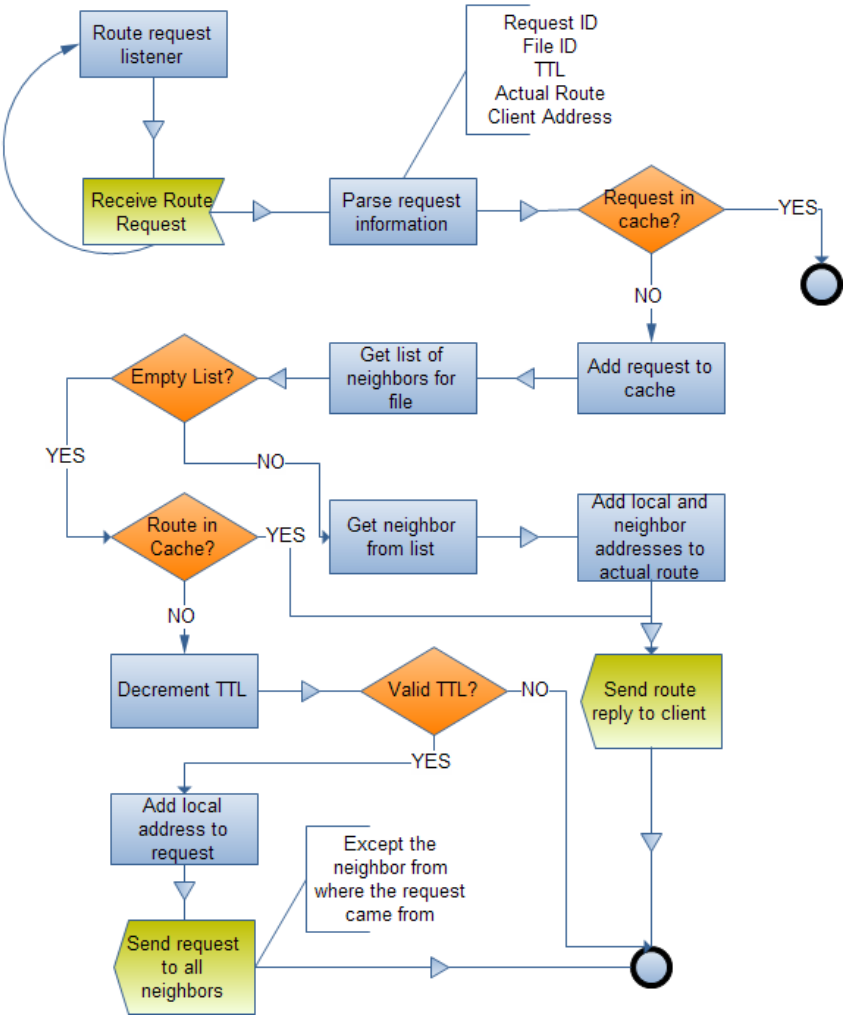


Figure 3.9: Receiving Route Requests

being out of range. Having these modules, we can now implement the transfer process.

3.6 Transfer Module

The transfer process, is made using UDP protocol. We decided to use UDP for the transfer process, due to the problems presented on section 2.3 on page 15. If we choose TCP protocol for the transfer process, we have to adapt the protocol in order to operate with success on a mobile multi-hop scenario. Another possibility is to implement a new layer just like the ATCP, between IP and TCP. Both approaches implies changes to default protocols or implementation of new layers. In order to use unchanged protocols, due to the problems presented by TCP, we choose UDP for the transport process. Keeping in mind the objective of working off-the-shef, we already designed a routing module that does not depend on additional software. Choosing UDP, we do not need to perform protocol changes. To achieve the objective of having and independent system, UDP seems the best choice. Besides, on section 2.3 we stated that UDP does not have the problems presented by TCP on mobile multi-hop scenarios.

A big limitation is imposed with this choice: each packet is limited to a maximum of 64KB of data. In case we have parts bigger than 64KB, then we have a problem: How to send the part using UDP protocol? The answer is quite simple: dividing each part in smaller parts, that we call "chunks".

Upon receiving a file part request, the server node starts by reading the file part using the File Module and then divides the byte array into chunks. Chunks are sent sequentially to the client node. On an initial version, ACK packets were used in order to confirm the reception of each individual chunk. The serving node waits for the ACK packet before sending the next chunk. Later the application was changed in order to enable application to decide when to wait for ACKs to send the next chunk based on the "capability" of the next hop. The use of ACKs and the changes performed are further explained on Section 4.6 on page 69.

An extra chunk is sent, containing only the checksum for the part transferred. The checksum value is transferred in order to ensure that the file part was received with no errors. In case all chunks are transferred, then they are merged and the checksum is calculated to compare with the checksum received. If the two values match, there were no transfer errors and the file part is saved. When some chunks are missing, or the checksum does not match, a new request is made. For better understanding about how the transfer operation works, you can look at Figure 3.10 that represents the transfer process on the client node.

Conceptually, the transfer is made by file parts, not chunks. Chunks are only used to fit a part in UDP packets.

When a device wants to fetch some file, it starts by consulting the list of neighbors. The list of neighbors includes the file ids that are available at each one. Suppose that device "A" wants to download the file with id "12345". If there is a neighbor "B", that contains the file "12345", then "A" issues a transfer request to "B" for part 1 of file "12345". After receiving part 1, "A" requests part 2, and so on, until "A" has all parts of the file.

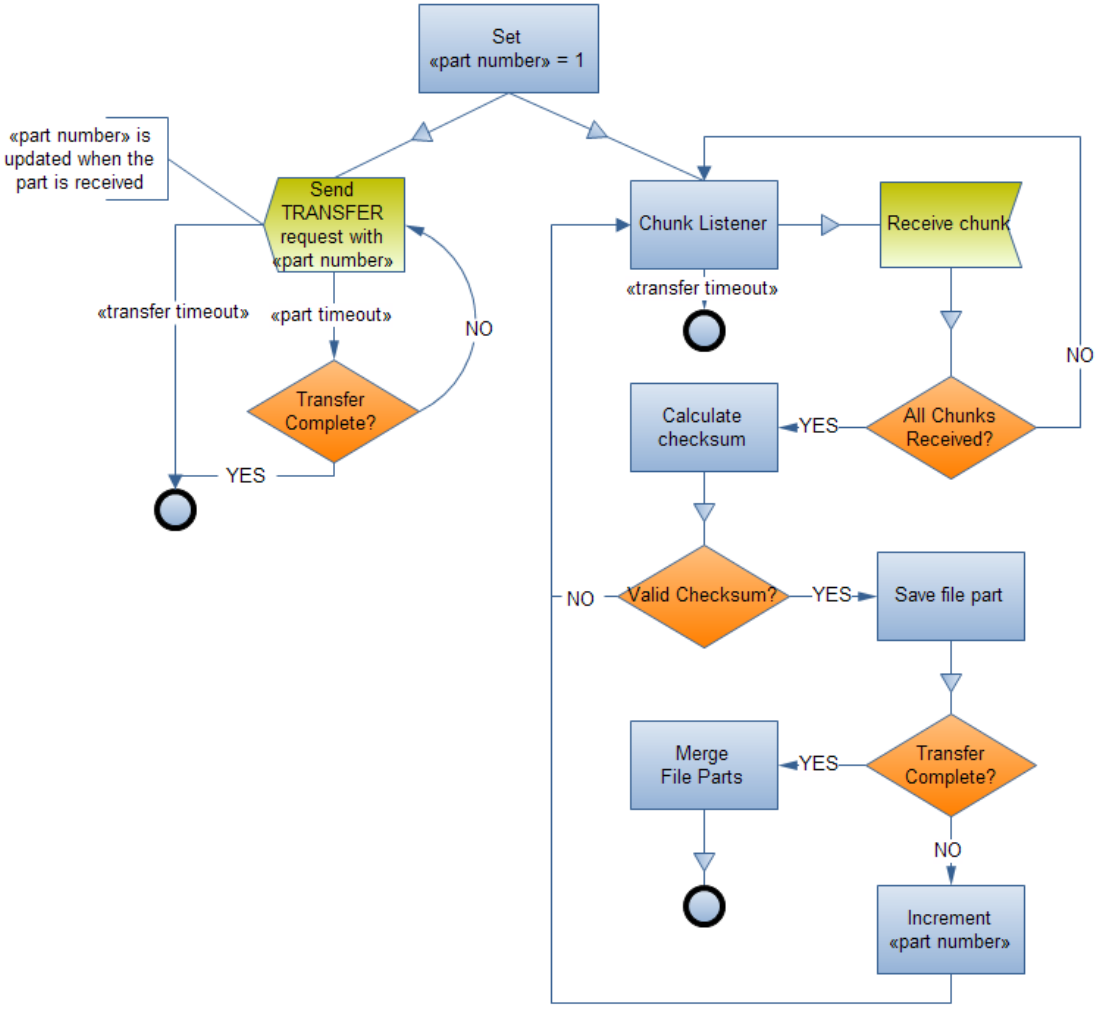


Figure 3.10: Transfer process (Client side)

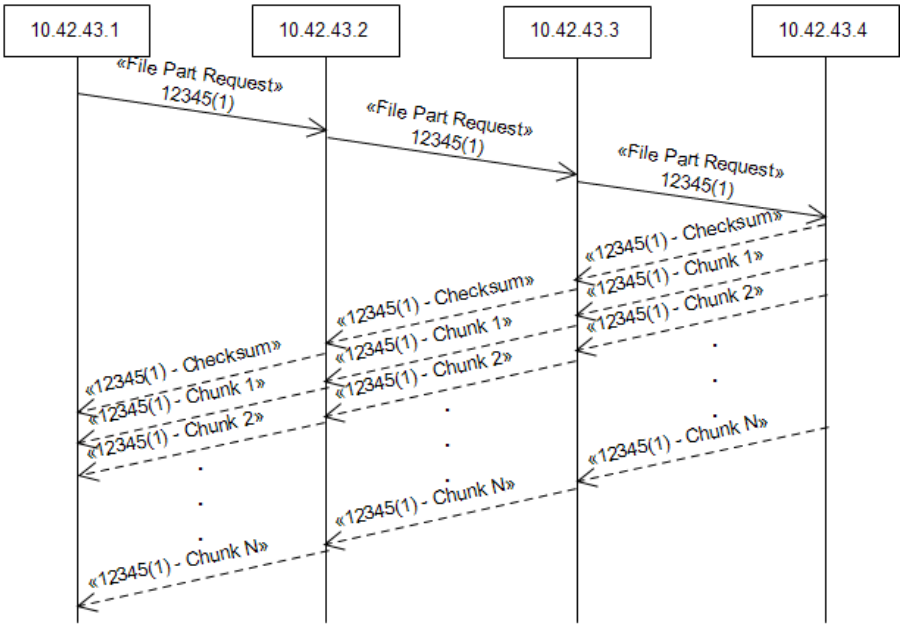


Figure 3.11: Transfer process diagram

The process is quite simple if there is some neighbor with the file, but that is not always the case. Given that the file is not available at the neighborhood, the node has to route the requests through other nodes in order to receive the file. For that process, the first step consists on consulting the route cache. If there is a route available for the file, then a request is built with the route appended and sent to the next hop. For example, if the route reaches node with address "10.42.43.10" passing though "10.42.43.2", then the request is sent to "10.42.43.2" because it is the next hop. In case there is no route available, a route discovery process is issued to gather routes for the file. Upon receiving a file request, in case the node is not the final destination, it just passes the request to the next hop as stated by the aggregated route. When the request reaches the final destination, the node replies with the file part requested just like it is serving a local neighbor. The difference is that the serving node is not serving the node that actually wants the file, but it is instead serving a local neighbor that forwarded the request. You can see how the protocol works looking at Figure 3.11 that represents a transfer process for a file part forwarded between intermediate nodes. Moreover, on each chunk it is aggregated the route needed to deliver it to the request issuer. For example, if the request is made by node with address "10.42.43.1" using a route passing through "10.42.43.2" and reaching then "10.42.43.10", each chunk from the response has a route appended telling exactly the nodes that the request should pass through, in this case, "10.42.43.1", "10.42.43.2" and "10.42.43.10".

When a download process is started, all file parts are not necessarily fetched from the same node. Imagine that a node A is downloading a file from node B. B moves away and becomes unreachable from A. The download process continues because before each file part is requested, the node checks where the file is available. The process of checking if there is a neighbor available for the file and then check if there is a route available is performed before each part is requested. That implies that file parts are not necessarily requested to the same node from the beginning to the end. Being the transfer based on file parts, soliciting all file parts to the same destination does not make sense. Each part is solicited to the actual best destination, i.e. the destination with less hop-number.

When a file part is requested, the client node cannot wait indefinitely for the part to arrive. Some file part chunks may be lost and the part must be requested again. When the file part is received, the next part is immediately requested. When the timeout occurs, the same part is requested again. Actually this timeout is a static value. In a future version, this time may be dynamic. A way of having a dynamic value is to store the time the last parts took to arrive. With the information about the number of hops from the route used to transfer the parts, the download timeout value could be calculated based on a function that depends on the hop-number value, the average round trip time (RTT) of the network and the file part size.

Besides the file part timeout value, there is also a transfer timeout. Each time a download process is started, a timestamp value is recorded stating the time it started. When a file part is received without errors, this timestamp value is updated. In case no file part is received, or the file parts are constantly received with errors then a timeout occurs. This cancels the transfer, as no file parts were received from a certain amount of time. This is also a static value, but unlike the file part timeout, is not relevant to have a dynamic timeout value. The file part timeout should be dynamic because the time to download from a direct neighbor and a node at 3-hops is not the same. Besides, there may be different RTT for each route available. The value of the file part timeout has a direct influence on the transfer performance. The transfer timeout has no impact on the performance, it is just a variable to identify files that are not accessible.

The File Module is used to write to disk each downloaded part. Also, it is responsible for merging file parts on transfer completion. The transfer process is only responsible for fetching the file and store it to disk. The delivery to the user is a responsibility of other presentation interfaces that use the API to transfer the file. On the case of our system, the mini HTTP server is the responsible for leveraging the files to the user.

In short, the file transfer uses the neighborhood module to download files from direct neighbors and the route module when the file is not available in range. Each route is used to address file requests across the network and to deliver file part chunks through intermediate nodes till the destination.

With the Transfer Module, we can now successfully transfer files between direct nodes and routing data between intermediate nodes. But there is some very important feature remaining. We need a mechanism to enable users to interact with our system. We need an API with all the required methods to enable the interaction with the application like starting a download process or consulting the state of a transfer. This will allow other applications to use our system by invoking the methods available in the API. Besides, this can be our starting point to implement an interface to enable users to interact with the system.

Method	Description
void initialize_server(List<AdHocFile> files, List<String> layout)	initializes the server
void initialize_server(List<AdhocFile> files, List<String> layout, String ipAddress)	
void initialize_client()	initializes the client
void initialize_client(String ipAddress)	
void start_file_download(UUID fileID)	starts the download of the file
boolean downloading_file(UUID fileID)	checks if file is being transfered
String show_file_download_status(UUID fileID)	shows the transfer status for the given file
String get_file_name(UUID fileID)	gets file name from UUID
String get_file_path(UUID fileID)	gets the path of a downloaded file
String get_file_folderName(UUID fileID)	gets the destination folder of the file
void set_log(string logLevel)	sets the log level
List<AdHocFile> get_all_available_file_ids()	returns the list of available files on the system
List<AdHocFile> get_available_files_by_type_id(int typeID)	gets the files available with typeID = typeID
FileTypeManager get_types()	gets the types of files available
List<String> get_layout()	gets the layout of the homepage
boolean isStarted()	checks if the application was already started (it's started when it receives the base information on the START process)
boolean is_file_available(UUID fileID)	checks if the file is available (already downloaded)

Table 3.2: API methods

3.7 API

The methods made available in the API are presented on Table 3.2. With these methods, any application is able to use our system. Now we can use these methods as a starting point to implement the user interface. To enable applications to use our system, they just need to import our project to the libraries and import the API. It can be imported using the following code:

```
import API.AdHocP2P
```

In order to use the custom classes created to deal with files and types, another imports must be done. You may need to use the following imports:

```
import files.AdhocFile
import files.FileType
import files.FileTypeManager
```

3.8 Mini HTTP Server Module

This is the module responsible for the interaction with the final user. It represents the frontier between user and the application. The application is accessible through HTTP, using any browser for the effect. We choose HTTP because it serves a normal HTML page with contents available. With this approach we do not need to implement a new interface to access our system, we just use HTML. Each user can access the system by using its preferred browser. Because our contents are presented as a normal html page, each user that is already familiarized with content access in the World Wide Web (WWW) can easily access our system without needing to adapt to a new application interface. Besides, each entity interested in using our application can adapt its own webpage to work on our system. Konark[16] also uses an HTTP server, although it is used for a totally different purpose. The Konark micro HTTP server is used to provide services to other nodes. Each node makes its services available through this server. We use the mini HTTP server for a totally different purpose, the server is used by the node itself, being unavailable for other nodes as the requests are filtered in order to allow only localhost requests. The HTTP server on our prototype works as the interface of the application, allowing the initial server to 'draw' its own interface.

