Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Mungai Evans

Application Design on Information Centric Networks

Master's Thesis
Espoo, June 18, 2013

Supervisor: Professor Martti Mäntylä
Advisor: Kari Visala M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Mungai Evans

Title:
Application Design on Information Centric Networks

Date: June 18, 2013 **Pages:** vii + 55

Major: Service Design and Engineering **Code:** T-86

Supervisor: Professor Martti Mäntylä

Advisor: Kari Visala M.Sc. (Tech.)

Over a span of three decades the requirements by users and applications of the Internet have changed from what they were in the beginning. It is used more for distribution than resource sharing which is what its architecture was designed for. This has brought new challenges. To solve this, academic research in the field of Future Internet Architectures (FIA) has led to different new Internet Architecture proposals known as Information Centric Network (ICN) architectures. However, little has been done to validate how these architectures fit into the real world.

The aim of this thesis is to validate how the PURSUIT architecture, one of the FIA proposals, fits in the real world. This will be done through porting an open source collaborative editor on top of Blackadder, the prototype of the PURSUIT architecture. This architecture follows a publish/subscribe model where data is published into, and subscribed from the network by applications. Unlike Host Centric Networks (HCN) networks, ICN networks do not identify hosts, rather the data is named and any interested party would use the names to access data from the network. Due to this nature of ICN networks, this thesis will propose a server-less design as the approach to developing distributed applications on top of ICN networks, through porting a collaborative editor. The porting experiment will follow this server-less based design as opposed to the more common client/server model that the collaborative editor has been designed in.

After the porting exercise, we shall evaluate our findings through measuring quality metrics and performing static analysis. The quality metrics will show that there will be increase in complexity of the design mainly because of having the client applications and the network dissolve the functions that the server performs. A new dimension of concurrency control of state within a distributed network will be realized. In a client/server model, the requests to the server are serialized, hence not having the concurrency control challenge. However, this will not be researched on in this thesis. We shall propose to have this as a future research area.

Keywords: Future Internet Architecture, Information Centric Networking, Host Centric Network, Server-less, Porting, Impedance mismatch, publish/subscribe, named data

Language: English

Acknowledgements

I would like to express my gratitude to all those who assisted me in completing this thesis. In particular, I would like to thank the PURSUIT research group for giving me this opportunity. I would also like to give special thanks to Kari Visala for his guidance all through this research project. He was of great help in guiding me towards delivering scientific work rather than focusing only on the engineering aspects of the research. He also had very constructive feedback during the writing phase. I would like to thank Apurva Jaiswal as well whom we worked with on the experiment in this thesis. We spent long hours brainstorming, designing, programming and occasional constructive arguments.

Furthermore, I would like to acknowledge the crucial role of the PURSUIT team, which developed and maintains the Blackadder prototype, for their assistance in our experiment. I would also like to thank the developers of Gobby collaborative editor for their assistance as well in the experiment.

Last but not least, I would like to acknowledge the support of my social support system. In specific, I would like to express my deepest gratitude to my fiancée, Catherine Kanyi, for her moral support and prayers all through this project. When things were tough and I almost gave up, she encouraged me. You are a strong foundation that I stand on. I would like to lastly thank my friends and family for their moral support and encouragement all through the time I spent in this research.

Espoo, June 18, 2013

Mungai Evans

Abbreviations

ICN	Information Centric Networking
HCN	Host Centric Network
FIA	Future Internet Architecture
UML	Unified Modelling Language
IDE	Integrated Development Environment
PC	Personal Computer
XML	Extensible Markup Language
API	Application Programming Interface
GUI	Graphical User Interface
TCP/IP	Transmission Control Protocol and Internet Protocol
XMPP	Extensible Messaging and Presence Protocol
DNS	Domain Name System
ISP	Internet Service Provider
TTL	Time To Live
UDP	User Datagram Protocol
CDN	Content Delivery Network
DHT	Distributed Hash Table
VPN	Virtual Private Network

Contents

List of Figures

Chapter 1

Introduction

This chapter will introduce this thesis by giving some background information of the research area and the motivation of doing this thesis in Section 1.1. Section 1.2 describes the research problem to be focused on in this thesis. Section 1.3 details the contributions of this thesis and finally, Section 1.4 gives a walkthrough of the thesis structure.

1.1 Motivation

Over a decade ago just when the Internet came to be, one of the reasons connectivity was needed was to share resources. Such resources were expensive machines such as printers, storage machines, supercomputers etc. [21]. This has long changed with the drop in prices of hardware such as memory, processors and hard disks. With these technological developments, the need for sharing resources has decreased in relevance. Today the Internet is used more for (re)distribution of content, i.e. publishing videos through services like YouTube, sharing content through social networks like Facebook, sending and receiving email, making voice calls etc. This is in contrast to what the Internet was designed to do.

Information Centric Networking (ICN) also known as Data Oriented Networking [26], or Content Centric Networking [21], is a new networking paradigm aiming to redefine the current Internet architecture. This paradigm focuses on WHAT, i.e. the data that a user requests for, rather than HOW, i.e. the means at which the data is delivered to the user, or WHERE, i.e. the location the data comes from. The current Internet architecture was designed to

connect machines, or network nodes together through having them negotiate a communication line set up through dialling. Data retrieval was possible only when these two nodes were able to converse. This type of a network is known as a host centric network (HCN). It provides data by having users define HOW they would like to receive the data, i.e. TCP or UDP packets, and from WHERE the data would come from, i.e. an IP address.

This inapplicability of today's Internet to serve current user needs has been widely known as an impedance mismatch [47], [44]. Impedance mismatch is a conflict between what the network service model offers, and what applications require from it e.g. APIs designed to be used by applications are tightly coupled with the network protocol i.e. the socket API is used to send and receive TCP and UDP packets and is bound to either IPv4 and the later IPv6 [25]. Impedance mismatch has led to middleware being developed on top of the current network architecture in an effort to bridge this gap. Such middleware include web server technologies that offer services to clients through dedicated servers, peer-to-peer technologies and Content Delivery Networks (CDN) for improved content distribution etc. Research in this area has led to several projects that have come up with different solutions in form of architectures in an effort to solve this problem. This field of research is known as future internet architectures (FIA).

1.2 Research Problem

One of the targets of this thesis is to validate a claim made within the PURSUIT research group based on the PURSUIT architecture [44]. This will be done through porting a non-trivial application on top of Blackadder [23], the prototype implemented based on this architecture. A non-trivial application in this case means an application that uses network services for complex tasks such as maintaining state of data. Some applications have been implemented previously [1], [22] on other prototypes to validate this claim, however the nature of these applications has made them easy to map on top of ICN. BitTorrent for instance is straight forward to implement [22] because it adopts a publish/subscribe design where pieces of data are distributed to peers interested in downloading and uploading content [6]. Each piece is identified using a SHA1 hash of the piece of data. With this type of approach, mapping the hash values to publication identifiers is straight forward because the hash values already fulfil all the properties required by data identifiers in an ICN network as we shall see in Section 5.1.

The other target of this thesis is to find out how the design of applications changes for applications targeting ICN. ICN paradigm focuses on WHAT data is needed rather than HOW it is delivered, or WHERE it originates from. Applications developed today, that require network services, follow the HCN paradigm as this was the first networking paradigm that is still used to date. Requirements by users, and hence applications, have changed overtime prompting a paradigm shift towards ICN. A one-to-one match of application design would not be possible because of the fundamental differences in both networking paradigms. As a result, the research questions that need to be answered in this thesis are:

1. *Is the claim that there is an improved impedance match between ICN service model and what applications require from the network true [44]?*
2. *Does the complexity of an application design change when designing it for ICN as opposed to designing it for HCN?*

1.3 Contributions

The main contribution made in this thesis is an application design suited for applications targeted to be used in a distributed network. In today's Internet architecture, most of such applications are designed following the client/server model. This is due to the inherent nature of HCN networks which enables to have clients address servers using IP addresses as identifiers of the servers in the network. In ICN however, the architecture does not provide a mechanisms to name and identify network hosts. This poses new challenges in how applications use the network.

In the experiment performed as part of this research work, a server-less design will be followed instead of the client/server model. The server-less design is not new as a design approach [28], however, the fit of the design on ICN is the contribution made in this thesis. We will show that this design suites applications designed to be deployed in an ICN network in comparison to a HCN network. We will also validate the claim that ICN improves impedance mismatch [44]. The need for new algorithms to handle the new challenges introduced by this approach in application design will be recognized but the research of this algorithms will be proposed as future research areas.

1.4 Thesis Structure

This thesis is divided into seven chapters. In Chapter 2, the research methodology and research approach is discussed. Chapter 3 discusses the background information relevant to this thesis. Here the PURSUIT architecture and the Blackadder prototype are analysed as well as FIA research background. Chapter 4 discusses the porting experiment. The study design is detailed which explains the experiment environment. The application to be ported in the experiment is also introduced in this chapter. In Chapter 5, the selection process and analysis of the application is documented as well as the entire porting exercise. Chapter 6 holds the discussion part where findings and their generalizations are discussed. The last chapter concludes this thesis by highlighting what was done, challenges in the experiment and by making proposals for future work.

Chapter 2

Research Methodology

This chapter discusses the research approach and methodology used. Section 2.1 describes how the literature review of the topic and research field in general is conducted. Section 2.2 shows and justifies the framework used in this research exercise while Section 2.3 defines methods used to evaluate the research findings.

2.1 Literature Review

A literature review was conducted to get an understanding of ICN. Materials were mainly gathered from ongoing research projects in the area of "Future Internet Architectures". Publications and deliverables were read through as well as documentations of the architectures designed. The focus was mainly on the problems that these architectures aim to solve and their differences with respect to ways of naming data as will be shown in Chapter 3. A review on the current Internet architecture was conducted as well. This was not done in depth because the material covering ICN had good summaries on the architecture and comparisons with ICN. The knowledge gathered here was also to be used to validate results found during the research.

Study on application designs was done as well. This was mainly in search of the best application design that utilizes distributed networks, and used at a large scale.

2.2 Research Framework

Guideline	Component
Design an Artefact	The research aims to produce an application design model as an artefact.
Problem Relevance	The relevance is discussed in Section 2.1.
Design Evaluation	The artefact (application design) will be evaluated through testing the ported application on a test environment. Static analysis, through lines of code measurements and class design analysis, will also be performed evaluating the application design against the requirements of developing an application on an ICN service model, Blackadder in this case.
Research Contribution	Claims made based on the PURSUIT architecture will be validated using the application design artefact and proposals made.
Research Rigour	The research will be built on a vast background of ongoing work in the future internet architecture field. Scrutiny of the work by other researchers will be used and iteratively the research will be improved. Where possible, mathematical models will be used to the extent possible.
Design as a search process	Porting is done iteratively in search of the application design that best fits with ICN.
Communication of research	The work will be presented as a thesis document and the application will be contributed as open source as part of the Blackadder open source prototype.

Table 2.1: Guidelines to follow in this research which follow the research framework designed by Hevner et al.

In research, an important part of the activity is how the knowledge is captured, documented and communicated. This in the research field of Information Systems is known as *design theory [13]*. In addition, Hevner et al. use the term *design science [13]* to define the research process of producing design theory. They define design science as a *paradigm that seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artefacts.* This research aims to create an application design as an artefact that will in turn be used to evaluate the claim that the PURSUIT

architecture *provides an improved impedance match towards application-level concepts [44].* Our research fitted well with design science research framework designed by Hevner et al. [14] and for this reason, we chose it to guide our research. The framework defines seven guidelines to follow during the research process. Table 2.1 shows the seven guidelines including components of this thesis.

2.3 Evaluation Metrics

To quantify the results found in the experiment, the complexity of the design was measured. Two metrics were used for this purpose.

1. Lines of code: This metric refers to the number of the executable source code lines that are part of the software component under investigation. An increase in the number of lines of code is directly proportional to the effort needed to maintain that software component. It also potentially indicates an increase in complexity, but the increase cannot be quantified accurately [46]. This is because the complexity of each line of code is not considered when using this metric. In this project the SLOCCount utility [45] was used to collect this information.

