

Project CS 2012 Product Report  
Uppsala University

Daniele Bacarella  
Jon Borglund  
Paolo Boschini  
Kiril Goguev  
Farooqh Hassan  
Marcus Ihlar  
Alexander Lindholm  
Knut Lorenzen  
Harold Martínez  
Thomas Nordström  
Thiago Costa Porto  
Linus Sunde  
Kim-Anh Tran

## **Abstract**

In Information Centric Networking (ICN), content is delivered to users based on the name of the requested resource without taking its physical location into consideration. Based on the NetInf protocol, an Android application backed by an Erlang implementation of a Name Resolution Service was implemented and both products are presented in this report. By communicating with each other, the systems store, share and retrieve data objects in an ICN fashion. In situations of network congestion content is difficult or impossible to retrieve. Using ICN, the system can provide alternate transfer methods to facilitate the delivering of content.

## 0.1 Glossary

**API** - Application Programming Interface  
**CH** - Content Handler  
**CL** - Convergence Layer  
**ERNI** - Erlang NetInf  
**EH** - Event Handler  
**HTML** - HyperText Markup Language  
**HTTP** - HyperText Transfer Protocol  
**ICN** - Information-centric Networking  
**IO** - Information Object  
**IDE** - Integrated Development Environment  
**JSON** - JavaScript Object Notation  
**LRS** - Local Resolution Service  
**MAC** - Media Access Control  
**MH** - Message Handler  
**NDO** - Named Data Object  
**NRS** - Name Resolution Service  
**NetInf** - Network of Information  
**OTP** - Open Telecom Platform  
**SAIL** - Scalable and Adaptive Internet soLutions  
**SDK** - Software Development Kit  
**SQL** - Structured Query Language  
**TCP/IP** - Transmission Control Protocol/Internet Protocol  
**UDP** - User Datagram Protocol  
**URL** - Uniform Resource Locator  
**URI** - Uniform Resource Identifier

# Contents

0.1	Glossary . . . . .	1
<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	Background . . . . .	9
1.2	NetInf enabled applications . . . . .	9
1.2.1	A NetInf based web browser for Android . . . . .	9
1.2.2	An Erlang implementation of NetInf . . . . .	10
<b>2</b>	<b>Preliminaries</b>	<b>11</b>
2.0.3	Information-centric Networking . . . . .	11
2.0.4	Network of Information . . . . .	12
2.0.5	OpenNetInf . . . . .	13
2.1	Development Languages . . . . .	14
2.1.1	Java-Android . . . . .	14
2.1.2	Erlang . . . . .	14
2.1.3	Javascript . . . . .	14
<b>3</b>	<b>Goals and Scope</b>	<b>15</b>
3.1	Frontend . . . . .	15
3.1.1	NetInf Enabled Browser . . . . .	15

3.2	Backend . . . . .	16
3.2.1	Goals . . . . .	16
3.2.2	Scope . . . . .	17
<b>4</b>	<b>Product Description</b>	<b>18</b>
4.1	Frontend . . . . .	18
4.1.1	Elephant Web Browser . . . . .	18
4.1.2	NetInfService . . . . .	21
4.2	Backend . . . . .	25
4.2.1	Erlang NetInf Name Resolution Service . . . . .	25
4.2.2	NetInf Video Streaming Client/Protocol . . . . .	25
4.2.2.1	First Implementation . . . . .	26
4.2.2.2	Modified NetInf Streaming . . . . .	26
4.2.2.3	Pure NetInf Streaming . . . . .	27
4.2.2.4	Streaming Frontend . . . . .	27
<b>5</b>	<b>System Architecture</b>	<b>29</b>
5.1	Architecture Overview . . . . .	29
5.2	NetInfService . . . . .	30
5.2.1	Configuration . . . . .	31
5.2.2	RESTful API . . . . .	31
5.2.2.1	Publish . . . . .	33
5.2.3	Retrieve . . . . .	34
5.2.4	Search . . . . .	34
5.2.5	ResolutionController . . . . .	35
5.2.5.1	NameResolutionService . . . . .	35

5.2.5.2	LocalResolutionService . . . . .	36
5.2.6	SearchController . . . . .	36
5.2.6.1	UrlSearchService . . . . .	36
5.2.7	TransferDispatcher . . . . .	36
5.2.7.1	BluetoothProvider . . . . .	36
5.2.7.2	BluetoothServer . . . . .	37
5.3	Elephant . . . . .	37
5.3.1	Configuration . . . . .	38
5.3.2	Control Flow . . . . .	38
5.3.3	RESTful API Access . . . . .	38
5.4	Erlang NetInf NRS . . . . .	41
5.4.1	Architecture layers . . . . .	42
5.4.2	NetInf Messaging . . . . .	42
5.4.2.1	Named Data Objects . . . . .	42
5.4.2.2	Internal NetInf Messaging . . . . .	42
5.4.2.3	Publish . . . . .	43
5.4.2.4	Publish Message Workflow . . . . .	44
5.4.2.5	Get . . . . .	46
5.4.2.6	Get Message Workflow . . . . .	46
5.4.2.7	Search . . . . .	47
5.4.2.8	Search Message Workflow . . . . .	48
5.4.3	Dependencies . . . . .	49
5.4.4	Configurations . . . . .	50
5.4.4.1	Meaning of the config values . . . . .	51
5.4.5	Using config files . . . . .	51

5.4.6	Extracting the config parameters . . . . .	52
5.4.7	Convergence Layers . . . . .	52
5.4.7.1	HTTP . . . . .	53
5.4.7.2	UDP . . . . .	53
5.4.8	Notes on other CL . . . . .	54
5.4.9	Plug N' Play Database Wrapper . . . . .	54
5.4.10	Setup of Database Module . . . . .	55
5.5	Chunked data streaming . . . . .	57
5.5.1	Content dispatcher . . . . .	57
5.5.2	Stream handler . . . . .	57
5.5.3	HTTP client handling . . . . .	58
5.5.4	HTML5 interfaces . . . . .	58
5.5.5	Difference between implementations . . . . .	58
5.5.6	Advantages and disadvantages . . . . .	59
<b>6</b>	<b>Evaluation and testing</b>	<b>60</b>
6.1	Frontend . . . . .	60
6.1.1	Test Setup . . . . .	60
6.1.2	Hardware . . . . .	61
6.1.3	Limitations . . . . .	61
6.1.4	Results . . . . .	62
6.1.5	Discussion . . . . .	64
6.2	Backend . . . . .	65
6.2.1	Video streaming protocol evaluation . . . . .	65
6.2.2	Pure video streaming evaluation setup . . . . .	66
6.2.3	Modified video streaming evaluation setup . . . . .	66

6.2.4	Results . . . . .	67
6.2.5	Discussion . . . . .	67
6.2.6	Notes on Interoperability . . . . .	68
<b>7</b>	<b>Conclusions and Future Work</b>	<b>70</b>
7.1	Conclusions . . . . .	70
7.2	Future Work . . . . .	71
7.2.1	Elephant and NetInfService . . . . .	71
7.2.1.1	Dynamic Content . . . . .	71
7.2.1.2	Search . . . . .	71
7.2.1.3	Delete Functionality . . . . .	72
7.2.1.4	NetInfService . . . . .	72
7.2.1.5	Database and Bluetooth convergence layer . . . . .	72
7.2.2	NetInf NRS . . . . .	72
7.2.2.1	Precaching . . . . .	72
7.2.2.2	Access Control . . . . .	73
7.2.2.3	Interoperability . . . . .	73
7.2.2.4	Handle large file . . . . .	73
7.2.3	Security . . . . .	73
7.2.3.1	NRS required folder creation . . . . .	73
7.2.3.2	Polling Logic . . . . .	74
7.2.4	General . . . . .	74
<b>8</b>	<b>Appendix A: Installation instructions</b>	<b>75</b>
8.1	Frontend . . . . .	75
8.1.1	Configuring Eclipse with Android . . . . .	75



8.1.2	Installing and debugging the application . . . . .	76
8.2	Backend . . . . .	76
8.2.1	Dependencies . . . . .	76
8.2.2	Script . . . . .	78
8.2.3	Riak Database . . . . .	79
8.2.4	Running the NetInf NRS . . . . .	80
<b>9</b>	<b>Appendix B: Maintenance instructions</b>	<b>82</b>
9.1	Frontend . . . . .	82
9.1.1	Default Application Settings . . . . .	82
9.1.1.1	Elephant Web Browser . . . . .	82
9.1.1.2	NetInfService . . . . .	84
9.1.2	Development Environment . . . . .	85
9.1.3	Eclipse Project Structure . . . . .	85
9.1.3.1	Elephant Packages . . . . .	85
9.1.3.2	NetInfService Packages . . . . .	86
9.1.4	Javadoc . . . . .	88
9.2	Backend . . . . .	88
9.2.1	Default Application Settings . . . . .	88
9.2.2	Development Environment . . . . .	89
9.2.3	Code and folder structure . . . . .	90
9.2.4	NetInf NRS modules . . . . .	93
9.2.5	Generating documentation . . . . .	98
<b>10</b>	<b>Appendix C: NetInf Video Streaming Draft</b>	<b>99</b>
10.1	NetInf Video Streaming Protocol . . . . .	99

10.1.1	Introduction . . . . .	99
10.1.1.1	Proposed method of retrieving chunked NDOs	99
10.1.1.2	Testing criteria . . . . .	100
10.1.1.3	Extra notes . . . . .	102
<b>11</b>	<b>Appendix D: License</b>	<b>103</b>

# Chapter 1

## Introduction

### 1.1 Background

The current architecture of the Internet is based on a host-to-host based model of communication. Much of the content transferred over the network is generated by a single source and accessed by many recipients. This type of communication is not well-suited for the current Internet architecture [15].

An alternative to host-to-host based networking is Information Centric Networking (ICN). In the ICN model data is requested and fetched regardless of its location. Currently four major specifications of ICN exist [7].

### 1.2 NetInf enabled applications

The Network of Information (NetInf) is one of four major existing ICN specifications. NetInf is designed to run independently or on top of current network topologies such as TCP/IP, UDP, Bluetooth etc [9].

#### 1.2.1 A NetInf based web browser for Android

When many people in a confined area use 3G-devices to retrieve content simultaneously there is a high load on the common 3G uplink. The *Elephant* web browser is designed to enhance the browsing experience under such

conditions by employing an information centric approach to the retrieval of web content.

### **1.2.2 An Erlang implementation of NetInf**

In order to support the Elephant web browser an implementation of the NetInf specification has been developed.

## Chapter 2

# Preliminaries

### 2.0.3 Information-centric Networking

The Internet was originally designed based on a host-centric paradigm (one-to-one communication), where users explicitly connect to hosts in order to use services and retrieve resources. In the early days this worked well due to the low amount of users per host, but as the internet gained popularity the use of services began to increase. Over the past decades, the host-centric approach has become a growing impediment for services with large user bases, with workarounds like load-balancing and content delivery networks to circumvent bandwidth bottle necks in place. Today most traffic involves transferring of audio/video media and social networking content, both relying on one-to-many communication. Information-centric networking (ICN) is a research field aiming to redesign the internet in a fundamental way for today's and the near future's usage patterns. In ICN, the actual host providing a specific resource or service can be arbitrary and therefore unknown to the user. Instead of connecting to a host, the user queries the network as a whole. This enables low-level caching in every network node, so that repeated forwarding of identical information can be minimised and bandwidth can thus be used more efficiently. The main challenges in ICN are the ways of addressing information units and the integration with existing, host-centric networks. At this time ICN only exists in the form of independent research projects (e.g. NetInf), with no cross-industry standards on the horizon yet [15].

## 2.0.4 Network of Information

Network of Information (NetInf) was one of the first approaches proposed by the 4WARD project [3]. This ICN paradigm was intended to deal with the issues that the current Host-Centric Networks suffer from. Every object on the network is called a Named Data Object (NDO) and is self-verifying. This leads to the user being able to request a certain object, an NDO, and fetch it from any source without worrying who or where it gets it from. The NDO can verify itself by using its own hash value as part of its name along with the used hash-algorithm. A lot has changed in NetInf since the 4WARD project made the first draft. Currently the most recent versions are managed by the SAIL project [6]. Figures 2.1 and 2.2 show, at the conceptual level, two ways how NetInf can handle NDO requests.

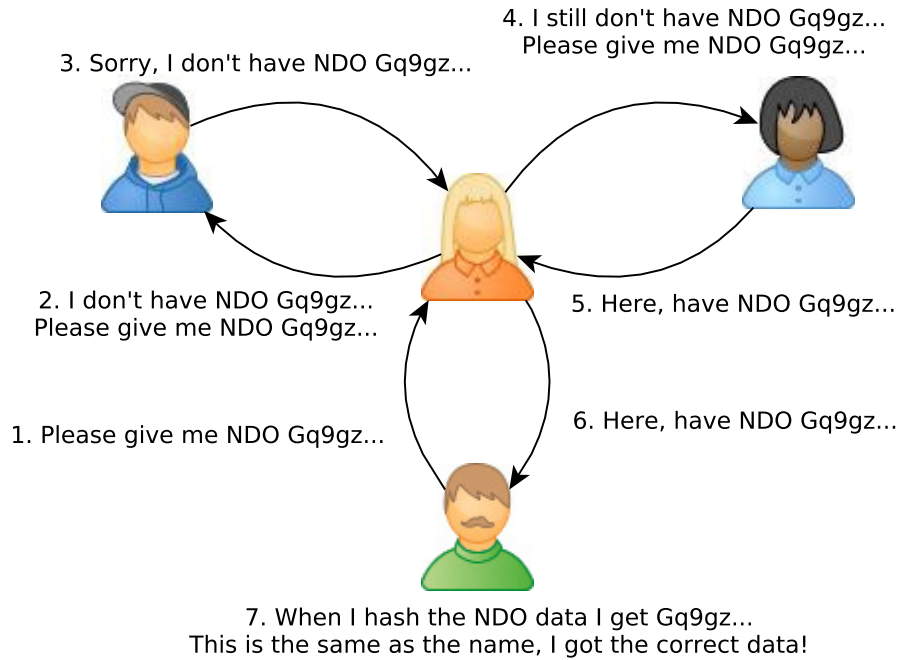


Figure 2.1: Handling an NDO request using forwarding

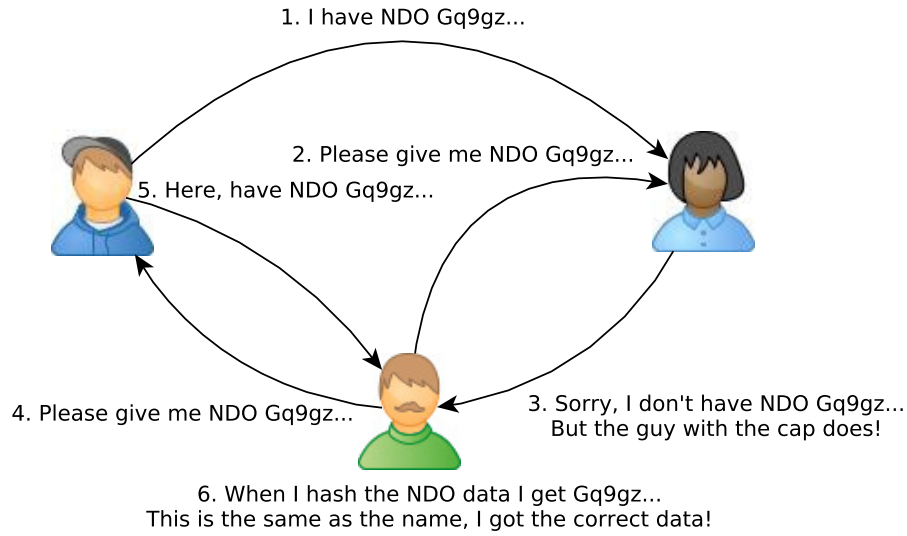


Figure 2.2: Handling an NDO request using locators

### 2.0.5 OpenNetInf

OpenNetInf [8] is an open source Java implementation of NetInf developed at the University of Paderborn. OpenNetInf is still in the very early development phase and mainly aimed at research. The frontend development team decided to use OpenNetInf as a starting point for the Android client's NetInf functionality, but still had to implement and extend it to provide additional functionality. One reason for choosing OpenNetInf, rather than starting from scratch, was to avoid reinventing the wheel. It was also a chance to contribute to an existing project. Another reason was the closely related work done in a previous master thesis [14] which used OpenNetInf.

## **2.1 Development Languages**

### **2.1.1 Java-Android**

Elephant was developed for Android, an operating system targeted at mobile devices. The fact that both OpenNetInf and a closely related previous master thesis [14] also used Java further solidified this decision. Java is an imperative object-oriented programming language. Access to Android API libraries and other tools needed for development are provided by the Android SDK.

### **2.1.2 Erlang**

Erlang is a concurrent, functional, fault-tolerant language with great scalability and ease of distribution. It was developed by Ericsson in the mid 80's and became open source in 1998.[12] These factors among others such as the customer being Ericsson Research made Erlang the language of choice for the NRS implementation. Another reason for choosing Erlang is that it uses the idea of modules and nodes as a primary platform for serving a function. This allowed the product to be broken up into several parts, supporting concurrent development.

### **2.1.3 Javascript**

Javascript [5] was used when the backend development team decided to create a simple HTTP interface to the NRS in order to show a proof-of-concept (NetInf streaming). Javascript was used to calculate the hash of files for streaming as well as for asynchronous communication with the NRS.



## Chapter 3

# Goals and Scope

### 3.1 Frontend

#### 3.1.1 NetInf Enabled Browser

The goal of the frontend was to produce a NetInf enabled browser for Android devices. The browser utilizes NetInf technology to retrieve web pages from other nearby devices and/or caching nodes in order to reduce the usage of shared 3G uplinks. The NetInf messages should conform to the NetInf HTTP convergence layer [9]. Web pages are split into several parts to make them available from multiple sources.

NetInf protocol requires that the browser have the possibility to:

- Inject existing web pages into the NetInf network as NDOs.
- Given a traditional web URL be able to find the corresponding NDO.
- Get NDOs from other devices.

The following problems are considered to be out of scope:

- Privacy
- Security

- Dynamic Content
- Battery Consumption
- Bluetooth Congestion

Privacy and security are both very important aspects, but would require complex considerations. They are areas that require future work.