Although our application was designed to operate with one single page, it could be adapted to suit other scenarios. For example, simulate an existing website, with different interconnected pages. One thing that it is not possible to do is to allow dynamic pages, like ASP or PHP, because each page is rendered locally at each node. In order to enable the existence of more than one page, implying more than one layout, the application needs to be changed at other modules, starting on the Neighborhood Module. This module is responsible for initializing each node with the layout and set of files. To enable more than one page, the application has to be changed in order to define a list of layouts, not only one layout that actually is used as the index page. Besides, for each layout a page must be assigned, for example, assign a layout to "index.html" other to "videos.html". With this adaptation, it is possible to simulate any static website.

After the application is running, users wishing to access the network content have to fire up their browsers and address to http://localhost.

This module includes a mini-HTTP server that responds to a limit of different HTTP requests. The default request is a GET / (index) method. To respond to this request, the server responds with the page layout and some javascript methods needed to interact with the server as shown on Figure 3.12. The layout is a set of html code lines as a typical html page. Some special tags are put between the normal html code. These tags are used to identify files. When this special tag is found on the layout, then some special treatment to the code is needed before sending it on the response.

The special tag includes a file id, a file name a type description and size described in bytes. For example, the tag *<!!!! fileid=3e8fc78c-5a0d-49e0-8229-7963eec4198f filename=foto.jpg type=images size=11293 !!!!>* represents the file with id *3e8fc78c-5a0d-49e0-8229-7963eec4198f*, that is named *foto.jpg* being of type *images* and size *11293* bytes. The layout page is processed line by line, so the tag must be on a single line. Each time the tag is found on a line, the module uses the API to know if the file is already available. Case the file is available, the content is delivered using the

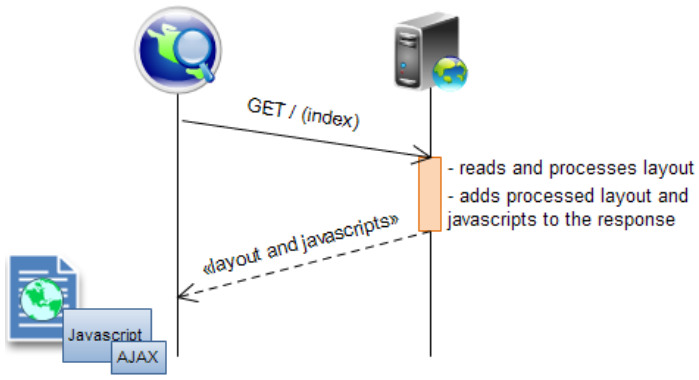


Figure 3.12: Getting page index

proper html code. For example, if the file is of type images, then the file tag is replaced with **. Similar is the case when the type is others, the response would be * filename * (a link to the file). Case the file is not available, another check is made, this time it is checked if the file is currently being downloaded. If that is the case, the tag is replaced with the file download status, for example, 23.0%. When the file is not available, and is not currently being downloaded, the file tag is replaced with a link to start the download process for that file. This link uses a javascript method to initiate the transfer and continuously update the download status. When the download status reaches 100%, the file is delivered.

Each tag found on the layout is replaced with special html code, to access the file, to start the transfer process or simply to show the transfer status. This is done by replacing the tag with a span class. For example, when the tag *<!!!! fileid=3e8fc78c-5a0d-49e0-8229-7963eec4198f filename=foto.jpg type=images size=11293 !!!!>* is found, the first step is to replace it with *[file html code]*. This is done to easily replace the span content on the various stages. First the *[file html code]* is the link to start the download process. For the tag presented, that would be something like: *Download foto.jpg*. In the first step, the file tag *<!!!! fileid=3e8fc78c-5a0d-49e0-8229-7963eec4198f filename=foto.jpg type=images size=11293 !!!!>* is replaced with:

```
<span id="3e8fc78c-5a0d-49e0-8229-7963eec4198f">
  <a href="javascript:startDownload('3e8fc78c-5a0d-49e0-8229-7963eec4198f');">
Download foto.jpg</a>
</span>
```

When the download is started, the *[file html code]* is replaced with the download status. Supposing that the current download status is 54%, then the html code would be something like:

```
<span id="3e8fc78c-5a0d-49e0-8229-7963eec4198f">54.0%</span>
```

When the download finishes, the span html is replaced with the file wrapper. That would result on having the following code:

```
<span id="3e8fc78c-5a0d-49e0-8229-7963eec4198f">
  <img src="/get/3e8fc78c-5a0d-49e0-8229-7963eec4198f/foto.jpg" />
</span>
```

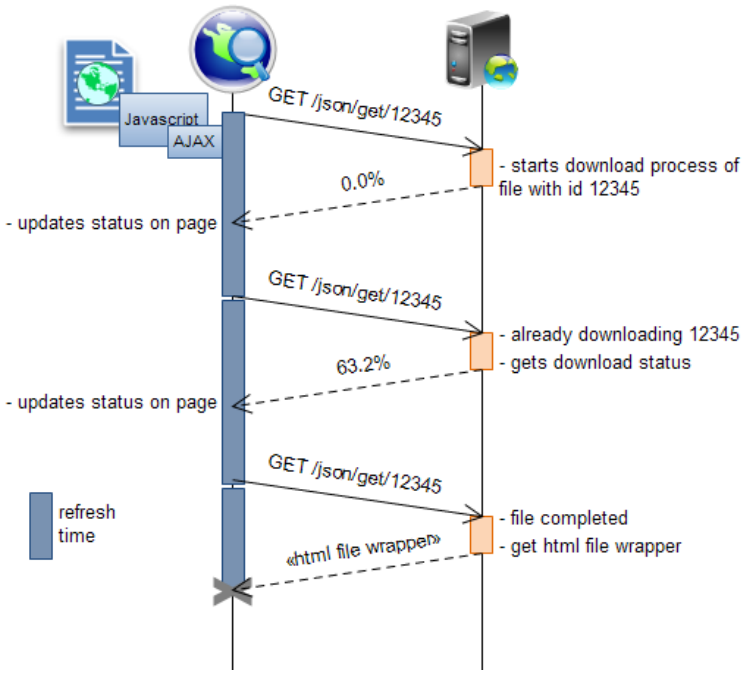


Figure 3.13: Downloading file

Initially div classes were used instead of span, but div class does not support inline by default. If we have on the same line several different div classes, they will not appear inline as intended. This is because span is an inline element and div is a block element. Block elements cannot be put inside inline elements. For example, having something like * ... 'div content' ... * is not possible. To enable the user to put the contents inside inline elements, the span was adopted over div.

In order to download a file, upon pressing the link presented on the page, an http get method must be issued to the server. To achieve that, it is used AJAX that provides a method for exchanging data asynchronously between browser and server to avoid full page reloads. There is a dedicated handler to respond to AJAX made requests. The server responds with the current download status or with "error" case there is an error on the download process, or with the file content as delivered by the default page. The download status must be updated frequently, so the script responsible for initiating the download process is also responsible for getting the current download status. In Figure 3.13 you can see a diagram representing a download process issued by pressing the download link presented on the page. The AJAX script is able to update the content of the file span using the simple code, being fileID the id of the file:

```
document.getElementById(fileID).innerHTML = http.responseText;
```

The http.responseText variable is the result from the request issued by the script. This variable includes html code that is updated upon reception. This code varies from the link to start the download process, the file download status and the file wrapper.

When the server receives the request, it starts the download process using the API. The download has just started, so the server responds with 0.0%, indicating that the download as started. The script that started the download, repeats the request every 2 seconds. When the download reaches 100%,

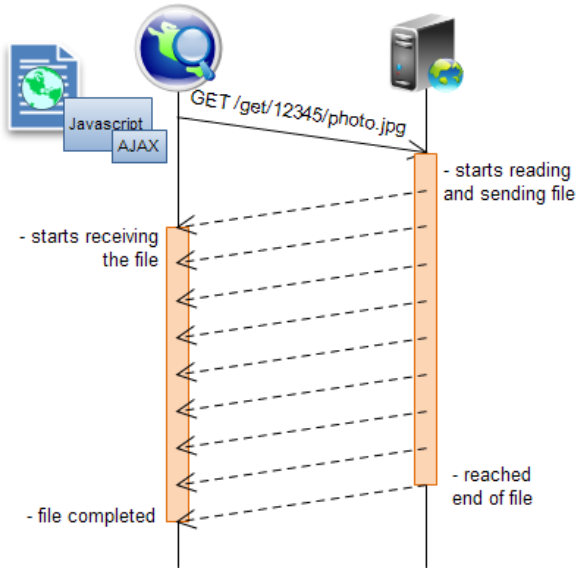


Figure 3.14: Accessing file

the server creates the file wrapper and delivers the content to the client. Along with the file wrapper, a special html comment is also included in order to inform the AJAX script that the download finished and stop the refresh process. For example, if a file with id 3e8fc78c-5a0d-49e0-8229-7963eec4198f is currently being downloaded, upon file completion, along with the file wrapper html code, the tag <!– fileOK:3e8fc78c-5a0d-49e0-8229-7963eec4198f –> is also included. This code does not appear on the webpage, but can be parsed by the AJAX script that stops the refresh process for the file 3e8fc78c-5a0d-49e0-8229-7963eec4198f.

The file is not delivered directly, due to some security limitations on the access to local files on a website imposed by some browsers like Firefox[1]. For example, the most practical way to access a file on the webpage, would be something like putting a reference to the local file, i.e. Click here to open doc.txt. This method is very practical, but would not work on some browsers. In order to access the local file, the http server listens to get requests on the form /get/file-id/filename. For example, when the server receives a request /get/3e8fc78c-5a0d-49e0-8229-7963eec4198f/foto.jpg it starts by parsing the request and then reads file with id 3e8fc78c-5a0d-49e0-8229-7963eec4198f returning a file named foto.jpg. The Figure 3.14 shows how a file is accessed by the browser.

Assuming a spontaneous mobile mesh network with devices that do not have the application, a mechanism for delivering the application could be implemented. Imagine that the server just started and there are several devices connected using an ad-hoc network. Suppose now that the server is the only node having the application. Easily we can adapt the mini HTTP server to allow non-localhost requests only for a specific URL. For example, we can adapt the system to accept requests for the path /application/. If the server has the address "10.42.43.1" then each node can fire up their own browsers and address to "http://10.42.43.1/application/". The mini HTTP server responds now with a page with some instructions on how to use the application and a download button. This way, all

[1]You can read more about it here: http://kb.mozillazine.org/Links_to_local_pages_don%27t_work

server neighbors are able to download the application. When these neighbors start the application themselves, their own neighbors can now access the application. For example, node with address "10.42.43.2" has just downloaded the application from the server. It starts the application and now its own neighbors can access the application accessing to "http://10.42.43.2/application/". This is an easy way to reuse resources from our system to spread itself across the network.

In case the layout has errors, or the user forgets to add the files to the layout, the files may become unavailable. To overcome this issue, there is a default layout that can be used by users accessing http://localhost/generic/. This is a default layout that presents content divided by their types. On the first access to the generic layout, all files are listed. A sidebar menu allows user to navigate through each category of files.

Another important function is needed, a way to setup the set of files and the layout of the system. For a normal client node, the http server is already fully functional, but for the server, there must be a mechanism to set the layout and to add files. When the application runs in server mode, an interface is presented that allow the user to add a new file to the system and simultaneously write the layout page code. After a file is added, the user has to classify the file on a set of defined types presented. After the layout is written and the files added, when ready, user boots up the system. Besides the mechanisms already explained about this module, all the processing is made through the API available.

3.9 Log Functionality

The Log functionality was implemented to allows us to gather information about the application performance. With this functionality we are able to determine how many messages were received and sent. How many files were downloaded, when a download was started and finished. We also know how many data was transferred in terms of bytes. This module was developed to help on the application testing. We need to test how the application performs and to enable that we need to register important events somewhere so we can evaluate the global performance.

Although the log functionality was developed to help on the testing phase, it can be used by experienced users to understand what is happening in the application. This allow each user to produce log files in order to search for problems or just to analyze himself the performance of the application.

To activate log functionality, the application must be started with an extra parameter. To do this, each node has to start the application with the parameter "–log:X", where X is an integer value between 1 and 63. This value is converted in binary code in order to map which events are going to be recorded on the log file. The events registered are appended to the file "log.txt". The file is created in case it does not exist yet. Case the file is already available, the events are appended to the file, being the content of the log preserved on later runs.

Table 3.3 represents the meaning of each bit, being *Bytes* the most significant bit. The *Bytes* bit, sets the application to log the amount of bytes received. With *Downloads* bit, the application records the events related with downloads, like download start and completion and counts the number

Bytes	Downloads	Chunk	Direct Transfer	Forwarding	Routing

Table 3.3: 6 bits representing which events are logged with Bytes being the most significant bit

of downloads. The *Chunk* bit is used to log the receiving and sending of chunks, including the number of chunks received and sent. *Direct Transfer* logs the download requests from direct neighbors. *Forwarding* logs everything related with forward requests, like a file part request or reply forwarded. The *Routing* bit sets the application to log route requests and route replies.

If someone intends to log the routing messages only, just needs to start the application with "–log:1", because the integer 1 equals 000001 in binary, so only the Routing bit is set. Otherwise, if someone wants to log Bytes and Chunk the parameter would be "–log:40", because 40 is 101000 in binary. To log all events, the parameter is "–log:63".

This method was used in order to reduce the number of parameters. For example, if each type of events is associated with a parameter, in order to log all events we would need something like "–logBytes –logDownloads –logChunk –logDirectTransfer –logForwarding – logRouting" or something like "–log:Bytes,Downloads,Chunk,DirectTransfer,Forwarding,Routing". Being the log functionality developed to be used by advanced users only, this is a simple way to enable it.

Besides, each event is tagged with a timestamp. This enables us to calculate time elapsed between events, for example, having the time when a download has started and the time the same download finishes we can estimate the transfer time. One log file will look something like:

```
[14/07/2011 18:37:57,018] Started Download of file 4972b164(12702823 bytes)
[14/07/2011 18:37:57,023] Transfer Message sent to 10.0.0.3
[14/07/2011 18:37:57,023] Route: using route with 1 hops to request file
[14/07/2011 18:37:57,103] Chunk received from 10.0.0.3
[14/07/2011 18:37:57,165] Chunk received from 10.0.0.3
...
[14/07/2011 18:37:57,652] Downloaded part [1] of file 4972b164 [SUCCESS]
...
[14/07/2011 18:38:11,296] Downloaded part [25] of file 4972b164 [SUCCESS]
[14/07/2011 18:38:11,302] Finished download of file 4972b164(12702823 bytes)
[14/07/2011 18:38:40,672]--Terminated--
[Chunks received: 243 ] [Chunks sent: 0 ] [Forward Chunks received: 0 ]
[Forward Chunks sent: 0 ] [Forward Requests received: 0 ]
[Forward Requests sent: 0 ] [ Transfer Requests received: 0 ]
[ Transfer Requests sent: 25 ] [ Route Replies received: 0 ]
[ Route Replies sent: 0 ] [ Route Request received: 0 ]
[ Route Request sent: 0 ] [ Downloads started: 1 ] [ Downloads completed: 1]
[ Downloaded parts: 25 [SUCCESS] - 0 [CHECKSUM ERROR] [ Download timeouts: 0 ]
[ Total Bytes Received: 12748929 ]
---------------
```

Chapter 4

Testing Results

In this chapter, the results obtained from a series of tests made in order to validate our prototype are presented. First the testing scenario is described. Then, the testing topology based on Common Open Research Emulator (CORE)[1] is presented and it is explained how the tests are made and how the results are collected. The log functionality of our prototype is used to produce some log files. These log files are then used to fetch the necessary data for the results presented.

4.1 Testing Scenario

Due to the nature of the application developed, made with the objective of being used in an ad-hoc network, with possibly a large number of mobile nodes, the tests were made with the help of an emulator. In order to gather results from a real scenario, many resources would be needed that were not available at this phase. The emulations were performed using CORE[1] (version 4.1), a tool that can emulate entire networks (wired or wireless) on one machine. CORE was installed on a single Ubuntu 11.04 Linux virtual machine running on AMD Opteron 275 at 2,2Ghz with 512MB RAM.

The help of many volunteers would be needed in order to test our application on a real scenario. Besides, many devices would be necessary to build the testing scenario. Nevertheless, the logistics necessary to change the topology would be too complex. Due to the complexity and dependence on external entities for achieving a real test scenario, we choose to perform the first set of tests with the help of CORE[1] emulator.