2. Cyclomatic Complexity metric: This metric, which was introduced by Thomas McCabe [29], is based on the topological structure of the code. Thomas uses the graph mathematical construct to analyse the complexity of source code where each node corresponds to a block of code which has one sequential flow, while the edges correspond to branches followed after making decisions during the execution of the code. This new formed graph is known as the program control graph. Each independent path in the graph is formed by a linear combination of nodes and edges i.e. a→b→d→e→f in Figure 2.1. Such a path represents a testable functionality in the program. The minimum number of such paths that cover the whole software component, hence representing the amount of testing required, is known as the *Cyclomatic Complexity Metric [46].* This value is calculated using the formula below

$$V(G) = e - n + 2 \text{ [29]}$$

where G is the program control graph, e is a branch in the execution flow and n is a piece of sequentially executable code. If for example

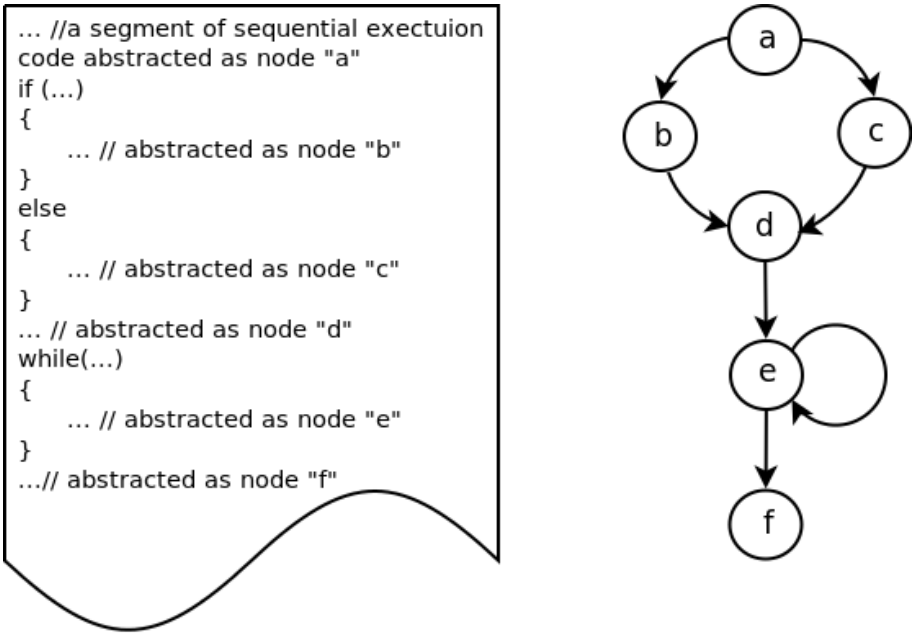

Figure 2.1: Source code and its program control graph example [46]

this formula is applied on Figure 2.1, the cyclomatic complexity of this code would be $V(G) = 7 - 6 + 2 = 3$. A high cyclomatic complexity represents a software component that is complex and hard to maintain.

In this project, we will use a plugin of the Eclipse [33] IDE known as metriculator [27]. This plugin helps generate code analysis metrics such as cyclomatic complexity on projects written in C or C++ programming languages.

Chapter 3

FIA Background

There have been, and still are several funded research groups working in the FIA research area. They have different approaches in their architecture proposals. This chapter will review some of the similarities and differences of the architectures on a high level. The PURSUIT architecture will also be reviewed but on a deeper level as it will be the basis of our experiment.

3.1 Real world problems solved

FIA architectures were motivated by three problems experienced in the Internet today [26], [21].

1. Persistence: Data should always be accessible once published into the Internet. This means that data availability should not depend on the location the data is stored. Today's Internet has been challenged by this problem and can commonly be seen by how many "broken links" there are.

2. Availability: Data and services should be highly available to users, i.e. reliable sources and low latency. The current Internet architecture has had ad hoc ways to solve this by introducing CDNs such as Akamai [2] and the use of peer-to-peer mechanisms like BitTorrent [3].

3. Authenticity: Since the current Internet architecture is not data centric, securing data to ensure it has come from the right source requires securing the actual channel that the data travels in. This again is an

impedance mismatch as security is a requirement of the data not the channel.

3.2 Naming

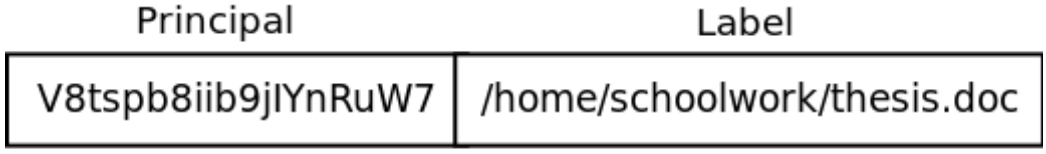

Figure 3.1: DONA flat label name structure

In ICN, data is identified by names within the network. There are two different ways that have been proposed by different architectures in this research area. The first one is naming data using flat labels. Data Oriented Network Architecture (DONA) [26] uses flat labels to name data.

DONA names structure consists of two parts as shown in Figure 3.1 above; a principal and a label. The principal part is a cryptographic hash of a public key which is part of a public-private key pair, owned by an entity equivalent to a domain or host in today's networks. The label part is an identifier created by the owner of the principal and should be unique within the domain of the principal. The label part is equivalent to resources within one domain in the Internet today. DONA names are not dependent on any location. This means that the data can be stored in any location without changing the identifiers i.e. if a student moves their web site from the school servers to a different hosting, the resource can still be accessed using the same name, therefore remaining persistent in the Internet. Users can also verify source of requested data once they receive it. This is made possible by the fact that DONA names are *self-certifying.* A user receives a triplet containing the data, owner's public key and a signature of the key which an application can use to validate the source of the data. Trust of public keys is guaranteed by third party Certificate Authorities [25].

The second way of naming data is by the use of hierarchical names i.e. `/aalto.com/dissertations/mungaievans/thesis/version1.` Such a data name, unlike a flat label, follows a hierarchy representing the data in the in-

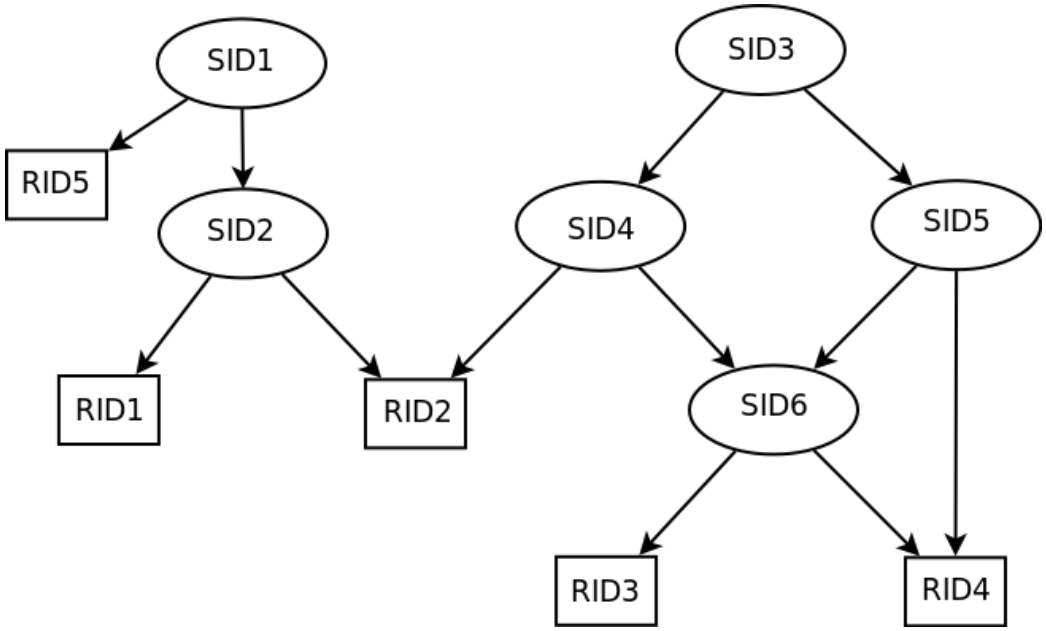

Figure 3.2: Information graph of data in a PURSUIT network

formation space. Content Centric Networking architecture (CCN) [34] uses hierarchical names to name data. CCN proposes adding version information of named data equivalent to TCPs sequence numbers. This can be used in use cases such as publishing video content where each frame would contain a version number and through computing the next version, a client would stream the entire video.

3.3 PURSUIT

The PURSUIT project [36] has put about five years into producing knowledge on ICN starting from the initial PSIRP project [35]. The work has led to an architectural design being produced and a prototype implementation known as Blackadder [23]. The architecture defines three core functions [43].

Before describing the three core functions, we will define what is known as *identifiers* which is the basis of these core functions. Since ICN is based on naming of data, *identifiers* define how information is identified within a network. The architecture follows a hierarchical type of an acyclic information graph as shown in Figure 3.2 below.

There are two information identification primitives. RID, also known as the *Rendezvous Identifier,* in the information graph above is a primitive which identifies a piece of data within the network. It is always a leaf node in the graph. The SID primitive known as a *scope identifier,* holds other identifiers that could either be RIDs or other SIDs. It can be termed as a container of identifiers. An SID never directly points to any piece of data in the network. The information graph is created and managed by the owner of the information as the architecture does not enforce any structure. Information items are identified using absolute path identifiers starting from the root SID to the leaf RID that points to the data i.e. SID1/SID2/RID2 and SID3/SID4/RID2 would identify different pieces of data in the network even if the leaf RID is the same. Each identifier is a 64 bit flat label. The architecture does not define any naming scheme other than the fact that each label needs to be unique. A complex naming scheme such as the P:L structure [10] or simple generated strings would suffice depending on the requirements of the application using the network.

The first function is *rendezvous*. This architecture follows a publish/subscribe paradigm. Each node within the network publishes data identified by a unique name consisting of RIDs and SIDs. Nodes interested in this data subscribe to receive that data using the name identifying the data. Matching the publisher and the subscriber so that communication may take place is handled by this function.

The second function is *topology management*. Once the *rendezvous* function pairs a subscriber to a publisher, a path of how the data would be delivered is required. This task is handled by this function. A topology graph is created considering aspects such as security and access policies. The scope of this topology graph is within a single network topology which includes one topology manager, a rendezvous manager and many number of nodes. Several of these networks can be interconnected but each would be autonomous sharing information with the other network topologies.

The third function is *forwarding*. Once the *topology manager* creates a topology graph, this function handles how the information is delivered. This is achieved through the use of Bloom Filters [41]. These are additional bits attached to a data packet header that contains information of how the packet should be forwarded. This is in contrast to IP packets that use routing tables based on the destination IP addresses attached in the headers.

Network caching of publications is also supported in this architecture. Storage of data is handled by storage devices as seen in Figure 3.3. With respect

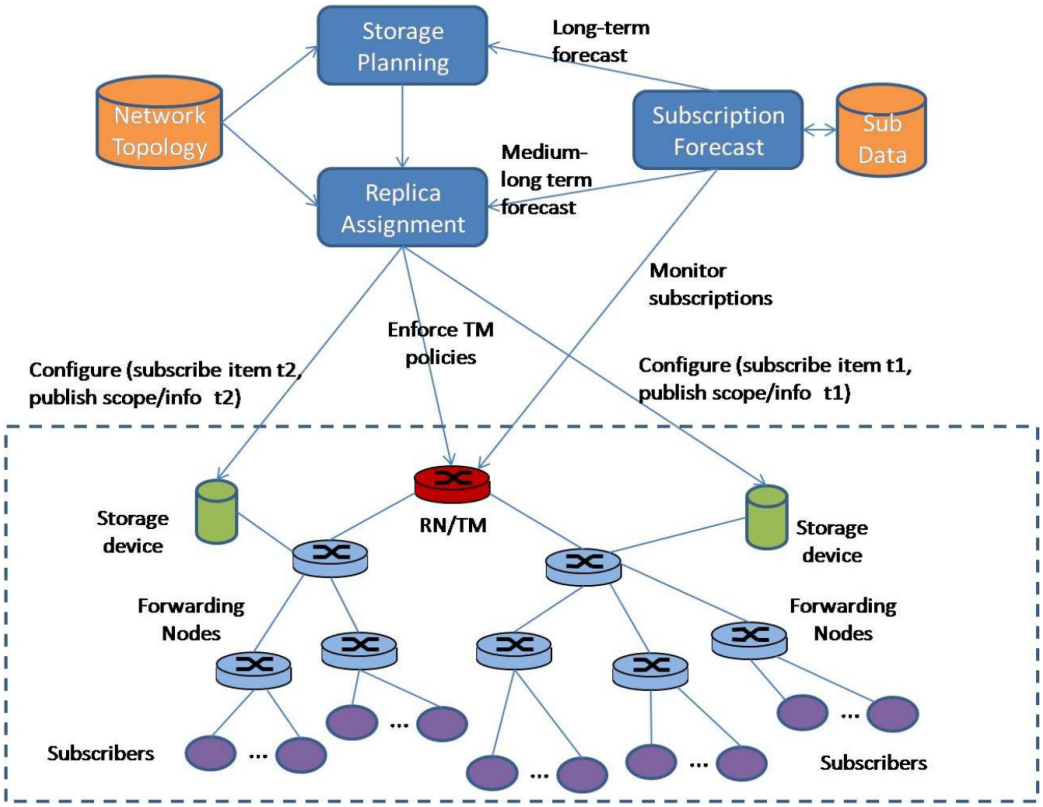

Figure 3.3: Storage and replica assignment architecture [43]

to the network, these are ordinary nodes that subscribe to, and republish the data they cache. A client that subscribes to data cached in a storage is serviced by such a storage node. However, the storage node has special algorithms [43] used for decision making on whether a node is suitable to store data or not. This is based on metrics such as distance from the storage device, the nodes popularity and the request rate of the data. In addition there is a storage planning component that is used to plan the distribution of the storage devices. This would for example be maintained by an ISP and provided as a service.