Dynamic content is relevant since a lot of content that is of interest to many users is dynamic and changes often (e.g. newspapers, Facebook, Twitter). However this adds a lot of complexity to the problem since dynamic content means the mappings from traditional web URLs to NDOs are constantly changing.

Battery consumption is a serious issue in this type of application. The application uses Bluetooth which is a battery draining technology. The simplest solution here is to let the user disable Bluetooth. Hopefully the problem of heavy battery consumption will be solved by future technological advancements.

A final problem that is not taken into consideration is possible congestion in the Bluetooth network due to a large amount of devices running simultaneously.

## **3.2 Backend**

This section describes the goals and scope set by the backend team for this product.

### **3.2.1 Goals**

The following goals were set for the Erlang implementation of the Name Resolution Service (NRS):

1. Build a Name Resolution Service (NRS) for the Erlang NetInf application.

2. Be able to publish, store and retrieve Named Data Objects (NDOs) in a NetInf network.
3. Make each NetInf node a caching node that can store NDOs.
4. Be able to stream video using the Erlang NetInf application.

### **3.2.2 Scope**

The scope of the NRS application is limited to providing all the functionalities outlined in the NetInf Protocol draft document. [9] This document outlines what information a NetInf Get, Publish and Search message should contain. It also defines how a Get, Publish and Search response message should look like. Apart from that it also covers specifications for the HTTP and UDP convergence layers. The team made sure that the application followed all these specifications accurately. Providing video streaming was not part of the scope at the beginning of this project but at the customers request preliminary(proof-of-concept) work was done. However readers should note that the video streaming is not meant to be a complete product and the development team encourages further research into it.

## Chapter 4

# Product Description

### 4.1 Frontend

The following section describes the product the frontend team developed during the course. The product consists of two different applications: Elephant, the web browser, and an implementation of the NetInf services. The architecture of these applications are described in Sections 5.3 and 5.2 respectively.

#### 4.1.1 Elephant Web Browser

The color of the spin-progress-bar indicates the medium used for fetching data.

- Red indicates the uplink connection (3G or Wi-Fi)
- Green indicates the database
- Black indicates a NRS caching node
- Blue indicates Bluetooth

Other than these colors, grey means that a search is currently in progress. The application also offers the possibility to interrupt the loading process



Figure 4.1: Loading a web page via uplink

by tapping on the the same icon used to start loading a page, as it toggles between a refresh icon and cancel icon.

Figure 4.1 shows an example view of the web browser.

To make sure that Elephant is able to retrieve resources from other mediums than the uplink, the NetInf service application must be up and running. This is because the NetInf service enables communication to other nodes in the network and to keep track of which devices have a certain resource to serve via Bluetooth.

Elephant contains some customizable settings that can be found in the menu entry on the top right of the application. These settings make it possible to take advantage of the NetInf service. In comparison to other available browsers, Elephant makes use of Information-Centric Networking as well as Location-Based Networking in an attempt to lower network congestion. In the setting page, see Figure 4.2, the user can decide if they want to share visited pages as well as upload web pages to a caching node. The first menu entry gives the possibility to register the local device as a locator that can serve other devices via Bluetooth. By enabling the second menu entry the device will not only register as a locator, but will also transfer the actual files to a NRS cache node, if any.

The last menu entry for the setting is for opening the NetInf Service's settings, so that the user does not have to switch applications manually when changing the service settings.

Functionality	Description
Search	The application searches for the hash value of the resource requested, specified by a URL. This search includes searching for the hash value within the local database and the remote Name Resolution Service (NRS). It returns the value, if found.
Get	A Get request that contains the hash value of an NDO triggers a content retrieval of the corresponding resource. The application tries to retrieve the content either from the Local Resolution Service (LRS), the NRS or from a remote Bluetooth device.
Publish	The application can register the local device as a locator for a resource specified by a hash in the NRS. This way, remote devices can try to retrieve that resource from the local device using Bluetooth.
Full put	The full put is a publish request that contains the actual content corresponding to the resource that is published. Thus, the NRS, to which the local device is connected to, can store the content and serve it itself.

Table 4.1: Functionalities

Finally, Figure 4.3 shows a simple help view presenting a brief description of the functionalities of the application, so that the user can have a better understanding of how the web browser works.

#### 4.1.2 NetInfService

The NetInfService is a stand-alone application that can be used by other applications in order to make use of Information-Centric Networking. All functionalities that are provided are listed in Table 4.1. If an application needs the ICN services, the NetInf Service application has to run in the background.

The services are configurable within a simple and self-explanatory user interface that is shown in Figure 4.4. If a user wants to change the NRS that the device is communicating with, the address as well as the port can

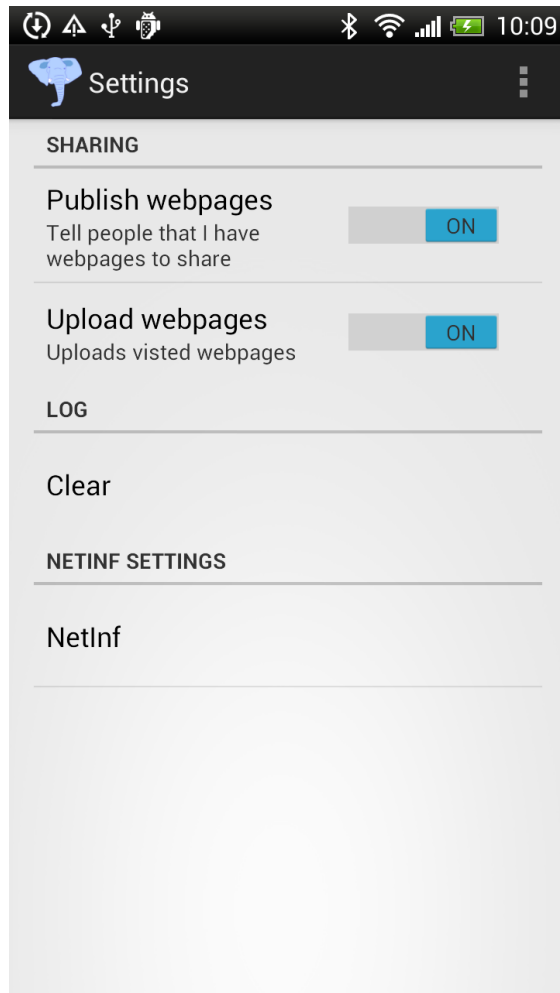


Figure 4.2: Settings view of Elephant



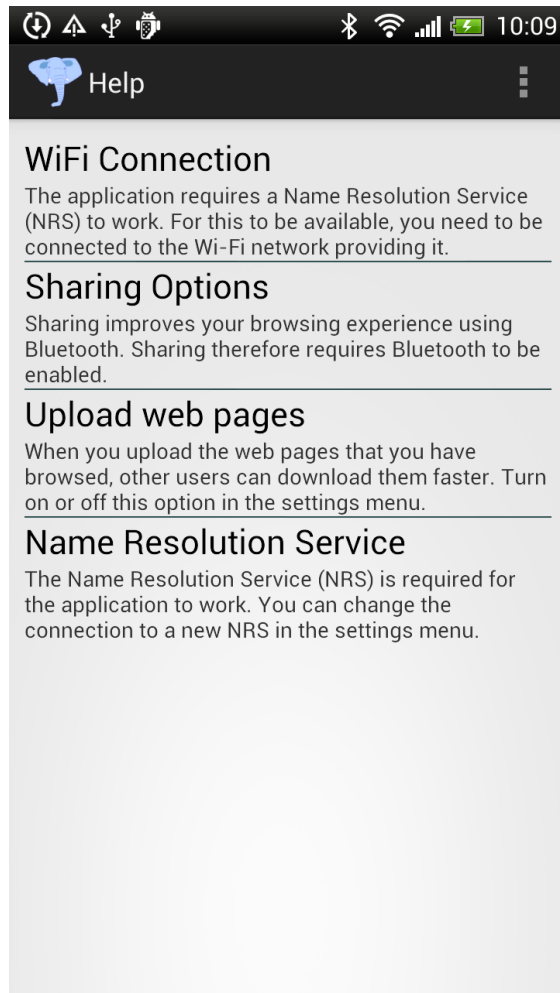


Figure 4.3: Help view

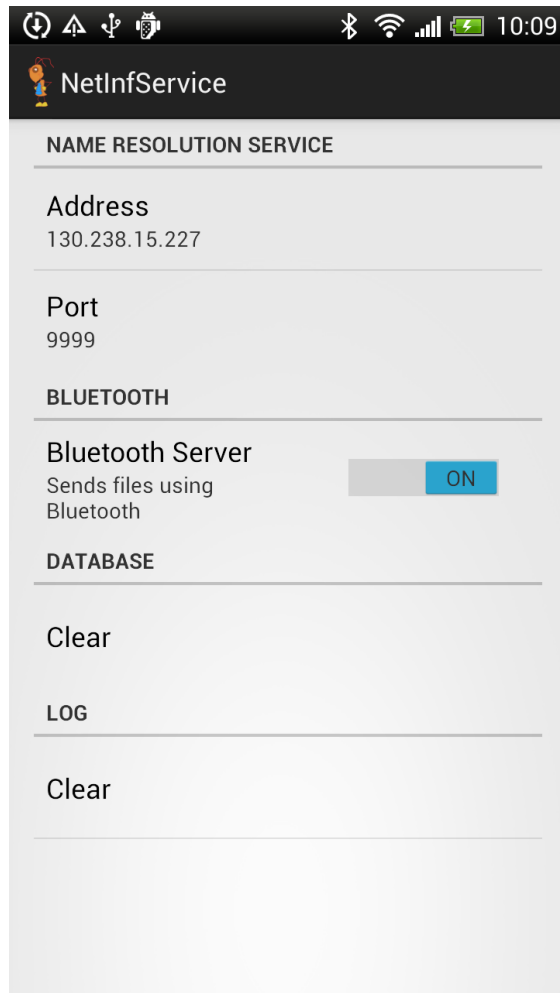


Figure 4.4: NetInf Service Settings

be changed accordingly. In addition a user can decide whether they want to share their downloaded content with other remote devices. Only if the *Bluetooth Server* is turned on, data will be shared. Finally, the database as well as the logs created during the run can be cleared. The database stores every single resource that is published. At some point, the database will contain a huge amount of old data that is not usable anymore. Clearing the database will then improve the run time since the search among stored content will perform faster. Log files are used for debugging purposes. In case a log is old, clear it and rerun the application in order to gain new logs.

## 4.2 Backend

The customer agreed on having an Erlang NetInf NRS. The backend product implements the, as of writing, current draft of the NetInf Protocol[9] in a purely functional language. The product promises a high level of scalability and fault-tolerance. The client initially asked for only the NRS as a product however the backend team was able to complete the initial product in a timely manner, allowing for applications of this network technology to be explored.

### 4.2.1 Erlang NetInf Name Resolution Service

The first of the two deliverables from the backend team to the customer. The Erlang NetInf NRS provides a new way to organize and retrieve data on the Internet. Based on an initial NetInf NRS protocol draft from development teams such as SAIL and Ericsson Research[9]. This product allows for flexibility and extension of the existing protocol.

Erlang's concept of modularization allowed the team to break up the NRS functionality into distinctive convergence layers, runtime database switching, and even allow for a proof-of-concept data streaming client/protocol.

### 4.2.2 NetInf Video Streaming Client/Protocol

The last of the deliverables from the backend team. The customer requested a proof-of-concept video streaming protocol and HTML client inter-

face which lies on top of the Erlang NetInf NRS technology. The streaming protocol is a completely new addition to the NetInf draft[9]. The team developed a way to utilize the code and transporting mechanism of the first product in order to stream video content. Along with the protocol outlined in Appendix C 10.1, the team has created a HTTP interface client which allows the user to see the streaming protocol in action in addition to accessing the NRS functionality. This particular product was not specified in the initial conversations with the customer in September, but was added late in the development cycle and is a proof-of-concept.

#### 4.2.2.1 First Implementation

In addition to normal NRS functionality, a HTTP transfer-dispatcher had to be implemented in order to transfer chunks between clients. The streaming works by clients subscribing to a stream from a specific NRS. Once connected the stream the client with a constant interval will ask the NRS where to find these chunks. All the chunks are transferred via the transfer-dispatcher. The playback of the video chunks is done by polling the local NRS, this implies that every client has its own NetInf node running. See figure 4.5. The benefit of using this approach is that only one NDO containing the filename has to be published. The receiver can then derive the chunks locations, by appending the chunk number to the end of the locators provided in the filename NDO.

#### 4.2.2.2 Modified NetInf Streaming

Due to a request from the customer a more true NetInf implementation of streaming was implemented. Instead of using the transfer-dispatcher between the client nodes a workaround was added that disabled content validation, this resulted in fetching chunks via NetInf messages. The polling logic is still the same as first implementation, seen in figure 4.5. Instead of using ordinary HTTP locators, the receiver is required to modify the *NetInf GET requests* to get the chunks. This is done by replacing the hash algorithm with the custom hash name *demo*, this fictitious hash name is used to avoid the content validation and database lookups. For example, to get the first chunk of *ni:///sha-256;abc*, the request should contain *ni:///demo;abc1*. The HTTP transfer-dispatcher is still used to transfer the chunks to the HTML-interface.

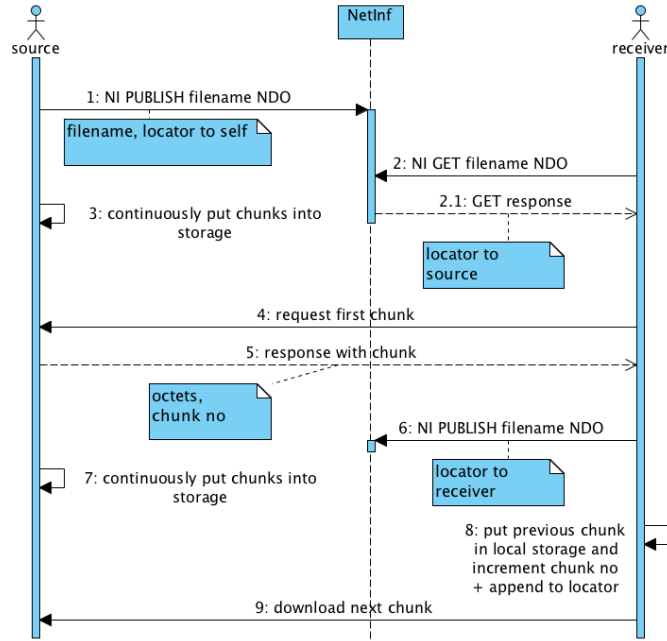


Figure 4.5: Original/Modified Chunked Data Transfer

#### 4.2.2.3 Pure NetInf Streaming

To be able to evaluate the modified NetInf streaming another implementation was added. This implementation uses NetInf searches and gets for chunks. See Figure 4.6. In this implementation the stream source is required to publish each chunk to the NRS, and modify the NDO metadata with the stream name and stream chunk number. The receiver is then required to search for each chunk to find it.

#### 4.2.2.4 Streaming Frontend

To merge the chunks, a simple HTML5 frontend was created. HTML was chosen to make the player platform independent. In both implementations the player starts a JavaScript that continuously polls the local NetInf node for the chunks through the HTTP dispatcher. The difference is that the pure NetInf player uses the NRS search to build the playlist, while the modified NetInf version only increases the chunk number.

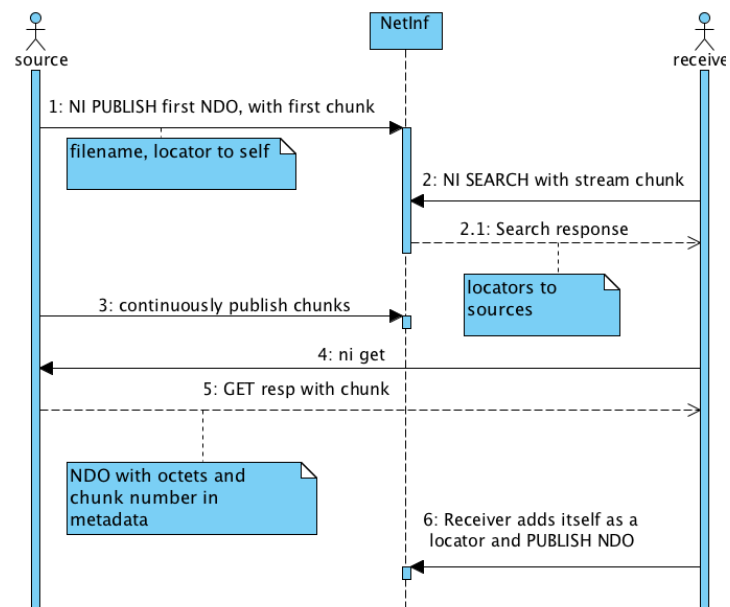


Figure 4.6: Chunked Data Transfer With Pure NetInf

## Chapter 5

# System Architecture

### 5.1 Architecture Overview

The system is a combination of frontend and backend subsystems that communicate using network protocols including HTTP and Bluetooth. Figure 5.1 shows how messages can flow between the different subsystems. All messages are of a request-response type, therefore messages are always flowing in both directions between subsystems. The messages contain NetInf get, publish or search requests except for messages emitted from the Elephant web browser to the Internet, these are normal HTTP requests.

The following sections describe the different subsystems in detail.