In order to collect data relative to the tests done, the application was changed to register important events in a log file. These events include the reception of a file part, a file part chunk, route request message, route reply, etc. Each event is tagged with a timestamp to calculate the elapsed time between events. The log functionality is explained on section 3.9.

The log functionality also records how much messages were received and sent, allowing the calculation on how much traffic was received in total. It also allows the calculation on how much overhead is generated for a file download process between 2 peers in the network. With this approach, running the application in the network produces a log file for each node. When a test finishes, the log files

must be collected to interpret the results produced. To collect the essential data from the log files, a parser was also developed specifically designed to parse the log file structure of our application and test scenario. The parser is responsible for getting the global information about a download process that occurred in the network, like download time, total bytes transferred and routing messages sent, etc. The experiments can be separated according to the distance in terms of number of hops between the principal involved nodes, client and server. The parser is also responsible for creating CSV files that can afterwards be easily imported into R script to generate the graphics presented on this chapter.

However, enabling log functionality on the application and implementing a parser to gather the information from the log files is not enough. There is also a need for an automatic mechanism to run tests and generate data. This step is very important, because tests done in a manual way would imply to start the application on every node and wait for the download process to finish. Also, the server and client nodes would need to be chosen manually. After the download finishes, the application needs to be stopped at every node and the log files collected one by one. Without automating the tasks, results would take much longer and would be less exhaustive. Because we want the tests to be somehow significant, there must be a large number of tests. Besides, there is not much time to exhaustively perform all the tests. To achieve that automatism on testing, a bash script was written to perform a series of consecutive tests. The script generates 2 distinct random numbers that will be mapped into a server and a client node of the network. After that, the script starts the application at every node of the network. If the node number equals the number of the server or client numbers generated, the server and client application is booted respectively. The other nodes run the application in idle mode, i.e. they run as normal client nodes without downloading any file. After that, the script waits for the application to finish. When the application exits, the script copies all the log files to a new folder and repeats the process until the loop ends. The script performs 200 consecutive tests. When the bulk of tests is made, the only manual steps needed are the execution of the parser application that generates the CSV files, and the execution of the R script that generates the respective graphics.

For all the tests, only one file with 12702823 bytes of size was used. The same file was used across all the tests, to allow us to compare the results from the different tests made, like download time and bytes transferred. The network topology is composed by 10 nodes, that are connected as shown on Figure 4.1. This topology was created to perform the first tests. These tests were performed in order to verify the functionality of the prototype and to further debug the application. The performance tests were made using the same topology afterwards. The majority of the tests were made with the objective of finding the best parameter values.

Parameters like part download timeout may have a huge impact on the performance. It represents the time that each client waits for the reception of a file part, before considering that an error occurred with the transfer, issuing a new request for the file part in question. This wait time cannot be an arbitrary value. If it's too small, unnecessary new requests would generate unnecessary traffic. If it's too big, the download process will slow down proportionally to the excessive timeout value, when errors in file parts occur.

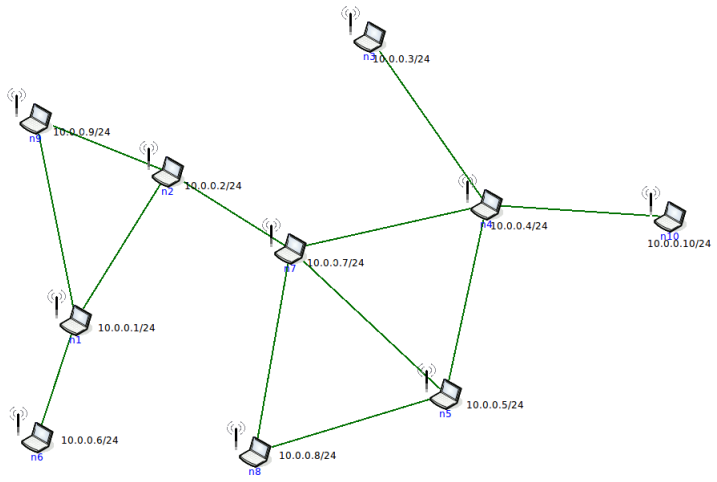


Figure 4.1: Network Topology

4.2 Application Parameters

Several application parameters were defined in order to tune the application behavior. We chose a subset of the most important ones to be evaluated in the actual test scenario. To demonstrate their relevance, the selected parameters are explained below.

4.2.1 Chunk Timeout

The chunk timeout variable represents the time that a node waits to deliver a chunk from a file part. When a client requests a file part from a given node, the part is divided into chunks that are sent to the requester. After the first chunk is sent the node waits for a confirmation of reception from its neighbor before delivering the next chunk. Case there is no confirmation, the next chunk is sent anyway after *'chunk_timeout'* milliseconds. This waiting time is used to ensure that packet loss is minimal, delaying the delivery to a time when the next node is able to receive data. On the preliminary tests, we observed that with no waiting time, for example, in a part with 10 chunks, sometimes only the first two and the last two chunks were received and all intermediate chunks were lost. This chunk timeout mechanism was implemented due to this mysterious packet loss. The initial idea was to send all the chunks from a part without waiting for confirmation, but due to the problem exposed, this mechanism seems essential to ensure the prototype is functional.

When chunk timeout value is too low and packets are lost, chunk packets may always fail on each request, making the download process impossible. When the value is too high and there is no confirmation or the confirmation is lost, the download process will slow down a lot. So this is a very important parameter to the performance and functionality of the prototype. In that sense, tests have to be made to observe the performance varying the parameter value. This enables the evaluation of the impact of each value and find optimal values to the present scenario.

4.2.2 Download Wait Time

This variable is similar to the chunk timeout, but this timeout is set on the client node, not on the serving node. When a node starts a download, it will wait for some time to receive a file part. If the file part does not arrive, the client cannot wait indefinitely. After the download wait time passes, if the part was not received, the client assumes that some chunks were lost and it must re-send the request for the file part in question. Similarly to chunk timeout, case the value is too small, the number of retransmissions grows, generating more overhead and slowing down the download process. If the value is too big, upon each part download failure, the download process will hang for the time equivalent to the download wait time value, slowing down the download.

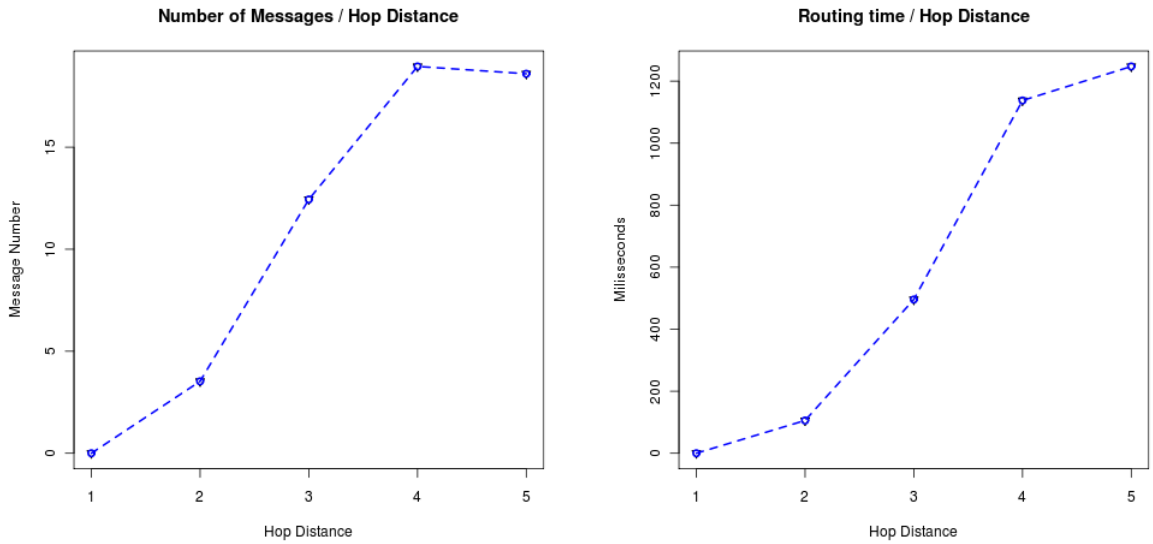
4.2.3 Part Size and Chunk Size

Other important variables are the part size and chunk size. Although they might seem to be less important to the download process, tests have been made to evaluate how the network reacts to different file parts and chunk sizes. At first glance, the impact of part size seems meaningless, because only chunks are transferred, not file parts. However, it must have some impact on performance because the processing that is needed to deal with file parts of 512KB, will not be the same as a file part of 50MB, as we believe. Another potential impact that part size can have, is related to retransmissions and the consequent overhead. If there is an error on a single chunk of a given file part, the whole part must be retransmitted. In this case, the difference between bigger and smaller file parts has impact on performance. In case of retransmissions, the latest generates less overhead while the first generates more. Case network performs well, and there are no errors, smaller parts generate more overhead, because with smaller file parts there will be more parts for each file implying more checksum traffic to validate each transferred part.

The case for chunk size is similar. If the size is too small, more chunks are transferred on each part, increasing the risk of packet loss. If the size is too big there is a bigger risk of corrupted packets. So, apparently there is no easy choice between a small, medium or big size. Some tests have to be done to understand how the network responds to different chunk sizes.

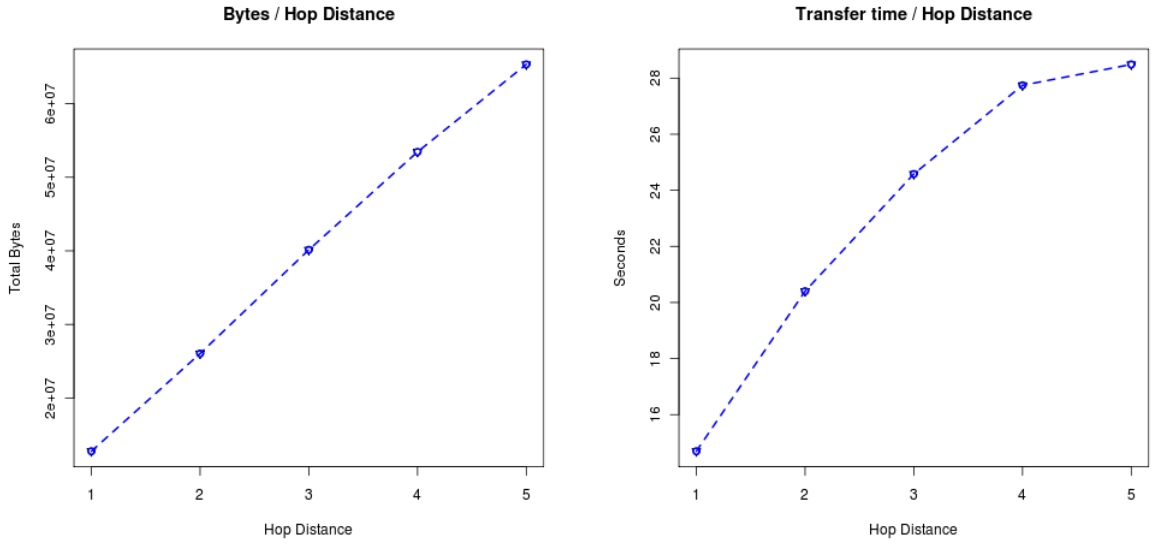
4.3 Initial Results

Two principal functions of our prototype, routing and file transfer were initially evaluated on our tests. For the routing process, the number of routing messages used to find a route was measured. Also it was measured the time elapsed from the moment a route request is issued until a reply is received. For the file transfer process, the time until the download finishes was measured, including the routing process and all data transmission overhead generated on the network. These metrics allow us to evaluate how the prototype performs. The main focus was on time, but the overhead is also important to see if the amount of bytes transferred is actually linearly proportional to the number of hops. Besides that, it is possible to conclude on the number of retransmissions observing the overhead results.



(a) Number of Messages vs Hop Number (b) Routing Time vs Hop Number

Figure 4.2: Routing Results (First Tests)



(a) Transfered Bytes vs Hop Number (b) Transfer Time vs Hop Number

Figure 4.3: Transfer Results (First Tests)

The first results were obtained from a bulk of 200 tests using default parameters chosen. The parameters were chosen somehow arbitrarily, except the chunk size. The chunk size was initially set to 60KB, in order to take advantage of the UDP packet maximum size of 64KB. The remaining variables were set to values that seamed adequate but without any prove of being the most suited ones. The download timeout was initially set to 1,5 seconds. The part size was set to 512KB. Finally, the chunk timeout was set to 300 milliseconds.

The tests were made using the script explained before, that generates 200 experiments choosing a server and a client randomly. The obtained results are shown on Figure 4.2 and Figure 4.3.

Readings the results, one of the first observations was that more routing messages were used to 4 hop transfers compared to 5 hop transfers. That led us to conclude that the number of messages depends more on the topology than on the number of hops. The routing time results were expected, with more time to find routes with bigger hop number. The overhead is linear to the number of hops. That is an expected result, because if a file is forwarded through other nodes, the file data must pass through each node. If we have, for example, a file of 5MB of data and it is transferred directly between two nodes, it is expected that the overhead would be approximately 5MB. On other hand, if the file must pass through an intermediate node, it is expected that the intermediate node will receive approximately 5MB that will forward to the destination, generating another 5MB. That applies to a bigger number of intermediate nodes, it's expected that a file of 5MB that is transmitted over 3 nodes, will produce a minimum overhead of 15MB, 5 for each node. Finally, in terms of download time, there was an initial expectation that the download time would be bigger as the number of hops increases. In fact, the results show exactly that in Figure 4.3(b).

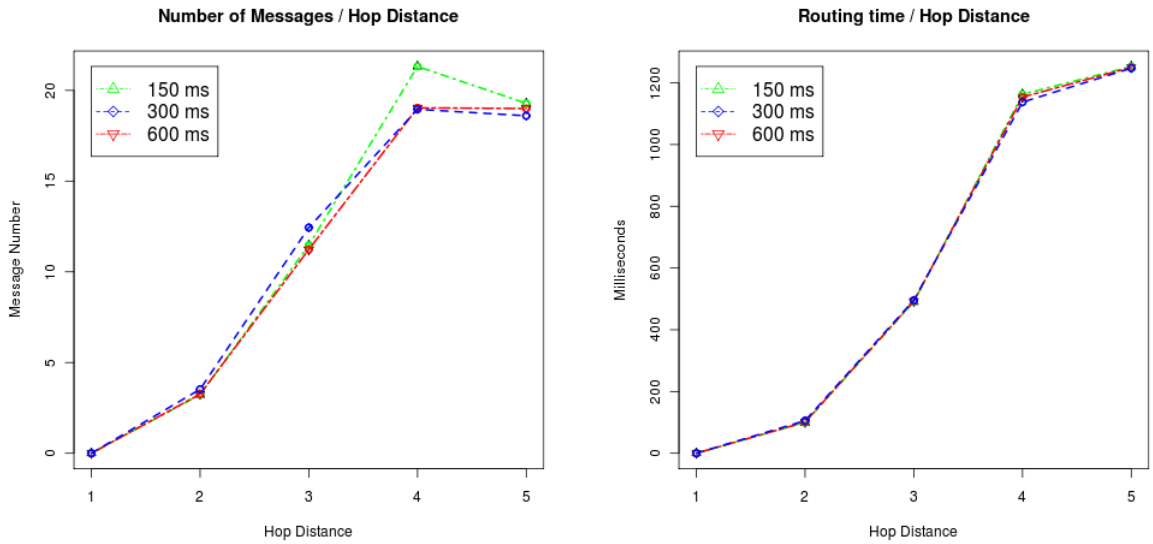
4.4 Parameters Analysis

The second set of tests were made in order to observe the prototype behavior when changing different application parameters. Each bulk of tests was made doubling and reducing to half each variable.

4.4.1 Chunk Timeout

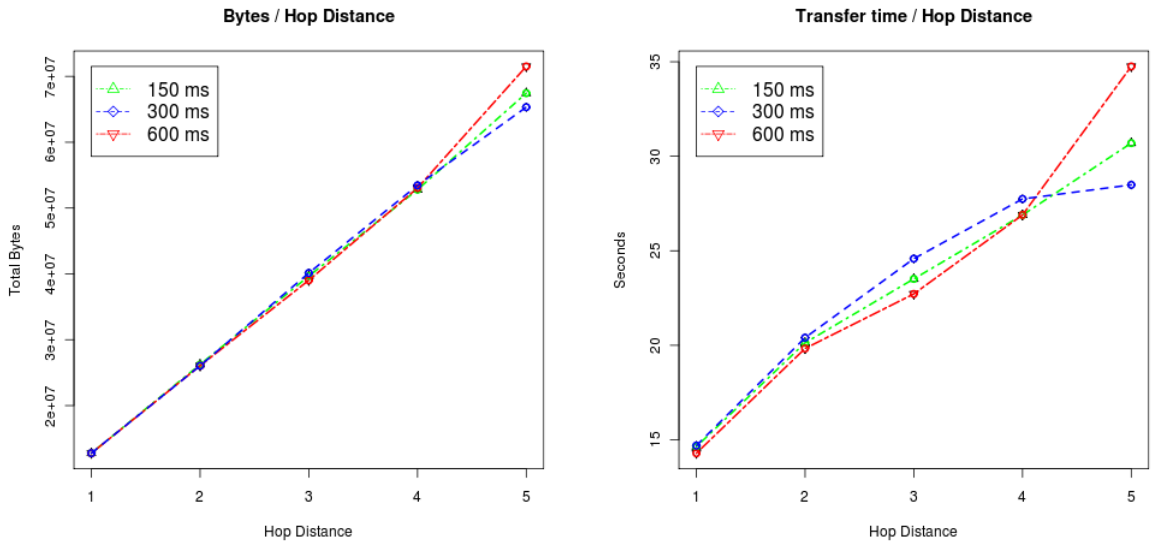
Results presented on Figure 4.4 and Figure 4.5 were obtained by changing the "chunk timeout" variable to 150 and 600.

