NELSON DA SILVA TEIGAS MORAIS

PROTOTYPING AN INFORMATION CENTRIC ARCHITECTURE FOR OPPORTUNISTIC NETWORKS

Orientador: Prof. Dr. Paulo Mendes

Coorientador: Bruno Batista

Universidade Lusófona de Humanidades e Tecnologias

Escola de Comunicação, Artes e Tecnologias da Informação

Lisboa

2013

NELSON DA SILVA TEIGAS MORAIS

PROTOTYPING AN INFORMATION CENTRIC ARCHITECTURE FOR OPPORTUNISTIC NETWORKS

Tese apresentada para a obtenção do grau de Mestre em Engenharia Informática no curso de Mestrado em Engenharia Informática e Sistemas de Informação, conferido pela Universidade Lusófona de Humanidades e Tecnologias

Orientador: Prof. Dr. Paulo Mendes
Coorientador: Bruno Batista

Universidade Lusófona de Humanidades e Tecnologias

Escola de Comunicação, Artes e Tecnologias da Informação

Lisboa

2013

Dedicatória

Á minha mulher, que esteve sempre presente e ao meu lado em todos os momentos.

Às minhas duas filhas, por todo o amor que me dão desde o dia em que nasceram.

Aos meus pais, por tudo o que me ensinaram ao longo destes 40 anos, e a quem devo muito do que sou.

Ao meu irmão, amigo eterno.

Acknowledgments

Colleagues, Friends and Teachers

During my bachelor and master degrees was fortunate to have found people who helped me in several ways to achieve my goals. For them, here are my words of appreciation and recognition.

To my two colleagues and friends that were with me during my bachelor degree, Hélder and Gustavo, big fellows in the several challenges that we have faced together, and from whom I keep with great pleasure the moments we shared.

To my colleague and friend, Ricardo Barbosa, with whom I exceeded what I thought were insurmountable barriers.

To my friend Vítor Castro, fellow of great discussions and with whom I explored many of the projects I have done.

To teacher Dr. José Rogado for all the knowledge transmitted on all these years, and for the opportunities provided to exceed myself.

To teacher Vítor Santos, for everything he taught me and for the trust he always raised on me.

To teacher Dr. Paulo Mendes, without him this dissertation would not be a reality. He opened my horizons to areas that were outside of my knowledge.

To teacher Bruno Batista for all the support given during the work done on this dissertation.

Resumo

Morais, Nelson. **PROTOTYPING AN INFORMATION CENTRIC ARCHITECTURE FOR OPPORTUNISTIC NETWORKS**. Lisboa. 2013, 97 páginas. Monografia, ULHT – Universidade Lusófona de Humanidades e Tecnologias.

A arquitetura atual das redes informáticas, que serve de suporte à maioria dos sistemas de informação, assenta num modelo de comunicações ponto a ponto e no qual a informação é transmitida em pacotes de dados que contêm informação sobre a origem e o destino dos mesmos. Este paradigma não se ajusta às necessidades de processamento de informação com que nos deparamos diariamente nos dias de hoje. Atualmente, a informação é ubíqua e os meios de acesso à mesma são efetuados cada vez mais com recurso a dispositivos móveis que comunicam sobre redes intermitentes ou pouco estáveis. A última década tem sido propícia ao aparecimento de propostas baseadas em novas formas de distribuição e acesso à informação com base em redes movidas pelo conteúdo da própria informação. Esta tese analisa uma dessas propostas e utiliza-a como base para a criação de um protótipo de uma rede desse tipo, direcionada para sistemas que operam em multiplataformas e sobre redes oportunistas.

Palavras-chave

Redes, Centradas, Orientadas, Conteúdo, Informação, Dados, Multiplataforma, Oportunistas

Abstract

Morais, Nelson. **PROTOTYPING AN INFORMATION CENTRIC ARCHITECTURE FOR OPPORTUNISTIC NETWORKS**. Lisbon. 2013, 97 pages. Monograph, ULHT – Universidade Lusófona de Humanidades e Tecnologias.

The current architecture of computer networks, which supports the majority of information systems, is based on a model of peer to peer communications and in which information is transmitted using data packets that contain information about its origin and destination. This paradigm does not fit the needs of information processing that we face every day nowadays. Currently, information is ubiquitous and the mobile devices are used even more to establish communications over intermittent or unstable networks. The last decade has been conducive to the emergence of proposals based on new forms of distribution and access to information based on networks driven by the content of the information itself. This dissertation examines one of these proposals and uses it as a basis for the creation of a prototype of such a network, targeted to systems that operate on multiplatform and opportunistic networks.

Keywords

Networks, Content-Centric, Information, Data, Multiplatform, Opportunistic

Abbreviations

ADU – Application-layer Data Units

CCN – Content Centric Networks

CCNx – Content Centric Network project

CPU – Central Processing Unit

ECATI – Escola de Comunicação Arquitetura Artes e Tecnologias da Informação

DONA – Data Oriented Network Architecture

DI – Dependency Injection

DTNs – Delay/disruption Tolerant Networks

FIB – Forwarding Information Base

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

ICON – Information Centric architecture for Opportunistic Networks

IoC – Inversion of Control

ISP – Internet Service Provider

IL – Intermediate Language

JIT – Just In Time

MSIL – Microsoft Intermediate Language

NDN – Named Data Networking

OSI – Open Systems Interconnection

PIT – Pending Interest Table

SCF – Store Carry and Forward

RFCOMM – Radio frequency communication

RPC – Remote Procedure Call

SITILabs – Research Unit in informatic Systems and Technologies

TLS/SSL – Transport Layer Security / Secure Sockets Layer

UDP – User Datagram Protocol

ULHT – Universidade Lusófona de Humanidades e Tecnologias

Uri – Uniform Resource Identifier

Contents

Table Index

Figure Index

1 Introduction

Nowadays a large amount of people uses the Internet for some kind of activity – business or pleasure – on a daily basis. The Internet moved from the bulky desktop computer to more pervasive devices like our smart phones or some form of embedded device that, for instance, can be found in the increasingly popular intelligent buildings, in our homes – fridges, laundry machines, domotic systems, etc. – or on the street for monitoring purposes, advertising, etc.