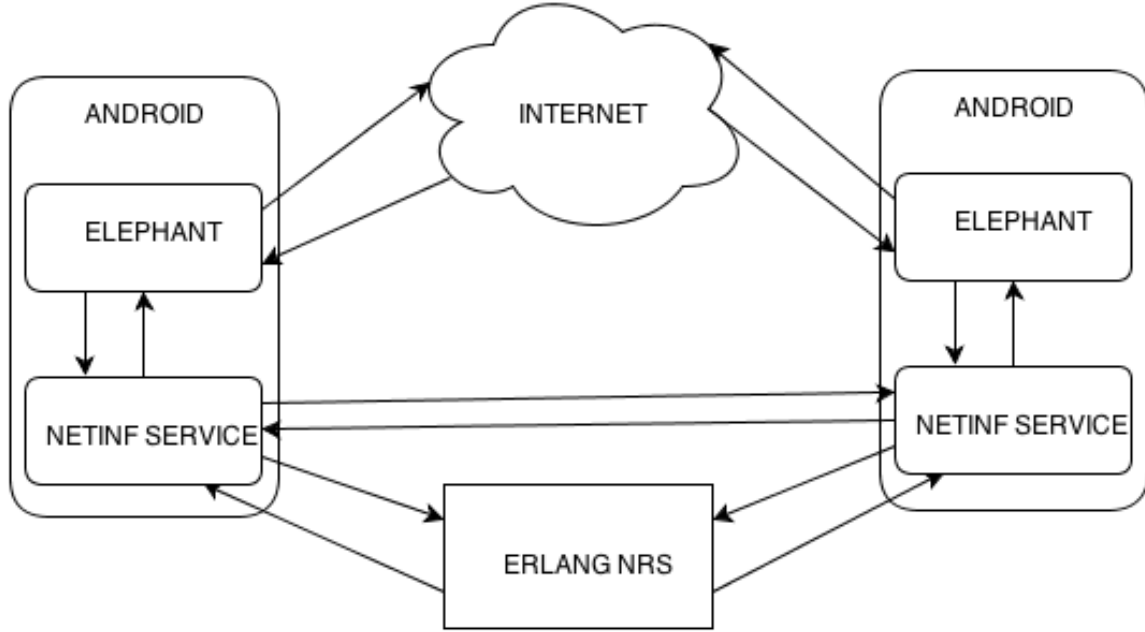


Figure 5.1: The Elephant browser initially requests an object from the NetInf service, if the object is not found the request will be forwarded to an Erlang NRS. The Erlang NRS responds with object or locators to the objects if the object is present in the NetInf network. If the object is not found within the NetInf network, the Elephant browser will try to fetch it from the Internet and then publish it to the NetInf Service which in turn will send a publish request to the NRS.

## 5.2 NetInfService

NetInfService is the first of two Android applications. It provides a NetInf node which can be accessed through a RESTful API. This allows any application running on the same device as the NetInfService to access NetInf functionality through simple HTTP requests. The interface is described in detail in Section 5.2.2. NetInfService is based on the work done in a previous master thesis [14]. The NetInfService uses and extends OpenNetInf to provide this functionality. An overview of the design of NetInfService can be seen in Figure 5.2, the individual components are described below.



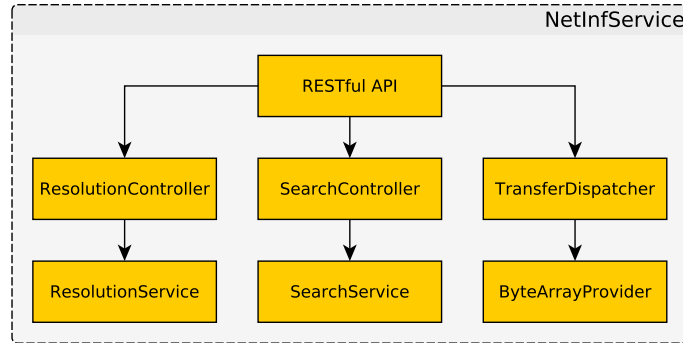


Figure 5.2: NetInfService Overview

Figure 5.3 shows a picture of the internal control flow of the NetInfService, and provides the packages and/or source files, as well as some text and arrows describing the control flow. The digits on the connections between boxes illustrate the order actions are taken.

### 5.2.1 Configuration

The default settings for NetInfService are stored in the properties file "assets/config.properties". This includes but is not limited to NRS IP address, NRS port and RESTful API port. Some of these settings can be changed when running the NetInfService on an Android device. This does not change the default values in the properties file but the changes are instead stored using the default Android shared preferences file, which is persistent.

### 5.2.2 RESTful API

The RESTful API receives HTTP requests for NetInf functionality. Depending on the type of request, the ResolutionController, SearchController or TransferDispatcher is called. Publish requests are handled by the ResolutionController. Retrieve requests first trigger a get request using the ResolutionController and, if the get response contained locators, it will transfer the file using the TransferDispatcher. Search requests are handled by the SearchController.

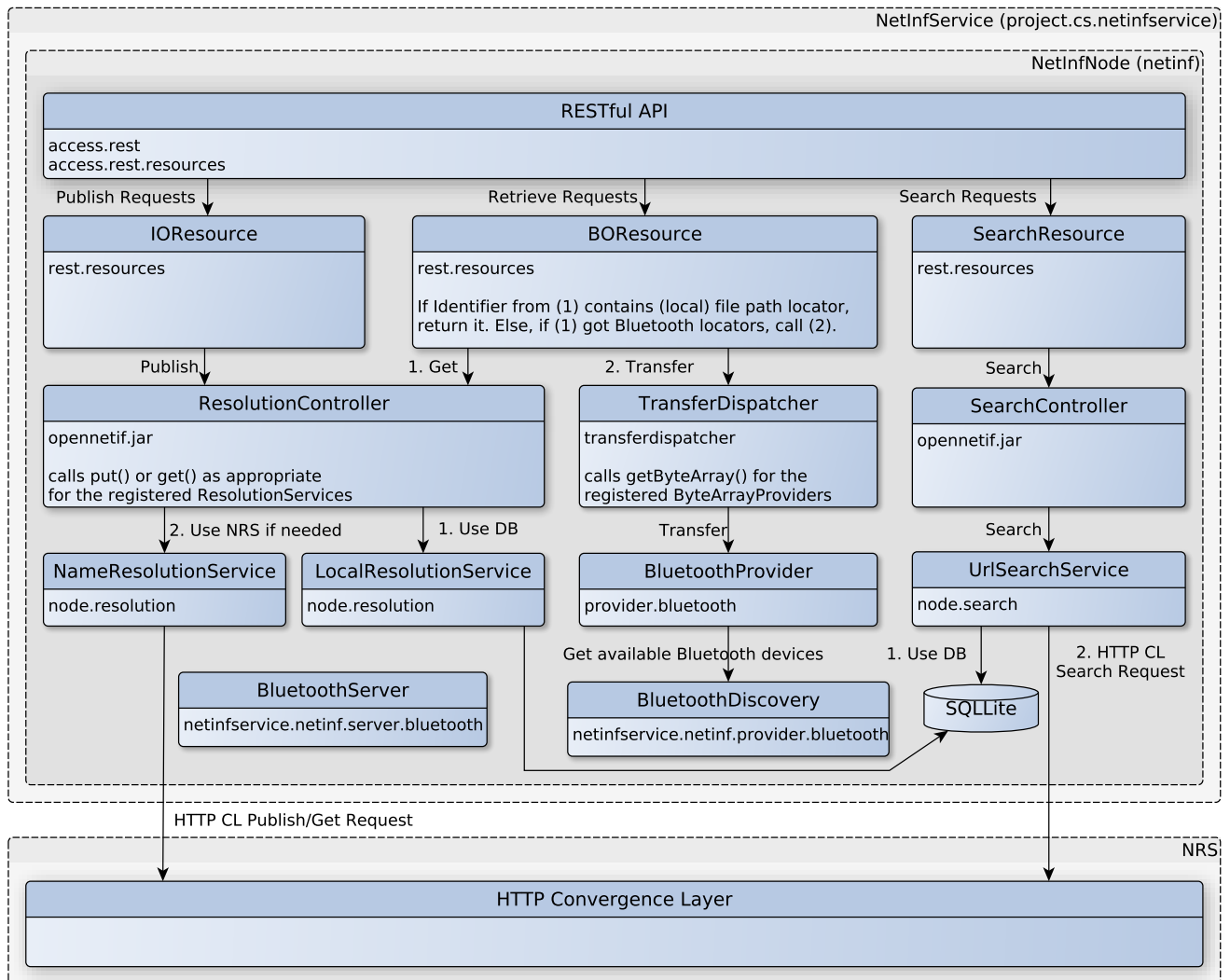


Figure 5.3: NetInfService Control Flow

The HTTP request uses the following format:

`http://{Host}:{Port}/{Prefix}?{key1}={value1}&{key2}={value2}...`

Since the NetInfService should be running on the same device as the application that is using it, the host would most likely be either localhost or 127.0.0.1. The default port is 8080, for changing these settings see Section 5.2.1. The key-value pairs must be URL encoded as they can contain illegal characters.

#### 5.2.2.1 Publish

Publish requests use the prefix publish.

They require the following key-value pairs:

key	value
hash	The hash of the object being published.
hashAlg	The hash algorithm used to hash the object.
ct	The MIME content-type of the object being published.

The following key-value pairs are optional:

key	value
btmac	The Bluetooth MAC address of the publishing device.
meta	Metadata as a JSON string.
filepath	Path to the file being publish.

- btmac: The Bluetooth MAC address of the publishing device, it will be added as a locator to the published object.
- meta: The metadata [9] of the object that is being published. Remember to URL encode illegal characters.
- filepath: The file path of the object that is being published. If this is present, a full put is done, otherwise not.

A successful publish returns HTTP status code 200. Any other code indicates that something went wrong and should give an indication of what went wrong.

Below is an example of how a HTTP publish request could look like, notice the URL encoding:

```
http://localhost:8080/publish?hash=TcoP1fQkoxsDq4B8uud%2Bsyvy0Inu0c7hVLOv7UWN4Nw
&hashAlg=sha-256&ct=text%2Fplain&btmac=F0%3AE7%3A7E%3A3F%3AD2%3A43
```

Unencoded it would look like:

```
http://localhost:8080/publish?hash=TcoP1fQkoxsDq4B8uud+syvy0Inu0c7hVLOv7UWN4Nw
&hashAlg=sha-256&ct=text/plain&btmac=F0:E7:7E:3F:D2:43
```

### 5.2.3 Retrieve

Retrieve requests use the prefix retrieve.

They require the following key-value pairs:

key	value
hash	The hash of the object being published.
hashAlg	The hash algorithm used to hash the object.

A successful retrieve returns HTTP status code 200. The HTTP response should contain a JSON object with two key-value pairs. The key path should contain a local file path to the retrieved object. The key ct should contain the MIME content-type of the retrieved object. Any other code indicates that something went wrong and should give an indication of what went wrong.

Below is an example of how a HTTP retrieve request could look like, notice the URL encoding:

```
http://localhost:8080/retrieve?hash=TcoP1fQkoxsDq4B8uud%2Bsyvy0Inu0c7hVLOv7UWN4Nw
&hashAlg=sha-256
```

Unencoded it would look like:

```
http://localhost:8080/retrieve?hash=TcoP1fQkoxsDq4B8uud+syvy0Inu0c7hVLOv7UWN4Nw
&hashAlg=sha-256
```

### 5.2.4 Search

Search requests use the prefix search.

They require the following key-value pairs:

key	value
tokens	The tokens being searched for.

The following key-value pair is optional:

key	value
ext	An optional JSON, meant for future extensions.

- tokens: Currently NetInfService only supports one token. This should be changed to accept a string of space separated search tokens to follow the specification.
- ext: The ext field is meant for future extensions. It is allowed, but currently ignored.

Below is an example of how a HTTP search request could look like, notice the URL encoding:

`http://localhost:8080/search?tokens=http%3A%2F%2Fwww.ericsson.com%2F`

Unencoded it would look like:

`http://localhost:8080/search?tokens=http://www.ericsson.com/`

### 5.2.5 ResolutionController

The ResolutionController handles a number of ResolutionServices. Each ResolutionService should provide publish, get, and delete functionality using some convergence layer or equivalent. Currently there are two ResolutionServices, the LocalResolutionService which uses a local SQLite database and the NameResolutionService which communicates with a specific NRS using the HTTP convergence layer.

#### 5.2.5.1 NameResolutionService

The NameResolutionService uses the Named Information URI [16] and the HTTP convergence layer [9] to communicate with a single NRS. Currently the NameResolutionService supports publish and get. Delete is not yet supported but the framework for its implementation is there.

#### **5.2.5.2 LocalResolutionService**

The LocalResolutionService uses a connection to the device's local SQLite database. The LocalResolutionService has higher priority than the NameResolutionService meaning that when performing a get request the LocalResolutionService will be used first and then if needed the NameResolutionService as well.

#### **5.2.6 SearchController**

The SearchController handles a number of SearchServices. Each SearchService should provide search functionality using some convergence layer or equivalent. Currently there is one SearchService, the UrlSearchService, which provides search functionality using a single token which is assumed to be a URL.

##### **5.2.6.1 UrlSearchService**

The UrlSearchService provides the functionality to search for a single token in the local database used by the LocalResolutionService. If the token is not found in the database, then a search will be performed using the HTTP convergence layer. This new search looks for tokens in the NRS, which uses the NameResolutionService.

#### **5.2.7 TransferDispatcher**

The TransferDispatcher handles a number of ByteArrayProviders. Each ByteArrayProvider should provide functionality to retrieve a file (as a byte array) in some way. Currently there is one ByteArrayProvider, the BluetoothProvider, which uses Bluetooth to transfer files between Bluetooth enabled devices.

##### **5.2.7.1 BluetoothProvider**

The BluetoothProvider provides the functionality to retrieve the bytes of an NDO through a specified Bluetooth locator. It first tries to establish

a connection to the remote Bluetooth device. If successful, it requests the bytes of an object by sending the hash value that describes the object. As a response, the provider retrieves the bytes of the object, if it was available on the remote Bluetooth device.

#### **5.2.7.2 BluetoothServer**

The BluetoothServer continuously listens to incoming Bluetooth file requests. If a remote device wants to connect to the local device, the BluetoothServer establishes a BluetoothSocket through which both devices can communicate with each other. The BluetoothServer expects to receive a string that represents the hash of a requested object. Using this hash, the BluetoothServer will send back the byte arrays of the corresponding object, if available.

### **5.3 Elephant**

Elephant is the second Android application. It is a simple browser which uses NetInf through the NetInfService to cache and share web pages.

The idea behind the Elephant browser is simple. When a traditional web URL is entered into the address bar and the refresh button is clicked, instead of downloading the web page from the Internet the browser first tries to use NetInf to retrieve the web page.

This is done by using the NetInfService. For the entered URL and each resource it links to:

1. Search for the hash of the URL/resource.
2. Get the file with the given hash.
3. Possibly publish the file so that other devices can retrieve the file from this device.

If the search or get fails for any reason, be it a timeout, no matches found or something else, the webpage is downloaded using the a standard HTTP request.

### 5.3.1 Configuration

The default settings for Elephant are stored in the properties file "assets/config.properties". This includes but is not limited to RESTful API IP address, RESTful API port and RESTful API timeouts.

### 5.3.2 Control Flow

Figure 5.4 shows an overview of the control flow of the interaction between Elephant and NetInfService. Figure 5.5 shows a picture of the internal control flow of the Elephant web browser, providing the packages and/or source files that are involved in different parts of the program, as well as some text and arrows describing the control flow.

The NetInfService mainly does its work by passing around Identifiers and NDOs (each encapsulating an Identifier).

An Identifier contains information about an NDO. The most important pieces of information in these applications are:

- Hash Algorithm
- Hash
- Content-Type
- Metadata (as a JSON string)

NDOs can contain attributes. In these applications the only used attributes are locator attributes. More specifically locators pointing to other Bluetooth devices and locators pointing to the local file system.

### 5.3.3 RESTful API Access

Access to the RESTful API is handled by the subclasses of NetInfRequest. There are three subclasses NetInfPublish, NetInfRetrieve and NetInfSearch corresponding to the API calls for publish, retrieve and search. NetInfRequest extends the class AsyncTask provided by Android, and hence all subclasses of NetInfRequest are also AsyncTasks.



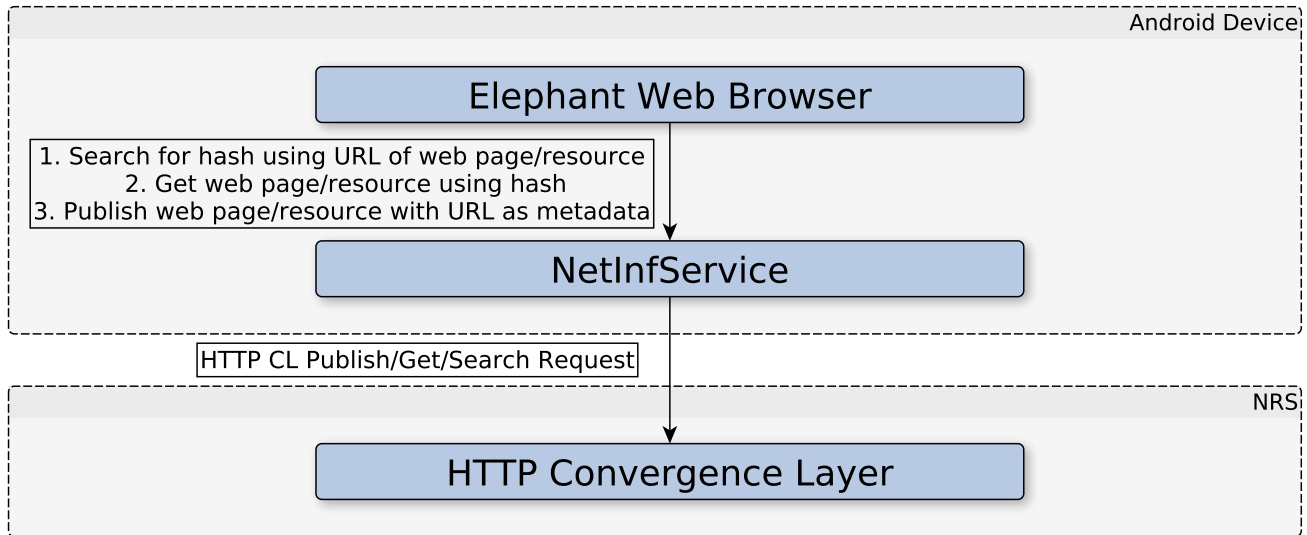


Figure 5.4: Elephant/NetInfService Control Flow

These classes can be used in two ways. Either by doing a blocking call:

```

1      // Create a new search
2      NetInfSearch search = new NetInfSearch("tokens", "ext");
3      // Execute the search
4      search.execute();
5      // Block until the search response is available
6      NetInfSearchResponse searchResponse =
7          (NetInfSearchResponse) search.get();
8      // Do things with the response...