3.3.1 Blackadder Prototype

The Blackadder prototype [23] is an implementation of a single node in the ICN architecture defined by PURSUIT. It is currently developed to run on Linux based distributions. It has been tested on Debian and Ubuntu distri-

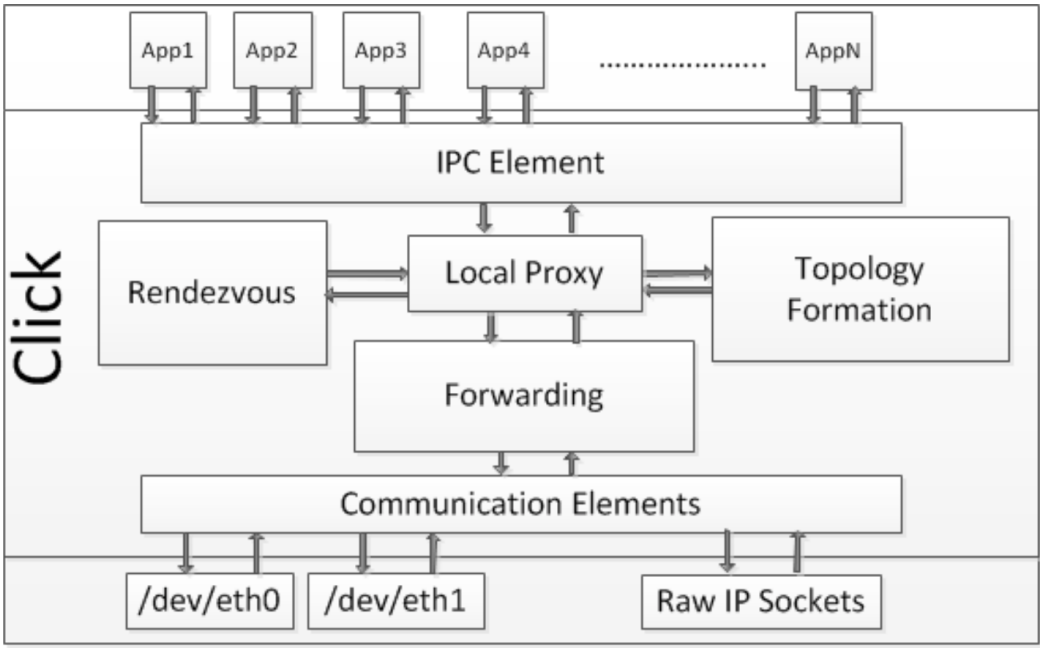

Figure 3.4: Blackadder node architecture [23]

butions but should in theory work on any distribution. Figure 3.4 shows the layered components of a single node.

All the Blackadder components are built as Click elements [24]. Click is an open source modular router developed for research purposes. The *IPC element* implements an inter-process communication mechanism using Netlink and TCP sockets that enables applications to communicate between each other while in the user-space. This element also interfaces with Blackadder which runs on kernel-space of the operating system. The *Communication Element* provides the interface between Blackadder and the transport layer. This implementation provides the use of Ethernet through creating Ethernet frames and sending them, or the use of raw IP sockets where IP packets are used. The *Local Proxy* is used to keep record of local publications within one node. The other components realize the core functions of the functional model as described in Section 3.3.

In deployment, a static network topology is defined in a configuration file. This file is deployed in every node that is part of the network. The topology is later translated into a graph in memory, using the igraph library [9], which is used to compute shortest paths between nodes. This takes place in the *Topology Formation* component that runs in every node. In the configuration,

one of the nodes is defined to handle the *Rendezvous* functions.

Blackadder exposes a set of APIs used by applications to publish and subscribe for information. Since the prototype has been developed in C++, the API defined is in C++ as well. There are however bindings developed that expose Java, python, ruby and C interfaces that can be used as well. The API has both synchronous and asynchronous functions. The asynchronous APIs have an "nb_" prefix. The API takes binary arrays generated from strings of data identifiers.

When publishing information, the applications have to publish scopes and information items using the APIs shown in Listing 3.1.

Listing 3.1: Publishing APIs

```
publish_scope(string &id, string &prefix_id, char strategy,
    char *LID)
publish_info(string &id, string &prefix_id, char strategy,
    char *LID)
```

"publish_scope" publishes a *scope identifier*(SID) into the network while "publish_info" publishes a *rendezvous identifier*(RID). Listing 3.2 shows examples of publications.

Listing 3.2: Example publications

```
publish_scope("0000", "", DOMAIN_LOCAL, NULL)
publish_scope("0001", "", DOMAIN_LOCAL, NULL)
publish_scope("1111", "0000", DOMAIN_LOCAL, NULL)
publish_scope("2222", "0000", DOMAIN_LOCAL, NULL)
publish_scope("3333", "00002222", DOMAIN_LOCAL, NULL)
publish_scope("000022223333", "0001", DOMAIN_LOCAL, NULL)
```

Scopes identifiers have to be published first before publishing any *rendezvous identifiers*. This is because every RID "is contained" in an SID. The above function calls would produce an information graph looking like Figure 3.5.

At this point the network only contains the information graph in Figure 3.5. The data still resides in the node. For data to be retrieved from the publisher into the network a subscriber has make a subscription. This can be done using the APIs shown in Listing 3.3.

Listing 3.3: Subscribing APIs

```
subscribe_scope(string &id, string &prefix_id, char strategy,
    char *LID)
```

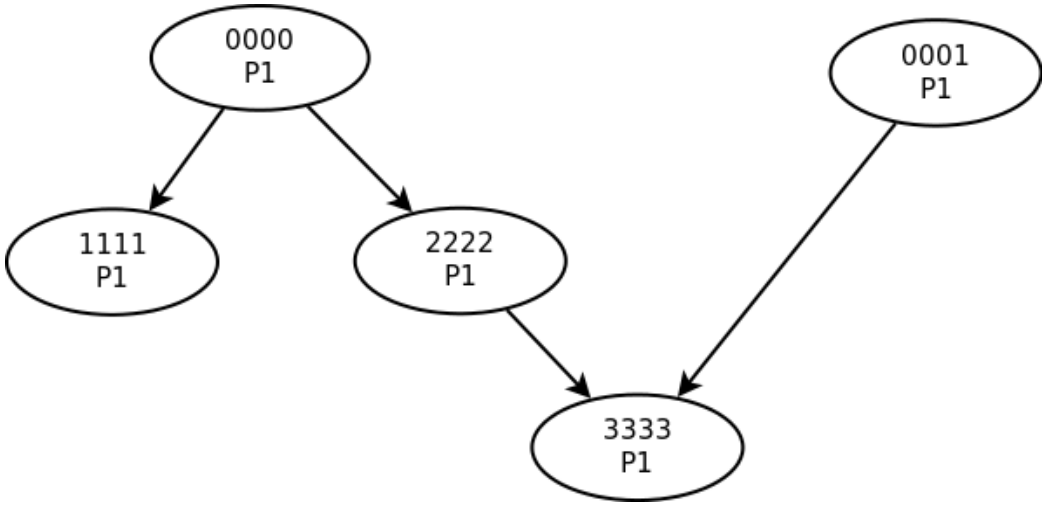

Figure 3.5: Information graph instance [23]

```
subscribe_info(string &id, string &prefix_id, char strategy,
    char *LID)
```

Once a node subscribes for data, Blackadder creates a `"Start Publish"` event which is sent to the publisher. The event contains the full path of the information graph item subscribed to i.e. `0000/2222/3333/RID`. The publisher handles this event by publishing the data using the API shown in Listing 3.4.

Listing 3.4: Publishing data APIs

```
publish_data(string &id, char strategy, char *FID, char *data
    , int data_len)
```

Unpublish and unsubscribe APIs exist as well. They reverse the publish and subscribe actions in the network. Other events generated by Blackadder are `"Stop Publish"` event which is received by a publisher once the last subscriber unsubscribes to an information item, `"Published Data"` event which is received by a subscriber once they receive data from Blackadder and `"Deleted Scope"` which is received by subscribers once a *scope identifier* is removed from the information graph.

Chapter 4

Experiment Set-up

This chapter will describe the experiment set-up. Section 4.1 will discuss the study design which will contain the process and justification of picking the application to port. Section 4.2 will describe the background of the Infinote protocol, the protocol used by the real-time collaboration software ported in this experiment.

4.1 Study Design

To study the effects of introducing ICN on application design, an experiment was performed where an existing application developed on top of TCP/IP, which is a good example of a HCN, was ported on the Blackadder ICN prototype. This experiment was done by a team of two members within the research group. Results of the experiment were collected through static analysis and dynamic testing. This thesis only covers the static aspects of the experiment where else the dynamic aspect is covered by Apurva Jaiswal in his *Efficiency Analysis of a Non-Trivial Application Ported Into ICN Paradigm* thesis. We expect to see structural changes of the application as well as a design approach in the static analysis results. These results are to be discussed in Chapter 6 to draw conclusions, and answer the questions stated in the research problem.

4.1.1 Application Choice

First, a suitable candidate had to be searched. We defined the requirements below that our candidate application needed to fulfil.

1. Open source application that is well documented and has a flexible license such as GNU General Public License which gives freedom for modification and republishing.
2. Implemented and heavily dependent on a HCN network that utilizes TCP/IP for example, as this will clearly show distinct differences once the changes are done.
3. Designed to communicate through connections between multiple nodes in a distributed system as this is the case in most Internet applications today.
4. Designed following a client/server model because most Internet applications today heavily follow this design meaning that findings in this research would most likely apply to most Internet use cases.
5. Programmed to run natively on a PC installed with a Linux distribution as opposed to running on a browser or mobile device because the Blackadder prototype is limited to Linux desktop environment.
6. Should have a relatively active development community for support purposes on issues specific to the application hereby reducing time spent on solving any blocking issues that may arise.
7. Should have a graphical user interface for demonstration purposes.

Search for candidates was done on various Internet search engines. Seven candidates were picked that fulfilled the above requirements. Of these candidates there were four collaborative editors and three multi-player games as shown in Figure 4.1.

To pick the most suitable candidate, further analysis was done on the candidates. This involved compiling, deploying and running the applications on the target environment. Aspects such as fewer external library dependencies, ease of deployment and ease of use were considered. Code complexity was also put into consideration. The most suitable candidate picked was the

Application Name	Language	Description	License	LoC	Document-ation	GUI
Gobby	C++/C, GTK+ 2.6 for GUI	Collaborative Document Editing Tool	GNU GPL	200,000	Very Good	YES
ACE	JAVA	Collaborative Document Editing Tool	GNU GPL	35,000	No active work	YES
Emacs	C, EMACS, LISP	Collaborative Document Editing Tool	GNU GPL	7,000	OK	YES
AbiWord	C++	Collaborative Document Editing Tool	GNU GPL	22,000	Good Document-ation	YES
Armagetr-onad Advanced	C/C++	Multiplayer Game	GNU GPL	13,000	Good	YES
PyScrabble	Python	Multiplayer Scrabble Game	LGPL MPL	3500	No active work	YES
Open Office	Multiple Language	Document Editing	GNU GPL	12,900,000	Very Good	YES

Figure 4.1: List of selected applications

collaborative editor called Gobby [12]. It was picked over the other applications as the code written was quite modular and was based on a well-defined collaborative protocol, Infinote [20].

This gave us an opportunity to focus our research on a collaborative protocol used by several applications [20] rather than a single collaborative editor. C/C++ programming language used was well known and preferred as well.

4.1.2 Development Environment

The development environment set up was based on version 0.3 of the Blackadder prototype [4] which was the latest at the time the experiment was performed, version 12.04 of Ubuntu (Linux) operating system, and version 0.5 of Libinfinity [17], the collaborative editing library used by Gobby editor. Source code analyses were done using ArgoUML [7] which is an application for producing UML diagrams, and Eclipse [33] which is an IDE. The programming language used was C and C++ for both the collaborative editor and the Blackadder prototype. The code was stored in Subversion revision control system hosted within the Pursuit project servers.

4.1.3 Test Environment

Initial testing of the application was done on local machines, that is, three PCs running on a custom Blackadder set up. Once the code was mature enough to have larger test runs, it was deployed in a larger Blackadder test bed [37]. The test bed is laid over IP through an open VPN [31] set-up which has over 25 virtual machine nodes spread in several European and US cities. Microsoft Excel spread sheet application was used to record readings from test experiments and produce analysis artefacts such as latency graphs. Since this thesis focuses on the static aspect of this experiment, the dynamic aspect which contains these artefacts will not be reported here.

4.2 Infinote Protocol

Infinote is a real-time text, open source, collaboration protocol defined for collaborative editors [20]. The protocol was defined as a venture to have a common way for having collaborative text editors implemented as opposed to having many different real-time collaboration protocols and implementations used within many applications as is the common case.

4.2.1 XML Messages Structure

The protocol is based on the client/server model where clients communicate with a server through XML messages. The protocol defines a form of file system created and maintained in the server. The file system contains a

directory structure that is used to organise files. Through certain type of messages, clients can request access to the directories within the server and make modifications. Clients will only have a view of the file system but the persistence of the file system lies in the server meaning that all the files created in the file system reside in the server's storage only.