Looking at the graphics on Figure 4.5, we can see that in terms of overhead, the results are very similar for the 3 cases. In terms of download time, the results obtained from using 150 and 600 are both lightly below the 300 results, except for 5 hops of distance. Although the differences, we believe that they are not significant. The "chunk timeout" is a variable used to prevent the node to wait indefinitely to send the next chunk, in case the confirmation packet is lost. Due to this fact, we can say that the timeout is only used on emergency cases, so in an ideal scenario without packet loss, this values is less important. A value bigger than the round trip time between two nodes seems adequate. It cannot be much bigger or the download will slow down very much, each time a confirmation packet



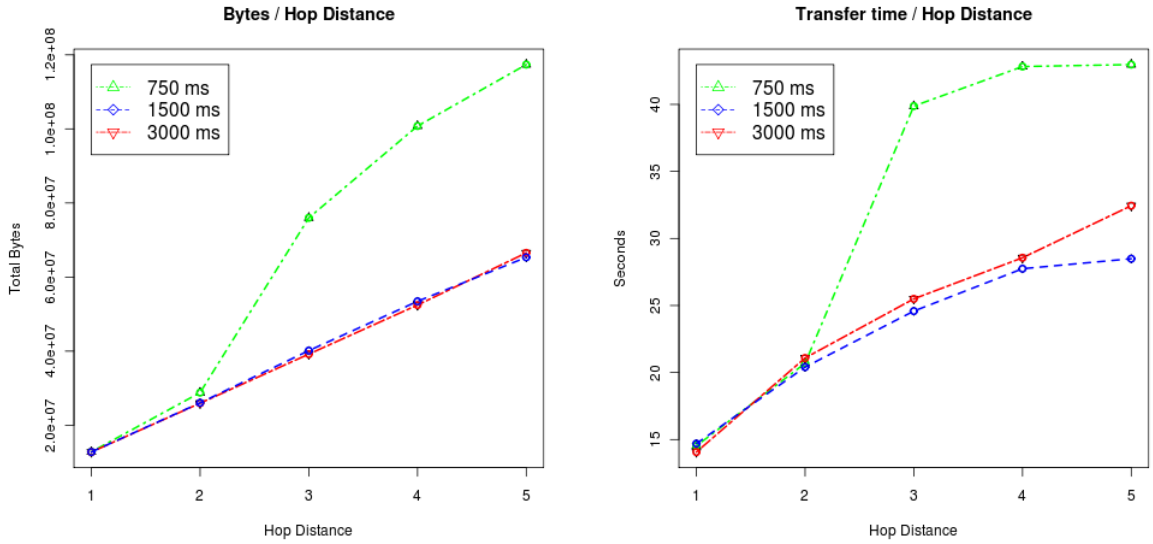
(a) Number of Messages vs Hop Number (b) Routing Time vs Hop Number

Figure 4.4: Routing Results (Chunk Timeout Comparison)



(a) Transfered Bytes vs Hop Number (b) Transfer Time vs Hop Number

Figure 4.5: Transfer Results (Chunk Timeout Comparison)



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

Figure 4.6: Transfer Results (Download Wait Time Comparison)

is lost.

In terms of routing time and routing messages, the results are very similar as shown of Figure 4.4. Along the tests performed afterwards, it was verified that the values were too similar and we concluded that changing the application parameters announced here has no impact on routing time or number of messages. From now on, we just compare the results in terms of transfer time and transfer overhead.

4.4.2 Download Wait Time

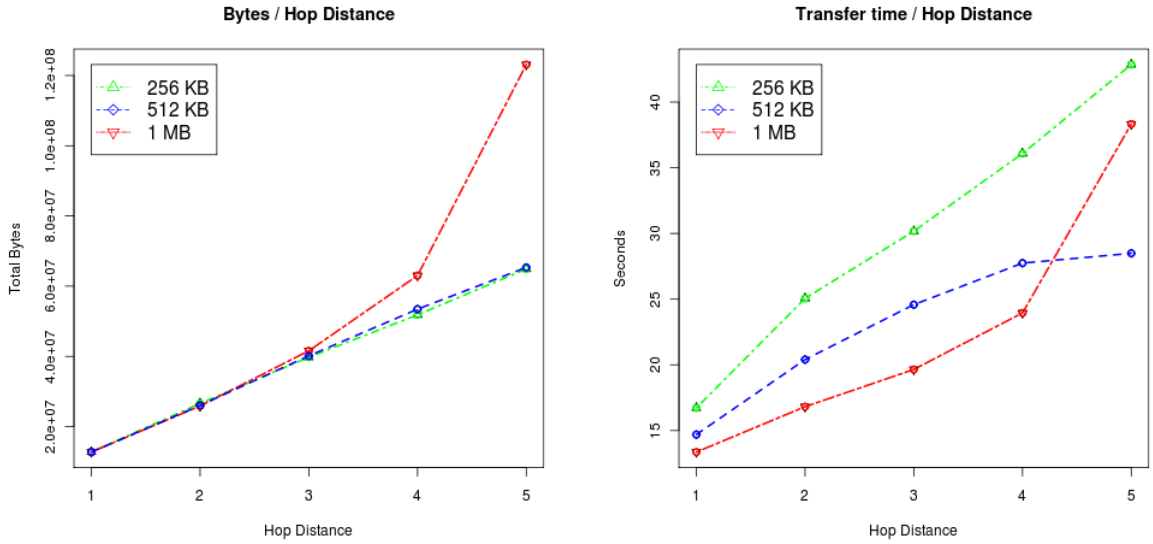
Download wait time was tested with 750 and 3000 milliseconds. half and double of the default value, respectively. The results obtained are shown on Figure 4.6.

When the value was shrunk to half (750 milliseconds), no changes were observed for 1 and 2 hops. For 3, 4 and 5 hops, the results are catastrophic compared to the default value used (1500 milliseconds). The transfer time increased a lot so as the total overhead. This means that there were a lot of retransmissions on the transfer, suggesting that many parts were requested more than once, if not all of them. This excludes the 750 milliseconds as a suited value for download wait time so as any below value.

Doubling the value, produced no significant changes. The transfer time was slightly bigger, suggesting that there were some chunks lost for a given file part, resulting on a bigger waiting time before a new request for the same part is issued.

4.4.3 Part Size

To evaluate part size, tests were made using part sizes of 262144 and 1048576 bytes. Figure 4.7 presents the results obtained.



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

Figure 4.7: Transfer Results (Part Size Comparison)

For parts with 262144 of size, the total overhead is similar to the default. In terms of transfer time, the differences are very evident. The time to download a file increases for every hop number. One explanation to these results that seems adequate is related with tasks that need more CPU to execute with smaller parts. Smaller parts imply that there will be more file parts to handle on each node. There are also more checksum packets transmitted and more processing to validate each packet. Another obvious thing is that there will be more file parts to request. To request a file with 5120KB of size, 10 file parts have to be requested for a file part of 512KB. If each file part has 256KB of size, 20 file parts have to be requested. Obviously, requesting a file part of 512KB is faster than requesting 2 file parts of 256KB. This is true because when we request a file part of 512KB then 512KB of data will be transmitted consecutively, while requesting 2 file parts of 256KB will imply receiving the first 256KB of data and the serving node stops sending data, waiting to receive the next file part request. This delay on each file part, results in slower downloads. Figure 4.8 illustrates this scenario, representing a transfer of 512KB of data using 512KB. A single file part request is sent and the data is transferred on a single file part. Now suppose the same transfer is done but using now file parts of 256KB as shown on Figure 4.9. A first request is sent and the first 256KB of data are forwarded. Then, the server stops sending and the new request is issued and the server sends the next 256KB of data. The difference from the two different file part sizes is now obvious.

When using file parts with 1048576 bytes of size, the total overhead is similar for 1, 2 and 3 hops of distance, but increases very much for 4 and 5 hops. Transfer time was bigger for 5 hops, but smaller for the other cases. Except for 5 hops, transfer times were generally lower, because bigger parts imply less checksum traffic, less requests and less file parts to process. That explains the lower values on transfer times, but how to explain the increase of total bytes transferred with 4 and 5 hops and slower downloads on 5 hops? Well, when we have file parts of 1048576 bytes of size, we are doubling the size of the initial part size defined. Surely it takes longer to download 1024KB of data than 512KB,

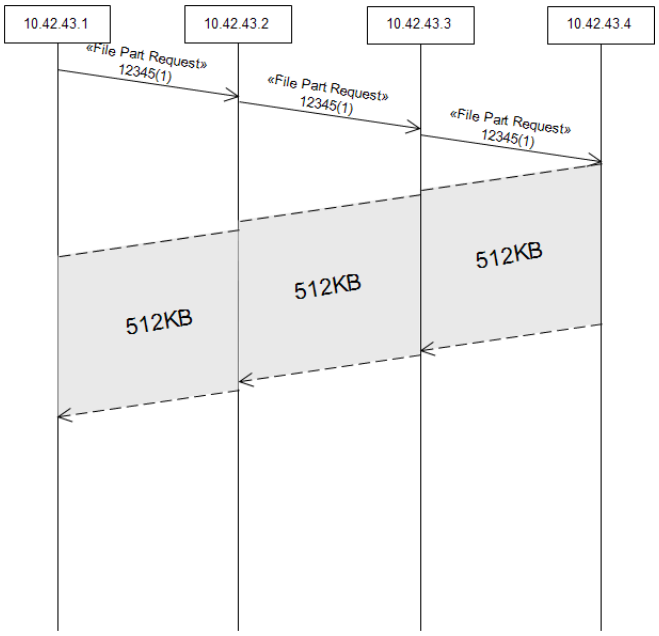


Figure 4.8: Transferring 512KB of data using 512KB file parts

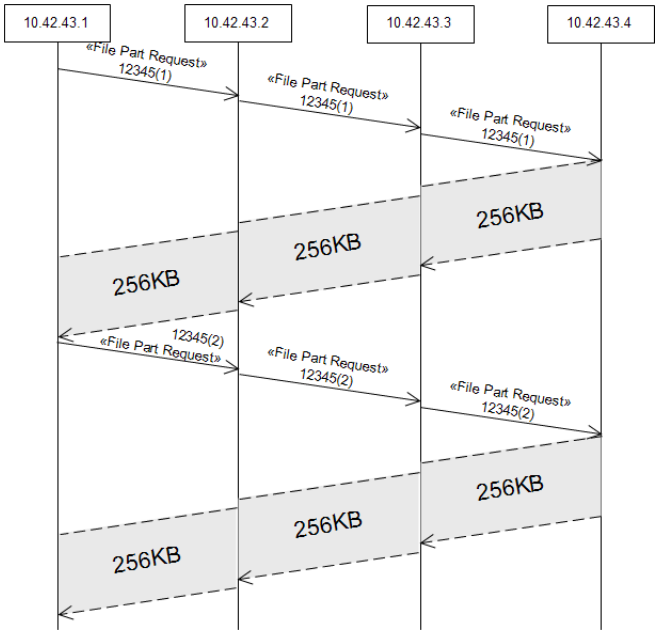
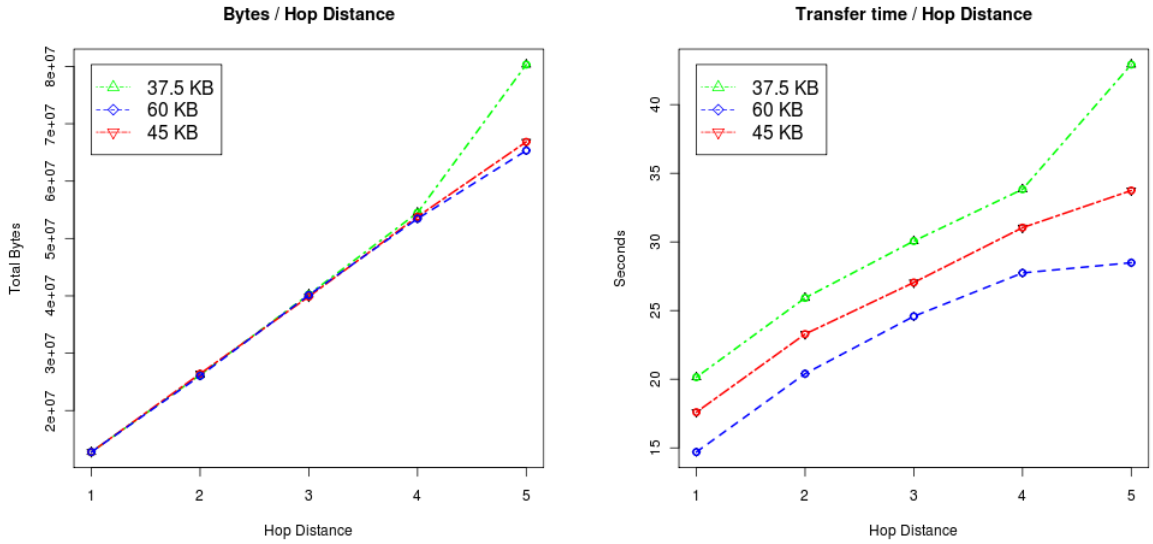


Figure 4.9: Transferring 512KB of data using 256KB file parts



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

Figure 4.10: Transfer Results (Chunk Size Comparison)

so maybe it is needed more than 1500 milliseconds to download a file part. That suggests that the download wait time value is closely related with the part size. That is somewhat obvious, because if we have bigger parts, we need more time to download each part. This also demonstrates that we need to adapt the download wait time, each time we change the file part size.

4.4.4 Chunk Size

Tests with chunk size were made using chunk sizes of 38400 and 46080 bytes. It was not possible to double the value due to the UDP packet size limitation. The results are shown on Figure 4.10.