The increase capabilities in terms of CPU, storage and communication capabilities of these devices means for one hand, that they can be consumers and producers of information, and on the other hand, exacerbates, among others, problems related with mobility and opportunistic networks where users take advantage of the diverse communication, computation, sensing, storage and other resources that surround them to produce and share information.

Looking to when and how people use the Internet – and network systems in general – a clear mismatch can be found. People only care about obtaining information and they do not care where the information is. From the network point of view the location is essential to find the requested content, thus when a person asks for content by name, that name has to be resolved to a location, an identifier that the network can understand. Moreover, the current network architecture, which has a dependency of the host location, is far from ideal in scenarios where hosts are very mobile, have a multitude of communication interfaces with very different technologies that need to work in a seamless way and have intermittent connectivity like ad-hoc and opportunistic networks.

This dissertation aims to address the problems of data communications found on such network architectures. It explores a network protocol where data flows across the network are based on the message content in contrast to the host location. Using this principle, we can persist data across the network as it flows between consumers and producers, because there is no host location dependency on the network packages. This avoids data retransmission through all network nodes every time a connection is lost. By removing the host location from the network packets, we can also reuse the content data of the network packets to deliver it to more than one

host and at different periods in time, improving network traffic by reducing the number of packet transmissions between consumers and producers. Mobile hosts can benefit with this new way of transmitting data. As packets that were not received due to network communication interruptions are now available on the network, mobile hosts have now a better chance to receive these packets since they will be delivered from a closer network node, instead of having to travel again from the producer.

1.1 Technical Background

1.1.1 Content Centric Networks

Current communication networks are evolving towards a more dynamic and mobile model, in which people aim to get access to collections of digital content and services, anyplace and at any time. However, the communication in mobile networks implements connections between machines, not location-independent access to content. This mismatch between the structure and use of networks causes complexity for applications and ineffective use of available bandwidth.

In recent years content-centric networking has gained momentum and several proposals arose, where some of the most prominent are:

- Content Centric Networking – CCNx [1]
- Named Data Networking – NDN [2]
- Haggle [3]
- NetInf [4]

This new paradigm that make content retrieval by name, not location, will change the architecture and the fundamental operation of networks.

1.1.2 Opportunistic Networks

Life style in modern society is creating an increasing demand for users to have constant capability to exchange information. However, providing pervasive connectivity to users that have a dynamic behavior is challenging since most social and technological networks display

topologies with patterns of connection between nodes that are neither purely regular nor purely random.

Examples of such networks are online social networks and Delay/disruption Tolerant Networks – DTNs [5]. DTNs are an example of opportunistic networks of self-organizing wireless nodes connected by multiple time-varying links, and where end-to-end connectivity is intermittent. Even in urban scenarios, it is possible to face intermittent connectivity due to dark areas, such as inside buildings and metropolitan systems, as well as public areas with closed access points or even places overcrowded with wireless access points. Unavailability of wireless connectivity can be also a result of power-constrained nodes that frequently shut down their wireless cards to save energy.

Due to intermittent connectivity, network architectures based on the knowledge of end-to-end paths perform poorly, and numerous opportunistic routing algorithms have been proposed instead [6] [7]. Some opportunistic routing protocols use replicas of the same message to combat the inherent uncertainty of future communication opportunities between nodes [8] [9]. In order to carefully use the available resources and reach short delays, many protocols perform forwarding decisions using locally collected knowledge about node behavior to predict which nodes are likely to deliver content or bring it closer to the destination [10]. In an information-centric approach, such destinations are reachable when they disseminate in the network their willingness to receive data, being such data defined by a set of parameters, such as type and users interests. To operate in an opportunistic network, nodes must have enough processing power and storage to keep data until another good intermediate carrier node or the destination is found [11], following a store-carry-and forward – SCF – paradigm.

1.2 Objectives

This dissertation aims to implement a prototype of an information centric architecture for opportunistic networks, called ICON [12]. The architecture is based on ideas extracted from the CCN / NDN and Haggle projects, which will be combined and enhanced. The goal is to extract from Haggle the modular flexibility of its architecture to operate in opportunistic networks and from NDN some of the basic communication elements, such as the forwarding and interest structures.

The prototype runs on Windows, Linux and Mac OS operating systems with Microsoft.NET Framework or Mono installed. To achieve such goal, the architecture was developed in C# targeting the Microsoft.NET framework subset implemented by Mono [13]. Mono is a software platform designed to allow developers to easily create cross platform applications. Mono is an open source implementation of the Microsoft.NET Framework based on the ECMA standards for C# and the Common Language Runtime.