The protocol defines a set of XML messages structured for sending and receiving real-time text. All messages have a root tag named `"group"` i.e.

```
<group name="NAME">...</group>
```

There are three types of messages exchanged in the lifetime of an application using this protocol namely *InfChat, InfDirectory* and *InfSession_N*, where *N* in this case refers to a positive number. *InfChat* messages are types of messages that contain instant messaging type of messages exchanged between clients. These messages can be used to enable users to communicate, through instant messaging, as they collaborate on a piece of document. In Gobby application, a chat window is used to show these messages and users can send instant messages between each other. These type of messages are formatted as shown below.

```
<group name="InfChat">...</group>
```

InfDirectory type of messages are used to communicate directory messages between the clients and the server. Clients can request to access the file system using an *InfDirectory* message like the one below. Here a client is asking to access the list of nodes (directories and files) within the directory identified by `id="0"`.

```
<group name="InfDirectory"><explore-node seq="0" id="0"/></
    group>
```

This message is handled by the server and the client can receive a success or failure message as shown below. Here the client receives two success messages from the server saying that there are two nodes in the file system. One is a directory *(InfSubDirectory)* named `"MyDoc"` and the other is a file *(InfText)*. The protocol gives flexibility to have more node types which could for example represent different file types.

```
<group name="InfDirectory"><add-node id="1" parent="0" name="
    MyDoc" type="InfText"/></group>
<group name="InfDirectory"><add-node id="2" parent="0" name="
    MyDir" type="InfSubdirectory"/>
```

Other requests such as adding a new directory, renaming a directory and removing a directory all together, follow the same request reply pattern.

InfSession_N type of messages are used to exchange messages between real-time collaborative sessions. Clients request the initial file from the server using a synchronisation request. In the case below the client requests for the file identified by `id="1"` which based on the messages above is the file named `"MyDoc"`.

```
<group name="InfDirectory"><subscribe-session id="1"/></group
    >
```

The client receives the message shown in Listing 4.1 from the server identifying what session number (2) this client would use to collaborate on the document, and the initial document content. From this point on the client would make requests targeting `"MyDoc"` using `"InfSession_2"` session provided by the server.

Listing 4.1: Message received from server

```
<group name="InfSession_2">
<sync-begin num-messages="4"/>
</group>
<group name="InfSession_2">
<sync-user id="1" name="John␣Doe" status="unavailable" time="
    " caret="0" selection="0" hue="0.75974399999999997"/>
</group>
<group name="InfSession_2">
<sync-user id="2" name="Jane␣Doe" status="unavailable" time="
    " caret="0" selection="0" hue="0.011233179294910934"/>
</group>
<group name="InfSession_2">
<sync-segment author="1">This␣is␣user␣1␣text␣and␣</sync-
    segment>
</group>
<group name="InfSession_2">
<sync-segment author="2">this␣is␣user␣2␣text</sync-segment>
</group>
<group name="InfSession_2">
<sync-end/>
</group>
```

During the collaboration all operations (delete, insert, add, undo etc.) requests and replies are wrapped within the *InfSession* tag as shown above.

Concurrency control and group undo operations during real-time collaboration are taken into consideration within the protocol. This is achieved by use of an algorithm developed specifically for real-time collaboration called adOPTed [39], [38]. Each client instance and the server as well, store a document state in memory once collaboration begins. The state of the document

is stored in the form of a state vector. Through the application of various transformation operations and distribution of concurrency identifiers [16], the consistency of the state vector is maintained.

4.2.2 Communication Structure

Infinote protocol has a modular design i.e. the protocol is not dependent on any specific type of network and the users of the protocol have no direct access or visibility to the transport layer(s) used by the protocol. This is achieved by the use of what is called the *communication group* [20]. A *communication group* contains a set of connections that use a particular transport method like TCP/IP to send and receive messages to each other. Each group instance handles a specified type of message corresponding to a group type i.e. *InfDirectory* type of messages would be handled by a group instance during runtime.

The method used by the *communication group* mentioned above is known as the *communication method* [20]. The protocol exposes *communication methods* to its users who in turn, use the methods to send and receive messages. Each method is identified based on what transport method it encapsulates i.e. TCP/IP, jabber etc. It is responsible for how messages are disseminated within the network. Example of different ways messages can be sent include broadcasting, uni-casting, multi-casting, peer-to-peer etc. Depending on the transport layer and the usage of the protocol, any of these dissemination methods could be used. A host creates and uses such a method to delegate sending and receiving of messages.

Since the *communication method* is used to implement any type of transport method, there are some properties that the protocol requires to be fulfilled. They are [20]:

- *Reliability: It needs to make sure that, when a host is sending a message, the message will arrive at the destination. If the underlying transport (such as TCP or UDP) does not guarantee reliability, then it needs to take care of that itself by acknowledging and resending messages if needed.*
- *Ordering: It needs to guarantee that messages arrive in the same order they have been sent. Again, if this is not already guaranteed by whatever transport the method uses it needs to make sure itself that messages are reported to the upper layer in order, for example, by adding sequence*

numbers to messages and holding back messages if an earlier sent message has not yet arrived.

- *Multi-casting: It needs to be able to send messages both to single group members to which a host has a direct connection and to all of the group. This is called the scope of a message, and it can be either point-to-point (to a single host only) or group (to all group members).*

With this pre-conditions, the layers above in the protocol have a lot of flexibility to generalize their behaviour.

4.2.3 Libinfinity Library

Libinfinity [17] is the library implementation of the infinote protocol written in C programming language and uses the GObject library [11]. It follows the client/server model in design just as the protocol defines. The library can be linked to any project implemented in C or C++. There are other library implementations [20] of this protocol that expose JavaScript, Python and Qt APIs.

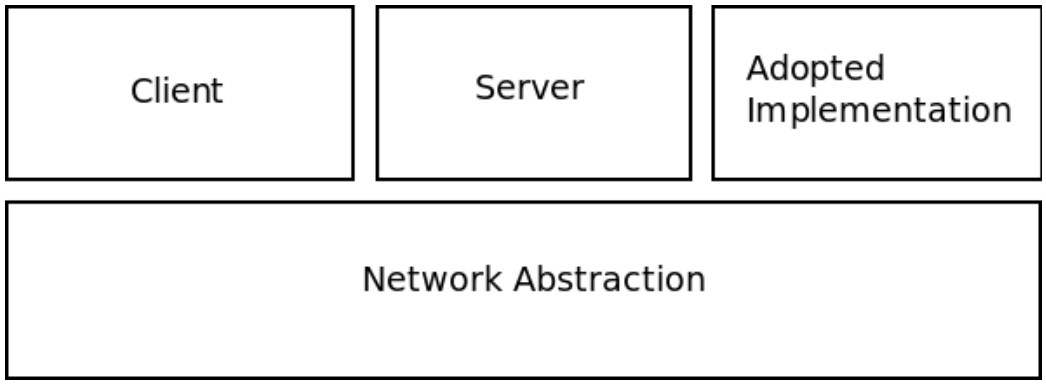

Figure 4.2: Layers of modules exposing APIs

The library has two layers of abstraction as shown in Figure 4.2 above. The top layer holds the APIs used by the clients and servers to access the document network document structure. The bottom layer abstracts the networking layer. This layer implements the dissemination strategies used by different network protocols as well as the classes that utilize the protocols. The rest of the section gives internal details through class diagrams of how the library is designed and implemented.

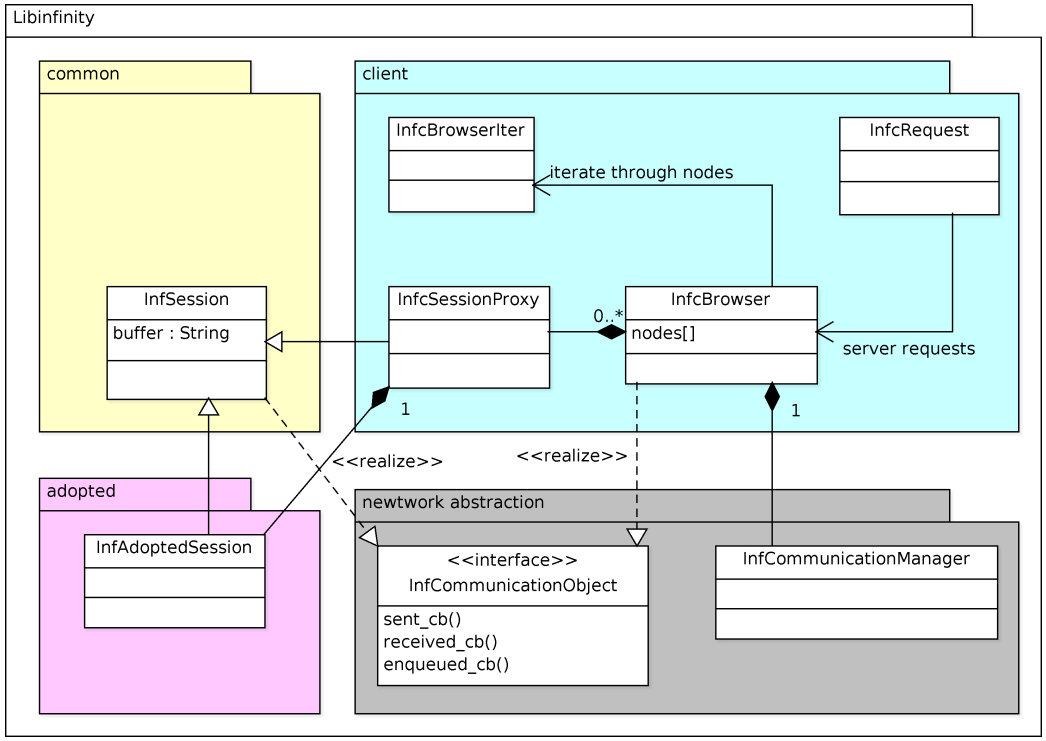

Figure 4.3: The class diagram of the client module

Figure 4.3 shows a minimalistic view of the client module and its dependencies. The module contains a set of APIs that are used by applications with a GUI which users can use. The services exposed by this API are browsing the file system and editing files. Clients use defined operations i.e. explore a folder, open a file etc. to communicate with the API through leaf classes that implement the *InfcRequest* base class. These operations are processed in the lower layers and in turn, sent to the server where they are processed. Once the operations are processed, the clients receive callbacks from the *InfcRequest* objects that initiated the operations.

The *InfcBrowser* and *InfcBrowserIter* are for accessing the remote directories. Operations such as adding, deleting, creating and editing of directories and files are handled here. The *InfSession* base class is a common class used by both the server and the client APIs to hold a collaboration session. Its purpose is to handle *InfSession_N* type of messages defined in the protocol. In the client module, the *InfcSessionProxy* leaf class implements this base class. It is used to request file contents from the server using an operation called *synchronizing*. *Synchronized* file contents are stored in the string buffer

property of the *InfSession* object. The *InfcSessionProxy* leaf class holds an instance of *InfAdoptedSession* leaf class. This class is used to handle all concurrency control and global undo operations of the adOPTed algorithm implemented in the adopted module.

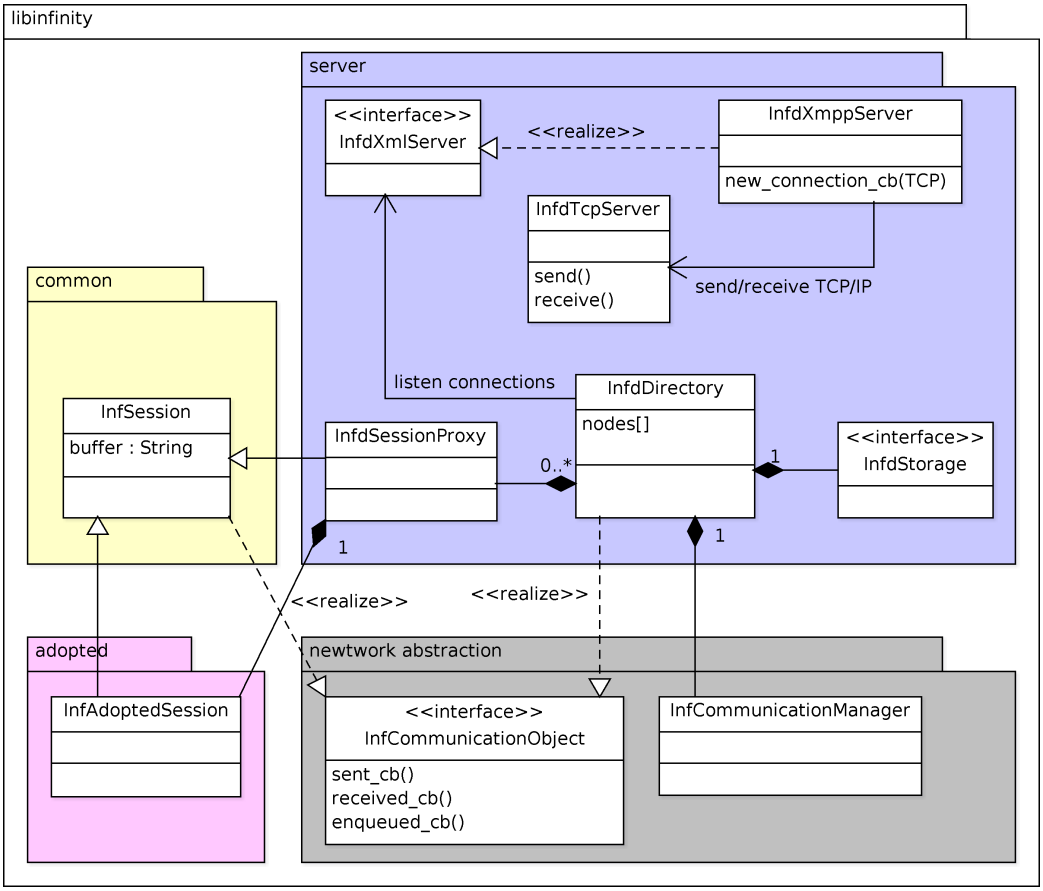

Figure 4.4: The class diagram of the server module

Figure 4.4 shows a minimalistic view of the classes the server module has and the dependencies as well. The server hosts the file system and stores all the files and directories used by clients. All clients communicate with the server in order to have access to the file system contents. *InfdStorage* interface is used to abstract the file system and exposes file system operations like delete, create, add and edit, to the server.