Reducing the chunk size to 46080 produced similar results in terms of total bytes transferred. The transfer time increased for each hop proportionally. That can be explained with longer CPU tasks, because there are more chunks to handle for each part. Besides that, more confirmation packets are needed to confirm the file part chunks. If the number of chunks doubles, the time the serving node waits for confirmations also doubles. If a confirmation message takes about 50 milliseconds to arrive after the chunk is sent, case the file part has 10 chunks, the node waits approximately 500 milliseconds on each file part. Case the file part has 20 chunks, the node waits approximately 1000 milliseconds. So, what we can conclude from here is that the transfer time is inversely proportional to the chunk size. That happens because we implemented an acknowledge mechanism to control the flow of chunks. This mechanism was not initially planned, but it was implemented due to some problems that we encountered when trying to send the chunks sequentially. These problems are further explained on section 4.6. So we have a scenario like the one illustrated on Figure 4.11. When we have more chunks, the waiting time to download a file part increases. We can deduce a formula to describe the overhead in terms of time, for each file part due to acknowledgements: $delay = RTT * (NC - 1)$ assuming that RTT between the different nodes is similar, being NC the number of chunks. In the example of

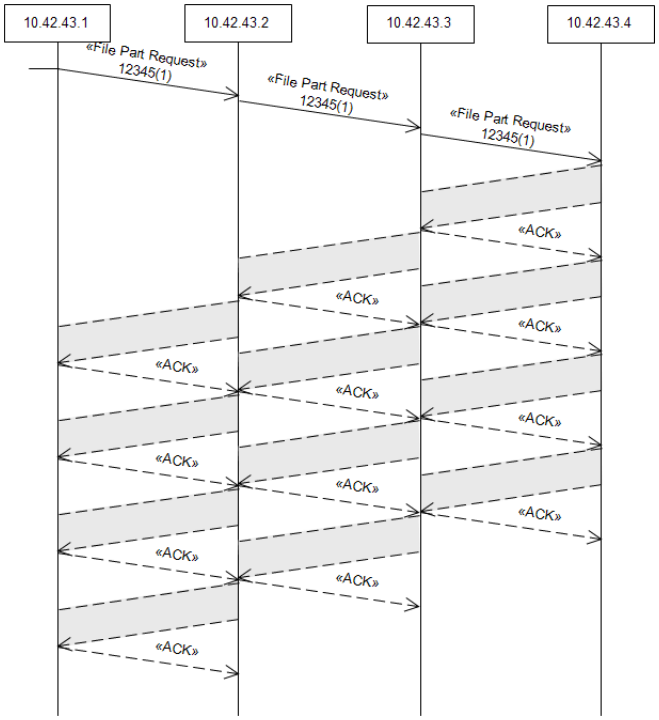


Figure 4.11: Transferring a file part

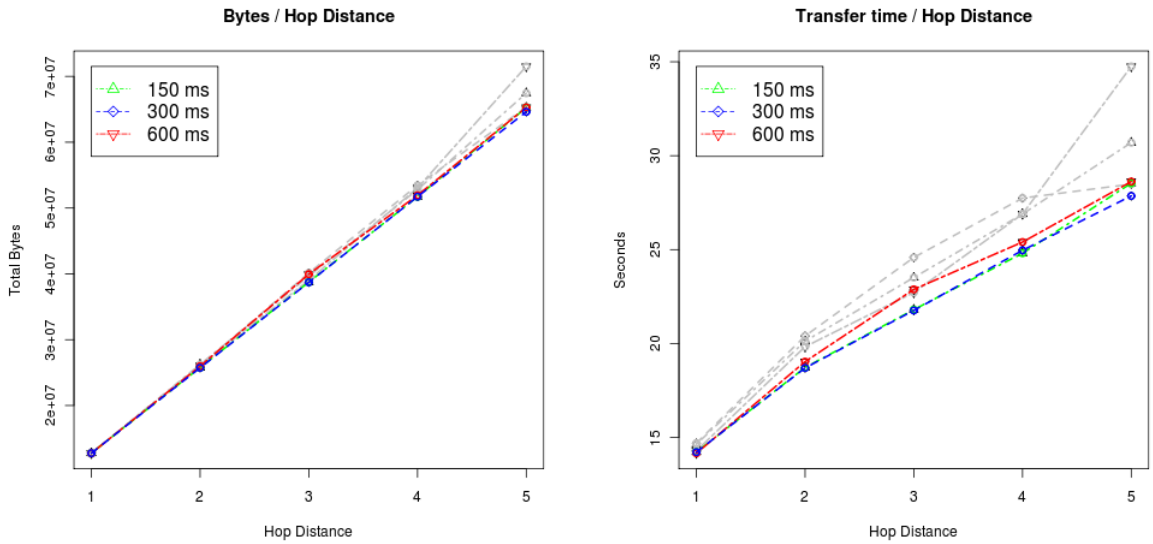
Figure 4.11 there are 4 chunks, so the delay due to acknowledge mechanism is approximately 3 RTT.

As shown on Figure 4.10(a), the overhead for 5 hops with chunks with 38400 bytes of size is slightly bigger, suggesting retransmissions. As stated before, on the part size comparison, if the time to download each part with smaller chunks increases, the overall transfer time increases. This implies that the download wait time must be adapted too, when reducing the chunk size.

4.5 Tests on New Hardware

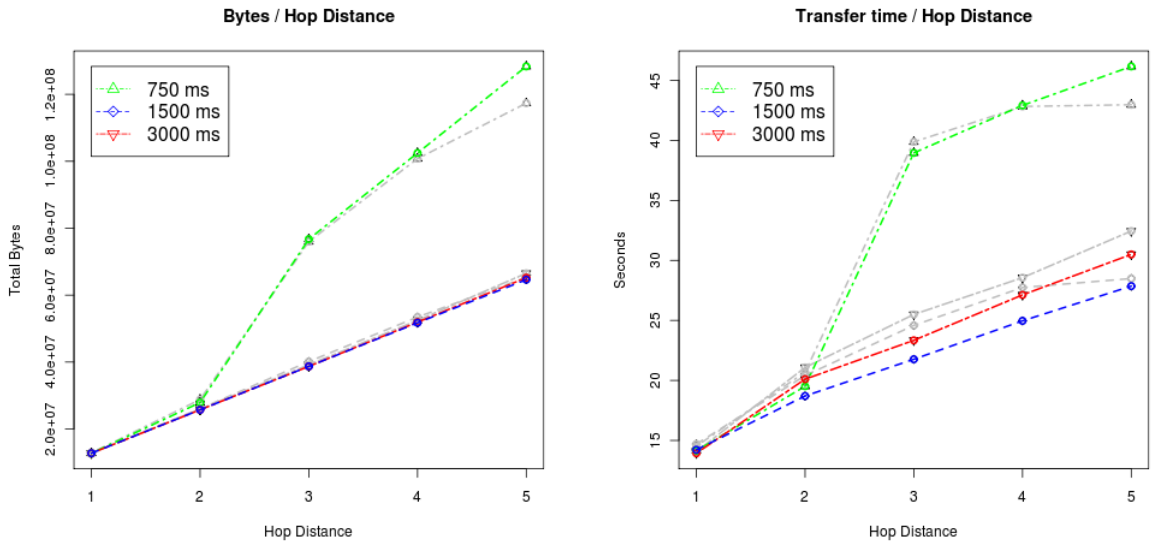
During the testing phase and after the first set of tests performed, the testing machine hardware was upgraded by system admin. The CORE installation migrated from the AMD Opteron 275 at 2,2Ghz hardware to an Intel Xeon E5335 at 2Ghz with the same 512MB RAM. At first that seemed to be a problem, because tests with new variables could not be directly compared with previous ones. But it was also an opportunity to analyze the impact of hardware change. So, as a starting point, we decided to repeat all the tests done previously with the new machine setting. The goal was to perceive the impact of different hardware on our prototype's performance. Some differences were expected, especially in terms of download time, because if the hardware of the simulator is different, also the time to run tasks is expected to be different.

Before running the simulations, we have done some modifications to the initial script. Initially, the server and client node were chosen in a random way. We verified that the number of simulations with 4 and 5 hops was very low compared to other hop numbers, especially for 5 hops. Some results for 5 hops were based on only 4 or 5 experiments. We decided to change the simulation script to force more runs with 4 and 5 hops. With this change, we expect to obtain less difference for 5 hops than



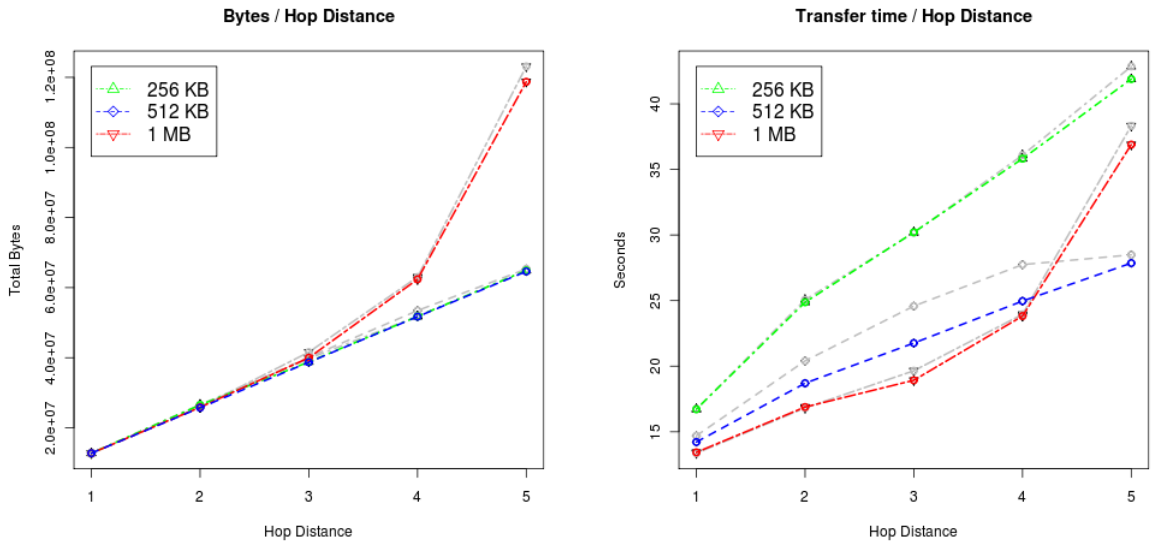
(a) Transfered Bytes vs Hop Number (b) Transfer Time vs Hop Number

Figure 4.12: Hardware Comparison (Chunk Timeout)



(a) Transfered Bytes vs Hop Number (b) Transfer Time vs Hop Number

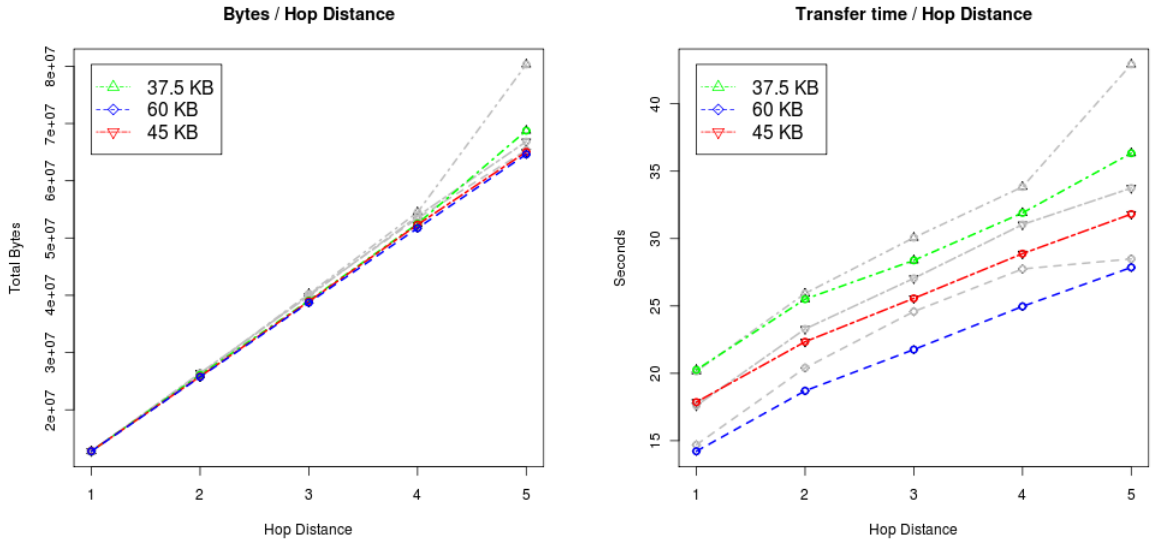
Figure 4.13: Hardware Comparison (Download Wait Time)



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

Figure 4.14: Hardware Comparison (Part Size)



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

Figure 4.15: Hardware Comparison (Chunk Size)

the results obtained so far.

Figures 4.12, 4.13, 4.14 and 4.15 show the new results obtained. The previous results are shown in gray color to illustrate the differences.

What we observe from these new results is that they are slightly better than the first ones. Besides, the strange behavior observed for 5 hops on the first tests was not observed in this new version. That suggests that the strange differences on the results observed for 5 hops previously were related the number of experiments done for 5 hops. Due to the hardware change, we conclude that this new hardware performs better than the first one. Besides, if we look at Figure 4.15(b), representing the comparison of results with different chunk sizes, we can see that latency changes for each chunk size are in similar proportions to the changes previously observed. That suggests that the time increase has more to do with the number of times each node waits for confirmation to send the next chunk than with the processing needed to deal with more chunks. As explained before, smaller chunk sizes, imply a bigger chunk number and more confirmations for each file part. Another important fact is that the results were more in a linear form than the previous ones. That reflects our changes done to the script, generating more 4 and 5 hop scenarios in order to get more significant results.

4.6 UDP Buffer Size

In a later stage of the testing phase, we discovered one of the major explanations to the packet loss verified before. We discovered that the Linux Operating System default UDP buffer size act as a limitation. According to [45], there is a limitation imposed by the operating system regarding the UDP buffer size. So, when we create a datagram socket its buffer size is limited by the operating system. When CPU is available for the application to receive the UDP packets, the buffer size does not need to be very large. The problem is that there is more than one application running at a time and even in the same application there are several different processes racing for CPU time. Authors in [45] state that UDP receiving buffer serves to match the arrival rate of UDP packets with their consumption rate by an application program. Upon receiving an UDP packet, in case the buffer is already full, it is discarded. Our application is build using UDP as the transport protocol. Obviously the buffer size as impact on the performance of our system. Sending all chunks from a file part without confirmation will result on chunks being lost due to the buffer size limit.

In order to change the maximum UDP buffer size on Linux operating system, we can use the following code:

```
sysctl -w net.core.rmem_max=8388608
```

Changing the buffer size on other operating systems is explained on [45].

Having this new knowledge in mind, we decided to adapt our application to take the maximum UDP buffer size of each node in consideration. So we adapted the HELLO messages to include the maximum buffer size in bytes. With this adaptation, each node knows how many chunks each neighbor can store, before starting to discard them.

Initially when we perceived the UDP packet loss issue, we decided to perform a very simple test. We implemented a simple server and respective client. The server was responsible of sending 10 chunks of 60KB of size, consecutively. The client was just responsible to receive them and count the number of packets received. Then we ran the test on localhost, in other words, the server and the client were on the same machine. Surprisingly, the client just counted 1 packet received! The following 9 packets were lost! We concluded that the network was not the problem, because the tests were performed in the local machine only. Still, we did not know the cause of this at the time.

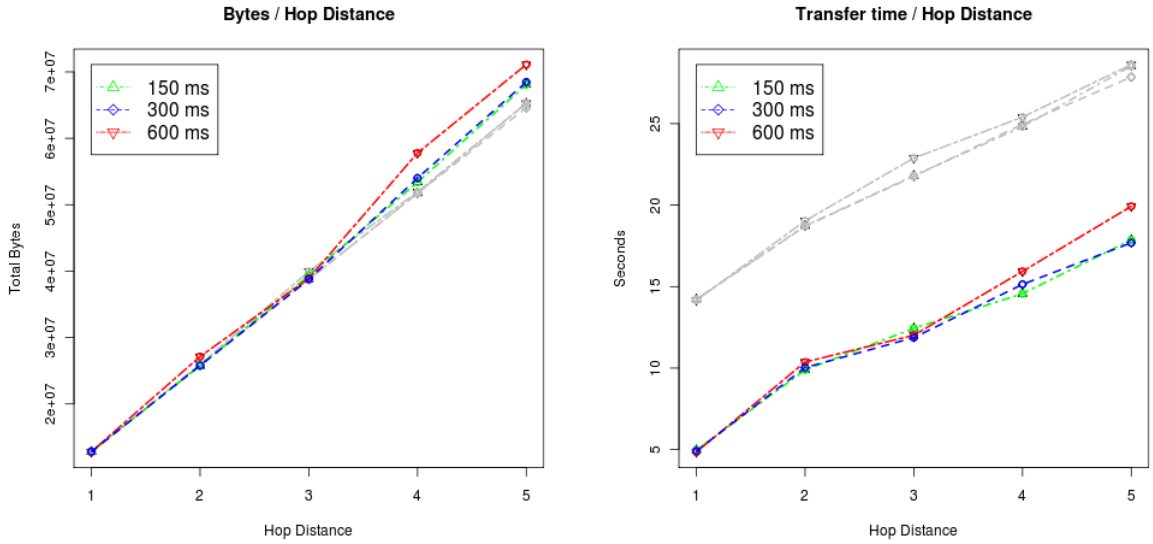
When we discovered the cause of this strange packet loss, we decided to incrementally increase the buffer size within java, without changing any OS parameter. The client was now able to receive 4 packets. Still, that is not enough, 6 packets were lost. At this time, we decided to change the maximum UDP buffer size imposed by the OS. We changed to a value approximate to 8MB. That is more than enough to receive the 10 packets of 60KB each. Then, we repeated the test and this time the client received 10 packets. That was a big victory, because we finally discovered an explanation to the mysterious packet loss verified before.

Resuming what has been changed, we can just say that now, each node calculates how many chunks a neighbor can store on its buffer and sends them right away without waiting for confirmation. If the buffer can receive a total part, then the serving node never waits for confirmations. Imagine that a file part is composed by ten chunks and the buffer size can receive five chunks. In this case, the serving node sends the first five and waits for confirmation for the fifth chunk, sending the next five chunks afterwards.

After these changes, we obtained new performance values. We set the maximum buffer size to 8388608 bytes and repeated the tests. The results obtained are shown on Figures 4.16, 4.17, 4.18 and 4.19.