To achieve the desired cross platform transparency in ICON, which is a key difference from Haggle and CCN / NDN projects, the ICON prototype code was compiled to target the Microsoft.NET Framework using the Mono subset library to ensure the cross platform execution. By compiling the code in this way, we get code compiled to an Intermediate Language – IL –, which will be compiled upon execution on the runtime environment by a process called Just In Time – JIT – compilation. For this reason the code compiled, is valid to be executed without requiring code changes or code recompilation on systems where Microsoft.NET Framework is installed, but also on systems where Mono is available. The development of the solution was done using Microsoft Visual Studio 2010 and 2012, but MonoDevelop can also be used as an alternative. The code was compiled to ensure compatibility with Monodroid for Android, which along with MonoTouch for iOS, are a subset of the Mono framework and consequently, a subset of the Microsoft.NET Framework. For these two platforms, further tests must be taken to ensure the system is fully operational under those environments, as the prototype developed under this project did not made any execution tests on such platforms.

1.3 Document structure

- The second chapter briefly discusses the Named Data Networking Project [14] and some of the most relevant projects around content centric networks, with a major focus on the CCNx project. Opportunistic networks and the project Haggle [3] are also introduced in this chapter.
- The third chapter introduces the Content Centric Network framework developed during this dissertation, which was called ICON – Information Centric architecture for Opportunistic Networks. It introduces its roots, architecture, and main components.
- The fourth chapter contains implementation details of the ICON Framework, including code extracts from the prototype source code, the development environment and tools used to achieve the final bits. It also identifies the major implementation issues found for the targeted platforms and explains why not all of initial goals were met.
- The fifth chapter describes how and on which environments the ICON Framework was tested.
- The sixth chapter presents the conclusions, including achieved goals, presentations done at the CCNxCon 2012 conference in the INRIA Sophia Antipolis, France [15] and at the CCNxCon 2013, Palo Alto, USA [16].

References and citations in this document were made using the IEEE style.

2 Content Centric and Opportunistic Networks

Content centric networks also known as content based networks, named data networks, information centric networks or even data oriented networks, are a new paradigm on how computer networks should be organized to better achieve today's requests for digital information.

Most of today's computer networks and Internet in particular, have their roots based on telephone networks, as such, the architecture is based on hosts and the connections established between them in order to transmit data. The Internet Protocol allowed us to establish communications worldwide, which combined with all the other protocols on top or around it exceed all expectations for ubiquitous connectivity. Nevertheless, this architecture with some of its surrounding protocols has now more than fifty years and as a consequence, today it is barely used for its original purposes. Nowadays, the challenges are different, the request for information has increased dramatically and people ask for data, and they do not want to know where it is hosted anymore. Through the years several changes were made to allow such requirements but current data communication patterns demand a new architecture to better address current needs.

In recent decades several projects were created to address this, the DONA – Data-Oriented Network Architecture [17], the TRIAD [18], the NetInf [4], the CCN – Content Centric Networking [19], are all distinct proposals on how to embrace this new challenge. Of those, the later was a fundamental reference for this dissertation.

In the context of opportunistic networks, project Haggle [3] propose a set of architectural principles, for pocket switched networks, based on asynchronous data-centric network architectures, which raise the application programming interface to a level where applications can provide the network with application-layer data units – ADU's, with high-level metadata concerning ADU identification, security and delivery to user-named endpoints. Such architecture, composed by a set of modular managers to build the core of the system, also inspired the work developed under this dissertation where a set of engines with their components build the core of the proposed framework.

2.1 Named Data Networks and the CCNx project

CCNx [20] is an open source project in early stage development exploring the next step in networking which is based on one fundamental architectural change: replacing named hosts with named content as the primary abstraction.

The architecture behind CCNx and the Named Data Networking Project – NDN [14] in particular, were major resources to the development of this dissertation. Explaining some of its core components and concepts with an emphasis on the most relevant parts herein is required step to understand the work developed under this dissertation.

Figure 1 and Figure 2 show most of the fundamental elements of the NDN architecture. Figure 2 represents a complete CCN Node and Figure 1 illustrates the only type of messages switched between those Nodes.