```

Or in a non-blocking way by overriding the function that is called when the response becomes available:

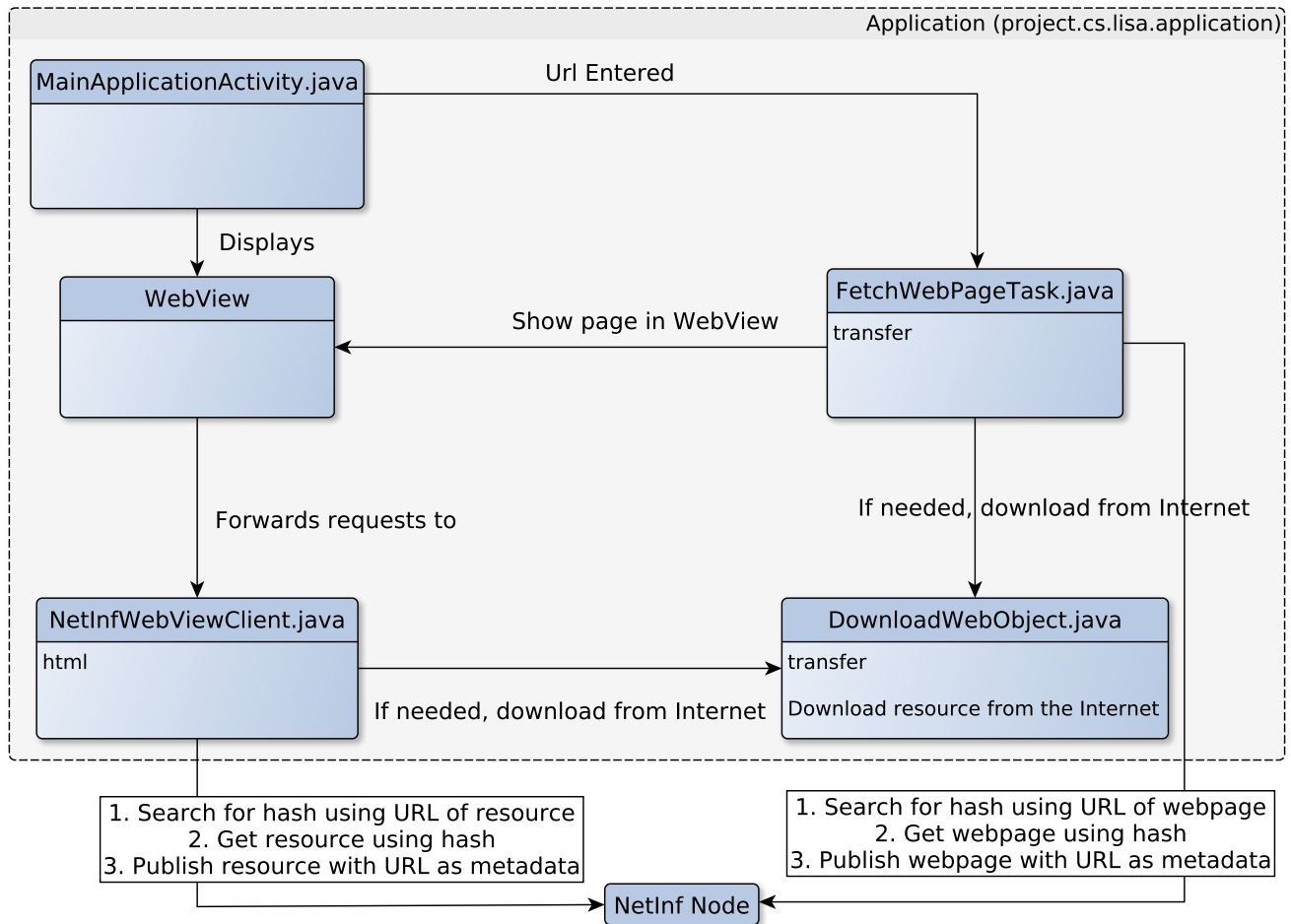


Figure 5.5: Elephant Control Flow

```

1      // Create a new search
2      NetInfSearch search = new NetInfSearch("tokens", "ext") {
3          @Override
4          public void onPostExecute(NetInfResponse response) {
5              NetInfSearchResponse searchResponse =
6                  (NetInfSearchResponse) response;
7              // Do things with the response...
8          }
9      }
10     // Execute the search
11     search.execute();

```

## 5.4 Erlang NetInf NRS

In the overall design there are two distinct process types: persistent and non-persistent. The persistent processes will run for the entire uptime of the system whereas the lifetime of a non-persistent process is the duration of its given task. Figure 5.6 below shows the current system design.

**Erlang NetInf – Process hierarchy**

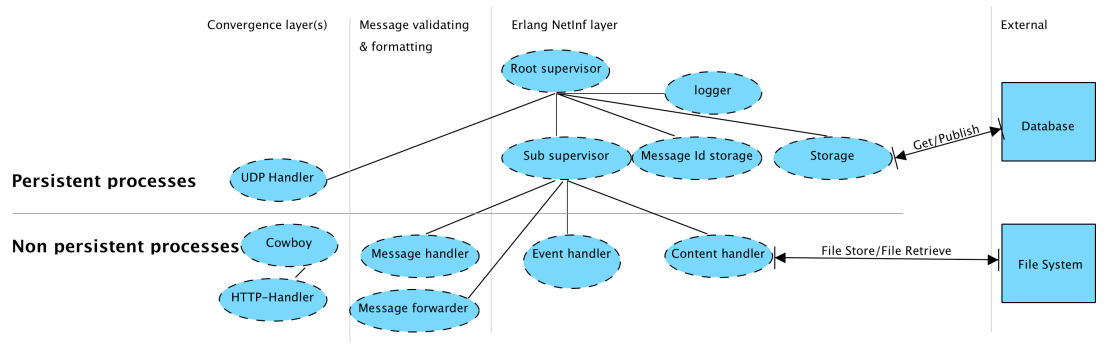


Figure 5.6: Current system design

### 5.4.1 Architecture layers

The system architecture is divided into four(4) distinct layers: a network (HTTP, UDP) convergence layer, message validation and formatting layer, an Erlang NetInf layer and finally an external storage layer consisting of a database as well as access to the file system. Within these layers lie the modules(Erlang processes) which are responsible for specific functions such as sending/receiving convergence layer messages, sanitizing messages, forwarding, accessing databases/file systems and logging.

### 5.4.2 NetInf Messaging

According to the draft specification[9] NetInf has three well defined messages which comprise of the core functionality of the system. The following sections describe the purpose and flow of each of these messages in addition to how the Erlang NetInf NRS handles them.

#### 5.4.2.1 Named Data Objects

NetInf describes any piece of information as a Named Data Object(NDO). In the current state of networks, the same piece of information is considered to be host centric and mutable with many copies of the same information lying around. The purpose of an NDO is to provide a convenient way for the protocol to be able to catalogue and perform operations such as storing, retrieving and finally searching for information while eliminating the need for host centric information.

#### 5.4.2.2 Internal NetInf Messaging

The Erlang NetInf NRS uses an internal record to represent a NetInf protocol message. For each request in the system, an instance of the following Erlang record is created and passed to various modules in order to have the specific operation performed. Afterwards the NRS constructs a response message and sends it back to the requester. There are two records defined in the module `nn_proto`, the first defines a request, primarily a Publish, Get, or Search message from outside of the system(external clients and other NRS').

While the second record defines the NDO. The message record includes the NetInf record (if available).

```
-record(message,
{
  msgid = undefined :: term(),
  time_to_live = undefined :: undefined | integer(),
  octets = undefined :: undefined | binary(),
  tokens = undefined :: undefined | [binary()],
  method = undefined :: undefined | get | search | publish | error,
  netinf = undefined :: undefined | proto() | [proto()] | term()
}).
```

The "netinf" field in the above record is dependent on the method and whether or not the message is a request or a response.

```
-record(netinf, {
  % name of the ndo
  name = undefined :: undefined | binary(),
  % list of possible locations and or URI's
  uri = [] :: [] | [binary()], % either empty list or a list of binary terms,
  % list of extensions stored in json format
  ext = {[[]]} :: {[[]]} | {[{binary(), any()}]},
  % timestamp of the ndo in json format
  time_stamp = undefined :: undefined | binary()
}).
```

#### 5.4.2.3 Publish

NetInf describes a Publish message which consists of the following fields:

**URI** - Contains the hash of the NDO. It is unique to the piece of information that is going to be published into the system. It is also mandatory.

**msgid** - A mandatory field as well and it is a unique number(string).

**loc1** - An optional field, this is a locator that can be used to access the information that will be shared.

**loc2** - Same as the above.

**ext** - An extension field, it is responsible for containing the metadata if any, in JSON format.

**rform** - The response format, The NetInf will format the response message using either HTML or JSON. JSON is the default if this field is not set. This is convergence layer specific.

**fullPut** - This field is set to determine if octets(binary data) is present with the publish message with either "true" or "false"

#### 5.4.2.4 Publish Message Workflow

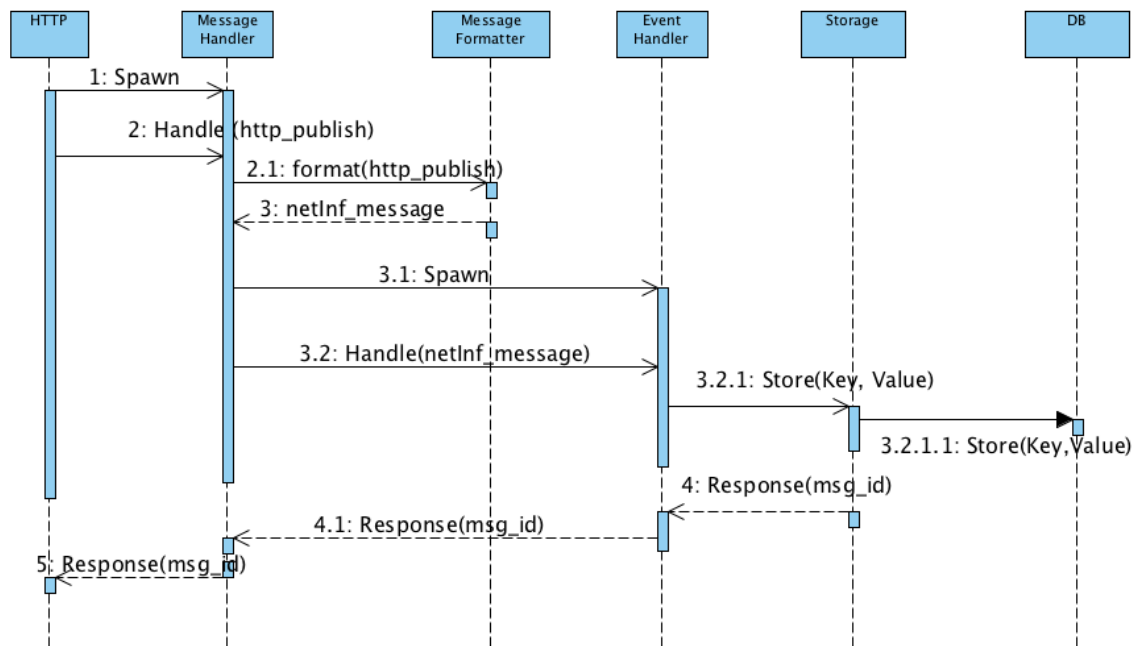


Figure 5.7: Publish Message workflow

A requester would like to share information. Using an external client the requester sends a NetInf Publish request message with the mandatory fields set as described above through a supported convergence layer(HTTP for

example). The Erlang NetInf NRS will then receive the message and create a 'message' record(see section: 5.4.2.2).

A convergence layer handler(CL handler) is spawned and receives the request. Once a request is in the system, the CL handler that received it will pass the request onto the message handler(MH). The MH is independent of the CL however, the formatting library used by the MH will depend on the CL the request was received on.

At this point the original request will be validated and sanitized for use in the internal NRS system and a NDO will be created. If the request is malformed the MH will construct a response message immediately and pass the new message back, informing the requester of the specific reason the request was rejected and then the MH will die.

In the case that the validation and sanitization succeeds the MH will have a newly sanitized and internal message representation of the original request(NDO). The request is now ready to be passed deeper into the system. The MH spawns an event handler(EH) and goes to sleep waiting for the message to complete the process.

The event handler will then read the NDO and determine if a content handler(CH) will need to be spawned in order to store the binary data(octets). The CH is only spawned if the "FullPut" flag in the original request was present and set to 'true'. Finally the EH will pass the NDO to the storage module and wait until the process is complete.

The storage module will call the appropriate function for the database(through the functions defined in the database wrapper nn\_database). In this case the database will store the published NDO. If a NDO with the same name exists, the two NDO's are merged and the result is stored.

Once the NDO is stored in the database a message is sent back through the chain of waiting processes. This occurs until the message formatter can create a response containing the NDO that was stored and any CL specific response codes. Note that any processes that were temporarily spawned will kill themselves after the job is complete. Figure 5.7 shows the flow of communication for Publish requests.

#### 5.4.2.5 Get

NetInf describes a Get message which consists of the following fields:

**URI** - Contains the unique hash of the NDO as well as the hash algorithm.  
The user requests the specific data object using this hash.

**msgid** - A mandatory and unique number for each message in the system.

**loc1** - Same as above

**loc2** - Same as above

**ext** - A field reserved for future extensions.

#### 5.4.2.6 Get Message Workflow

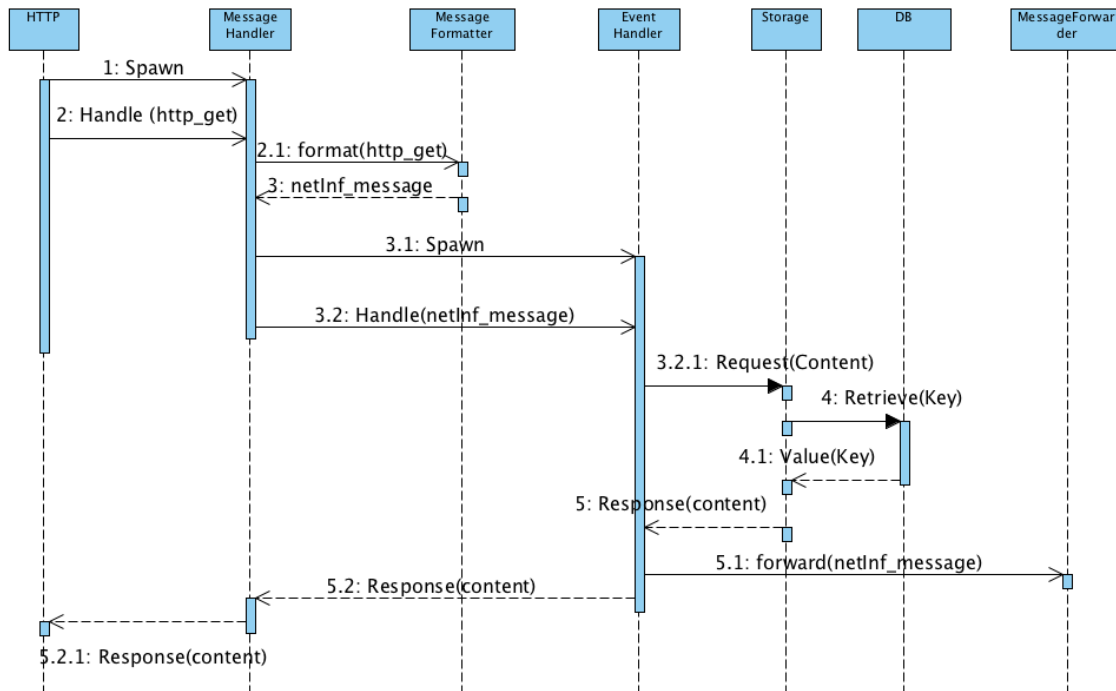


Figure 5.8: Get Message workflow



A requester would like to retrieve information, using an external client the requester sends a NetInf Get request (assuming they know the name of the NDO they are looking for) in the format described above. The process is similar to the publish, until the NDO is passed to the event handler. The event handler will call the database to lookup the NDO using the URI field. If found the NDO will be returned back through the system to the message formatter in order to construct the appropriate response message. However, if the NDO is not found a message forwarder will be spawned and the NDO request is broadcasted out on the UDP CL. If there are other Erlang NetInf NRS' on the network a UDP response packet will be sent and the original NRS which forwarded the request will eventually respond to the requester with the NDO or timeout. Figure 5.8 shows the flow of communication for Get requests.

#### 5.4.2.7 Search

NetInf describes a Search message which consists of the following fields:

**msgid** - Mandatory and a unique number.

**tokens** - Space delimited text. This is the text that the user will search for within the system

**rform** - Optional and will default to Json if not specified, however this is convergence layer specific.

**ext** - A optional field reserved for future extensions.

#### 5.4.2.8 Search Message Workflow

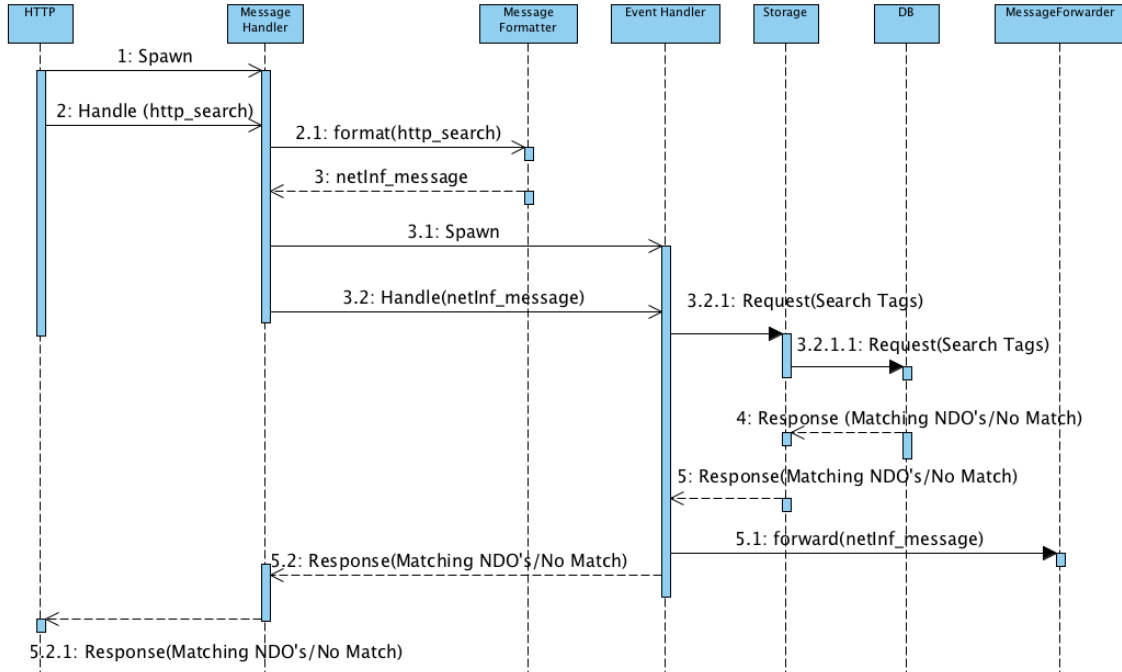


Figure 5.9: Search Message workflow

A requester would like to retrieve information but does not know the name of the NDO. A NetInf search request can be sent to the Erlang NetInf NRS to retrieve a list of NDO names and the metadata which match a particular criteria(search tokens). The process is similar to both the Get and the Publish however the event handler calls the appropriate search function in the attached database. If no match is found the request is automatically forwarded on the UDP CL where the same procedure takes place. If a match is found, a response is created and sent back to the original CL handler. Figure 5.9 shows the flow of communication for Search requests.