InfdDirectory class is used to manage directory requests coming from the clients. *InfdSessionProxy* class instances hold text collaboration sessions that are ongoing between the clients and the server. Each *InfdSessionProxy*

instance in the server corresponds to an open document being collaborated on. This means that if there are 5 clients collaborating on 2 documents, there will be 2 *InfdSessionProxy* instances for each document in the server.

InfdXmppServer, which realizes the *InfdXmlServer* is used to create new client connections to the server once such a request arrives. In this case the connections are TCP/IP based though this is not part of the protocol definition. *InfdTcpServer* class is used to listen to connection requests from the TCP/IP network and also send server reply messages.

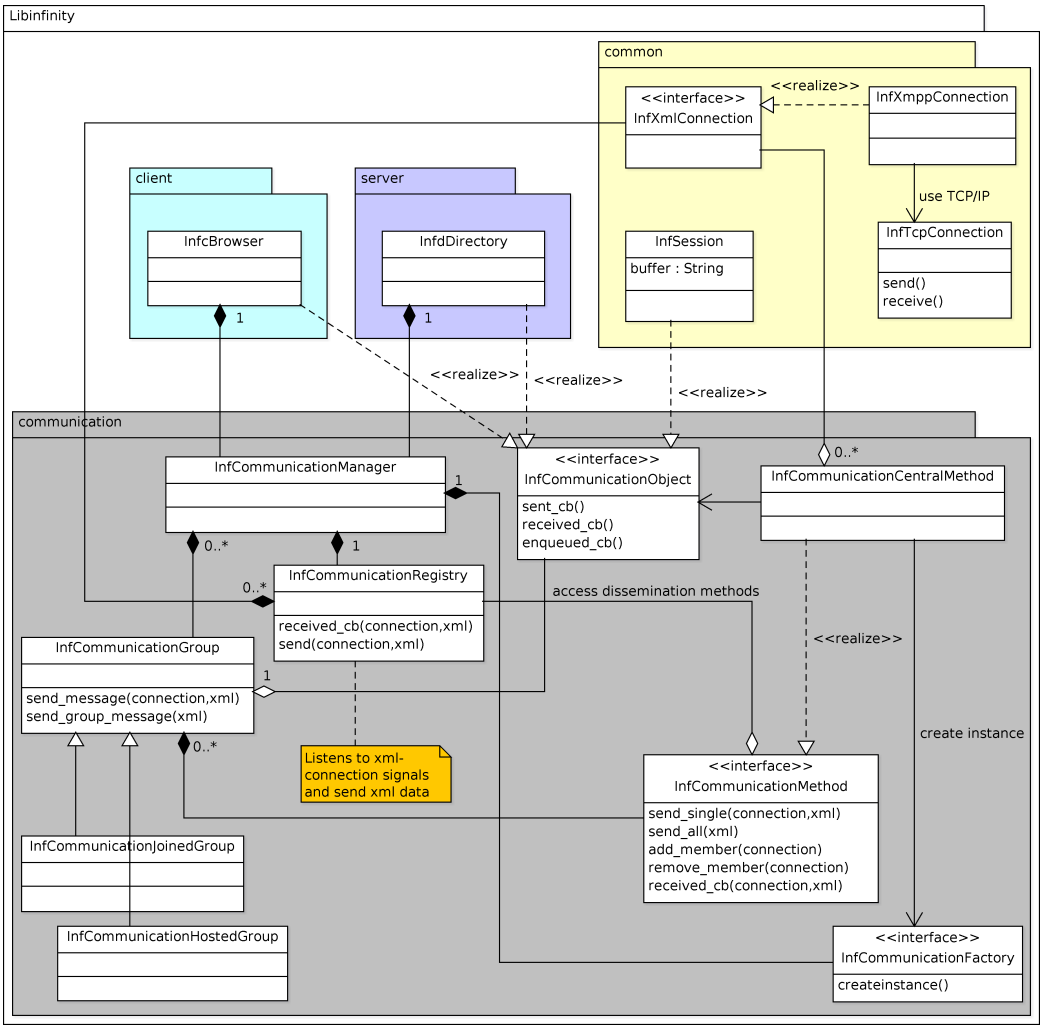

Figure 4.5: The communication module

The communication module in Figure 4.5 shows the interface between the transport layer and the application layers. The *InfCommunicationManager*

class is the main entry point of this module. Client and server APIs create an instance of this class and register themselves as listeners to events. This is made possible by the *InfCommunicationObject* interface that both *InfcBrowser* and *InfdDirectory* classes implement. *InfSession* base class implements this interface as well so as to receive events during collaboration sessions. This base class is used within the client and server modules.

On the lower levels, this module interfaces with the transport layers through the *InfXmlConnection* interface. This interface is used to implement concrete classes that have specific transport layer functionality i.e. TCP/IP. *InfTcpConnection* class is a TCP/IP implementation class which is used by *InfXmppConnection* class. *InfXmppConnection* class implements the *InfXmlConnection* interface and is used within the communications module. The class contains some XMPP logic which is used for maintaining user presence and providing users' personal information. The server has some additional classes i.e. *InfdXmlServer, InfdXmppServer* and *InfdTcpServer* which use these lower level classes. These extra classes are used to manage different client connections that are received through the transport layer. *InfdTcpServer* and *InfdXmppServer* are concrete classes that are used for TCP/IP and XMPP implementations in Libinfinity library, but are not part of the protocol specification.

An instance of *InfCommunicationRegistry* class is owned by the manager. This class orchestrates how the communication module classes work. When messages are sent from the higher levels, they end up in the registry class. Here the registry may send the messages to the network immediately or defer the operation. This logic involves different parameters such as whether an operation is a batch operation, number of messages pending within the queue or the so called scope (*group* or *point-to-point*) of a message as defined in the *multi-casting* properties of a communication method on the protocol definition.

InfCommunicationGroup class implements the *communication group* defined in the protocol. This is a base class having two leaf classes containing additional logic that is different for the server and client. Both classes store members within the same group as defined in the protocol.

InfCommunicationMethod interface creates the abstraction of the transport layer by defining a set of functions that generalize sending and receiving of data. Each concrete method class is required to fulfil the *communication method* properties defined in the protocol. Libinifinity library contains an *InfCommunicationCentralMethod* class that fulfils these properties for TCP/IP

transport protocol. The method is used by default for any transport layer implementation used within the library.

Chapter 5

Application Porting

This chapter will discuss the porting process of the Libinfinity library, the implementation of the Infinote protocol. Section 5.1 discusses how the naming scheme is designed and used to have the library requirements fit with the Blackadder service model. Section 5.2 shows how the design and source code of Libifinity changes to accommodate a server-less implementation. This is through detailing changes made to the XML data used to communicate between clients and through class diagrams detailing the modifications done to the library.

5.1 Data Naming in Libinfinity

The PURSUIT architecture defines a graph of flat labels for the naming scheme for the data identifiers. These identifiers correspond to `SID` and `RID` combinations in the information graph. In defining the naming scheme the mutability property of Blackadder's data space was considered. In this section the naming scheme will be built up in parts explaining the reasoning behind the decisions of defining each part. A forward slash is used as a delimiter between each part in the scheme.

5.1.1 Namespace

The information spaces created will contain data related to Gobby application. These spaces need to be differentiated from other sets of information spaces. For this reason, a namespace was defined for all data identifiers in

the Gobby application. Other applications using the library would have to define another namespace. The namespace identifier is the first part of the naming scheme.

```
"Gobby/..."
```

5.1.2 Decoupling identity from connections

The current implementation of Libinfinity uses TCP/IP to create connections between clients-clients and server-clients. These connections are used to identify nodes using addresses. The library builds on this implicit identity property to realize user identification. This identification is used in certain scenarios in the library. An example is when the server receives a connection request from a client, the IP address is used to uniquely identify the client that made the request. A connection is created and collaboration between the server and this client would be uniquely identified. If a client requests to synchronise a document from the server, it uses the servers IP address to identify the server. Communication takes place through message passing of XML data via the socket connection created between the server and the client. In this case the client handles the data received in a specific way as opposed to data received from a different socket. Another example is during collaboration of a document. All modifications made on the document come from different users running their own client instances. Identification of the users is built from the socket connections made between the clients. The IP addresses are unique in every client collaborating on the document i.e. for a document having five collaborators, there are at least five unique connections in each of the clients collaborating on the document. In ICN there is no concept of addresses and hence no connections between nodes in the network. To provide the identification property on ICN, a different approach had to be thought through. Since the only information the application provides to, and receives from the network is data and data identifiers, client identification could only be realized through these network pieces. This meant either embedding the user identifiers in the naming scheme or in the data. If the former was used the scheme would look like below.

```
"Gobby/<email_address>..."
```

This would mean that once a new user joins a collaboration, each user already joined in the network would have to be informed so that clients would subscribe to data identifiers published by this user. This would be unnecessary overhead because it would require having a mediator to publish this

information to all present clients. For these reasons the latter approach was used which involved the identity information being embedded in the data itself as shown using the "icn_id" attribute below.

```
<group name="InfDirectory" icn_id="first.last@mail.com">...</
    group>
```

5.1.3 Orthogonality

The data identifiers in the network are required to be independent from where and how the data is stored. This gives the library flexibility of how files are stored in the file system and reduces maintenance of data identifier to data mappings. In Libinfinity this would ensure that;

- If the location of the data storage changes, the original data identifiers would still point to the data after relocation.
- If the file names of the files would be changed in the storage without an explicit operation from the user, the original data identifiers would still point to the data after the change. An example of such a case would be when file names are shortened or are stored in a flat file system.
- If the storage medium would change, the original data identifiers would still point to the data after the change. An example of such a case would be for example when a hierarchical file system would be replaced with a relational database.
- If the ownership of data would change, the original data identifiers would still point to the data.

This implies that the data and the data identifiers require to be orthogonal [8]. According to the definition of the information structure of a data space, there are three basic elements (X, F, p) that form the data space.

> *X is a non-empty set of "objects"*
>
> *F is a non-empty set of "access paths" to the information kept in the objects*
>
> *p is a "rule" for transforming the objects*

Objects in Cremers' and Hibbard's [8] definition are represented by elements in the *"x"* axis in Figure 5.1. X therefore is the domain of the *"x"* axis.

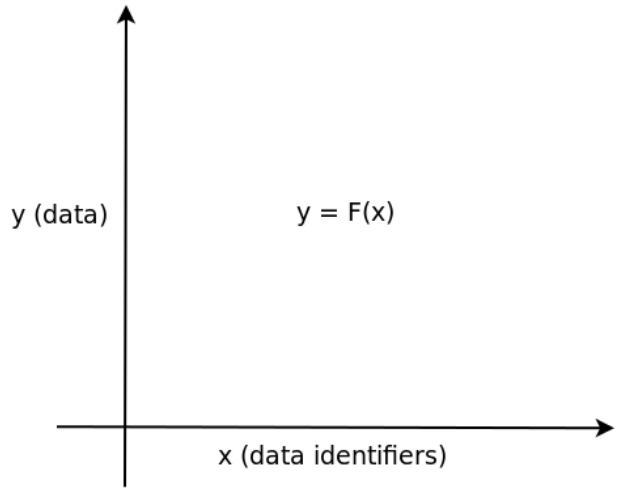

Figure 5.1: Graph representing the network data space

The *"y"* axis represents the information kept in the objects which in this case is the data pointed to by the data identifiers. F is represented by the two functions that are used to access the data using the data identifiers i.e. publish and subscribe. p is represented by the operations of the data i.e. create, read, update and delete. As we can seen, both the *"x"* axis and the *"y"* axis are independent and orthogonal i.e. if f is the publish function, g is the subscribe function and x and *x1* both are different names identifying named data in the information space, then;

```
f(x) != f(x1)
g(x) != g(x1)
f(x) != g(x1)
f(x) == f(x)
f(x) == g(x)
```

Orthogonality in this model only deals with names that point to data in the data space i.e. information items. Scope item names are not included in this domain even though they can be used to access data. They are considered as collections of data identifiers rather than data identifiers. The final naming scheme modelled looks like this.

```
"[NAMESPACE]/[ROOT_DIRECTORY_NAME]/..."
```

`NAMESPACE`: This is the namespace, defined above, which in this case will always be `"Gobby"`.