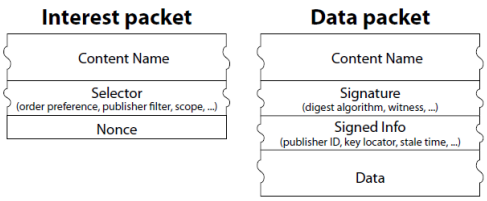


Figure 1 - CCN Messages

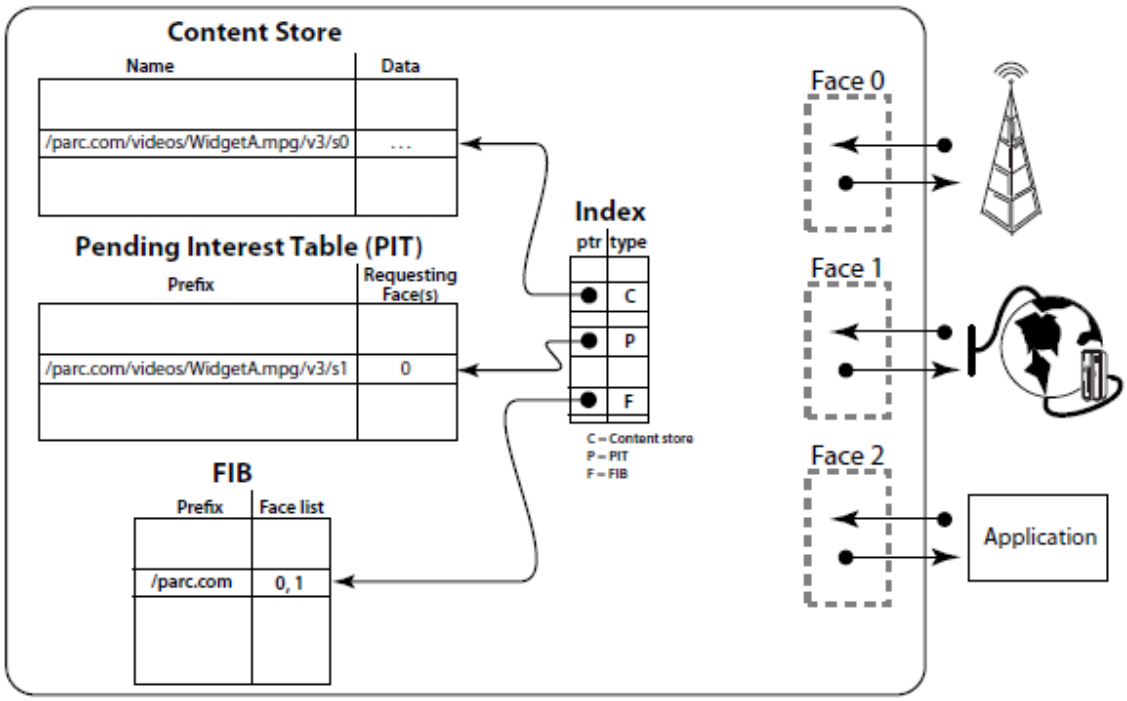


Figure 2 - CCN Node

Following the principles described in the Named Data Networking Project, CCNx Nodes exchange data with each other using a protocol that does not require any knowledge about the hosts where the data passes through in order to send or receive data. To achieve that, the CCNx architecture can be divided into six major areas:

- Names
- Security
- Routing and forwarding
- Caching
- Pending Interest Table
- Transport

Excluding the security, which will not be discussed here despite its major role in this architecture, all the other areas will be addressed to open the door to what was implemented in the ICON Framework, but before discussing each of these areas, we need to understand how communications are processed in a network based on CCNx Nodes.

Every communication in CCNx is initiated by the receiver, the one who will consume and process the data it requests. To achieve that it must send to the network its intention to get some data, and for that it issues an Interest to the network which eventually will be satisfied. An Interest message is essentially a Name that identifies the Content to be retrieved. A Name is mainly a set of hierarchical components, which uniquely identify some Content. Once the Interest is sent, routers on a CCNx network will store the Face where the message arrives and will forward the Interest to other Node(s) based on its Forwarding Information Base – FIB table. When the Interest arrives to the producer, the one capable to produce or retrieve the requested data, a Data message containing the name of the data, a signature, the signer info and the data itself, is sent back to the Face where the Interest message arrived. On the network, the Data messages make the reverse trajectory until they reach the consumer who asked for the Interest. This trajectory is done based on the Name contained in the Data message along with the information that was temporarily persisted on the Nodes to where the Interest was sent. To improve network efficiency, if a Node receives the same Interest while it is waiting for a response for an equal Interest, the Interest received will not be forwarded again but the Face where it has arrived will be stored internally in the Pending Interest Table – PIT. Once the Data packet is received, and a PIT entry exists for it in the Node, the Faces stored with the PIT entry

will have the Data packet forwarded to all of them and the PIT entry is removed. The Data packet is also stored locally on the Node's local storage, allowing new Interests to be served directly from this local cache whenever their Name corresponds to the Name of a Data packet previously stored. Finally, Faces represent communication interfaces to the Node's outside world, and this can be wireless and/or wired networks or applications.

This networking paradigm also redefines the Open Systems Communication – OSI model, by structuring the layers in such a way that removes the bottleneck from the current Network layer, and moves the universal component of the network stack from IP to chunks of named content. Figure 3 illustrates the comparison between the two models.

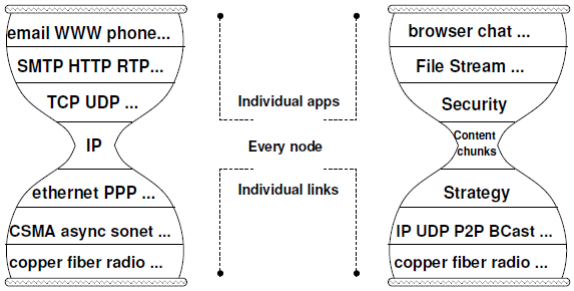


Figure 3 - OSI model redefined for CCN based networks

2.1.1 Names

Names represent the most important piece in the NDN architecture and are still under active research; in particular, how to define and allocate top level names remains an open challenge [14]. Top level names are the gate that opens the door to the definition of global unique names, but Name uniqueness is not required, unless for data retrieved globally, because applications can define profiles that specify naming conventions to be used to exchange their data. Applications can choose the naming schemes they want, and according to their needs, but the NDN design assumes hierarchically structured names, which in a human readable format can be represented like '/ulht/ecati/siti/videos/ICON_Video'. The name hierarchy allows the routing mechanism to scale and at the same time provides useful information for applications, like the relationships between pieces of data. The names are only meaningful for the applications and are completely opaque to the network, which means routers do not know the meaning of a name, only the boundary of its components is known. Dynamically generated data, like a live video stream for instance, require the Names to be defined in a deterministic way, which means, applications must agree on an algorithm to allow both the consumer and the producer to reach to the same name for the same piece of content, or, consumers must be able to retrieve data based on partial names.