### 5.4.3 Dependencies

The system architecture for the backend relies on a few external libraries. The following are important for the system to run well.

1. Erlang Cowboy [11]

Erlang Cowboy is a small light-weight HTTP server and library written in Erlang for the purpose of handling HTTP requests. The Erlang NetInf NRS product uses functions in Erlang cowboy to communicate with the HTTP convergence layer. Erlang cowboy is responsible for all the multi-part and HTTP requests that are created in the system.

2. Erlang RTS [2]

Erlang Runtime application system is the core Erlang system. Erlang NetInf NRS would not run on the computer system without the core part of Erlang.

3. Erlang covertool [10]

Erlang covertool is an Erlang library created by Ivan Dubrov in order to convert the Erlang "cover" reports into a specific format called cobratura XML. This is required for compiling code metric reports and passing them into Jenkins for display on the build server.

4. Erlang JSON library [18]

Erlang JSON is a library created by Yamashina Hio and Paul J. Davis. This library contains functions which allow Erlang to convert data structures to JSON data structures. The Erlang NetInf NRS uses this library extensively to support storing, retrieving and manipulating data in JSON format.

5. Riak [17] - with search hooks and a Key-Value(KV) bucket installed. (Only used for using the system with Riak database)

Riak is an open-source scaleable and distributed Erlang database. Created by Basho Technologies, The NetInf NRS uses this database when run with the Riak Database option. Riak was chosen because of it's ease of use and recommendation by the customer. It is the only 'real' database that is currently supported since the other 'database' shipped with the product is an Erlang list data structure.

#### 5.4.4 Configurations

Since the system is supposed to be modular, configuration files are also implemented.

Configuration files allow the user of the system to quickly start it with a pre-determined setup, things such as which database to use, which convergence layers are supported, and timers for various functionality are also stored here for quick use and editing. The benefit of using Erlang specific configuration files is the great way they are organized, giving a highly readable and easy to swap out functionality.

We have created three(3) config files which live in the "config" directory.

- list.config
- riak.config
- static\_peers.config

Note: the static\_peers configuration is used only for testing of the HTTP forwarding. This configuration contains a list of IP addresses to other NRS'.

For new databases the developers encourage a new configuration to be created in order to keep things simple.

The following is the syntax for a config file

```
[{netinf_nrs,  
 [   
 {key1, value1},  
 ...  
 {keyN, valueN}  
 ]  
 }].
```

And here is an example of a config file:

```
[{netinf_nrs,
```

```
[
{database, nn_database_riak},
{convergence_layers, ["http"]},
{ip_timer, 5000},
{discovery, on},
{nrs_port, 9999},
{ct_port, 8078},
{client_port, 8079}
]
}].
```

#### 5.4.4.1 Meaning of the config values

**database** This is used to define what database to use. The value must be the name of a module that implements the `nn_database` behaviour. Default is the Riak implementation.

**convergence\_layers** Deprecated. This was used to define which convergence layers the node should support. Currently UDP multicast is used instead.

**ip\_timer** Deprecated. This was used to define how often the node broadcasted that it was live to other nodes via UDP multicasts.

**discovery** Deprecated. This was used to control whether or not the discovery service was active for testing purposes.

**nrs\_port** This defines what port the NRS will use to listen for NetInf messages

**ct\_port** This defines the port used to transfer HTTP chunks to clients.

**client\_port** This defines the port the HTML client interface interacts with.

#### 5.4.5 Using config files

To run the application with a config file, the `-config` flag must be set on the Erlang command line.

```
erl -pa ebin deps/*/ebin -config configs/list
```

OR

```
erl -pa ebin deps/*/ebin -config configs/riak
```

Note: if the `netinf_nrs.app.src` file has some configuration options in the `env` section and there is a config file specified on the Erlang command line then the parameters in the config file will take precedence.

#### 5.4.6 Extracting the config parameters

Developers can use the following code to extract the values associated with a configuration parameter.

```
application:get_env(app-name, parameter-name).
```

The argument `app-name` is the name of the application, in this case `netinf_nrs`, and `parameter-name` is the name of one of the parameters defined in the config file or the `env` section of the `netinf_nrs.app.src` file.

For example to retrieve the value associated with the database a developer can use the following:

```
application:get_env(netinf_nrs, database).
```

This code would return `{ok, nn_database_list}` or `{ok, nn_database_riak}` depending on the configuration. However if the parameter name is not defined, the above code will return `undefined`.

#### 5.4.7 Convergence Layers

The NetInf NRS protocol introduces the concept of Convergence Layers(CL). CL's are specific protocols that can be used to talk to other nodes on the network. For example it is possible to use the `HyperTextTransferProtocol(HTTP)`, `Erlang Messaging` or `User Datagram Protocol(UDP)`. The CL's are implemented as a set of Erlang modules whose sole jobs are to receive and send messages of the specific type of the CL. These modules receive(CL handlers) `clean/format(CL specific formatter)` and forward into and out of the system.

#### 5.4.7.1 HTTP

The NetInf protocol draft discusses using HTTP as a primary layer of communication between nodes. All requests on this layer are arriving into the system as HTTP messages and then subsequently changed into internal NetInf messages. The HTTP CL consists of three(3) modules.

1. `nn_http_handler` - Uses Erlang cowboy to receive and send HTTP requests to/from the system
2. `nn_message_handler` - Specifically spawned with the attached HTTP formatter in order to process requests
3. `nn_http_formatting` - Handles converting requests/responses from HTTP to NetInf messages and vice versa.

#### 5.4.7.2 UDP

The NetInf protocol draft lists UDP as a CL, however its function is more like a discovery protocol for other NRS' of any type of implementation. The current UDP CL broadcasts NetInf messages on the network on a multicast IP 225.4.5.6 and port 2345.

The UDP CL is called when the NetInf NRS receives either a Get or a Search request from some other convergence layer and the requested NDO is not in the the NRS. The Forwarding module will forward the request using the multicast address. Similar to the HTTP CL, this layer consists of three(3) modules as well.

1. `nn_udp_handler` - Uses the Erlang `gen_udp` library to send/receive UDP messages into and out of the system.
2. `nn_message_handler` - Spawned with the attached UDP formatter in order to process requests.
3. `nn_udp_formatting` - Handles converting requests/responses from UDP to NetInf messages and vice versa.

Once the UDP Get/Search messages are sent out to the network, the system may receive back a response with the details asked for by the original request.

The `UDP_handler` in conjunction with the `nn_message_handler`(Spawned with UDP formatting) will then extract the details, create an internal NetInf Message and forward the message to the process that is currently waiting on the original request. In the case of an HTTP CL, the NetInf message would be forwarded back to the process which deals with the HTTP CL formatting/message handler.

Currently UDP Get and UDP Search requests are supported. The framework for UDP Publish requests are included in the current code base, but have not been used for the purpose of forwarding publishes.

#### **5.4.8 Notes on other CL**

There was a plan to include an Erlang specific CL, this would become a group of modules(handler, formatter) which deal only with Erlang specific messages. However this was a thought that later turned out to be of no real benefit, but extending the system to include this or other CL's could be easily done.

#### **5.4.9 Plug N' Play Database Wrapper**

The Erlang NetInf NRS includes functionality to allow for run-time database switching as well as providing an easy way to add interfaces to existing databases. At the time of writing this report the Erlang NetInf NRS provides support for Riak, as well as a default Erlang list 'database' implementation. The database interface is designed to be intuitive to implement. Figure 5.10 shows the interface.



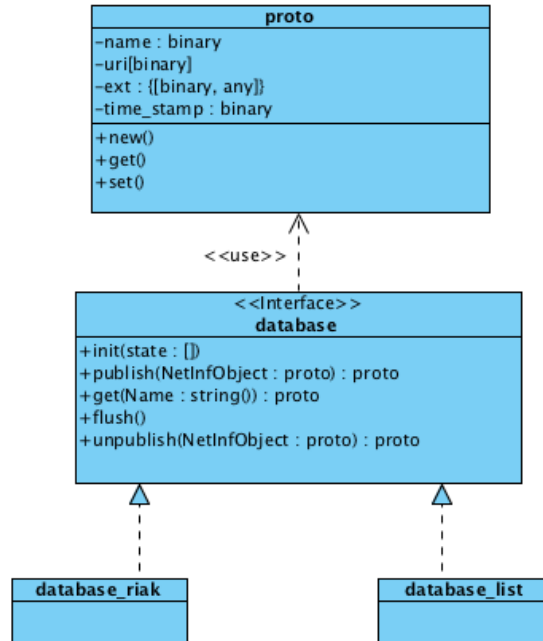


Figure 5.10: Database Interface

#### 5.4.10 Setup of Database Module

All modules which wish to implement a database connection will use the custom `nn_database` behaviour.

```
-module(nn_database).
```

The following functions must be implemented by all database wrappers:

##### Initialization

```
init() ->
{ok, pid()} | {ok, registered_name:atom()} | {error, string()}
```

The above function returns the connection for the specified database. Remember the `init` should return an identifier to the persistent process of the

specified database.

```
publish(NetInfObject::nn_proto:proto()) ->
  {ok, ReturnNetInfObj::nn_proto:proto()}
```

Takes the NetInfObject and returns: {ok, NewObject} where NewObject is the NDO created when merging the object being published with an object with the same name in the database. If there was no object with that name in the database, NewObject is the object being published.

### **Get**

```
get(Name::string()) ->
  {ok, nn_proto:proto()} | {ok, no_match}
```

Takes the NetInf Name of the object and returns: {ok, Data} where Data is the NDO that was found or no\_match if not found.

### **Unpublish**

```
unpublish(NetInfObject::nn_proto:proto()) ->
  {ok, ReturnNetInfObj::nn_proto:proto()} | {ok, no_match}
```

Takes the NetInfObject and returns {ok, ReturnNetInfObj} where ReturnNetInfObj is the NDO entry that was deleted from the specified database.

### **Search**

```
search(SearchList::list()) ->
  {ok, list()} | {ok, no_match}
```

Takes a Erlang list of search keywords and returns a list of the NDOs which match those key words.

### **Flush**

```
flush() -> ok
```

Deletes all the entries in the database.

See the module `src/nn_database_list` as an example of the wrapper implementation for use with a 'list' database in the source code.

## 5.5 Chunked data streaming

A chunked data stream can for example be a video or audio stream. The Erlang NetInf NRS includes two different ways to stream chunked data. The first is the modified version of NetInf that removes the overhead of publishing each chunk to the NRS. The other implementation uses pure NetInf to publish each chunk. Both of them use an HTML5 interface to playback the video stream.

### 5.5.1 Content dispatcher

To be able to transfer the video chunks to the local HTML5 interface a content dispatcher service was added. The difference is that the content dispatcher service serves the NDOs' octets directly through HTTP, that is without the multi-part response as done in the HTTP CL. The module for this is the *nn\_ct\_handler*. This service is spawned when the NRS system starts and runs on port 8078. To request the octets of the NDO *ni:///sha-256-64;abc* pass the url:

`http://localhost:8078/octets/ni%3A%2F%2F%2Fsha-256-64%3Babc`

### 5.5.2 Stream handler

The stream handler module *nn\_stream\_handler* handles fetching and polling of chunked octets between different NetInf nodes. In both streaming implementations the logic for fetching the chunks is to shuffle the list of available locators and fetch all available chunks in the ICN. When there are no more chunks, the stream handler retries on a regular interval. After a defined number of retries it will finally terminate itself.

### 5.5.3 HTTP client handling

The *nn\_http\_client\_handler* module is used to serve the HTML5 interface for NetInf Get, Publish and Search requests. It forwards these requests to the local NRS. The client runs on port 8079. The different interfaces that can be seen are:

The interface for regular NetInf interaction, located at *http://localhost:8079/*

The modified streaming, located at *http://localhost:8079/stream*

The pure streaming, located at *http://localhost:8079/streampure*

Other than the above viewable URIs there are a couple of other requests that are used to interact with the system.

Subscribe to a modified chunked stream *http://localhost:8079/subscribe*

Subscribe to a pure NetInf stream *http://localhost:8079/subscribe/search\_and\_get*

### 5.5.4 HTML5 interfaces

To combine the video chunks, two HTML5 video elements are used to pre-cache the chunks through the content dispatcher. This is done by alternating the visibility and playback of the two elements with JavaScript. With this method of playback the video chunks looks like one continuous stream. To make the usability of the interface smoother, some asynchronous network requests are used to communicate with the HTTP client. To the right of the interface it is possible to see the current state of the local NRS.

### 5.5.5 Difference between implementations

The main difference between the implementations is in the module *nn\_event\_handler*. When requesting a chunk with the modified version, the special hash algorithm name *demo* is used to avoid database lookups and content validation. To request chunk number 80 of the stream *ni:///sha-256;abc*, the NDO name will be *ni:///demo;abc80*. The metadata and locators in such a response are empty. To add a new chunk to the stream just increase the chunk number and add the physical chunk to the storage.

**The pure NetInf streaming** assumes that each chunk in the stream has been published to the NRS, with an ordinary NetInf publish request along with required metadata. The chunk metadata should include stream name and chunk number. For the stream *mystream* and chunk number *80* of *ni:///sha-256;abc*, the metadata must include:

```
{"meta": {"stream": "mystream", "chunk": "mystream80"}}
```

To obtain the chunk of a pure stream the receiver will have to search for each chunk and then get the NDO.

### 5.5.6 Advantages and disadvantages

The biggest advantage with the modified streaming is that the overhead of handling the chunks is reduced. A flaw is that the content validation has been disabled and it is possible to add chunks with modified content.

## Chapter 6

# Evaluation and testing

### 6.1 Frontend

The evaluation of the Elephant browser and the NetInfService applications tries to answer the following core questions:

1. How much uplink bandwidth is saved?
2. How much content is reused?
3. How much of linked resources are dynamically generated?
4. How fast is the browser?

#### 6.1.1 Test Setup

The first test consists of a set of web pages and four Android phones. Each phone will automatically retrieve the set of pages in a random order. Using the logging functionality of the applications, information about how (Internet, Bluetooth, NRS or Database) resources are retrieved is gathered. Information about how many bytes each resource consists of and how long it takes to retrieve is also acquired. Full put is enabled on all four phones, because the answer to question one does not depend on which method of transfer is used. The results are meant to give an idea of the answer to questions one and two.

The web page sets are of sizes 15, 20, 25 and 30. They are derived from the service Alexa [1], which is renowned for its web metrics. This service keeps track of the most visited web sites by country, and the top sites were used to create the sets.

The test also uses a Name Resolution Service that is reset between retrieving each set of web pages.

The second test setup consisted of two runs: First, retrieving all web pages in the set of 15 web pages using one blank phone. Second, the same phone retrieves the same set of web pages again. This time, the phone should already have the web pages in its database. The results are meant to give an idea of the answer to question three. The test is repeated two times, with the Name Resolution Service reset in between.

A third test uses four phones, each retrieves the 15 web pages set. This test is repeated three times once with full put enabled on all phones, once with full put enabled on two phones and finally with full put disabled on all phones. This is done to test the Bluetooth functionality. The goal of this test is to simulate the scenarios where there is no, limited, or full peer-to-peer interaction, respectively.

### **6.1.2 Hardware**

The tests are run on three Samsung Galaxy Nexus phones and one HTC One X phone using Android OS 4.1.1 Jellybean

The Name Resolution Service was run on an Intel Core 2 Quad CPU Q9400 @ 2.66GHz 4 with 4 gigabytes of volatile memory using Ubuntu 12.04 LTS.

### **6.1.3 Limitations**

The Name Resolution Service supports two types of databases for storing published NDOs. The first uses Erlang lists stored in volatile memory, the other uses a Riak database. The tests use the list database as it was the database used during the development of the browser application.

This means that the test is limited by the amount of free volatile memory of the system. A preliminary test using a set of 50 web pages caused the system to run out of memory, resulting in a crash. Because of this, no set

contains more than 30 web pages.

#### 6.1.4 Results

The results of test one can be seen in Figure 6.1. Each bar represents a specific set size and the colors show how much of the data was transferred with each technology.

Table 6.1 shows the total time spent and the time spent transferring the files while retrieving the 15 web page set.

The results of test two can be seen in Figure 6.2. The two leftmost bars represent the first run of the test and the two rightmost the second.