`ROOT_DIRECTORY_NAME`: This is a generated unique name within the

`"Gobby"` namespace. In the real world this name would be generated by the service provider owning the namespace as a way to govern their namespace.

The ellipsis at the end of the naming scheme is left to the clients to form the information structure they wish i.e. hierarchy of the directories and files. This can be as below.

```
"[NAMESPACE]/[ROOT_DIRECTORY_NAME]/Subdirectory1/File1"
"[NAMESPACE]/[ROOT_DIRECTORY_NAME]/Subdirectory2/
   Subdirectory1/File1"
```

A hybrid of hierarchical and self-certifying names [10] are a good match for instantiating this scheme. The principal `(P)` part would be used to represent the namespace and the root directory, while the rest of the name (ellipsis) would be the label `(L)` part. The label part would however have to be hierarchical in nature in order to match the information structure.

5.1.4 Using naming scheme in Blackadder

Figure 5.2 shows an instance of a directory created within the Blackadder information graph. Each scope contains a metadata information item which is used to store meta information of the directory or file referred to by the scope. As discussed earlier, each item (scope or information) is accessed through the formation of the absolute path i.e.

```
"Gobby/RootDir/Dir2/MyDoc/content"
```

The service provider provides the `"RootDir"` scope for the client to start using when forming a user's information graph.

The directory metadata information item contains directory properties defined by the clients. For the case of `"RootDir"` information item, the XML data pointed to looks like below.

```
<group name="InfDirectory" icn_id="...">
<subscribers count="2"/>
<node name="Dir1" type="InfSubdirectory" scope="Gobby/RootDir
   /Dir1"/>
<node name="Dir2" type="InfSubdirectory" scope="Gobby/RootDir
   /Dir2"/>
</group>
```

For the case of `"Dir2"` the XML data published and subscribed to by Gobby clients looks like below.

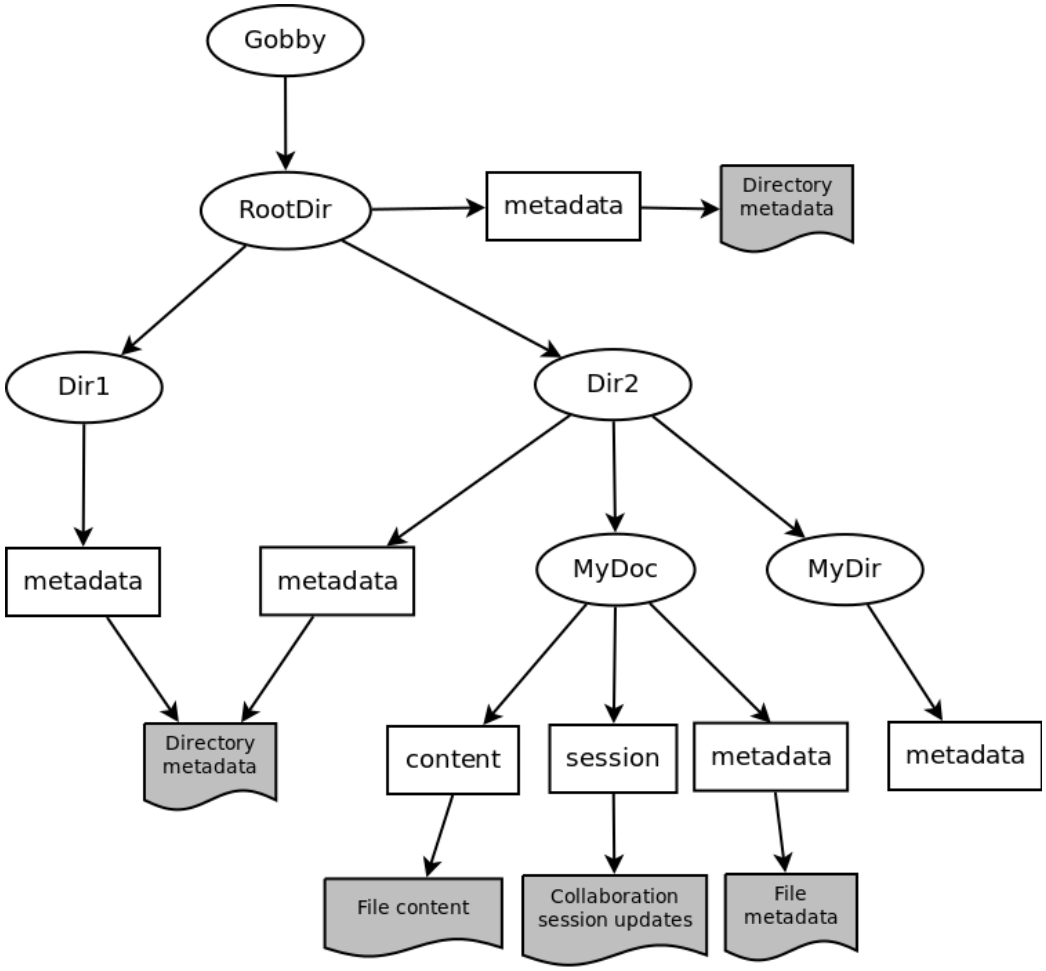

Figure 5.2: Graph representing the network data space

```
<group name="InfDirectory" icn_id="...">
<subscribers count="2"/>
<node name="MyDoc" type="InfText" scope="Gobby/RootDir/Dir2/
   MyDoc"/>
<node name="MyDir" type="InfSubdirectory" scope="Gobby/
   RootDir/Dir2/MyDir"/>
</group>
```

There are some modifications made to the original XML scheme as we shall see. The `"subscribers"` tag keeps count of how many clients have a directory open i.e. clients subscribed to the directory. Once a client subscribes to the directory it has to increase this value and republish the data and before unsubscribing from the scope, it has to decrease the count, republish the data

and unsubscribe to both the scope and the "metadata" information item. The "node" tag has been added replacing "add-node" and "remove-node" tags. "add-node" was initially used by the server to inform the clients to add a new node when a client first connects to it, or make additional changes to an existing connection while "remove-node" was used to inform the server to delete a node from its file system [16]. These operations will now be handled by the "node" tag by either adding or removing the tag to the XML data and publishing. If for example the user would like to delete "MyDir" directory from the information graph in Figure 5.2 above, the client would publish the following updated XML data after having removed the "node" tag containing "MyDir".

```
<group name="InfDirectory" icn_id="...">
<subscribers count="2"/>
<node name="MyDoc" type="InfText" scope="Gobby/RootDir/Dir2/
   MyDoc"/>
</group>
```

The publication would be made to the "metadata" information item as this XML data is metadata to "Dir2" parent directory. The metadata name of the item is formed by concatenating the scope name with "metadata".

```
"Gobby/RootDir/Dir2/metadata"
```

Prior to deleting the directory, the client must first try to unpublish both the scope ("Gobby/RootDir/Dir2/MyDir") and the "metadata" information item ("Gobby/RootDir/Dir2/MyDir/metadata") used to access "MyDir". The client has to verify that the directory is empty i.e. there are no extra nodes in the metadata XML. If it is not empty those nodes would have to be removed first and the recursion would continue. The "count" attribute would have to be 0 as well for the client to unpublish.

Scopes representing files have two other information items. The "content" information item is used to store the file content while the "session" information item is used for updates to the file during a collaboration session. To start editing a file, the client performs three steps. The first one is subscribing to the "metadata" information item in order to update the subscribers information.

```
<group name="InfDirectory" icn_id="...">
<subscribers count="2"/>
</group>
```

Next the client would have to subscribe to the ”content” item to receive the latest file contents. This XML data looks like below.

```
<group name="InfDirectory" icn_id="...">
<sync-begin num-messages="4"/>
<sync-user id="1" name="John␣Doe" status="unavailable" time="
    " caret="0" selection="0" hue="0.75974399999999997"/>
<sync-user id="2" name="Jane␣Doe" status="unavailable" time="
    " caret="0" selection="0" hue="0.011233179294910934"/>
<sync-segment author="1">This␣is␣user␣1␣text␣and␣</sync-
    segment>
<sync-segment author="2">this␣is␣user␣2␣text</sync-segment>
<sync-end/>
</group>
```

It can be seen that the content of the "name" attribute has changed from *"InfSession_2"* to *"InfDirectory"*. This is because the information containing the number of sessions open on the server is no longer maintained. This is also handled as a directory operation *("InfDirectory")* rather than a collaboration on the document *("InfSession_N")* as it was earlier. After receiving the file the client subscribes to the "session" information item. Here the client receives a delta of all the updates of the file that were published before the latest file was persisted. These changes look like below.

```
<group name="InfSession" icn_id="...">
<request user="1" time=""><insert-caret pos="43">␣</insert-
    caret></request>
<request user="1" time="1:1"><insert-caret pos="44">t</insert
    -caret></request>
<request user="1" time="1:2"><insert-caret pos="45">e</insert
    -caret></request>
<request user="1" time="1:3"><insert-caret pos="46">s</insert
    -caret></request>
<request user="1" time="1:4"><insert-caret pos="47">t</insert
    -caret></request>
<request user="1" time="1:5"><insert-caret pos="48">s</insert
    -caret></request>
<request user="1" time=""><move caret="48" selection="0"/></
    request>
<request user="1" time=""><delete-caret pos="48" len="1"/></
    request>
</group>
```

Again the "name" tag has changed for the same reasons explained above. The "request" tag is now used instead of the original "sync-request" because this data is no longer receive after a synchronisation request from the server, rather it is a cached list of current session requests. This initial "last updates" data received from any of the publishers currently making modifications on the open session is possible because the publishers receive

`"START_PUBLISH"` Blackadder event once a subscriber subscribes to an information item. After processing the initial XML data that client is ready to receive more changes from publishers or publish changes of the file.

Just before a client leaves from a session, it performs two operations. It first publishes the latest file to the `"content"` information item. This ensures that once collaboration is done the file is up to date in the information space. It then publishes the unsubscribe message shown below.

```
<group name="InfSession" icn_id="..."><session-unsubscribe/><
   /group>
```

At this point other clients collaborating clear out their cached session requests older than the unsubscribe message received because the collaboration changes that were done were persisted by the client that left. Any new session requests are stored in the now empty cache as usual.

Concurrency control is a concern as there is no server. All collaboration publications are handled by the adOPTed algorithm, but the directory updates and persistence of the files are not handled. This will be looked at in Section 5.2.2.

5.2 Server-less Implementation

Since in ICN there is no concept of addresses, and nodes have no direct access to other nodes for communication, the idea of a server disappears in this implementation. All that clients do is either publish data to the network or subscribe to receive data from the network. Through these publications and subscriptions, clients create an information graph which resides in the network.

Infinote protocol does not enforce a request-reply type of communication in its definition of server/client model [19], instead, communication is based on the *communication method* implemented. In Libinfinity library, the *"central" communication method* has been implemented. It follows a request-reply type of communication as the dissemination strategy. All messages from the clients are sent to a dedicated server, inifinoted [18], which in turn replies to the clients. Request-reply however only applies to *"InfDirectory"* type of messages where else, *"InfSession"* and *"InfChat"* messages are handled differently in the communication method. Removal of the dedicated server that enables a request-reply dissemination strategy brings new challenges when

reimplementing this library. Below are the functions the server provides, which will have to be realized in a server-less environment.

1. **Persistence:** In the current implementation of Libinfinity, the server provides the storage for all the files and directories. The hierarchical structure of the nodes is also maintained persistently by the server even when none of the clients are connected.

2. **Serialization of *InfDirectory* messages:** *InfDirectory* messages are used to send and receive directory messages meant to access the contents in the directory structure. Serialization of these messages is needed to avoid conflicts of operations on the directory structure and its contents, as each request is handled one at a time in order of arrival. This keeps the directory hierarchy consistent across all the clients and the server.

5.2.1 Persistence

In Blackadder, caching is provided by the network as shown in Figure 3.3. The data accessed through the information graph created by the client is cached by storage nodes within the network. However, due to violation of a network design principle known as the *end to end argument* [40], and another design philosophy known as *fate-sharing* [5], such a means of storage cannot be relied on to replace the persistence function of the server. There are two ways this function can be realised. According to the PURSUIT architecture the responsibility of the storage devices is held by an ISP or any organisation responsible for planning the network. If the ISP could provide storage services that could be offered through the storage devices in the network, such services would be used for storing the data. The second way would be to have users, or providers of the Gobby application, set up strategic nodes in the network that would work as storage devices. These nodes would store publications and republish them to clients closest to them. Using either of these two ways would not violate the *end to end argument* because the application would have control over the service, and ultimately the responsibility of the data stored. The *fate-sharing* philosophy would be maintained as well because both the clients and the storage services are controlled by the same party.