Below is a text representation of a Name sample, providing access to the first release of the third segment of a video produced by the Universidade Lusófona de Humanidades e Tecnologias – ULHT, in the Sistemas e Tecnologias Informáticas – SITILabs, of the Escola de Comunicação Arquitetura Artes e Tecnologias da Informação – ECATI School:

- /ulht/ecati/siti/videos/ICON_Video_Demo_Win8_Ubunto_MacOS.wmv/1/3

Each forward slash separates the components who define the Name and its hierarchy.

2.1.2 Routing and Forwarding

Routing and forwarding packets are based on names, which eliminates four problems that addresses pose in the IP architecture: address space exhaustion, NAT traversal, mobility, and scalable address management [14]. Has the names do not have an upper boundary defined, which means they do not have a limit on their size, the space address limit imposed by the IP architecture do not exist on this architecture. NAT is not required since host addresses are no longer exposed or transferred between network Nodes communications. Mobile devices, with frequently changing IP addresses will benefit from the fact that data transmission will not rely on such information to successfully transmit data. Data is transferred based on their Name and not from where it came from or goes to. Such a fact will improve the communications since data transfers will not be broken due to address changes. Sensor networks, which require address assignment and management is now dispensable, which will bring benefits for its maintainability. CCN Nodes maintain a FIB table, which stores the forwarding information required to send Interests to the network, this table is composed by a list of Name prefixes and each prefix contains a list of all the Faces that can satisfy it on the network, Figure 4 highlights the graphical representation of the FIB table on a CCN Node. Routing protocols, similar to those we have today for IP routing, can be used to announce the Nodes capability to forward Interests, this can be done using specific Name schemes, which once defined could be used to let Nodes know which Interests can be satisfied by whom. Due to the unbound nature of Names, where no length limit is established, there is an extra challenge

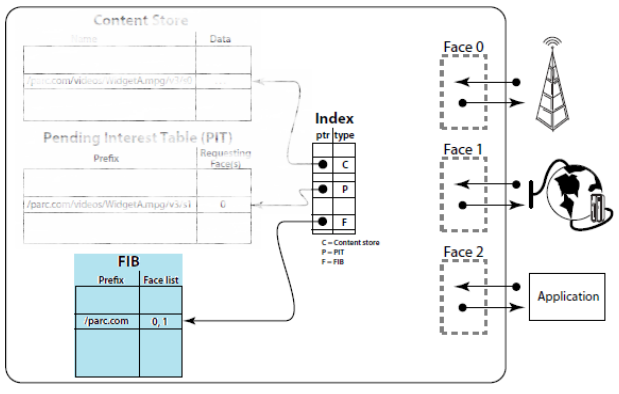


Figure 4 - FIB table on a CCN Node

of how to maintain the FIB table size scalable to the number of data names it stores, because the longer the Name length is more memory is required to store data in the FIB table. ISP-based aggregation and exploiting the Name space and network structures are two proposed approaches to address this problem, and belong to the research agenda of the NDN project. In contrast to IP, NDN also natively supports multipath routing and without the risk of looping within the network, since Data packets make the reverse path of the Interests requests they reply to. This opens a new world for algorithms to explore such capacity, which have its space reserved under the Forwarding Strategy layer of the NDN project [14].

2.1.3 Caching

Data packets are stored in the Node Content Store once data is received. As data is identified by their Name, Interests requesting the same data can be served by data stored locally, avoiding round trips to content producers. This will improve response times, as data will be closer of the consumers and spread across the network. Not only this will improve data retrieval for static data, it will also benefit dynamic content, mostly for multicast scenarios where the same data is transferred across several consumers.

When a cache miss occurs, a lookup on the FIB table is made to check if the Node can satisfy the Interest requested. If the FIB table contains an entry that can satisfy the interest, which means that there is a prefix in the FIB table that fully or partially matches the Name of the Interest, then an Interest message is sent to the Faces associated with that entry and a record is added to the PIT table. This record contains the Name and the Faces to where the Content message should be sent to satisfy the Interest. When a cache miss occurs, and no match if found in the FIB table, then the process ends and the Interest request will not be satisfied by the Node.

Cache management is an area still under development, but previous work developed on this area suggested a least-recently-used or least-frequently-used policies to be the desired policies, nevertheless a simpler approach based on random removal from the cache are also to be tested and validated to see if it performs as nearly as well. [21]

2.1.4 Pending Interest Table – PIT

The PIT contains the Faces where Interests have arrived and for which the Data packets must be sent once received. It plays a fundamental role on the overall architecture as it provides the ability the send fewer requests to the data producers while it caches all similar Interest requests until data is produced and retrieved by the producers. PIT entries should not live longer than the expected packet round trip in order to maximize the PIT usage. At the same time, they should not time out prematurely or packets will be lost, as received Data packets without a corresponding PIT entry will be discarded, although they are stored locally. It is also the base support for natural multicast, as it caches all the Faces who required a certain content, which, upon received, will be set to all the cached Faces of the PIT entry. Once the Data packets are sent to the PIT entry cached Faces, the PIT entry and the Faces it stores are removed. A PIT entry only exists if the Interest could not be satisfied from the local Content Store. The PIT provides another piece of important information that can be used to control the Node's traffic, the number of PIT entries identify the Node's congestion. This information can be used by the Forwarding Strategy to adjust network traffic.