The results of the third test can be seen in Figure 6.3

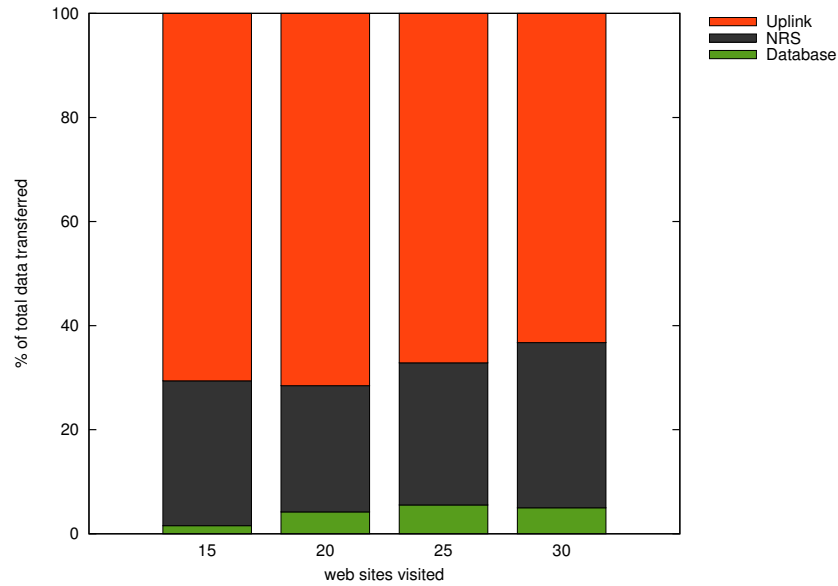


Figure 6.1: Percentage of data transferred over the different transport methods during test one



Phone #	Total time (s)	Time downloading (s)	Time downloading (%)
1	251	17	<b>6</b>
2	303	15	<b>4</b>
3	241	20	<b>8</b>
4	254	18	<b>7</b>

Table 6.1: Total time and time spent downloading for the set of 15 web pages used during test one

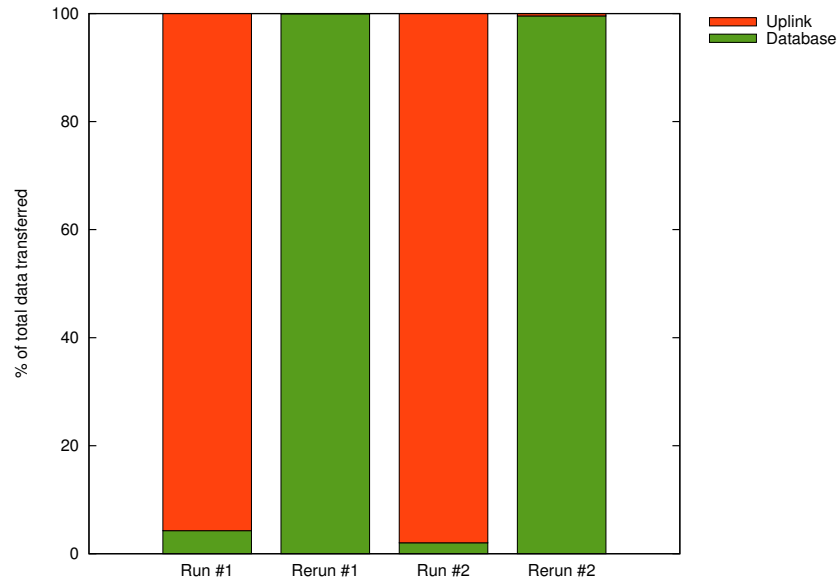


Figure 6.2: Percentage of data retrieved from the local database during test two

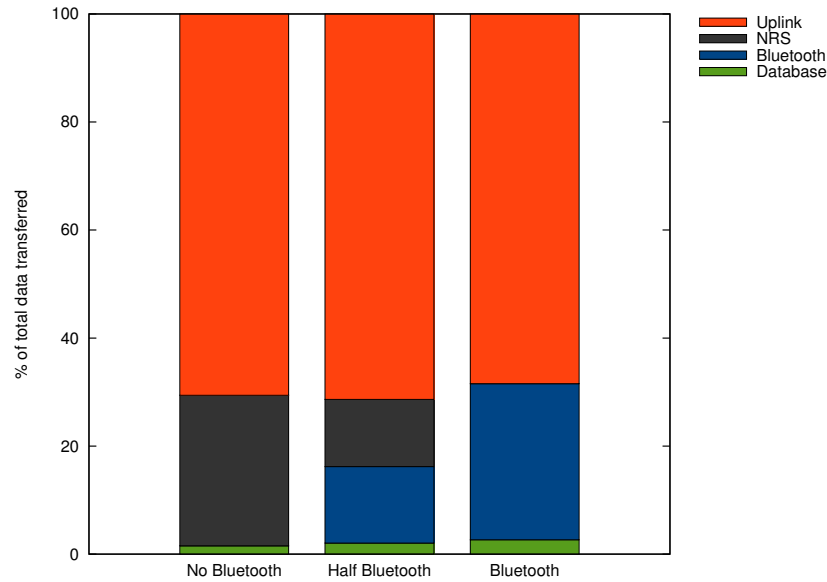


Figure 6.3: Percentage of data transferred over the different transport methods during test three

### 6.1.5 Discussion

Figure 6.1 shows that approximately 30% of the data can be retrieved without accessing the Internet. Precaching of popular web pages is expected to improve this result.

It was observed that if one phone had a jump start on another phone when retrieving a certain web page, the second phone shortly caught up with the first phone. The two phones would then try to retrieve the same resource at the same time. Since this resource will not be in the NRS, both phones will retrieve it from the Internet.

In Figure 6.1 it can be seen that a few percent of the resources are retrieved from the database. The reason behind this is that some resources are reused multiple times throughout the web pages. Since resources are cached in the database the first time they are retrieved, additional requests can use the cached version.

Figure 6.2 demonstrates that when accessing a web page a second time, a small part still has to be retrieved using the Internet. An example of when

this can happen is when a web page links to a resource using JavaScript to add a timestamp to the resource's URL. Because of the dynamic nature of this content, it will not be found when searched for. Therefore, there will be a small amount of resources that always will be retrieved from the Internet.

In Figure 6.3 the amount of data retrieved without using the Internet is similar whether or not full put was used. This is as expected because the data that is not made available through full put should be available through Bluetooth.

An unexpected behavior observed during testing is that the application spends most of the time searching. More specifically, the NRS does not respond in a timely manner to requests that result in no match. Unfortunately, the time spent searching is not logged. As can be seen in Figure 6.1 the time downloading is a small fraction of the total time spent. It is strongly suspected that most of the total retrieval time is spent waiting for search results.

A second unexpected behavior observed is that the NetInfService randomly pauses until it regains focus. When this happens all NetInf functionality becomes unavailable, which causes all resources to be retrieved from the Internet. It is suspected that this is caused by how the Android OS handles background applications.

## 6.2 Backend

The following section describes how the Erlang NetInf streaming functionality was evaluated and tested.

### 6.2.1 Video streaming protocol evaluation

One of the problems discussed was reducing congestion when broadcasting content. This was solved by implementing an alternative way of sending chunked data as seen in Section 5.5 according to the video streaming draft in Appendix C 10.1. In the following section this implementation is referred to as the *modified streaming*. The following tests were conducted:

- Testing the modified version of NetInf video streaming.

- Testing the pure version of NetInf video streaming.
- Comparison between both implementations of the NetInf Video streaming

The testing was done by transferring 500 chunks of bogus data between a number of different NetInf nodes running the NRS. All the receiving nodes started the transfer at the same time. To be able to publish  $N$  number of chunks in an easy and fast manner, the *nn\_evaluation* module was implemented.

### 6.2.2 Pure video streaming evaluation setup

The nodes that were included in the pure version of the streaming were the following: Central NRS node, one streaming source node that published all 500 NDOs to itself with *fullput* set to True, then also published the NDO to the central NRS with *fullput* set to False. The central NRS can not have the octets because otherwise it would provide all clients with the octets directly, hence prevent the network load to be more balanced.

Five client nodes then retrieved each chunk by:

1. searching NRS for the chunk with chunk number and stream name.
2. getting the NDO metadata and locators from the NRS.
3. fetching NDO with octets from one of the locators.
4. publishing the NDO to itself
5. adding itself as a locator and publish to the NRS.
6. repeating the procedure for the next chunk.

### 6.2.3 Modified video streaming evaluation setup

In this setup one node served both as the source and central NRS while there were five other client nodes. The source first published the stream NDO to itself, then used the content dispatcher to put all the chunks into the storage. The clients then get the streamed NDO, added themselves as

locator and published it to the NRS. To retrieve the chunks each client need to:

1. append the current chunk number to the NDO name.
2. send the request to one of the locators.
3. if status of the response is 404, repeat step 2.
4. store the octects in its storage.
5. increase the chunk number

#### 6.2.4 Results

The Figure 6.4 shows that the pure NetInf streaming is faster when it comes to transferring all the chunks.

The CPU load of the central NRS was recorded with the built in system monitor, the result of the pure version is in Figure 6.5 and the modified in Figure 6.6.

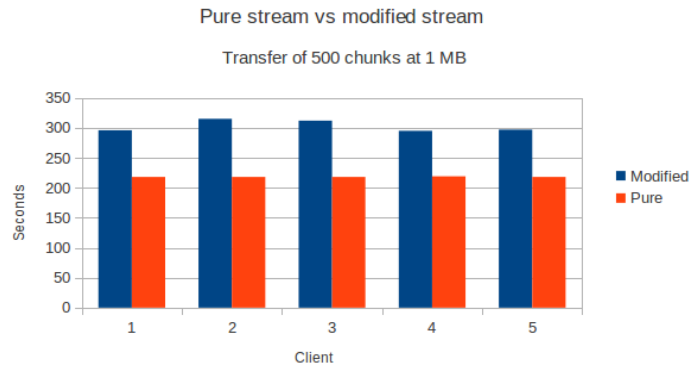


Figure 6.4: Pure streaming vs modified streaming

#### 6.2.5 Discussion

As can be seen in Figure 6.4 the transfer of the chunks took less time using the pure streaming, this was not unexpected since the modified version has

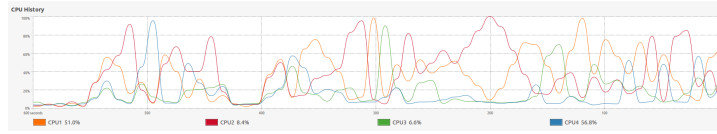


Figure 6.5: Central NRS CPU usage during pure streaming

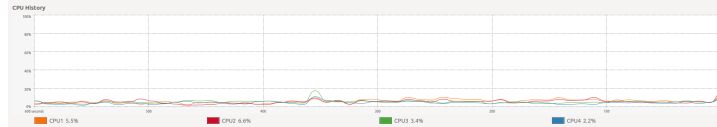


Figure 6.6: Central NRS CPU usage during modified streaming

a lot of overhead in this case, if it tries to get chunks from another client which does not have them yet, this will yield an 404 response and the client will need to try another source. The pure NetInf however will get a hit every time.

Comparing CPU load in Figure 6.5 and Figure 6.6, shows that even with only five receiving nodes the pure central NRS was under considerably high load. While in the modified versions load was almost not noticeable. The CPU load on the pure NRS is caused by the number of database lookups.

## 6.2.6 Notes on Interoperability

There are already existing implementations created by SAIL and Ericsson Research for the NetInf protocol. In the beginning of the product life-cycle the customer requested the development team to evaluate the interoperability of this product with other systems. However as the product evolved the customer requested that the interoperability be left to them to evaluate and that this development team continue with evaluating the video streaming instead.

Therefore the development team did not evaluate interoperability, but there is confidence that with minor tweaking of the code (due to differences in the various draft versions of the NetInf protocol specification) this product and others will become interoperable.

The following list describes the evaluation performed for testing the NetInf NRS application.

- Evaluation of the search time and get time for the supported databases
- Measuring the number of requests per frontend phone client

## Chapter 7

# Conclusions and Future Work

### 7.1 Conclusions

The main goals of this project was to develop applications based on the principles of information centric networking using the NetInf protocol. These goals were achieved and the teams were able to build the applications using Java and Erlang/OTP. The backend product(NetInf NRS) is a concurrent and fault tolerant application as per the principles of OTP. Both teams had clear goals when they started the project and achieved it comfortably in the end. Infact, the backend team added functionality to support streaming videos using the application. This functionality was not part of the original plan but because the team achieved all the other major goals well before time.

Both frontend and backend teams performed testing and evaluation of the application. The backend team evaluated the streaming with pure NetInf messages and the modified version of streaming based on the draft. Other results that the backend team observed during the evaluation of streaming was that the list database is very slow. This is because the search time is  $T*N$  where  $T$  is the number of search tokens and  $N$  is the number of meta data attribute stored. But as mentioned in the future work section, some other database should be implemented to see if the application works better. In the modified version of streaming we can improve the polling strategy to



transfer all the chunks faster. Also the content validation is disabled in the modified version of video streaming and that can cause unauthorized content to be published. The front-end evaluation showed that apart from two problems the application seems to be working as expected. The first problem is the slow searches, which is mentioned above. The second is the problem with NetInfService randomly pausing, which might result from how the Android OS handles background applications.

## **7.2 Future Work**

### **7.2.1 Elephant and NetInfService**

#### **7.2.1.1 Dynamic Content**

Currently, the dynamic content problem is ignored. Given a traditional web URL, the browser application maps it to an NDO. As long as the web content is static, the mapping from URL to NDO will be a one-to-one relation. However, if the web content is dynamic, the mapping will be one-to-many. If this is the case a search could return several matching NDOs. At a first glance adding a timestamp specifying when the page was retrieved could seem to solve the problem. While this is true for some dynamic web pages, it does not hold in general. For example a web page could be generated differently depending on who or from where it was accessed. Furthermore, a dynamic web page linking to other dynamic resources might be dependent on getting the correct version of the linked resources. In the second case a timestamp could help if all resources belonging together are marked with the same timestamp and not the individual access times. Currently the first search result is used by default.

#### **7.2.1.2 Search**

The Elephant browser relies heavily on NetInf searches as described in Sections 4.1.1 and 5.3. The time needed to perform a search increases as the number of published NDOs grows. The NRS supports two ways of storing published NDOs either using an Erlang list or a Riak database. Preliminary tests had problems with increasing search times using both these approaches. Searching is expensive because it is being performed in a list database. Bet-

ter results can be obtained when using databases that are suited to handling a larger amount of data, such as Riak.

#### **7.2.1.3 Delete Functionality**

Currently the NetInf delete functionality is currently not provided by the NetInfService.

#### **7.2.1.4 NetInfService**

NetInfService was implemented as its own Android application which is supposed to run in the background. While this makes it easy to create other applications using the provided functionality, there are currently problems with the application randomly stopping and not resuming until it is brought to the front. The suspected reason is that the Android OS might pause applications in the background to save system resources, or stop them when system settings are changed and then restart them when they are brought to the front. If this is an unavoidable problem for Android applications running in the background then NetInfService needs to be changed, perhaps into an Android Service.

#### **7.2.1.5 Database and Bluetooth convergence layer**

Other suggestions for future work include testing the application with another database like SQLite or building a Bluetooth Convergence Layer for users to be able to send NetInf messages via Bluetooth.

### **7.2.2 NetInf NRS**

#### **7.2.2.1 Precaching**

If the NetInf network starts without any objects cached, it is probable that a lot of Internet access in the beginning while the content is entering the network. This could be prevented by precaching content in the NRS. By investigating which web pages are frequently accessed and when they are accessed, the NRS could download these popular web pages in advance. If

the search request always uses the NRS, this information will be continuously available to the NRS and it could automatically download the pages it expects to be accessed when there is bandwidth to spare.

#### **7.2.2.2 Access Control**

Currently, any user can publish their content on the NRS. One functionality for the future is to implement some kind of access control mechanism. Only authorized users would be able to publish content to a particular NRS and only a particular group of users would be able to access the published content.

#### **7.2.2.3 Interoperability**

Further work could be done in testing the interoperability between different NetInf implementations. Different implementations of NetInf exist, in different programming languages. These implementations should be able to communicate with each other if they have been implemented using the same version of the protocol draft.

#### **7.2.2.4 Handle large file**

The current system has some unexpected behaviour when files transferred exceed 10 megabytes. An improvement to the application could be to make it more stable when handling larger files.

### **7.2.3 Security**

Security is a field that was out of scope for this project. However, it is an area that should not be overlooked in the future. Questions like how to handle private data within the network, who can publish or retrieve data within the network, who to trust as a content source, amongst others.

#### **7.2.3.1 NRS required folder creation**

Currently the NetInf NRS requires a few environment folders (logs and files) to be present without crashing the system. The product relies on a separate

”make” file which creates these folders. In the future the folder creation can be moved to be within the NetInf NRS product.

#### **7.2.3.2 Polling Logic**

The polling logic needs to be implemented in the video streaming client, this is how often the receiver should get a new chunk or check if a new chunk exists.

#### **7.2.4 General**

An important concept of ICN is the peer-to-peer communication between devices. During our project we only focused on transferring content through Bluetooth, as this was a well known and reliable technology for emulating peer-to-peer communication. In the future we see other technologies that could be faster and more convenient for the realization of ICN, such as transferring data through physical contact between devices.

As far as it concerns the ICN draft, a suggestion would be to rewrite the HTTP Convergence Layer specifications in terms of consistency. The HTTP Convergence Layer uses a mix of JSON and HTTP forms, which makes it overcomplicated to work with it.

## Chapter 8

# Appendix A: Installation instructions

### 8.1 Frontend

This section will describe how to configure the environment to run the frontend application on an Android device. This description includes configuring Eclipse with Android in order to continue development. The guide assumes that the Java SDK is installed.

The frontend team have been working with:

- Eclipse Indigo, Service Release 2
- Android Version 4.1, API Level 16

#### 8.1.1 Configuring Eclipse with Android

There are two ways to set up Eclipse with Android. If Eclipse is installed, follow the instructions "Installing the Eclipse Plugin" at [13] in order to configure the Android support for the Eclipse environment.

If Eclipse is not installed yet, Android has released a Bundle that contains Eclipse and all necessary tools for developing Android applications. This bundle can be found at "Get the Android SDK" at [13].

### 8.1.2 Installing and debugging the application

The latest version of the frontend code can be found on Github <sup>1</sup>.

In order to run the application on an Android device, connect the device to a computer, import the project and simply run it in Eclipse. Eclipse should recognize the device on its own and immediately offer a list of available devices, on which the application can be installed on.

For debugging, USB debugging on the device must be enabled. This setting can be found under the settings menu of the device.

Note that the system might not detect some devices. This issue occurred with using the HTC ONE x. In that case, a preliminary set up needs to be done. This set up includes creating a *udev* rules file. More information can be found under **”Setting up a Device for Development”** at [13].

## 8.2 Backend

This section describes how to install/setup the Netinf NRS and the NetInf Streaming on a server, it is intended for both end users and developers.