We can see that for 1 hop distance, the download time drops brutally to approximately 4 seconds. The download time value was generally much lower for every tested scenario compared to the tests done before. Although, the global overhead was bigger, suggesting retransmissions. Being the downloads faster with greater overhead, it can be concluded that the retransmissions were due to checksum errors. If packet loss occurred, the application should wait for timeout before requesting the same file part again, implying slower downloads. The amount of parts with errors is bigger in this new version. We can conclude that these errors occur because there are more data being transferred simultaneously on the network. There is only one CPU to emulate all 10 nodes from our test topology because we are using CORE. This new version of the application is much more CPU intensive relative to the last version. Since each node does not wait for confirmation to send the next chunk, the server and intermediate nodes are constantly sending data. That can explain why there are more errors and also the difference in terms of time and overhead. The results on this new version vary much more than the previous one. In order to understand if the overhead results are due to network congestion or due to errors induced by the emulator, some tests on a real scenario are needed.

In terms of chunk timeout, we can see that the results for 150 and 300 milliseconds are very similar. The results for 600 milliseconds are slightly worse. There is more overhead and slower transfers. In



(a) Transfered Bytes vs Hop Number

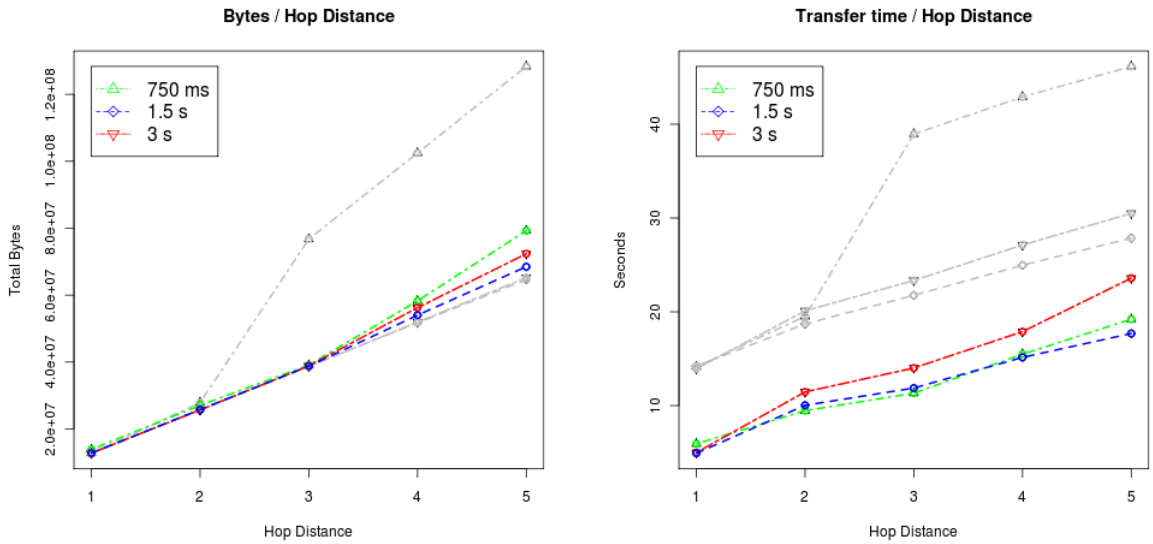
(b) Transfer Time vs Hop Number

Figure 4.16: Buffer Comparison (Chunk Timeout)

this new version, the application only waits for confirmation of the last chunk. In theory, it should not have impact if we choose a bigger value. Doubling the time each node waits for confirmation implies that threads will be alive for more time. This implies that CPU is needed for longer periods on each node, slowing down the global performance. That seems to be the best explanation for the differences observed. Although, we can only be sure of this assumption with tests on a real scenario. On a real scenario we have one CPU for each node available, instead of having one CPU to emulate all nodes as we did on CORE.

The results of download wait time with 750 and 1500 milliseconds are very similar. Although, the overhead for 750 milliseconds is bigger suggesting that there are more retransmissions due to timeout. For 3 seconds, the download time is bigger, suggesting that there were packet loss and the application waits more time to request a retransmission. Nevertheless, the 3 seconds value generates less overhead when compared to 750 milliseconds because the timeout occurs less. For this scenario it seems that the 1500 milliseconds value suits best because the transfers are faster with smaller overhead.

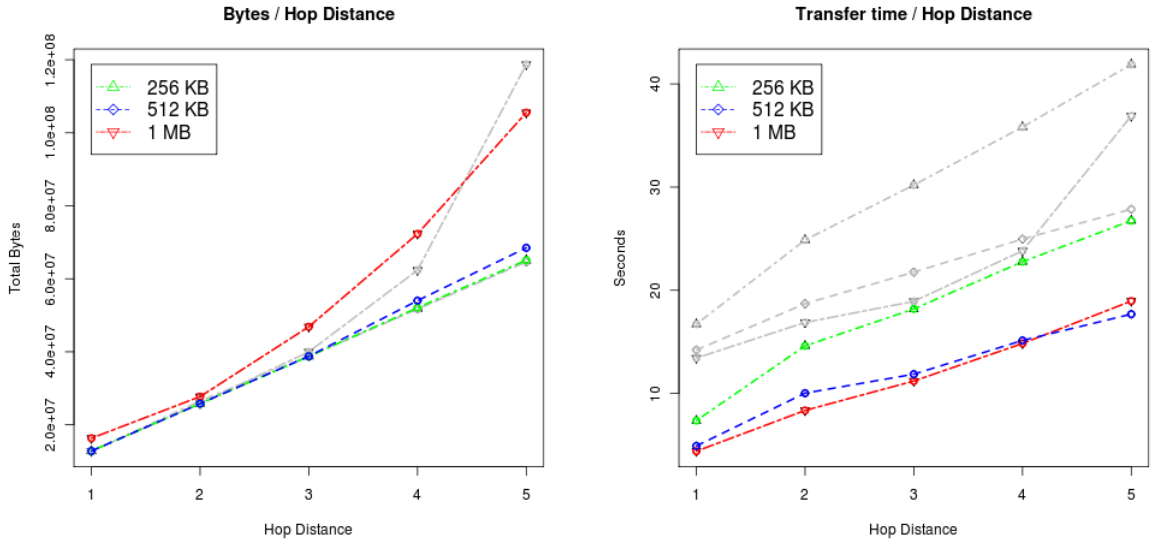
Reducing the part size produced the expected results. We can see from Figure 4.18 that parts with 256KB of size result on slower transfers. Although being slower, there are less errors and packets loss because the overhead is minimal. Parts with 1MB of size result on faster downloads, except for 5 hops. In theory, as we concluded before, bigger parts imply faster downloads and less overhead because there are less file parts to request and there is more data being transferred consecutively compared to smaller parts. The transfer is generally faster with file parts with 1MB of size, but the overhead is bigger. In theory, that should not happen, but having bigger file parts implies that each part has more chunks. The risk of a chunk getting lost or corrupted for each part increases. On section 4.4.3 we concluded that 1MB file parts take more time to transfer compared to 512KB parts. We concluded that the download wait time value may not be suited for file parts with 1MB. This is still true, but the time results obtained now are much lower than the ones obtained before for the same 1MB file parts.



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

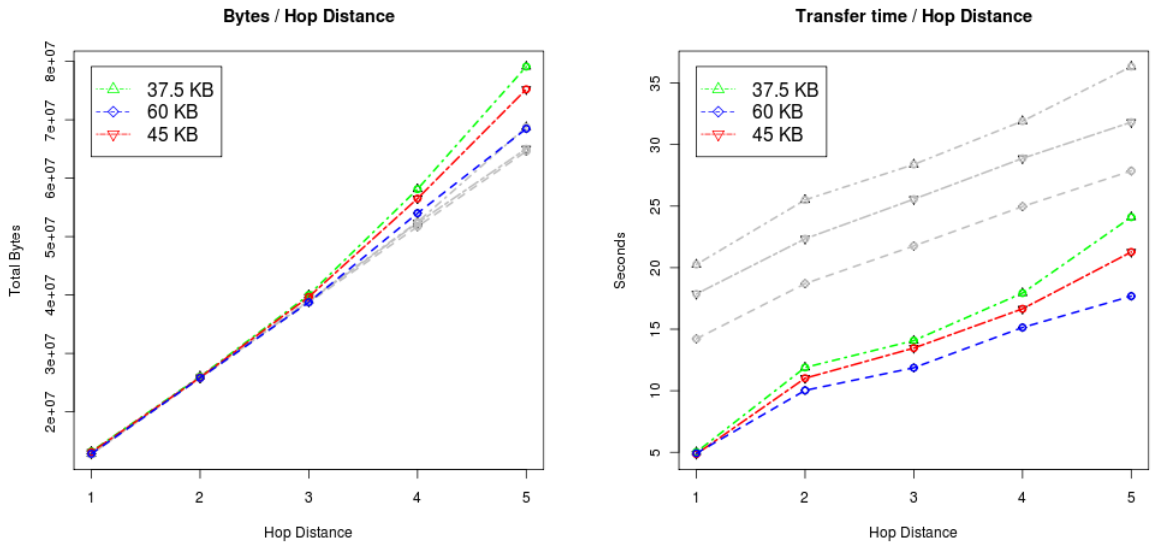
Figure 4.17: Buffer Comparison (Download Wait Time)



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

Figure 4.18: Buffer Comparison (Part Size)



(a) Transfered Bytes vs Hop Number

(b) Transfer Time vs Hop Number

Figure 4.19: Buffer Comparison (Chunk Size)

We can conclude that now the file parts are transferred faster. This may lead us to conclude that the there is no retransmissions due to timeout, since the transfer is globally faster. But we can have faster transfers with more retransmissions. This happens because if we have file parts with 1MB, there is a bigger chance of chunks being lost. Each time a chunk is lost, a timeout occurs and the part is requested again. This delays the reception of the respective file part. This delay can be compensated with successfully transferred file parts. Having file parts of 1MB, each file part takes longer to transfer, but when a part is transferred with success, it takes less time to transfer than two parts with 512KB of size. So, we can have faster transfers even in the presence of more retransmissions resulting in more overhead. There are more chunks being transferred simultaneously, doubling the part size we are doubling the number of chunks being transferred for each part. The chunks are sent consecutively between source and destination without waiting for acks, the CPU is needed during more time by each node when a file part is being transferred. This CPU intensive tasks can be too much for the emulator, resulting on delays on part delivery. Resuming, the results observed for 1MB of part size can be explained by CPU performance. The results in terms of transfer time are good, but the overhead results are unacceptable. We are convicted that the overhead results are due to the problems already referenced of having one CPU to perform all tasks of all 10 nodes on the topology.

Interesting results are also obtained from varying the chunk size. Each time we reduce the chunk size, both transfer time and overhead increase. In theory, smaller chunks would produce slightly faster downloads. As we do not have confirmations for each chunk on this test, the latency for receiving the first chunk should be smaller, resulting on a file part being transferred faster. To better understand why smaller chunks should result on slightly faster downloads, we can look at Figure 4.20 representing a file part transfer with 32KB chunks and Figure 4.21 representing the same file part transfer but with 16KB chunks. We can see that in both cases, the delay of time from the moment the request is sent to the moment the file part begins to be received is approximately 3 RTT plus the time to transfer

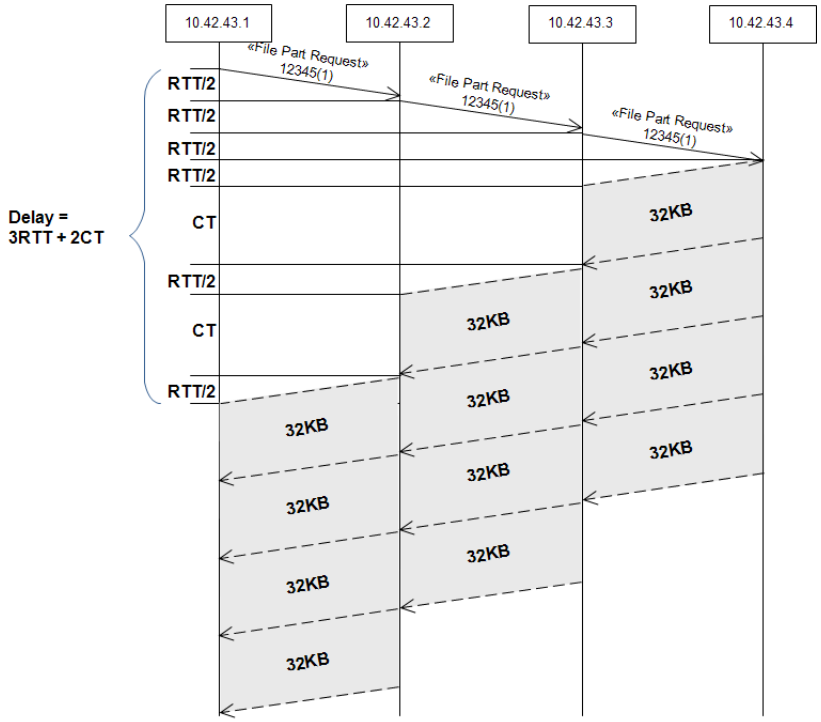


Figure 4.20: Transfer of a 128KB file part with 32KB of chunk size

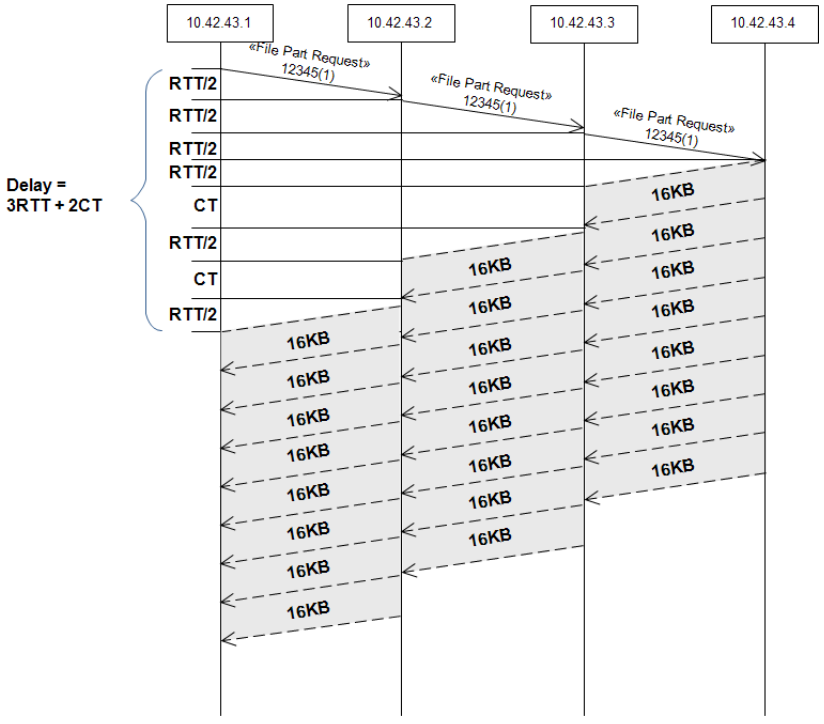


Figure 4.21: Transfer of a 128KB file part with 16KB of chunk size

2 chunks between any 2 nodes. But we can see that the data from the transfer with 16KB of chunk size, starts to being received by "10.42.43.1" a little sooner than the 32KB transfer. The delay time can be expressed by the following formula: $delay = N * (RTT + CT) - CT$, being N the number of hops and CT the time to transfer a chunk between two nodes, assuming that RTT between the nodes is similar. We can also assume that bigger chunks result on bigger CT values, because bigger chunks take more time to transfer. We can conclude that, in theory, bigger chunk sizes should imply bigger file part transfer delays.

We now observe that the results are similar to the previous ones, but the difference is not proportional to the results obtained before. As the chunks are sent sequentially without waiting for confirmation for each one, having more chunks with smaller size should not result on slower transfers. But we observe that smaller chunks produce worse results and in theory that should not happen. Again, the explanation for this fact is related with CPU tasks. Having more chunks for each file part imply that the CPU is needed more frequently at every node. The nodes now request CPU time more frequently and concurrently between each other, because there are more chunks and each chunk is transferred faster. Again we have some results that in theory should not happen. The most logical explanation is once again, related with CPU performance. We expect that in a real scenario, the application performs as expected without the CPU problems detected from testing the application with the help of an emulator.

4.7 Real Scenario

Although the unavailability of time and resources to perform deep tests on a real scenario, we decided to perform some mini-tests anyway. This is important to prove that the application works on a real scenario, not just on the emulator. Besides, it may help us to find emulator related issues affecting the results. We kept the default variables, except the part size that we adopted the 1MB value, as it produced the best results in terms of time for 1 and 2-hop scenarios. To test the application on transfers of 1-hop distance we used two machines. We used one Asus Eee PC 1005HA with an Intel® Atom™ N280 processor with 1GB DDR2 of RAM and one Toshiba Portégé R830-10R with an Intel® Core™ i7-2620M processor with 4GB DDR3 of RAM. We created a wireless ad-hoc network connecting the two machines. The IP addresses were set manually at both computers. The Eee PC acted as server and the Portégé as client. We performed 10 consecutive tests and collected the log files from both server and client.