In both of these implementations, some intelligence had to be added to these storage nodes. Not all publications are meant to be stored i.e. *"InfSession"*

and *"InfChat"* should not be persisted. Due to this, a TTL attribute had be added to the publication as shown below.

```
<group ttl="forever|day|week|time"></group>
```

The TTL attribute defines how long a publications should be persisted. By default none of the publications are persisted. Defining a `"forever"` TTL value informs the storage node to persist that file permanently unless explicitly deleted by a client. This value is used in all the *"InfDirectory"* messages.

```
<group ttl="forever" name="InfDirectory" icn_id="...">
<sync-begin num-messages="4"/>
<sync-user id="1" name="John␣Doe" status="unavailable" time="
   " caret="0" selection="0" hue="0.75974399999999997"/>
<sync-user id="2" name="Jane␣Doe" status="unavailable" time="
   " caret="0" selection="0" hue="0.011233179294910934"/>
<sync-segment author="1">This␣is␣user␣1␣text␣and␣</sync-
   segment>
<sync-segment author="2">this␣is␣user␣2␣text</sync-segment>
<sync-end/>
</group>
```

The storage devices are not restricted to how they store the data. It could be stored as plain text files or in a database. The device could also opt not to store the data if there is not disk space for example. However, logic to verify successful storage of files that need to be persisted is needed in the library. An additional step to verify storage of the data is made by subscribing for the same data. If the data does not exist in the network the operation performed be the client is reverted.

5.2.2 Concurrency Control

The serialization function ensures that the directory state stays intact. Without the server concurrent operations can cause the distributed files to have different states. Take for example a client A that is a subscriber and publisher to a directory named `"Assignments"` and at the same time, client B is a publisher and subscriber of the same. If both client A and client B publish an operation to rename the directory i.e.

Listing 5.1: From client A

```
<group name="InfDirectory" icn_id="...">
<subscribers count="2"/>
<node name="Assigments␣2013" type="InfSubdirectory" scope="
   Gobby/RootDir/Dir2/MyDir"/>
```

```
</group>
```

Listing 5.2: From client B

```
<group name="InfDirectory" icn_id="...">
<subscribers count="2"/>
<node name="School␣Assigments" type="InfSubdirectory" scope="
   Gobby/RootDir/Dir2/MyDir"/>
</group>
```

there would be inconsistencies in the directory name if both messages reach the clients at the same time. Another example would be concurrent updates by several clients of the metadata XML data. Since these data is used heavily to maintain the state of the directory structure, having inconsistencies here is of great risk. Files would easily be left in the network never to be accessible if for example a `"node"` tag defining the data identifier of a file is deleted as a data conflict consequence. Concurrency control of these messages has to be added on the clients to avoid these potentially fatal state inconsistencies.

5.2.3 Source Code Changes

Figure 5.3 shows a class diagram that shows changes implemented in Libinfinity. The parts in red are the changes we introduced.

A new *InfCommunicationIcnMethod* class has been introduced to abstract ICN as a communication method. Infinote protocol has defined three properties that this class has to fulfil.

1. Reliability: It is claimed that ICN improves reliability [21], [26] of data delivery. This is due to the inherent support of in-network caching that reduces changes of data loss. However, this cannot be proved in Blackadder's case because the test bed [37] that was used to test this implementation had Blackadder deployed on a TCP/IP network which is a reliable network [15].

2. Multi-casting: This is inherent in ICN because publications are by default disseminated by multi-casting [26], [43], [21].

3. Ordering: Just like reliability, ordering is supported in Blackadder because it is deployed on top of TCP/IP which enforces ordering of packets [15].

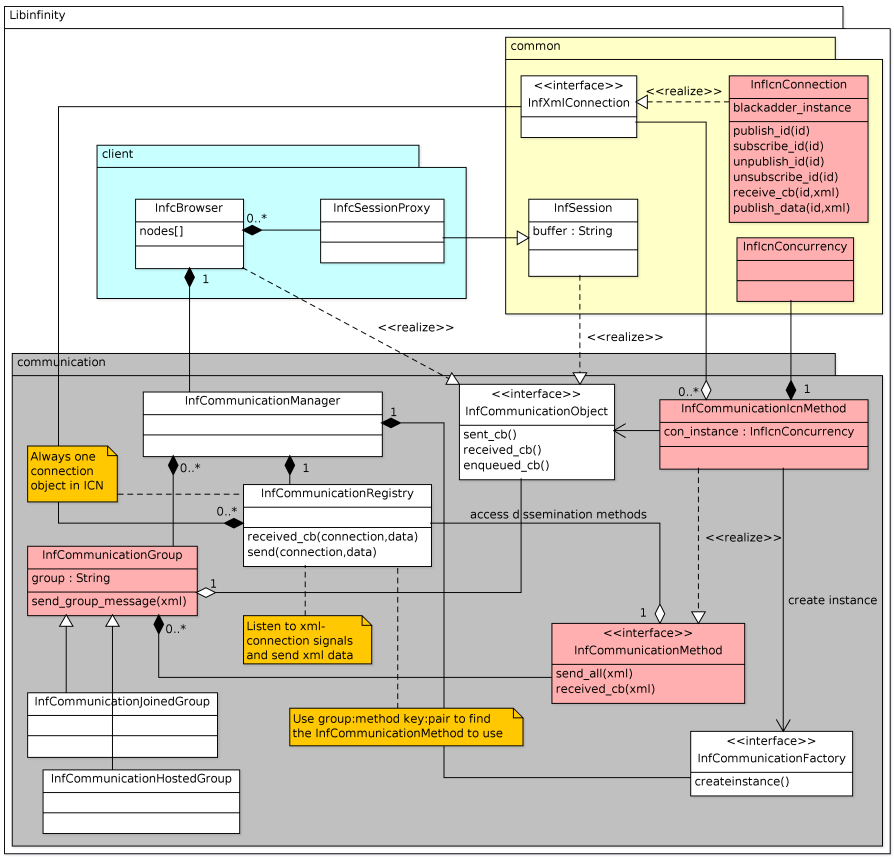

Figure 5.3: Blackadder implementation in the communication module

In addition to these properties, concurrency control as discussed above had to be supported by this method. A new concurrency control class, *InfIcn-Concurrency*, was added in Libinfinity as shown in Figure 5.3. An instance of this class is used in *InfCommunicationIcnMethod* to handle concurrency control issues. However, the real implementation of the class was not done as concurrency control in ICN as a whole was thought to be a separate research effort. This class is just a place holder.

Modifications were made in *InfCommunicationGroup, InfCommunication-Manager* and *InfCommunicationMethod* classes to remove the dependency of *InfXmlConnection* objects used for user identification purposes. In place of this, user information passed from the client module to *InfCommunication-*

Manager was embedded into the XML messages as discussed in Section 5.1.4. Dissemination methods used in these classes are uni-cast in nature i.e. no send or receive operation targets a single node in the network.

In *InfCommunicationRegistry* only one instance of *InfXmlConnection* class exists. This is because all the sending (publishing data) and receiving (receiving subscribed data) is handled by one *InfXmlConnection* object which wraps a Blackadder class instance in the *InfIcnConnection* concrete class implementation.

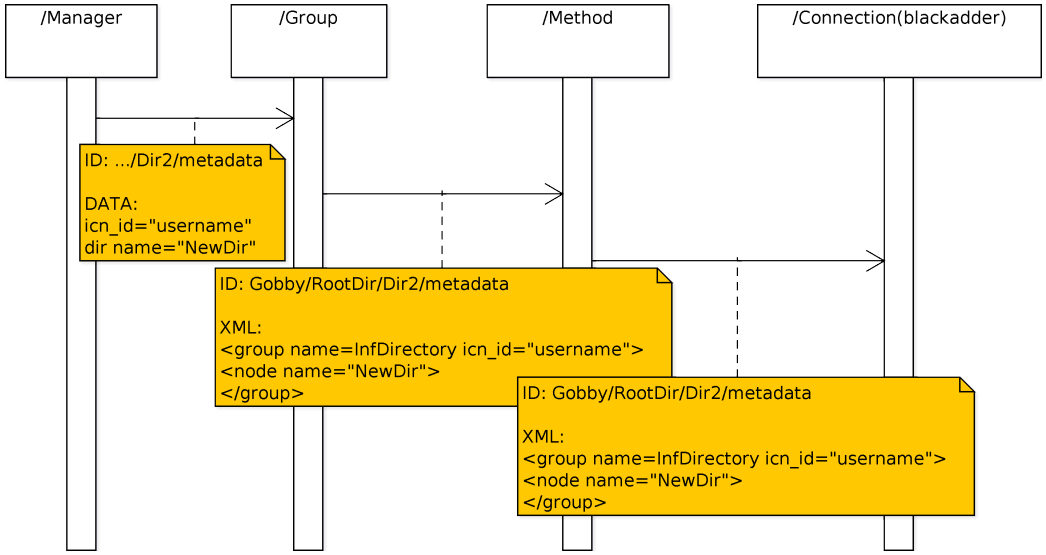

Figure 5.4: Sequence of a directory name update

When publishing, the data to be published is received from the client module through the *InfCommunicationManager* class. The data identifier is formed and together with the data passed down to the *InfCommunucationGroup* classes as shown in Figure 5.4. The group classes create the XML data that is then passed down to the *InfCommunicationIcnMethod* class. Here resolution of other publications that may have been received concurrently is handled and then the message is published through the *InfIcnConnection* class.

In the subscription case, both the data identifier and the type of operation are passed down to the *InfCommunicationManager* class as shown above. Based on the operation, one of the *InfCommunicationGroup* classes, which handle *InfSession, InfDirectory* and *InfChat* groups of messages, receives the data identifier. In the case above, opening of a file means subscribing to receive

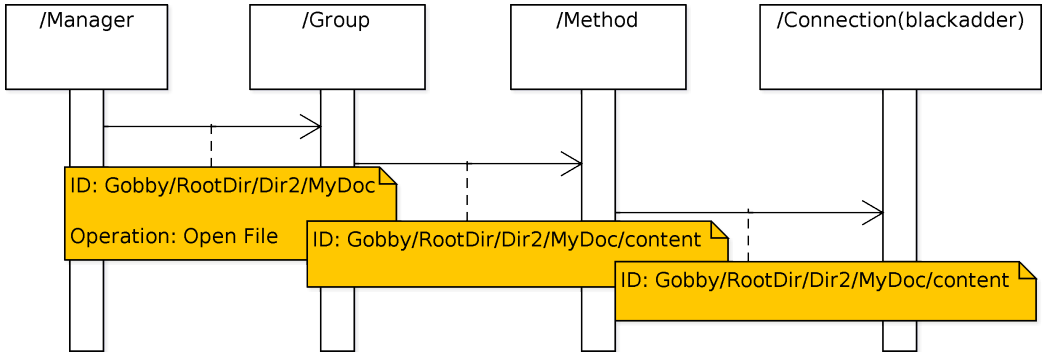

Figure 5.5: Sequence of an operation to open a file

the initial file content. The group class appends the `"content"` suffix to the data identifier to form the full identifier pointing to the latest file content. This is in turn passed down to the *InfCommunincationIcnMethod* class and then further down to the *InfIcnConnection* class where subscription for data is made.

Data identifiers provided to the Blackadder API are required to be of a specific length. This is defined in a configuration value compiled with the application. In this porting exercise the length value used was 8 bytes. This value is not enough to hold the possibly longer data identifiers. For this reason FNV hashing function [30] was introduced to generate statistically unique hash values of fixed length of the data identifiers. This function was chosen mainly because of its speed in generating hash values. This functionality was added in the *InfIcnCommunication* class implementation.

Chapter 6

Discussion

This chapter will discuss the findings of this porting exercise. In Section 6.1 we will analyse the increased complexity of Libinfinity library and discuss the reasons for the increase. We will validate the claimed improvement in impedance match in the PURSUIT architecture [44] and try to generalize these findings in Section 6.2. We will also discuss the challenges faced when conducting this experiment.

6.1 Improved Impedance Match

The server-less implementation of this application requires a hierarchical representation of data to clients realized, which the network provides through the information structure defined in the PURSUIT architecture [44]. The naming scheme we have defined maps the directory and file concepts into scopes and each concept contains related information items that are used to point to the data stored in the network. Directories contain subdirectories and files which are represented by scopes and their sub-scopes. A client that requires services such as getting all the information stored in one directory, which in the current Internet architecture is provided through traditional file systems hosted by servers, would only need to subscribe to the directory's scope. This architecture improves the impedance match between applications and the services provided by the network because access to the information structure earlier provided by middle ware such as relational databases, file systems or server applications is now provided directly by the network. This makes it easier to map information concepts between the applications and the network as opposed to using application level protocols such as HTTP

Metric	Original Libinfinity	Ported Libinfinity
Lines of Code	73,891	74,283
Cyclomatic Complexity	78	85

Table 6.1: Complexity metrics of Libinfinity library

and Tabular Data Stream [42].