### 8.2.1 Dependencies

In order to run the system properly the following components are required to be installed on the system, the user can either run the script from the section below, or install these manually. Furthermore, the developers support Ubuntu 12.04 LTS 32 bit. Attempts to run this software on other operating systems and architectures may produce unexpected results.

For the NetInf NRS system:

- Make
- Rebar
- G++ compiler

---

<sup>1</sup>Project CS Frontend application. <https://github.com/project-cs-2012>

- Erlang - version R15B03

For the NetInf Video Streaming, in addition to the above the following component(s) are required

- Google Chrome Web browser - or any other browser that supports HTML 5 video tag.

To install Make and G++ manually please run the following commands in the terminal.

```
sudo apt-get install build-essential
sudo apt-get install g++
```

To install Rebar and Erlang manually please follow the following steps

Download and install: Rebar from Github <sup>2</sup>.

unzip the archive and run the following command

```
cd rebar-master
./bootstrap
sudo cp rebar /usr/bin
```

To Download and install Erlang - version R15B03 for the 32 bit architecture. It can be retrieved from the Erlang-Solutions website or use the following commands.

```
wget https://elearning.erlang-solutions.com/couchdb//
rbingen_adapter//package_R15B03_precise32_1354121173/
esl-erlang_15.b.3-1~ubuntu~precise_i386.deb

sudo dpkg -i esl-erlang_15.b.3-1~ubuntu~precise_i386.deb
```

---

<sup>2</sup>Rebar Github. <https://github.com/basho/rebar/archive/master.zip>

### 8.2.2 Script

For the convenience of end users and developers, there is a packaged install/setup script available after obtaining the backend code. This script is responsible for quickly installing the entire system with all the dependencies.

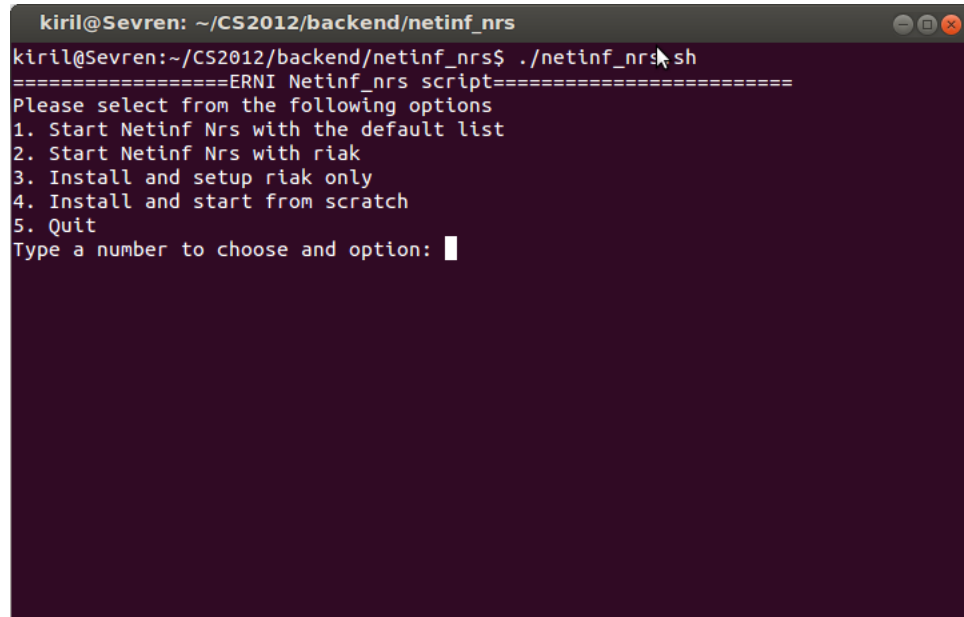
**If the script is not immediately runnable please run the following command:**

```
chmod a+x netinf_nrs.sh
```

The script can be run by using the following on a command line terminal.

```
./netinf_nrs.sh
```

The script will be put a into a menu loop shown below and instructs the user to type a number in order to choose an option. Choosing an option will preform the task and then cause the script to exit normally.

A screenshot of a terminal window with a dark purple background. The window title is "kiril@Sevren: ~/CS2012/backend/netinf\_nrs". The prompt is "kiril@Sevren:~/CS2012/backend/netinf\_nrs\$". The user has entered "sh" after the prompt. The output shows a menu titled "=====ERNI Netinf\_nrs script=====". Below the title, it says "Please select from the following options". There is a numbered list: "1. Start Netinf Nrs with the default list", "2. Start Netinf Nrs with riak", "3. Install and setup riak only", "4. Install and start from scratch", and "5. Quit". At the bottom, it says "Type a number to choose and option:" followed by a white cursor. The terminal window has standard Linux window controls (minimize, maximize, close) in the top right corner.

```
kiril@Sevren: ~/CS2012/backend/netinf_nrs
kiril@Sevren:~/CS2012/backend/netinf_nrs$ ./netinf_nrs sh
=====ERNI Netinf_nrs script=====
Please select from the following options
1. Start Netinf Nrs with the default list
2. Start Netinf Nrs with riak
3. Install and setup riak only
4. Install and start from scratch
5. Quit
Type a number to choose and option: █
```

The following options are available to the user



- Start Netinf NRS with the default list:  
Assumes the system has all the dependencies installed and only starts the NetInf NRS with the list database.
- Start Netinf NRS with riak:  
Will check that Riak is running and present in the system and then start the NetInf NRS with the Riak database. If Riak is not present then the script will download and install all the required components.
- Install and setup riak only:  
Use this option only when Riak needs to be downloaded and installed on the machine. This will not start an NRS.
- Install and start from scratch:  
This option assumes a bare machine and checks that all the dependencies are satisfied. It will auto download and install anything that is required and then start the NetInf NRS with the default list database.

### 8.2.3 Riak Database

Riak is a database written in Erlang. It is known for being distributed and fault-tolerant. Riak was chosen above other database implementations since it was suggested by the customer and the development team had great support available.

In case there is something wrong with the script process on the target machine please follow the manual installation instructions below.

Install libssl0.9.8 with:

```
sudo apt-get install libssl0.9.8
```

Next install the Riak database:

```
wget http://downloads.basho.com.s3-website-us-east-1.amazonaws.com/riak/CURRENT/ubuntu/lucid/riak_1.2.1-1_i386.deb
```

```
sudo dpkg -i riak_1.2.1-1_i386.deb
```

In order for the search to work in the Riak system and from the NRS please enable search, Riak Search has to be enabled in the app.config (/etc/riak/app.config) file. Simply change the setting to true in Riak Search Config section (shown below).

```
%% Riak Search Config
{riak_search, [
    %% To enable Search functionality set this 'true'.
    {enabled, false}
]},
```

Then run the in the terminal:

```
riak restart
```

Followed by the command below to index the bucket:

```
search-cmd install netinf_bucket
```

Lastly, please make sure that the NRS is started with the Riak database.

### 8.2.4 Running the NetInf NRS

To run the NetInf NRS without the script, please run the following commands after navigating to the netinf\_nrs folder:

- Using the list database

```
erl -pa ebin deps/*/ebin -config configs/list -s netinf_nrs
-eval "io:format(\"NetInf NRS is running ... ~n\")."
```

- Using the riak database

Please make sure the Riak daemon is started before. If it has not been started use the first command shown below before the erl command.

```
riak start
```

```
erl -pa ebin deps/*/ebin -config configs/riak -s netinf_nrs  
-eval "io:format(\"NetInf NRS is running ... ~n\")."
```

## Chapter 9

# Appendix B: Maintenance instructions

### 9.1 Frontend

#### 9.1.1 Default Application Settings

As mentioned in Sections 5.2.1 and 5.3.1 the default settings for NetInfService and Elephant are stored in two properties files located at "assets/config.properties" inside each projects folder. These files mostly contain internal application settings but also some settings that can be changed through the applications' setting menus. If settings are changed through the setting menus the default in the files are not changed. Instead the changes are stored using the Android concept shared preferences which in short stores the settings in a for each application assigned file.

##### 9.1.1.1 Elephant Web Browser

The properties file belonging to Elephant contain the following:

- **hash.alg = sha-256**

Constant describing the hash algorithm used. Changing this does not change the algorithm, only the constant used in some functions.

- **access.http.host = localhost**  
The address of the NetInfService RESTful API.
- **access.http.port = 8080**  
The port used by the NetInfService RESTful API.
- **sharing.folder = /DCIM/Shared/**  
The folder used to store downloaded and shared data.
- **restlet.retrieve.file\_path = path**  
The key in the JSON response to a RESTful API retrieve request that contains the file path.
- **restlet.retrieve.content\_type = ct**  
The key in the JSON response to a RESTful API retrieve request that contains the content path.
- **restlet.search.results = results**  
The key in the JSON response to a RESTful API search request that contains the search results.
- **default.webpage = ul.se**  
The default web page of the web view.
- **httprequest.timeout = 2000**  
The maximum time NetInfRequest subclasses wait for a response from the RESTful API.
- **httprequest.encode = UTF-8**  
The encoding is used for HTTP requests.
- **http = http://**  
The HTTP schema.
- **timeout.netinfsearch = 2000**  
Application specific timeout for searches.
- **timeout.netinfretrieve = 2000**  
Application specific timeout for retrieves.
- **timeout.netinfdownload.webobject = 2000**  
Application specific timeout for downloading from the Internet.

### 9.1.1.2 NetInfService

The NetInfService properties file contains several settings that are also in the Elephant browser settings. Duplicates are not repeated here, instead see Section 9.1.1.1.

- **identity.nodeIdentity = ni:HASH\_OF\_PK=123 UNIQUE LABEL=456**  
The OpenNetInf node identity.
- **nrs.http.host = 130.238.15.227**  
The NRS address.
- **nrs.http.port = 9999**  
The NRS port.
- **nrs.http.search.timeout = 20000**  
The timeout when doing a search using the NRS.
- **metadata.\***  
Some field names used as metadata.
- **bluetooth.interval = 300000**  
Period of time, after which a Bluetooth discovery is triggered.
- **bluetooth.timeout = 10000**  
For how long the Bluetooth discover is allowed to run.
- **bluetooth.number\_attempts = 2**  
How many times Bluetooth tries to establish a Bluetooth connection.
- **bluetooth.buffer = 1024**  
The size of the buffer used to retrieve files over Bluetooth.
- **lrs.priority = 77**  
The priority of the LocalResolutionService. Used to determine in which order to use ResolutionServices, higher means more important.
- **nrs.priority = 42**  
The priority of the NameResolutionService. Used to determine in which order to use ResolutionServices, higher means more important.

- **nrs.timeout = 2000**

The HTTP timeout when doing publish or get from the NRS.

- **nrs.max\_message = 100000000**

The maximum number to use as message ID.

### 9.1.2 Development Environment

The Elephant browser and the NetInfService applications were developed on Ubuntu using the Eclipse IDE and Android SDK.

For instructions on how to set up Eclipse IDE to use the Android SDK as well as compiling the applications see Section 9.1

### 9.1.3 Eclipse Project Structure

There are three Eclipse projects:

- Application which contains the code for the Elephant browser
- NetInfService which contains the code for the NetInfService application
- NetInfUtilities which contain some utility functions used by the two other projects

Following are the different packages the applications are structured into and the functionality that is associated with each.

#### 9.1.3.1 Elephant Packages

- **project.cs.lisa.application**

Contains the Android Activities for the main view and the settings screen as well as auxiliary code for these.

- **project.cs.lisa.application.dialogs**

Contains several popup dialogs either asking for input or displaying information.

- **project.cs.lisa.application.hash**

Contains SHA-256 hashing functionality.

- **project.cs.lisa.application.html**

NetInfWebViewClient changes the behaviour of the WebView by telling the WebView what to do when new resources are to be downloaded. NetInf is used to search for, download and publish web pages and/or resources. If necessary the Internet is used.

- **project.cs.lisa.application.html.transfer**

The FetchWebPageTask fetches the HTML source of a web page using NetInf if possible and loads it into the WebView. Then the WebView will load the resources using the NetInfWebViewClient.

DownloadWebObject is used by the NetInfWebViewClient to download the web page or resource from the Internet if not accessible by NetInf.

- **project.cs.lisa.application.http**

Contains the code handling the HTTP communication with the NetInfService's RESTful API. The classes NetInfPublish, NetInfRetrieve and NetInfSearch handle publish, retrieve and search respectively. The classes NetInfPublishResponse, NetInfRetrieveResponse and NetInfSearchResponse represent responses to publish, retrieve and search respectively.

- **project.cs.lisa.application.networksettings**

Contains code for checking and setting the Android devices network settings properly.

### 9.1.3.2 NetInfService Packages

- **project.cs.netinfservice.application**

Contains the Android Activities for the main view and the settings screen as well as auxiliary code for these.

- **project.cs.netinfservice.database**

Handles the SQLite database used by both the LocalResolutionService as well as the UrlSearchService.



- **project.cs.netinfo.service.netinfo.access.rest**  
Handles the HTTP server providing the RESTful API.
- **project.cs.netinfo.service.netinfo.access.rest.resources**  
Contains the implementation of the RESTful API. IOResource handles publish, BOResource handles retrieve and SearchResource handles search.
- **project.cs.netinfo.service.netinfo.common.datamodel**  
Contains extensions to OpenNetInf NDOs. This includes the metadata and content-type fields.
- **project.cs.netinfo.service.netinfo.node**  
StarterNodeThread is used to start a NetInf node.
- **project.cs.netinfo.service.netinfo.node.exceptions**  
The InvalidResponseException is used when an invalid response is returned to a NetInf message.
- **project.cs.netinfo.service.netinfo.node.module**  
The Module class binds abstract interfaces to concrete class implementations using Guice [4].
- **project.cs.netinfo.service.netinfo.node.resolution**  
The resolution package contains the different ResolutionServices, see Section 5.2.5.
- **project.cs.netinfo.service.netinfo.node.search**  
The search package contains the SearchServices, see Section 5.2.6.
- **project.cs.netinfo.service.netinfo.provider**  
Contains the ByteArrayProvider class, see Section 5.2.7.
- **project.cs.netinfo.service.netinfo.provider.bluetooth**  
The BluetoothDiscover class is used to periodically search for and store nearby Bluetooth devices.  
The BluetoothProvider class is described in 5.2.7.1

- **project.cs.netinfservice.netinf.server.bluetooth**

The BluetoothServer listens for incoming Bluetooth pairing requests. As soon as the local device has been successfully paired with a remote device, the BluetoothServer waits for a file request containing a hash. If the specified file is available, the file will be transferred to the remote device.

- **project.cs.netinfservice.netinf.transferdispatcher**

Contains the TransferDispatcher class, see Section 5.2.7.

- **project.cs.netinfservice.util**

Contains utility classes.

The IdentifierBuilder and IOBuilder classes ease the construction of Identifiers and InformationObjects respectively.

#### 9.1.4 Javadoc

Javadoc was used during development to document the functionality of classes. For more detailed description of classes and their methods refer to the Javadoc.

## 9.2 Backend

The following section describes the maintenance and default settings of the backend team's NetInf NRS as well as the NetInf Video Streaming.

### 9.2.1 Default Application Settings

The NetInf NRS application is controlled using one out of two methods, first through the Erlang application src file(netinf\_nrs.app) found in the netinf\_nrs/src directory. The second from the configuration files loaded at run time from the configs directory.

By default the following settings are used when there is no specification on the Erlang command line which config file is to use, please refer to section 5.4.4 to understand what each setting is used for.

In short, the Erlang NetInf NRS will be started with the list database.

```
database nn_database_list
convergence_layers ["http"]
ip_timer 5000
discovery off
nrs_port 9999
ct_port 8077
client_port 8079
list_timer 3600
```

### 9.2.2 Development Environment

To those wishing to continue the development of the Erlang NetInf NRS and the NetInf Video Streaming, the following section details how the development environment was set up.

Please note that the applications were developed on the Ubuntu 12.04 LTS platform, deviating from this may cause the application to behave in unexpected ways.

The recommended editor used was Emacs with the erlang-mode, this editor and mode can be installed using the following commands:

- `sudo apt-get install emacs`
- `sudo apt-get install erlang-mode`

Useful emacs commands include:

- `ALT+X`  
Sets up Emacs for the meta-command mode. Erlang-mode can be set by typing `ALT+X erlang-mode`

- CTRL+X+F Opens a file or create a file.
- CTRL+X+S  
Emacs quick short cut for saving files
- CTRL+C+K  
Emacs quick short cut for compiling and saving Erlang files
- CTRL+X 1-3  
Emacs quick short for dividing the windows into 1 whole window(1), horizontally(2) or vertically(3).

Please note that Emacs has auto-completion for commands when pressing tab. Other useful tips include Erlang-mode skeletons which allow the developer to import comment sections and whole skeletons for generic servers and behaviours.

In addition to running the NetInf NRS, developers should note that Erlang comes with a utility to monitor the processes spawned in applications called AppMon. To start AppMon, start an Erlang shell and use the following command.

```
appmon:start().
```

### 9.2.3 Code and folder structure

The NetInf NRS and the NetInf Video Streaming application are organized in the following way:

**netinf\_nrs** The main folder which holds the following folders as well as the other important files for the application.

**configs** The folder which contains all the configuration files for the NetInf NRS application. Please place all the new configuration files here.

**curludp** This folder contains a text file which is read by udp\_test.sh. It has no other uses.

**deps** This folder is created automatically by running the rebar (or the script, which invokes makes and eventually rebar). It contains all the dependency code required for libraries that were used in the NetInf NRS application.

- doc** This folder is created automatically by running the `rebar doc` command (see 9.2.5). Use the `index.html` file to get to the first page of the documentation. Note that this is normally not present unless the `doc` command has been run.
- ebin** This folder contains all the compiled Erlang beam files, this is where the Erlang virtual machine will look for the compiled code modules.
- files** This folder contains all the stored binary content (NDO cache). The NetInf NRS will look here and determine if it has the content or not. If the NRS receives a get message, alternatively if the NRS receives a publish message with binary octets, it will save it here. This folder is required for the NRS to start properly and will be created using the make target `"set_env_folders"`.
- logs** This folder contains all the logging files, it also contains a folder named `old`. The logger service in the NRS will create a text file with information about the NRS and current activities up to a default size of 10MB (may be changed in the `nrs_logger.erl` file in the `src` directory). This folder is required for the NRS to start properly, this folder will also be created using the make target `"set_env_folders"`
- resources** This folder contains all the resources associated with the HTML client interface for the NetInf NRS video streaming.
- src** This folder contains all the source code for both the NetInf NRS and the video streaming.