In order to perform tests with 2-hop distance, we introduced an extra machine to the configuration. A Toshiba Satellite U300-130 with an Intel® Core™2 Duo T7500 and 2GB DDR2 of RAM. This extra machine acted as server, while the Portégé acted as the intermediate node and the Eee PC as client. We needed to ensure that the server could not interact directly with the client, otherwise we have 1-hop transfers like the tests already done. Instead of moving away the server from the client until there is no connectivity and then put the intermediate machine in the middle, we decided to keep the 3 machines in the same room. This represents an easy way to perform the tests, requiring no further physical space. Besides, another person would be needed to monitor one of the machines during the

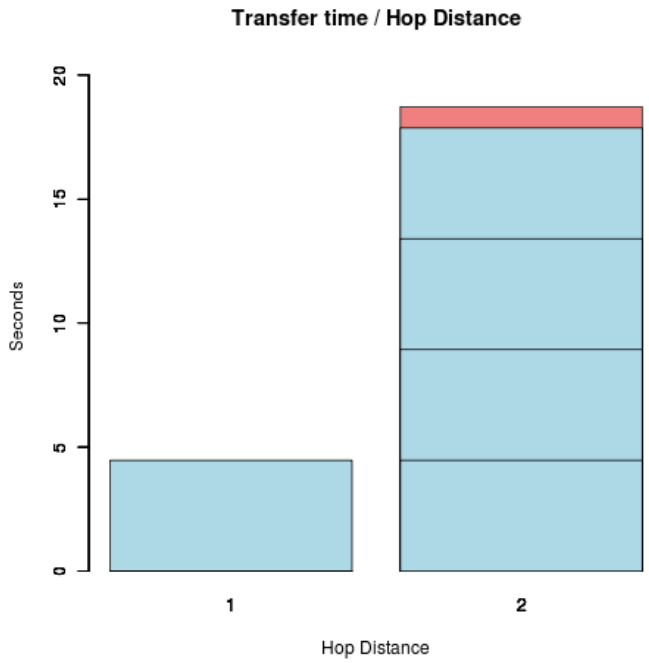


Figure 4.22: Real scenario time results

testing process. The human interactions needed to do the tests would be more difficult to perform if the computers are distant from each other. To have the 3 computers on the same room and perform the 2-hop distance transfers, we need some way of blocking the communication between the server and the client. The machines need to be connected to the same network, because the intermediate node needs to communicate with both server and client. But if we have the machines in the same network, if they are all in range with each other, they are all able to communicate directly. To simulate the 2-hop scenario, we decided to change the routing table on each machine. Using the "route add" and "route del" command, we are able to adapt each machine routing table. The routes were adapted in order to change the server and the client to be able to communicate only with the intermediate node. The intermediate machine is the only one that is able to communicate with the 2 other machines. After we changed the routing table, we executed ping commands to test if the client was able to contact the server and vice-versa and we got 'Destination Unreachable'. After that we performed 10 tests like before and collected log files in order to evaluate the results.

The results obtained from 1-hop and 2-hop tests are presented on Figure 4.22 and 4.23. The values represent the mean value from the bulk of 10 tests performed. Looking at Figure 4.23 we can see that the overhead for 2-hop transfer is slightly bigger than the double value from 1-hop results. The doubled value is expected because the traffic is forwarded on the intermediate node, replicating the traffic generated by the server. The result is slightly bigger than the doubled value suggesting some retransmissions. We can easily understand and accept the overhead results.

When we look to the results in terms of time we can see that the differences are not expected. The results from section 4.6 show that the transfer time for 2-hop transfers is slightly bigger than the double value from 1-hop transfers. Here we obtained a time more than 4 times bigger as you can see on Figure 4.22. In CORE we have an homogeneous network, each node is emulated on the same way,

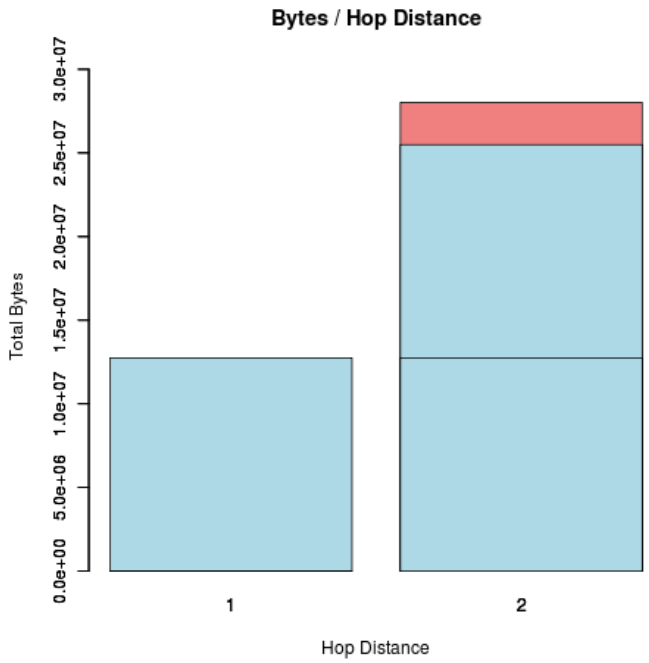


Figure 4.23: Real scenario overhead results

they are all equivalent nodes. On the real test we have an heterogeneous environment, with different machines. We conclude that the performance of the system is very dependent on the capabilities of each node. Although, the difference is huge and is not expected at all. After the results obtained, we decided to reset the routing tables and perform some pings between the 3 machines. Between Eee PC and Portégé, the ping time value was constantly $\approx$ 2ms. The ping time between Portégé and Sattelite as the ping time between Eee PC and Sattelite varied greatly. Sometimes we have times $<$1ms and suddenly we have some results like 50ms, 300ms, even 800ms! Surely there is some problem on Sattelite or with wireless signals. We were unable to find the cause for these weird results. The connections to Sattelite seem very unstable, that in turn result on slower transfers.

Other possibility that can explain the results for 2-hop is the fact that the machines are all in range. We changed routes in order to simulate a 2-hop scenario. The client may be able to listen to HELLO messages from the server and when it tries to update the list of available contents, it is not able to do that, because there is no route for it. In this case, each time the client receives an HELLO message from the server, it adds the server address to the list of neighbors and tries each time to update the list of content. This scenario represents more CPU processing on the client side that in turn may result on slower transfers. Although the client may be able to listen to HELLO messages from the server, we can anyway simulate the 2-hop scenario, because the client is not able to know which files are available at the server.

Despite the slower transfers, the number of checksum errors was generally low. For 1-hop tests there were no checksum errors, for 2-hops the number of checksum errors varied from 0 to 2 for each transfer. The number of retransmissions due to timeout on the client ranged from 1 to 4. But looking to the overhead we can see that the transfer time as more to do with the time that data takes to reach the destination, rather than the time due to checksum errors and timeouts. We can conclude that the

difference has more to due with latency of the network and latency of the CPU processing.

Ideally we should repeat the tests done on the previous sections to compare the differences obtained from the emulator and the real scenario. That was not possible due to limitations of time and resources but is certainly the next step of our prototype testing.

Chapter 5

Conclusions and Future Work

In this chapter we present the conclusions from the project developed. The main conclusions are presented on the next section. In section 5.2 we present a thorough discussion about some architecture and implementation decisions, presenting some ideas that can represent new solutions to be explored in the future to improve the system developed.

5.1 Main Conclusions

Our final result is a Java application that is able to run on a vast number of different devices and Operating Systems. The installation is very easy, the node just needs to run the JAR file. After connecting to the Ad-Hoc network and run the application, the user is now able to access the content. This is a non-intrusive application that does not change any system configuration.

We designed a content-centric network, that way the network is focused on content and not on something else. To enable a node to find content available somewhere in the network, we created an on-demand routing module based on content using a controlled flooding. That seems to us the best approach to route content, because content is delivered on-demand when the user presses a download link on the html page of our mini HTTP server.

The system created includes a mini HTTP server. With this interface, we can easily integrate with our API, allowing users to use their preferred browser. This improves the user experience since there is no need to learn how to use a new interface. Besides, the initial server that makes the content available can also 'draw' the interface using the well known HTML syntax. This enables the server to organize the contents on the page as desired. It is also possible to simulate any static html page with this approach.

The transfer performance using UDP is very dependent on the maximum buffer size limit restricted by the Operating System. If there were no limits regarding UDP buffer size, the transfers could be much faster because the buffer would be able to receive more consecutive data. The main objective of our system is to operate off-the-shelf and work with every different device that supports Java. In a future version, a module to tweak the Operating System to change the maximum buffer size could

be implemented, allowing the final user to choose from changing kernel properties and achieve higher performance or work without messing with the system and be satisfied with a medium performance.

The tests already done prove that the prototype really works. But an intensive testing phase is essential to ensure that the prototype is able to perform on a real scenario with several nodes. This would allow us to understand if the prototype is able to scale or not. The main objective from the start of this project was to achieve a functional prototype. We achieved that, but now we need to focus on a new objective, make it more robust and increase the global performance of the system. On the design process of the prototype, we tried to make each component on a simple way. Although, some other functions for the system have been thought, like authentication and cryptographic mechanisms to allow some nodes to update the contents available. This would imply some new challenges to solve and due to limitations of time we decided to start it small but achieve a complete working prototype instead of having optimized modules without having the whole application built and running.

Our routing approach is based on content and not on addresses like the typical protocols presented on section 2.2. This allows the creation of an abstraction level to the application. This way, there is no need of address handling. Each node does not have knowledge of the entire network, it just keeps contact with the direct neighbors and routes to content that are not available on local neighbors. With this abstraction layer, each node keeps information about the location where the content is available and not where are the other nodes. Each route is valid for a content and not for an address. For example, if there is a route pointing to node with address “10.42.43.10” for the file “12345”, the same route is not valid to file “54321”, even that “10.42.43.10” has the file. The route is associated with a content id and not an address. To access two different files on the same physical destination, there must be two different routes.

On the present version of our application, routes get invalidated based on timeouts. When a route is used to download a file part and a chunk is received, the route timestamp is updated. When the application tries to use a route and it does not receive chunks, the timestamp is not updated. Soon the route is deleted from the route cache and an alternative route is used. If there is no alternative route available, a new route discovery process is issued. The tests performed on section 4 are based on a static topology. First we tested the system to prove its functionality. After that we tested the impact of changing some application parameters. Other parameters could not be tested. That is the case of the route timeout values and the frequency of HELLO messages. Being a static topology, changing these variables as no impact on the performance of the application. In order to test the impact of different values for the route timeout and HELLO message frequency, the network as to be mobile.

The way our routing protocol was designed allows us to easily adapt to other scenarios as to implement new mechanisms, for example, the implementation of Route Errors similar to the DSR protocol. The versatility of the protocol allows us to perform several optimizations as content caching. Another possibility is to create routes based on some additional parameters and not based only on the file id. In theory, our routing module can be optimized to successfully deal with mobility, but to achieve that more tests are necessary. Both routing module and neighborhood module represent the starting point to optimize our system.

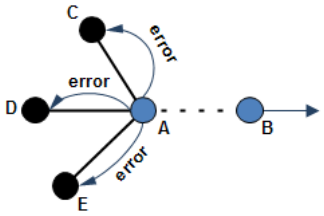


Figure 5.1: A sending error messages to C,D,E when B moves away

5.2 Discussion and Future Work

The routing is made based on an ID and not on a physical address. This makes possible the adaptation of our system to identify routes using different IDs. For example, we can change the ID from a UUID to a String. We can now represent as the ID any string that we want. This way, the application can easily be adapted to find addresses instead of files. To achieve this, we change the application in order to each node announce that it has a content with an ID equal to its address. For example, the node "10.42.43.1" announces that it has the content with id "10.42.43.1". The route discovery process generates routes to access content "10.42.43.1", resulting on routes to access the physical node. As we can see, the application can easily be adapted to operate with addresses. Besides, it can be adapted to work with the two types of routing simultaneously. Imagine node "10.42.43.1" having the files "12345", "54321" and "12321". This node announces that it has the content "12345", "54321", "12321" and "10.42.43.1". This allows the discovery process to find the files on the node and also find it by its physical address.

Routing protocol is based on IDs, so we can choose whatever we like as the ID. This enables us to adapt the application easily to fit on other different scenarios. For example, we can change the application in order to make each node announce some parameters as it was announcing content. Each node could announce something like, "cpu=1000Mhz", "ram=512MB" and "file=12345". Afterwards, some node can issue a route request for the ID "cpu>500Mhz & ram>300MB & file=12345". For this to be possible, we can adapt the application to interpret the ID as an expression and not a static ID. This would allow each node to find routes for content only to nodes with certain characteristics. With the above example, the routes found would reach nodes having CPU>500Mhz and RAM>300MB and the file 12345.

The routing protocol includes a route cache and its routes expire after some time. The intermediate nodes also cache routes and each node proactively maintains a list of neighbors. When a node detects that a neighbor is not available anymore, it can proactively invalidate each route that passes through that neighbor. Besides, it could alert its neighbors about the route failure. This notification could be similar to Route Errors from DSR. For example, in Figure 5.1 node A detects that B moved away and is now unreachable and sends errors messages to neighbors C, D and E. The error messages could be identified with the route ID or the node ID. If we choose the route ID, some other routes on neighbors that use the node will not be invalidated. The Route Error should include the ID from the node that is not available. When neighbors receive this information, they start by invalidating the routes that use the node ID from the Route Error. After that we can choose from rebroadcast the route errors

or not. If we decide to not rebroadcast the message, upon receiving a file request using the route that is no longer in cache because it was invalidated, a route error can be forwarded to the source, invalidating the route on the intermediate nodes. When the index of network mobility is very high, the overhead of broadcasting Route Errors can be very high, so instead of using Route Errors, we can keep the application unchanged, using only timeouts to invalidate routes.

To evaluate the performance of our system on mobile scenarios, some more test should be done. These tests were not made due to restrictions of time. The problem is that we need more than some minor tests. The tests in mobile scenario would imply much more time than the ones that we have already done. After getting the first results from the performance in a mobile scenario, we need to scrutinize the results obtained to interpret them successfully. After that, we need to change the application to use Route Errors to compare with the previous results. Besides, we need to infer a relationship between RTT and number of hops of each route to obtain a function that defines how much time each route should be considered valid.

The performance of the system on a mobile scenario depends mostly on the performance of the neighborhood and routing module. The performance depends on the time that modules detect network changes, as neighbors becoming out of range. To find the best parameter values for these modules, the tests on mobile scenarios are very important. We believe that the first results may not be the best, because we do not know what are the best parameter values. We should perform similar tests to the ones performed on the previous chapter. We could change the HELLO interval and evaluate the results. Similarly we can change the route timeout and compare the results allowing us to understand which values produce the best results. In theory, the transfer module operates on the same way on static or mobile scenarios. The network is abstracted by the neighborhood and routing modules. The performance depends directly on how these modules react to mobility. If we assume that neighborhood and routing modules have fresh information about neighbors and routes to content, the application is capable of achieving performance similar to static networks. We believe that tests on mobile scenarios will allow us to infer the best parameters or even deduce a function to be used to calculate the time that each route is considered valid and the time that each neighbor should be considered valid. A more mobile network, will imply more frequent HELLO broadcasts and smaller timeout values compared to less mobile networks. To determine if the network is more mobile or not, some history based mechanisms should be implemented to deduce the mobility index that can be used to calculate the best application parameter values. Currently, we have all the application parameters as fixed values. That was done in order to simplify the implementation process, as using dynamic values would imply mechanisms to infer them. Although the system uses static parameters, we have conscience that is not the best option. The parameters should be calculated dynamically to adapt to the conditions of the network.

In order to maintain updated routes during a transfer, the application should be adapted to perform route discovery constantly while the transfer takes place. This way, by the time a route becomes invalid, new routes are already available. Besides, if the content becomes available at a smaller hop number, the new route could be used to request the remaining file parts. Looking at Figure 5.2 we can see node A communicating with node B. Suppose that node A is transferring a file X from B. Node A has

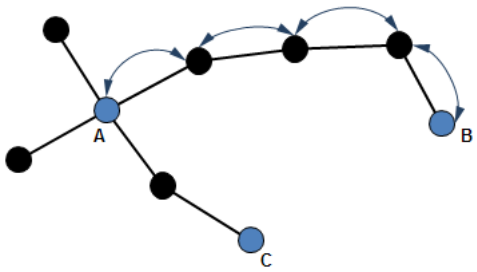


Figure 5.2: Node A transferring a file from B

already transferred 56% of the file and node C just got closer and it already has file X. Node C is closer to A than B, so node A can now request the file to C instead. For this to be possible, the application needs to be changed in order to forward the route request messages, even if there is a route in cache available. This is necessary to find more recent routes. For example, we can change the application in order to respond with a route reply in case there is a route in cache and forward the request anyway.