2.1.5 Transport

The NDN architecture does not have a separate transport layer. Applications, their supporting libraries and the Forwarding Strategy layer take the responsibility of today's transport protocols. Data can be transferred on top of any data link layer or superior of the OSI model, as such, no reliable transmission channel is required, and the highly dynamic networks like those where mobile devices mostly operate are natively supported. As a consequence, reliable communications must be supported at the application level, where apps have the task to resend Interest requests when Data packets are not received, and if they are still meaningful. A lower level alternative exists at the Forwarding Strategy layer, which, with the gathered information, will be able to choose from the available Faces the best one to be used to retransmit the Interests.

2.1.6 *ICON Framework and the CCNx project*

The ICON Framework is based on some of the concepts defined on the CCNx project, which were found useful for its definition and implementation. In essence, the protocol

definition to support Content Centric Networks, along with the fundamentals of some of the structures behind it, where brought to the ICON Framework. However, the ICON Framework, pretends to be modular, cross-platform and simple to use, so despite the CCNx project had influence it, some changes and enhancements were introduced to achieve its goals. Both sections containing the ICON Framework description and implementation, will provide more details on the similarities between both projects, but for now these are the concepts that the ICON Framework will get from the CCNx project:

- A hierarchical Name definition to identify and request for content;
- A Interest message that describes the intent to get some content;
- A Content message that carries content identified by a Name and without identifying the source or destination hosts in anyway;
- A data transmission protocol based only on Interest and Content messages;
- The Faces that represent the abstraction of communication interfaces;
- A Content Store, to persist data locally. Data stored in this cache can be sent to other requesters when they ask for the same content;
- A Forwarding Information Base, to act as the routing mechanism;
- The Pending Interest Table, to manage pending Interests sent to the network and that are waiting to be satisfied;
- Some of the principles behind the Strategy Layer, like managing the cache retention policy. The ICON Framework Decision Engine is where we can find some of the functionalities similar to those that exist on the Strategy Layer

In terms of serialization, the ICON Framework did not opt for the solution implemented in the CCNx project, where a custom serialization was defined. The ICON Framework uses an already known serialization process that provided the efficiency required by the platform, more details on the serialization process used can be found in The ICON Framework section.

To achieve modularity and a network node model based on three engines, the concepts brought by the CCNx project were also organized differently. The ICON Framework and the ICON Framework Implementation and Usage sections both provide details on what is an ICON Framework node and how it was designed to support modularity.

Cross-platform using the same compiled code is also a difference between the CCNx project and the ICON Framework. The ICON Framework was developed in C# and compiled using the .NET Framework to support such functionality.

2.2 Opportunistic networks and the Haggle project

The Haggle project approach to opportunistic networks is based on dissemination of searches to share content directly between intermittently connected mobile devices. The project is based on two pillars, the first is searching technologies and the second is a new network architecture and paradigm that provides a better support for opportunistic networks.

In Haggle, data that is stored locally or that is disseminated to the network contains metadata attributes that will be used to perform queries against it. The metadata is based on a set of attributes in the form of key value pairs, which at the end describe the data where they are attached to. In Haggle this tuple of metadata plus data is called a data object. Searching is then based on data objects by looking at their metadata, and when a search is performed, it results in a ranked list of data objects relevant to the query.

Data objects are stored in Haggle nodes, which are the devices where the Haggle platform is running, and data objects are stored in a relation graph where they are connect to each other by a relation. This relation might have a weight to define the strength between two data objects, and this strength is determined by looking at the number of shared attributes that exist in both data objects, so the more the shared attributes the higher will be the value of this weight. Haggle nodes are also considered data objects and are called node descriptors; the node descriptors contain on their metadata the combined application interests of a node. Application interests are the set of registered attributes that an application can register, and will work as keys to perform the search queries against the data objects. Node descriptors are sent to every a new node that a node discovers.

Haggle defines a set of primitives to perform its search-based resolution. A first set of primitives allows the insertion of data objects and node descriptions in the relation graph of the node. A second set exists to map data objects to nodes and vice versa, and these are called resolve primitives. The resolve primitives are named r_d to resolve data objects and r_n to resolve

nodes. Figure 5 shows how these primitives are used in Haggle applications, where the pushed-based data dissemination service that exists in Haggle sends to the applications the data objects that match their interests.

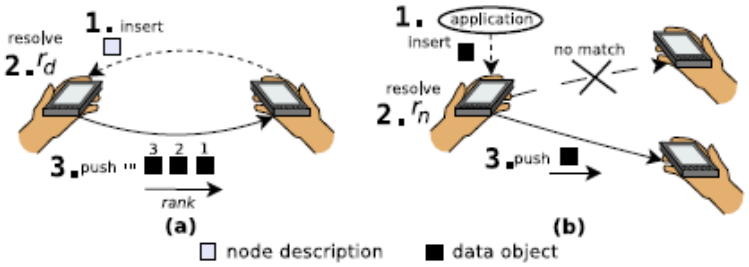


Figure 5 - Search-based resolution primitives

2.2.1 Haggle Network Architecture

The Haggle network architecture is based on a modular paradigm where the core system results from the aggregation of a kernel and a set of managers responsible for the different aspects of the system. Figure 6 shows the Haggle architecture where a kernel exists in the middle, and processes the messages that are sent to the event queue by the set of managers that live around him. Managers communicate with each other via the message queue of the kernel. The kernel also contains a local store where the relation graph of the node is persisted. Search queries are done on top of this storage to retrieve a ranked set of data objects that satisfy the search criteria. Core types are specified in the kernel, like the attributes that define the metadata of a data object, the node descriptors and the interfaces that abstract the communication with other peers. Interfaces can be physical, like an Ethernet or Wi-Fi network card, or can be a logical representation provided by an inter-process communication mechanism, like a socket or a pipe. This architecture promotes modularity because any manager can be easily replaced or added when new functionality is required.