Please note that inside the main `netinf_nrs` folder several files exist consisting of a `Makefile`, `rebar.config` file, `udp_test.sh` -udp testing script, `readme` and the `netinf_nrs` startup/install script.

**Makefile** This is the make file with several targets shown below. It is primarily used for compiling the NetInf NRS project and invoked by the main `netinf_nrs` script.

- `all`  
Creates the required folders for the environment, compiles both erlang source code and the JSON c++ and finally runs `eunit` but this does not start the NRS.

- `all_no_test`  
Same as the above, however it does not invoke the eunit tests.
- `eunit`  
Runs the eunit tests using the rebar and skips all the dependency tests (only tests NetInf NRS).
- `integration_test`  
Compiles the Erlang source code and the dependencies if needed and then runs only the `integration_test` code.
- `integration_test_riak`  
Same as the above but will attach the Riak database to the Erlang NetInf NRS instance and run the `integration_test` code on that.
- `makeec`  
Compiles only the C++ JSON dependency code in the `deps` folder.
- `set_env_folders`  
First removes the following required folders: `logs` and `files` and then re-creates them.
- `compile`  
First tests if the `"deps"` folder already exists and then compiles the dependencies, otherwise it will download all the required dependencies and then compile them.
- `compile_deps`  
Cleans the `"deps"` folder, then downloads all the dependencies again and compiles them.
- `start_script_riak`  
Runs the steps in the `all_no_test` target, then starts the NetInf NRS with the Riak database attached.
- `start_script`  
Runs the steps in the `all_no_test` target, then starts the NetInf NRS with the default list database attached.
- `clean`  
Removes the environment folders (`logs` and `files`) and removes the ebin compiled folders as well as the crash dump if the Erlang virtual machine has crashed previously.

**rebar.config** This file defines all the settings for rebar in this particular product. The dependencies as well as options for eunit and various plugins to rebar can be configured.

**udp\_test.sh** This file is a script for testing the UDP convergence layer. Please note that it is best tested with the discovery turned off in the config file. At least two different computers are required to run these tests. All instructions are in the script.

**netinf\_nrs.sh** This file is the main setup/install and run script. Please use this to install all the required components on the machine to ensure maximum compatibility. More details about this script can found seen in 8.2.2

#### 9.2.4 NetInf NRS modules

The main code modules are located in the "src" directory of the main folder. Each file has comments inside which can be generated into documentation please see subsection 9.2.5

**Erlang-application file** Erlang applications require a definition file in order for the Erlang virtual machine to be able to understand which modules need to be preloaded and what configurations if any need to be supplied to the application.

- **netinf\_nrs.app.src** - This file gets read by the Erlang virtual machine at compile time. Developers can set various options for the default settings in the "env" section of the file.

**Supervisors** Erlang uses supervisors to organize which process must stay alive for a application to function as intended, below is a brief description of all the supervisors required in the NetInf NRS application

- **nn\_sup** - This is the main supervisor which starts the persistent processes mainly: sub, msg id, and client supervisors as well as the storage, discovery(deprecated), logger, stats and the udp handler.
- **nn\_sub\_supervisor** - This is the sub supervisor. It is responsible for starting the following non-persistent processes: event\_handler, message\_handler, content\_handler, http\_forwarder, udp\_forwarder and finally the content\_transfer\_handler.
- **nn\_client\_supervisor** - This is the client supervisor. It is responsible for starting the http client stream handler.

- `nn_msgid_sup` - This is the message id supervisor. It is responsible for starting the modules associated with message id storage.

**Convergence-Layers** As stated in section 5.4.7 the application was designed to be modular, the idea of convergence-layers allowed the application to group three distinct modules together to create the convergence-layer in Erlang.

- `nn_http_handler` - This module contains code to process and send/receive http requests from the outside world(using the cowboy library). It is the first part of the HTTP convergence layer.
- `nn_http_forwarder` - This module contains code to send and receive http requests to other NRS' when a search has failed in the NRS system. Note, this feature(HTTP forwarding) is only used when the system has been started with the `static_peers` configuration. See 5.4.4 for more details.
- `nn_http_formatting` - This module is responsible for taking a HTTP message and converting it to the internal representation of the NetInf Message and vice versa. It interacts with the `message_handler` passing converted messages back and forth. The `message_handler` that has been spawned with HTTP as the convergence layer uses this module.
- `nn_udp_handler` - This module contains code to send/receive Net-Inf UDP messages. It is spawned by the sub supervisor and serves as the entry and exit point of the system for the UDP convergence layer.
- `nn_udp_forwarder` - This module contains code to convert and forward messages from another convergence layer into UDP specific messages and then passes them to the UDP handler to send out of the system.
- `nn_udp_formatting` - This module contains code to convert UDP NetInf messages and extract information into a NetInf Message for use in the `message_handler`. The `message_handler` that has been spawned with UDP as the convergence layer uses this module.
- `nn_message_handler` - This module is responsible for accepting messages from a convergence layer handler. It is spawned with the specific convergence layer name so that the module can redirect requests to the appropriate formatter. The handler also for-



wards messages to the event handler for further processing in the system.

**Database behaviour & Storage interface** This application contains a custom behaviour to allow developers to quickly create wrappers for databases as well as functionality to change the database at run-time. The following modules are involved:

- `nn_database` - This is the custom behaviour implemented for creating database wrappers. Each new database wrapper must implement this behaviour. Erlang will then warn the developer if there are missing key functions in the implementation of the new database wrapper. The section PNP Database Wrapper 5.4.9 has more information.
- `nn_storage` - This is the interface between the database wrapper and the content caching. This module is responsible for facilitating requests from the event\_handler.

**Databases** The NetInf NRS application has support for various plug and play database wrappers. As long as the developer adheres to the required input and output of the database behaviour this application can be extended to work with any database.

- `nn_database_list` - This module implements the callback functions defined in the `nn_database`. It is also a quick database consisting of a persistent Erlang list data structure. The module also has a timer which causes the list structure to be saved to disk every hour. This can be controlled in the `configs/list.config` file under the appropriate variable.
- `nn_database_riak` - This module implements the callback functions defined in `nn_database`, it is a wrapper for talking to a Riak process (Riak is a standalone database).

**Content-Caching** The NetInf NRS also includes a method of caching binary objects sent into the system via NetInf messages, the following are the two modules involved in this functionality.

- `nn_content_handler` - This module is responsible for handling the binary octets(files) coming into the system it is also the interface for storing and retrieving the files associated with NDOs.

- `nn_hash_validation` - This module validates the hash of the NDO coming in against the one that is currently stored in the files folder. Please note that the files folder must be present in the system otherwise the application will crash. As stated previously, the make target "set\_env\_folders" can be used to create the required environment folders.

**NetInf Video Streaming** The NetInf NRS application supports a video streaming protocol on top of the existing application. The main files used in this protocol are described here. Note that the video streaming also relies on the resources folder as well to provide the http client interface to the user.

- `nn_subscribe` - Responsible for subscribing to a stream, it is only used in the NetInf video streaming.
- `nn_stream_handler` - Responsible for polling and fetching the chunks for the users.
- `nn_stats` - Responsible for keeping track of various statistics.
- `nn_ct_handler` - Responsible for transferring of content without NetInf
- `nn_http_client_handler` - Responsible for exposing the `http_client_interface` to users.
- `nn_http_ct_handler` - Responsible for transferring of content over HTTP.

**Logger** The NetInf NRS application supports a file based logging method, which creates a file named `log.txt` in the logs folder. The logger comes with three(3) levels: verbose, warning and error. Developers can choose which level to log at in the configuration file. By default the log file sizes are set to 10MB and then the log file gets moved to the `old/` folder. You can increase this in the `nn_logger` module under the macro `LOG_FILE_SIZE`.

- `nn_logger` - This module is responsible for opening the log file and writing to it.
- `nn_logger_server` - This module is responsible for handling the requests to log the event from various modules.
- `nn_log_handler` - This module contains the `gen_event` server which the logger modules connect to, the logger server accepts the requests from the handler.

**Message ID storage** The distributed nature of the NetInf NRS and multiple messages required the developers to be able to keep track of which message is associated to a specific process id and convergence layer handler. The message storage is a persistent table that maps this and allows the application to forward requests to specific handlers.

- `nn_msgide` This module contains code to store one message id to process id mapping.
- `nn_msgids` This module contains code to initiate to insert, lookup and delete the message ids
- `nn_msgid_store` This module contains the interface to the `nn_msgids`. Developers should call this module to interact with the message id table.

**Utility & Misc** The following modules are used to expose a variety of functions through out the system.

- `nn_util` - This module contains many useful functions that were being used in multiple modules. It is recommended to read through the functions here as a function may have already been created and exposed to the developers.
- `nn_merging` - This module contains all the functions for merging metadata.

**Integration test** The NetInf NRS application required a test in order to check if all the modules were working as intended using the black box testing technique.

- `nn_integration_test` - This module contains the black box level tests for the entire NetInf NRS system.

The majority of the above modules have unit tests for them in the same folder, they are denoted with the same starting name but also have the `_test` as well.

### Other Important Modules

`netinf_nrs` - holds the main code for starting and stopping the application along with all the required dependencies(Ranch, Crypto, Cowboy).

`nn_app` - Starting point of the `nn_application`. Initiates a HTTP listener, starts the main supervisor and reads all the configuration settings from the config files as well as the env from the `netinf_nrs.app.src` file.

`nn_event_handler` - This module is responsible for passing messages between the storage interface and `content_handler`.

`nn_proto` - This module contains the internal representation of a NetInf message based on the draft. It also contains functions to get and set the messages. It is used in many modules and it is part of the core NetInf NRS architecture.

`nn_discovery_service` - This module is deprecated.

`nn_discovery_client` - This module is deprecated.

### **9.2.5 Generating documentation**

Code documentation for the application can be generated by running the following commands in the terminal when in the main `netinf_nrs` folder.

```
rebar doc skip_deps=true
```

This will create a new folder "doc" in the main `netinf_nrs` folder. The documentation should be read from the file named `index.html`

## Chapter 10

# Appendix C: NetInf Video Streaming Draft

### 10.1 NetInf Video Streaming Protocol

#### 10.1.1 Introduction

The purpose of this draft is to outline a design and protocol specification for enabling of streaming and chunking data within the current and existing netinf architecture.

##### 10.1.1.1 Proposed method of retrieving chunked NDOs

The stream source will PUBLISH an NDO containing the filename as content. The object is marked as a stream in the metadata. It also contains a locator to the stream source.

In order to access the stream, a receiver will first perform a NetInf-GET on the above filename object in order to retrieve the locator.

In the next step the client can get the chunks. This is done by replacing the hash algorithm in the NDO-name with demo and sending a NetInf-Get to the source. The stream provider will then return an NDO with the octets of the most recent chunk and the chunk number in the metadata. This implies that the regular hash validation has been disabled.

For further chunks, the receiver will increment the chunk number and append it to the locator.

If a stream is published with the name

```
ni:///sha-256;Wk0CMB2aEQHrARjTldfhRE50gZkZmCHzokcoVMnfp2Y
```

You can get latest chunk by fetching

```
ni:///demo;Wk0CMB2aEQHrARjTldfhRE50gZkZmCHzokcoVMnfp2Y
```

To get first chunk, append a 1 to the end of the hash

```
ni:///demo;Wk0CMB2aEQHrARjTldfhRE50gZkZmCHzokcoVMnfp2Y1
```

To get the 32th chunk, append 32 to the end of the hash

```
ni:///demo;Wk0CMB2aEQHrARjTldfhRE50gZkZmCHzokcoVMnfp2Y32
```

If the receiver also acts as a cache, it will send a PUBLISH with the filename object and a locator pointing to itself to the NRS. Figure 10.1 shows the flow of the actions.

#### 10.1.1.2 Testing criteria

For a simple test setup, in addition to the NetInf node, a receiving and a publishing client is required. A more sophisticated test could involve multiple receivers, to demonstrate the caching.

A use case for testing is one where the source of the content is a pre-encoded video file of a particular size.

The example below assumes a video file of the size 700 megabytes.

A client (Publisher) will have a default chunk size which is used to chunk up the source file. For this example, 1 megabyte chunks will be used.

Thus, it can be calculated that the publisher will create 700 chunks of the size 1 megabyte, numbered 1-700.

A second client (receiver) knowing the name of the video file will then send a Get request to the NetInf node (to keep things simple, it is assumed that the object's name is already known).

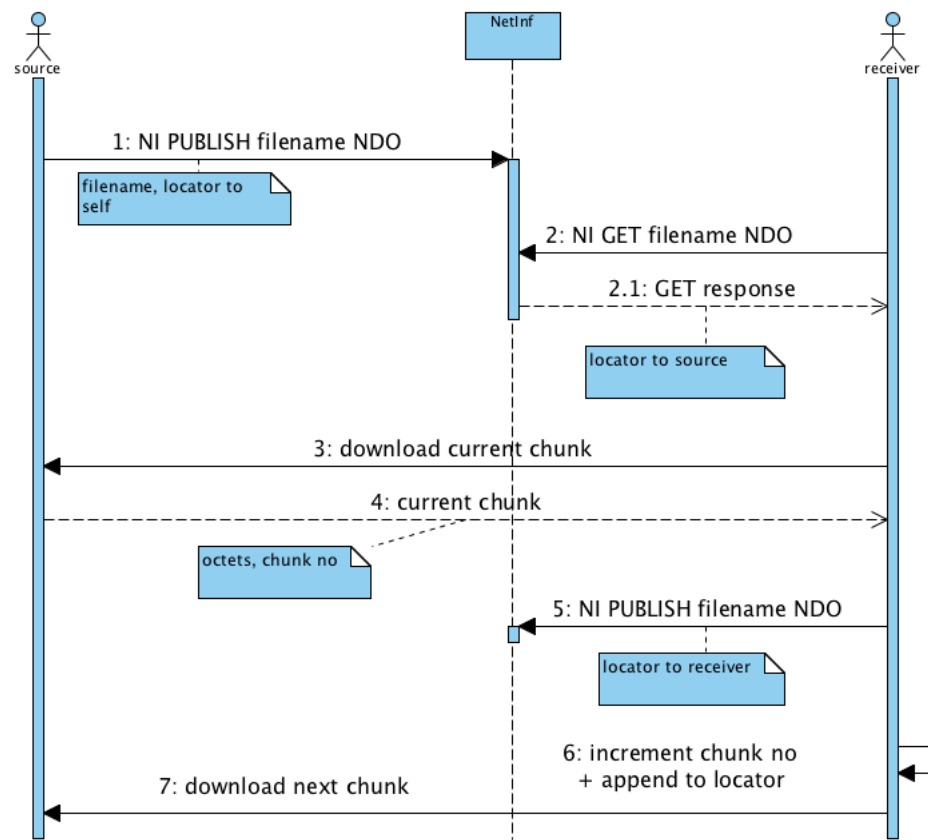


Figure 10.1: Streaming Sequence Diagram

The client will now start generating consecutive GET requests with the sequence numbers constantly increasing and expect to receive the appropriate chunk.

As well as poll the NRS for new locators, to get better load balance between nodes contributing to the stream.

This process occurs until the GET requests generate a 404 because the unique sequence number has increased past the last published chunk number.

#### **10.1.1.3 Extra notes**

Chunk size is going to be a configurable option in the publishing client.



## Chapter 11

# Appendix D: License

Copyright 2012- 2013. Ericsson, Uppsala University

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Uppsala University. Projekt CS course, Fall 2012

Projekt DV/Projekt CS, is a course in which the students develop software for distributed systems. The aim of the course is to give insights into how a big project is run (from planning to realization), how to construct a complex distributed system and to give hands-on experience on modern construction principles and programming methods.

# References

- [1] Alexa - the web information company. Retrieved January 16th, 2013, from <http://www.alexa.com/>.
- [2] Erlang programming language. Retrieved January 15th, 2013, from <http://www.erlang.org/>.
- [3] The fp7 4ward project. Retrieved January 8th, 2013, from <http://www.4ward-project.eu/>.
- [4] Google guice (n.d.). Retrieved January 15, 2013, from <http://code.google.com/p/google-guice/>.
- [5] Javascript — mdn. Retrieved January 15, 2013, from <https://developer.mozilla.org/en-US/docs/JavaScript>.
- [6] Sail. Retrieved January 17th, 2013, from <http://www.sail-project.eu/>.
- [7] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. IEEE Communications Magazine, 2012.
- [8] E. Bauer M. Becker F. Beister N. Dertmann-R. Hrestic M. Kionka M. Mohr M. Mühe D. Murali F. Steffen S. Stey E. Unruh Q. Wang C. Dannewitz, M. Herlich and S Weber. Opennetinf documentation design and implementation. Technical report, University of Paderborn, September 2011.
- [9] S. Farrell D. Kutscher and E. Davies. The netinf protocol. Handed to us by Ericsson, 2012.
- [10] Ivan Dubrov. Coverttool. Retrieved January 15th, 2013, from <https://github.com/idubrov/coverttool>.

- [11] Loc Huguin. Erlang cowboy. Retrieved January 15th, 2013, from <https://github.com/extend/cowboy>.
- [12] Martin Logan, Eric Merritt, and Richard Carlsson. Erlang and OTP in Action. Manning, November 2010.
- [13] Android Developers (n.d.). Android developers. Retrieved January 8th, 2013, from <http://developer.android.com>.
- [14] H. Otaola and M. Sosa. Using multiple transport networks in netinf enabled android devices. Master's thesis, KTH, School of Information and Communication Technology (ICT), Sweden, 2012.
- [15] Kostas Pentikousis, Prosper Chemouil, Kathleen Nichols, George Pavlou, and Dan Massey. Information-centric networking [guest editorial]. IEEE Communications Magazine, 50(7):22–25, 2012.
- [16] C. Dannewitz B. Ohlman A. Keranen P. Hallam-Baker S. Farrell, D. Kutscher. Naming things with hashes draft-farrell-decade-ni-10. Handed to us by Ericsson, 2012.
- [17] Basho Technologies. Riak. Retrieved January 15th, 2013, from <http://docs.basho.com/riak/latest/downloads/>.
- [18] Paul J. Davis YAMASHINA Hio.