One of the major advantages of our protocol is that routes are based on an ID and not based on addresses. Each ID may be available at more than one physical location at a time, thus increasing the file availability. In a protocol based on addresses, when the destination moves out of range, the system as to deal with this failure at the application level. The application needs to find where is the content available now, and choose one destination to interact with. Besides, the information about the entire network must be dealt by each node. On our approach we do not need to keep information about the entire network, but only on the ones that really matter. The application does not know the entire network, it just knows content. The addresses to reach the content are abstracted by our routing protocol.

Globally, the routing protocols are based on addresses. This is done to allow applications that assume end-to-end communication between the network nodes to continue to operate. Typically, applications that were designed to operate on wired networks or on the Internet. The routing protocols try to abstract the network to the application. The packets generated by the application are the same, being responsibility of the routing protocol to forward correctly the packets in order to reach the destination. In fact, this seems the best approach to adapt existing applications to operate on a multi-hop mobile network. For applications that are being designed from scratch to operate specifically on ad hoc networks, the routing approach based on addresses may not be the most adequate. The traditional routing protocols abstract the ad hoc network to the application that in turn interprets that all nodes are available in the same way. With this abstraction layer, each node treats all other nodes as neighbors implying that each one has to deal with the information related with all of them. For example, if we look at Figure 5.3 we can see a sample topology represented. The same network is interpreted by node with address “10.42.43.1” as illustrated in Figure 5.4. With this level of abstraction, node “10.42.43.1” treats other nodes in the same way. Suppose now that we are running a peer-to-peer system like the BitTorrent and “10.42.43.1” chooses node “10.42.43.6” and “10.42.43.11” to transfer some data. The application does not know that the communication with “10.42.43.11” is supported by “10.42.43.6”. This can lead to bad choices from the application, since the topology is abstracted by the routing protocol.

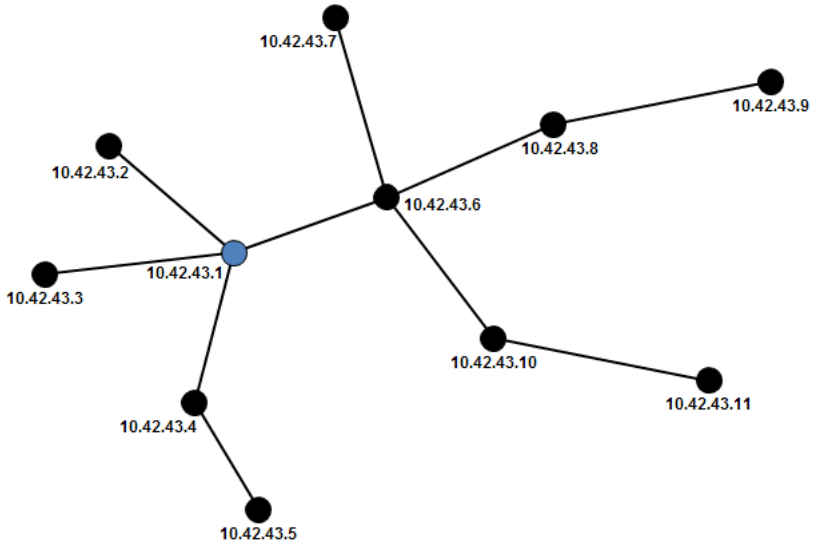


Figure 5.3: Network Topology Example

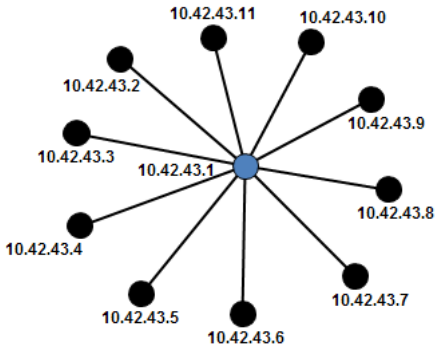


Figure 5.4: Abstraction layer created by typical routing protocols

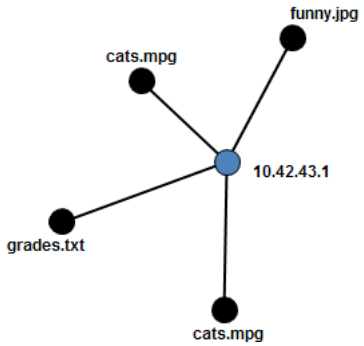


Figure 5.5: Abstraction layer created by our routing module

Our routing module creates an abstraction layer at a higher level. The application does not see other nodes, it sees instead links to content like represented on Figure 5.5. The module deals with the routing process and sees the network as shown on Figure 5.3, this enables the possibility of performing cache between intermediate nodes that could not be made if we worked at the application layer using a typical routing protocol. The caching process could be achieved on several different ways. When a node receives a transfer request, it could start sending file parts to neighbors without being requested making the content available closer to the destination. Each intermediate node could also cache some file parts to serve other nodes later. Another possibility is the use of some heuristics to calculate where and when the content should be cached. In section 2.4.3 we talk about social-aware networking and some works are presented. These works use some heuristics to calculate where the content should be sent to, in order to successfully spread the content to interested users. Some similar heuristics could be used to calculate when and which nodes should cache content. For example, each node could maintain a history of requests and based on that decide if it should cache content or not. Suppose that some node X received recently several requests from different neighbors, for the same file that is available elsewhere. In this case, node X could proactively or reactively start caching the content to deliver to the nodes interested. That is a mechanism difficult to implement when a generic routing protocol is used, due to the abstraction layer created (see Figure 5.4). Caching is an important feature that was not implemented on our system but makes part of our plans for the future. In order to take advantage of our content -centric system, file parts could be cached across the nodes to increase file availability and speed up the transfer process. With part caching, a node that solicits a file part to a distant neighbor, any node in the middle would be able to respond with the file part if already has it.

The routing module could be optimized to achieve better results. Besides the optimizations already discussed, we could also adapt the application to make use of more than one route simultaneously. When there is more than one route available to a given file, in case they are disjunctive, the client could make parallel requests. Suppose that a client node starts a download process for a given file, having 2 disjunctive routes on its cache. In case the both routes are similar in terms of hop-number, the client could request the first file part using one route and the second file part using the other. The application could then be adapted to follow this algorithm and after some testing, we could decide if it represents a good improvement or not.

However our protocol seems the best suited to applications designed for ad hoc networks with the

objective of sharing content, there are some limitations. If we need to adapt an existing application to operate on ad hoc networks, our routing protocol is not able to do that. The routing process is made at the application layer, packets are not really forwarded at the transport protocol. The code of our routing protocol must be embedded in the application. The routing is based on content, so any application designed to operate with addresses directly cannot use our routing protocol.

At first glance, our routing protocol only scales to the number of contents available. Each route is valid for one file only, so if we have a thousand files then a thousand routes are needed to download all files. However, our application was designed to deliver files on-demand. When the application starts, the files are not automatically downloaded, no routes are created or used. Only when the user intends to download a file, it issues a transfer request that in turn may generate a new route for it. Typically, each user accesses a file only one time. The route is used and if no other neighbors intend to download the same file, the route is deleted from route cache. Each user issues transfer requests sequentially, so it is possible to download, for example, one hundred files having only half a dozen routes in cache simultaneously. Considering that each route is generally used only one time by each node, there is no need to pro-actively maintain routes in cache for the files available. Besides, the user mobility would invalidate constantly the routes created. Resuming, our routing protocol was designed to be reactive on-demand, because each route is typically only used one time and some others are never used, a user may be just interested, for example, on 10% of the files that are available. That makes clear our decision of implementing an on-demand routing protocol based on content. Ultimately, the decisions made were based on the approach of allowing each user to select individually which files it wants to access. The system philosophy is: "I want to access this specific file and I want it now!".

As an alternative approach to the transfer protocol, we could use a sliding window mechanism. Recovering Figure 4.9 from section 4.4 on page 64, we can see that there is a delay from downloading a file by parts. We could change the application in order to make the server send more consecutive parts, waiting for confirmations from the client to adjust the windows size or retransmit unconfirmed parts. But with this mechanism, we are assuming that each client solicits the whole file to the same destination. This is incompatible with the mechanism of using different routes simultaneously or starting downloading the file from a closer destination. In the present version of the application, each serving node is stateless, it treats all requests independently. If we decide to adopt a sliding window mechanism, the server as to keep information about the transfer, track if the client sent confirmation messages and calculate if there are packets lost and retransmit them automatically. This would increase the CPU load at the nodes. We think that adopting a stateless approach and requesting different file parts using different routes and changing routes on the middle of the transfer would produce better results on a mobile scenario. This is because the content may be available on different destinations at different hop distance during the transfer process. Obviously, if we assume a static network and only one destination available during a transfer process, the sliding window approach would produce better results. That is the scenario that we used to test our application in both cases, using CORE and the real scenario tests. We believe that the sliding window mechanism would produce better results on that specific scenario only. When we have a mobile network, during a single transfer, the serving node may become unreachable and the source may become available elsewhere. If we download the file

using parts, we can start requesting the next file parts to other destinations and we can even request different file parts using different routes simultaneously.

The tests performed were based on a simple topology with a maximum of 5-hop distance. We think that is sufficient to understand how our application performs and how would perform on a real scenario. Although, we need to test with different topologies in the future to ensure that the system's behavior is similar.

Bibliography

[1] J. Ahrenholz, C. Danilov, T. Henderson, and J. H. Kim. Core: A real-time network emulator. *Proceedings of IEEE MILCOM Conference*, 2008.

[2] I. Akyildiz, X. Wang, and W. Wang. Wireless mesh networks: a survey. *Computer Networks*, 47(4):445–487, March 2005.

[3] G. Anastasi, M. Conti, and E. Gregori. *IEEE 802.11 Ad Hoc Networks: Protocols, Performance and Open Issues*. IEEE Press and John Wiley&Sons, New York, NY, 2003.

[4] Stefano Basagni, Imrich Chlamtac, Violet R. Syrotiuk, and Barry A. Woodward. A distance routing effect algorithm for mobility (dream). In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '98, pages 76–84, New York, NY, USA, 1998. ACM.

[5] N. Beijar. Zone routing protocol (zrp). *Ad Hoc Networking, Licentiate course on Telecommunications Technology*, 2002.

[6] Chiara Boldrini, Marco Conti, and Andrea Passarella. Design and performance evaluation of contentplace, a social-aware data dissemination system for opportunistic networks. *Computer Networks*, 54(4):589 – 604, 2010. Advances in Wireless and Mobile Networks.

[7] Levente Buttyán and Jean-Pierre Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. *Mob. Netw. Appl.*, 8:579–592, October 2003.

[8] A. Carzaniga and A. L Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems (NSF-IMWS)*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, AZ, USA, 2001.

[9] K. Chandran, S. Ragbunathan, S. Venkatesan, and R. Prakash. A feedback based scheme for improving tcp performance in ad-hoc wireless networks. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 472 –479, may 1998.

[10] Imrich Chlamtac, Marco Conti, and Jennifer J.-N. Liu. Mobile ad hoc networking: imperatives and challenges. *Ad Hoc Networks*, 1(1):13 – 64, 2003.

[11] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Definition Language (WSDL). Technical report, March 2001.

[12] Bram Cohen. Incentives build robustness in bittorrent. Technical report, bittorrent.org, 2003.

[13] Bram Cohen. The BitTorrent Protocol Specification, Version 11031. http://www.bittorrent.org/beps/bep_0003.html, January 2008.

[14] P. Costa, C. Mascolo, M. Musolesi, and G.P. Picco. Socially-aware routing for publish-subscribe in delay-tolerant mobile ad hoc networks. *Selected Areas in Communications, IEEE Journal on*, 26(5):748 –760, 2008.

[15] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on tcp throughput and loss. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1744 – 1753 vol.3, march-3 april 2003.

[16] 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, 2003.

[17] Ólafur R. Helgason, Emre A. Yavuz, Sylvia T. Kouyoumdjieva, Ljubica Pajevic, and Gunnar Karlsson. A mobile peer-to-peer system for opportunistic content-centric networking. In *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, MobiHeld '10, pages 21–26, New York, NY, USA, 2010. ACM.

[18] Klaus Mochalski Hendrik Schulze. Internet Study 2008/2009. www.ipoque.com/resources/internet-studies, 2009.

[19] Gavin Holland and Nitin Vaidya. Analysis of TCP performance over mobile ad hoc networks. *Wireless Networks*, 8(2/3):275–288, 2002.

[20] Ting-Chao Hou and Victor Li. Transmission range control in multihop packet radio networks. *Communications, IEEE Transactions on*, 34(1):38 – 44, jan 1986.

[21] Philippe Jacquet, Paul MÃ$\frac{1}{4}$hlethaler, Thomas Clausen, Anis Laouiti, Amir Qayyum, and Laurent Viennot. Optimized link state routing protocol. In *IEEE INMIC'01, 28-30 December 2001, Lahore, Pakistan*, pages 62–68. IEEE, IEEE, December 2001.

[22] David B. Johnson, David A. Maltz, and Josh Broch. Ad hoc networking. chapter DSR: the dynamic source routing protocol for multihop wireless ad hoc networks, pages 139–172. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[23] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. *SIGPLAN Not.*, 42:179–188, June 2007.

[24] Young-Bae Ko and Nitin H. Vaidya. Location-aided routing (lar) in mobile ad hoc networks. *Wirel. Netw.*, 6:307–321, July 2000.

[25] Evangelos Kranakis, Harvinder Singh, and Jorge Urrutia. Compass routing on geometric networks. In *IN PROC. 11 TH CANADIAN CONFERENCE ON COMPUTATIONAL GEOMETRY*, pages 51–54, 1999.

[26] P. Leach, M. Mealling, and R. Salz. Rfc 4122: A universally unique identifier (uuid) urn namespace, 2005.

[27] Wen-Hwa Liao, Jang-Ping Sheu, and Yu-Chee Tseng. Grid: A fully location-aware routing protocol for mobile ad hoc networks. *Telecommunication Systems*, pages 37–60, 2001.

[28] J. Liu and S. Singh. Atcp: Tcp for mobile ad hoc networks. *Selected Areas in Communications, IEEE Journal on*, 19(7):1300 –1315, jul 2001.

[29] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45748-8_5.

[30] Arezu Moghadam, Suman Srinivasan, and Henning Schulzrinne. 7ds - a modular platform to develop mobile disruption-tolerant applications. In *Next Generation Mobile Applications, Services and Technologies, 2008. NGMAST '08. The Second International Conference on*, pages 177 –183, 2008.

[31] Henrik F. Nielsen, Noah Mendelsohn, Jean J. Moreau, Martin Gudgin, and Marc Hadley. SOAP version 1.2 part 1: Messaging framework. W3C recommendation, W3C, June 2003.

[32] R. Ogier and P. Spagnolo. Mobile Ad Hoc Network (MANET) Extension of OSPF Using Connected Dominating Set (CDS) Flooding. Technical report, 2009.

[33] L. Pelusi, A. Passarella, and M. Conti. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. *Communications Magazine, IEEE*, 44(11):134 –141, 2006.

[34] C. Perkins, E. Royer, and S. Das. RFC 3561 Ad hoc On-Demand Distance Vector (AODV) Routing. Technical report, 2003.

[35] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. pages 234–244, 1994.

[36] S. Rajagpalan and Chien-Chung Shen. A cross-layer decentralized bittorrent for mobile ad hoc networks. *Mobile and Ubiquitous Systems, Annual International Conference on*, 0:1–10, 2006.

[37] K. Ramakrishnan and S. Floyd. RFC 2481 A Proposal to add Explicit Congestion Notification (ECN) to IP. Technical report, 1999.

[38] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: a platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association.

[39] Mohamed Karim Sbai, Chadi Barakat, Jaeyoung Choi, Anwar Al Hamra, and Thierry Turletti. Bithoc: Bittorrent for wireless ad hoc networks. 2007.

[40] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101 –102, August 2001.

[41] James Scott, Jon Crowcroft, Pan Hui, and Christophe Diot. Haggle: a networking architecture designed around mobile users. In *Proceedings of the Third Annual Conference on Wireless On-demand Network Systems and Services*, pages 86, 78, 2006.

[42] H. Takagi and L. Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *Communications, IEEE Transactions on*, 32(3):246 – 257, mar 1984.

[43] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, DEBS '03, pages 1–8, New York, NY, USA, 2003. ACM.

[44] C. K. Toh. *Ad Hoc Mobile Wireless Networks: Protocols and Systems.* Prentice Hall PTR, December 2001.

[45] R. Valzah, T. Montgomery, and E. Bowden. Topics in High-Performance Messaging. *29West, Inc.*, 2009.

[46] S. Xu and T. Saadawi. Does the ieee 802.11 mac protocol work well in multihop wireless ad hoc networks? *Communications Magazine, IEEE*, 39(6):130 –137, jun 2001.

[47] Shugong Xu and Tarek Saadawi. Revealing the problems with 802.11 medium access control protocol in multi-hop wireless ad hoc networks. *Comput. Netw.*, 38:531–548, March 2002.