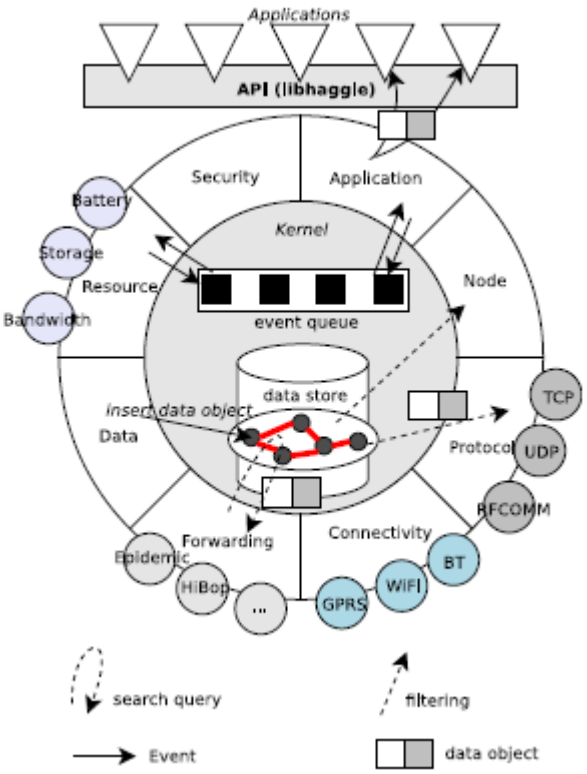


Figure 6 - Haggle network architecture

Applications communicate with Haggle via the libhaggle API. This hides the complexity of the network from the applications. To support this, Haggle uses a Application manager that acts as a proxy between the application and the network architecture to exchange data objects or events. Other managers have different responsibilities:

- The Connectivity manager for instance, is responsible to discover local and remote network interfaces and provide the infrastructure with the capacity to communicate with other nodes;
- The Protocol manager is responsible to send and receive data objects, it uses TCP for communications when using the Ethernet and Wi-Fi interfaces, RFCOMM for Bluetooth and UDP for local inter process communications;
- The Resource manager handles information about the device, like power battery, available storage and bandwidth, and can change the behavior of the node to ensure that the resources are used appropriately and in an efficient way;
- The Node manager gathers information about discovered nodes, and tries to exchange node descriptors every time a new node is found on an interface, it also manages an internal list of active nodes and adds the discovered nodes to the relation graph of the node

2.2.2 ICON Framework and the Haggle project

The network architecture design of the Haggle project depicts a clear separation of responsibilities in a network node. By assigning such responsibilities to specific managers, which can be replaced or extended with new ones if needed, the architecture provides a modular framework that shows how it can be improved or raised to provide new functionalities. This modular design inspired the network architecture that can be found on the ICON Framework. The ICON Framework section exhibits changes that were made to this network model, showing how the overall set of components are organized on the network node model definition that it defines. In a brief introduction to these changes, we will see that instead of having a central kernel, which all the other managers communicate with to interact with each other, the ICON Framework network node model defines an aggregator kernel composed by three engines, each containing their own set of components to perform specific tasks. This provides a more

aggregated view of the network node, but at the same time, maintains the modularity aspects that provide the flexibility demonstrated on the Haggle project.

The ICON Framework will not use the search-based data dissemination framework present in Haggle. The reason behind this decision relies in the fact that the Haggle project is more focused in exploring a better way of spreading data in opportunistic networks, using for that, a push mechanism to send data to the devices that express interests matching that data. Such data is then selected using a ranked search algorithm to identify the data objects to be pushed. The paradigm for the ICON Framework is different: the goal was not to merely disseminate data across nodes, but to have a framework that can be used in opportunistic scenarios, sustained by a network protocol resilient to the network interruptions that exist in such environments, so that data can be sent across the network devices in a more efficient way than what we have today with the traditional network protocols. To achieve such goal, the choice was to use some of the concepts that guide the CCNx project, which seemed more appropriate, instead of using the search-based framework presented in the Haggle project.

These are the concepts that the ICON Framework will use from the Haggle project:

- Network design architecture;
- Modularity;
- Flexibility;
- Abstraction of network complexity;
- A network kernel;
- API to be used by applications

3 The ICON Framework

This chapter describes the ICON Framework, its architecture, and main components.

3.1 Introduction

The ICON Framework is heavily inspired on the CCN Node Model [19] found on the CCNx architecture, in particular it is based on the fundamental components and principles present on the CCN Forwarding Engine Model, which are the Faces, the Content Store, the Pending Interest Table and the Forwarding Information Base. More than getting these structural elements from the CCN Model, also the core messaging components of the CCN Model – Interest and Content Object messages – where brought as vital infrastructure elements to the ICON Framework architecture – Interest and Content – , as these are also the only type of messages transferred between ICON Nodes.

The modularity features found on the Haggle platform [3], which brought more flexibility to a system where opportunistic communications are the main focus, were also present since the beginning of the architecture design of the ICON Framework. As opposed to the CCN Model, where components modularity cannot be found, Haggle based their entire architecture on individual components – Haggle Managers, who communicate with each other using the event driven messaging pattern [22].