The fact that the network requires the location of the data and the means to deliver it, rather than the data identification that an application needs, has been claimed to be an impedance mismatch [44], [47]. This experiment shows an improvement in impedance match because data is now accessible through data identifiers. Gobby applications neither have to provide the location, nor the means of delivery, to the network to request for data from the directory structure. All that is required is a data identifier in the form of a name formed from the defined naming scheme, as opposed to the IP addresses used in the original design of Libinfinity.

6.2 Code Complexity in Libinfinity

Applications such as Gobby, are required to maintain state in a distributed environment and hence follow a client/server model where the server facilitates in maintaining the state. This is however not possible in an ICN environment due to the inherent property of not addressing network nodes like the server. This adds some complexity in porting such an application to an ICN environment because new challenges such as concurrency control, and in general maintaining a distributed state in a server-less environment, have to be considered. Client applications would require to use additional concurrency control algorithms when accessing information from the network hereby increasing implementation complexity. There are efforts to include concurrency control algorithms in ICN prototypes such as the CCNx synchronisation library [32] where clients can register data that has state and the network would keep the state up to date once changes are detected.

Data published and subscribed to increases complexity of Libinfinity as well. In this porting exercise, identity was embedded in the data published and subscribed to by clients. This requires, not only changes in Libinfinity library, but changes in the protocol as well. The increase in code, and hence complexity, is apparent from the Lines of Code complexity metric in Table 5,

and the new classes that were added in Figure 4.8. Metadata is published to maintain consistency in the information graph i.e. number of subscribers of a file or directory is maintained in the file's or directory's metadata. This extra effort of maintaining state brought to the client applications introduces some level of complexity as well. Both of these changes contribute to the increase in cyclomatic complexity as shown in Table 5.

This protocol is quite modular compared to the comparisons we made in the application selection process, but it still includes some aspects that follow a HCN paradigm. A generalization could easily be made from this claiming that many application designs, even if network agnostic, are approached with a HCN design mind-set. This will bring difficulties in either porting, or designing such applications from scratch. The mind-set needs to change to accommodate a different design approach. However, it still remains to be seen to which direction application design will head. Should we focus application design around data or endpoints with identifiers? How well will the change in mind-set be embraced and adopted?

Chapter 7

Conclusion

The aim of this research work was to validate the claims of an improved impedance of application requirements and the network service offered by ICN. This was to be done through porting a non-trivial application on top of ICN. The results of the porting exercise were also to be used to draw conclusions on the impact of ICN on application development. This was to be based on the difficulty of porting an application, significance of structural changes introduced in the application design and the delta between implementations (ICN based implementation and HCN implementation).

A suitable application to port on ICN was selected which was a real-time collaborative editor based on a protocol called Infinote. Modifications on Libinifinity, the infinote protocol implementation, were made to have the library support ICN. A naming scheme was designed to fulfil both the requirements defined by the protocol and those defined by the PURSUIT architecture. This naming scheme was implemented into Libinfinity library to aid clients to create information graphs in the network as a form of directory and file management. Other major structural changes made in the library were removing the server. This was due to the fact that, ICN being a host agnostic networking paradigm, the concept of a server could not be realized as clients using the library had no host to connect to as a server. This change introduced a new set of challenges related to concurrency control. This was deemed to be a separate research topic but was acknowledged as a necessity in having the library fully functional. When recording measurements, test use cases that would not lead to state inconsistencies were selected.

There were challenges during the porting process, and the research exercise as a whole. First of all, the time allocated by the research project was

not sufficient to select a larger scope in the research, and to implement all the changes into the application. To be specific, concurrency control was a problem not tackled. The field of study was also fairly new for both of the researchers involved in the project. This meant that some effort was spared for learning about the research field, instead of focusing on the research problem. Another challenge that was experienced was with the delta between the Blackadder prototype and the PURSUIT architectural definition. The prototype is an ongoing project and at the time we forked a version to work with, not all features were present. Just to highlight one, the lack of in-network caching was missing and this would have generated results related to availability and latency worth analysing.

This thesis motivates future research work in concurrency control. There has been some effort [32] on this problem but as it can be seen in the research results, this will be an impediment in main stream applications as well.

The source code changes made in this Libinfinity library were contributed back to the open source community through the Blackadder project [4].

Bibliography

[1] Bengt Ahlgren, Borje Ohlman, Erik Axelsson, and Lars Brown. Subversion over OpenNetInf and CCNx. In *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*, pages 1056–1063, 2011. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6115163`.

[2] Akamai. Akamai homepage. http://www.akamai.com/, 2012. URL `http://www.akamai.com/`.

[3] BitTorrent. BitTorrent - delivering the world's content. http://www.bittorrent.com/, 2013. URL `http://www.bittorrent.com/`.

[4] Blackadder. Blackadder in GitHub. https://github.com/fp7-pursuit/blackadder, 2012. URL `https://github.com/fp7-pursuit/blackadder`.

[5] David Clark. The design philosophy of the DARPA internet protocols. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 106–114, 1988. URL `http://dl.acm.org/citation.cfm?id=52336`.

[6] Bram Cohen. The BitTorrent protocol specification. http://bittorrent.org/beps/bep_0003.html, 2008. URL `http://bittorrent.org/beps/bep_0003.html`.

[7] CollabNet. ArgoUML UML modelling tool by CollabNet. http://argouml.tigris.org/, 2009. URL `http://argouml.tigris.org/`.

[8] Armin B. Cremers and Thomas N. Hibbard. Orthogonality of information structures. *Acta Informatica*, 9(3):243–261, 1978. URL `http://www.springerlink.com/index/J785V1636N365682.pdf`.

[9] Gabor Csardi and Tamas Nepusz. The igraph library for complex network research. http://igraph.sourceforge.net/, 2012. URL `http://igraph.sourceforge.net/`.

[10] Ali Ghodsi, Teemu Koponen, Jarno Rajahalme, Pasi Sarolahti, and Scott Shenker. Naming in content-oriented architectures. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, pages 1–6, 2011. URL `http://dl.acm.org/citation.cfm?id=2018586`.

[11] GNOME. GObject reference manual. http://developer.gnome.org/gobject/stable/, 2012. URL `http://developer.gnome.org/gobject/stable/`.

[12] gobby.0x539.de. Gobby collaborative editor. http://gobby.0x539.de/trac/, 2012. URL `http://gobby.0x539.de/trac/`.

[13] Shirley Gregor and David Jones. The anatomy of a design theory. *Journal of the Association for Information Systems*, 8(5):312–335, 2007. URL `http://www.layrib.com/dl/gregor_07_design_theory.pdf`.

[14] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1): 75–105, 2004. URL `http://dl.acm.org/citation.cfm?id=2017217`.

[15] IETF. RFC 793 - transmission control protocol. http://tools.ietf.org/html/rfc793, 1981. URL `http://tools.ietf.org/html/rfc793`.

[16] infinote.0x539.de. Infinote/Protocol gobby. http://gobby.0x539.de/trac/wiki/Infinote/Protocol, 2011. URL `http://gobby.0x539.de/trac/wiki/Infinote/Protocol`.

[17] infinote.0x539.de. Libinfinity v0.5 reference manual. http://infinote.0x539.de/libinfinity/API/libinfinity/, 2012. URL `http://infinote.0x539.de/libinfinity/API/libinfinity/`.

[18] infinote.org. Infinote/Infinoted gobby. http://gobby.0x539.de/trac/wiki/Infinote/Infinoted, 2011. URL `http://gobby.0x539.de/trac/wiki/Infinote/Infinoted`.

[19] infinote.org. Infinote directory. http://infinote.org/protocol/directory/, 2011. URL `http://infinote.org/protocol/directory/`.

[20] infinote.org. Infinote protocol definition. http://infinote.org/, 2011. URL `http://infinote.org/`.

[21] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12, 2009. URL `http://dl.acm.org/citation.cfm?id=1658941`.

[22] Konstantinos Katsaros, George Xylomenos, and George C. Polyzos. MultiCache: an incrementally deployable overlay architecture for information-centric networking. In *INFOCOM IEEE Conference on Computer Communications Workshops, 2010*, pages 1–5, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5466639`.

[23] Jimmy Kjallman, George Parisis, Dimitris Syrivelis, Borislava Gajic, Martin Reed, Christos Tsilopoulos, and Charilaos Stais. First lifecycle prototype implementation. Technical report, PURSUIT, 2011. URL `http://fp7pursuit.ipower.com/PursuitWeb/wp-content/uploads/2011/09/INFSO-ICT-257217_PURSUIT_D3.2_First_Lifecycle_Prototype_Implementation.pdf`.

[24] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000. URL `http://dl.acm.org/citation.cfm?id=354874`.

[25] Teemu Koponen. *A Data-Oriented Network Architecture*. Doctoral dissertation, Helsinki University of Technology, Espoo, Finland, October 2008. URL `http://lib.tkk.fi/Diss/2008/isbn9789512295609/`.

[26] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 181–192, 2007. URL `http://dl.acm.org/citation.cfm?id=1282402`.

[27] Ueli Kunz. metriculator | eclipse plug-ins, bundles and products - eclipse marketplace. http://marketplace.eclipse.org/content/metriculator#.UX6OXdfSVko, 2012. URL `http://marketplace.eclipse.org/content/metriculator#.UX6OXdfSVko`.

[28] Jack YB Lee and Raymond WT Leung. Study of a server-less architecture for video-on-demand applications. In *Multimedia and Expo, 2002.*

ICME'02. Proceedings. 2002 IEEE International Conference on, volume 1, pages 233–236, 2002. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1035761`.

[29] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.

[30] Landon Curt Noll. FNV hash. http://www.isthe.com/chongo/tech/comp/fnv/, 2012. URL `http://www.isthe.com/chongo/tech/comp/fnv/`.

[31] OpenVPN. OpenVPN - open source VPN. http://openvpn.net/, 2013. URL `http://openvpn.net/`.

[32] CCNx org. CCNx synchronization protocol. http://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtocol.html, 2012. URL `http://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtocol.html`.

[33] Eclipse Org. Eclipse - the eclipse foundation open source community website. http://www.eclipse.org/, 2013. URL `http://www.eclipse.org/`.

[34] PARC. Content-centric networking - PARC, a xerox company. http://www.parc.com/services/focus-area/content-centric-networking/, 2013. URL `http://www.parc.com/services/focus-area/content-centric-networking/`.

[35] PSIRP. PSIRP home page. http://www.psirp.org/, 2010. URL `http://www.psirp.org/`.

[36] PURSUIT. PURSUIT home page. http://www.fp7-pursuit.eu/PursuitWeb/, 2013. URL `http://www.fp7-pursuit.eu/PursuitWeb/`.

[37] PURSUIT. Test bed PURSUIT. http://www.fp7-pursuit.eu/PursuitWeb/?page_id=237, 2013. URL `http://www.fp7-pursuit.eu/PursuitWeb/?page_id=237`.

[38] Matthias Ressel and Rul Gunzenhauser. Reducing the problems of group undo. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139, 1999. URL `http://dl.acm.org/citation.cfm?id=320312`.

[39] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297, 1996.

[40] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984. URL `http://dl.acm.org/citation.cfm?id=357402`.

[41] Mikko Sarela, Jorg Ott, and Jukka Ylitalo. Fast inter-domain mobility with in-packet bloom filters. In *Proceedings of the fifth ACM international workshop on Mobility in the evolving internet architecture*, pages 9–14, 2010. URL `http://dl.acm.org/citation.cfm?id=1859987`.

[42] Sybase. TDS 5.0 functional specification v3.8, 2006. URL `http://www.sybase.com/content/1040983/Sybase-tds38-102306.pdf`.

[43] Dirk Trossen, George Parisis, Borislava Gajic, Janne Riihijarvi, Paris Flegkas, Pasi Sarolahti, Petri Jokela, Xenofon Vasilakos, Christos Tsilopoulos, Somaya Arianfar, and Martin Reed. Architecture definition, component descriptions and requirements. Technical report, PURSUIT, 2011. URL `http://fp7pursuit.ipower.com/PursuitWeb/wp-content/uploads/2011/12/INFSO-ICT-257217_PURSUIT_D2.3_Architecture_Definition_Components_Descriptions_and_Requirements.pdf`.

[44] Dirk Trossen, George Parisis, Kari Visala, Borislava Gajic, Janne Riihijarvi, Paris Flegkas, Pasi Sarolahti, Petri Jokela, Xenofon Vasilakos, Christos Tsilopoulos, and Somaya Arianfar. Conceptual architecture: Principles, patterns and sub-component descriptions. Technical report, PURSUIT, 2011. URL `http://fp7pursuit.ipower.com/PursuitWeb/wp-content/uploads/2011/06/INFSO-ICT-257217_PURSUIT_D2.2_Conceptual_Architecture_Principles_patterns_and_sub-components_descriptions.pdf`.

[45] David A. Wheeler. SLOCCount user's guide. http://www.dwheeler.com/sloccount/sloccount.html, 2004. URL `http://www.dwheeler.com/sloccount/sloccount.html`.

[46] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 352–356, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5477581`.

[47] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D. Thornton, Diana K. Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, and Christos Papadopoulos. Named data networking (ndn) project. 2010. URL `https://www.parc.com/content/attachments/named-data-networking.pdf`.