Despite the ICON Framework brought some of its concepts from the CCN Node Model and the Haggle platform, it innovates by combining them and introducing new ones in a new framework. The ICON Framework goes beyond the modularity concepts found in Haggle and extents this concept; it uses dependency injection to automatically load the modules at runtime, according to the application needs. Multithreading is also part of the ICON Framework foundations, ICON Nodes can process requests concurrently and they also create worker processes internally to monitor the PIT in several ways as will be described during the ICON Framework Implementation section. Applications have also the capacity to define their own custom implementations of the core modules, for cases where the provided ones do not fit the in the application requisites, and where such customizations might be preferred to run on specific devices or environments. If needed, the modules can also be switched at runtime to allow the Node to adjust himself to a new network environment, where a change in the Node

behavior provided by a different module implementation might be appropriate. Changing modules at runtime, by using the dependency injection functionality, can be done by the Decision Engine module based on the rules and context information already gathered. For this reason, modules in the ICON Framework can be developed in such a way that they can be customized to run on specific devices or network environments in order to take the best return from the available resources. For example, on a device where more than one communication interface exists, if the device goes below a configured threshold on the battery charge value, the Decision Engine might opt to load a Network Engine that uses communication interfaces with a lower power consumption, thus optimizing the battery consumption and avoiding a faster battery exhaustion.

Regarding the message format and encoding it does not follow the implementations found on CCN or Haggle. Instead of using a custom proprietary format, the ICON Framework uses an open format known as Protocol Buffers [23]. The protocol produces small messages and is fast to encode and decode data.

The FIB and PIT concepts present in the CCN node are essential concepts brought from this project to the ICON Framework. These concepts are found on the Metadata Store component, which is responsible to manage the forwarding information table and the pending interest table of an ICON Node. On the CCN node model these two concepts along with the content store are accessed via a common list where all these concepts are mixed and indexed, but the ICON Framework provides direct indexed access to each one of these lists and avoids the additional level that exists in the CCN node model.

The ICON Framework has the capacity to define multiple communication interfaces per Node. Using such capacity, we can use the Decision Engine to select the best communication interface, or even more than one if desired, to forward messages to the network. This is beneficial, since data exchange will not be dependent on a specific communication interface, which allow us to adjust the forward information table as the environment condition changes. With the capacity to use the Decision Engine to adjust the forward information table and the communication interfaces in use, the system is more capable to adapt himself to opportunistic routing, where forwarding decisions are made using locally collected knowledge about node behavior, to predict which nodes are likely to deliver a content or bring it closer to the destination [24].

The architecture design of the ICON Framework depicts tree main components that together define the ICON Framework network Node:

- Decision Engine
- Network Engine
- Data Engine

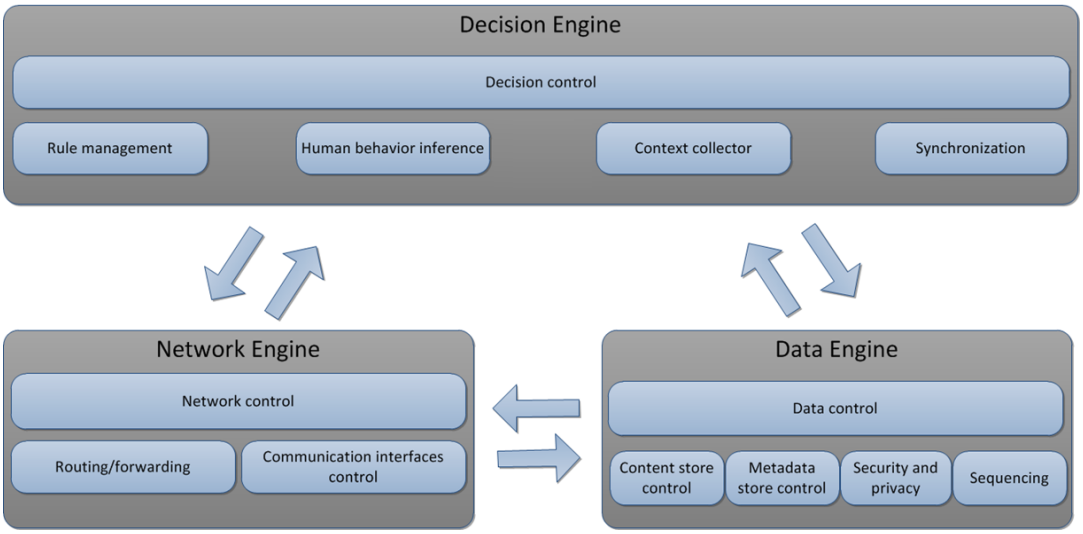


Figure 7 - ICON Framework architecture design that defines the network Node

Each one of these engines have their own responsibilities and are designed to deal with specific areas identified during the design of the ICON Framework architecture. The Node Model and the Implementation sections of this document describe how each of these engines are defined by components, illustrating the main responsibilities of each one. When compared to the Haggle node, we can see a simplified node structure based on three engines only, versus the larger set of managers present on a Haggle node. In Haggle, we can find managers for protocol management at the same level that we see managers for connectivity management, forwarding information management, data management or even node management. In contrast, ICON makes a clear distinction between data management, network management, and decision management, and defines three engines to handle such areas. ICON takes the principles of modularity found in Haggle and readjust them to a more clear view of the Node, where each engine can handle specific areas, which are then composed by components that are capable to interact with each other to manipulate the Node behavior as a whole. The ICON Framework also differs from Haggle by using dependency injection to load the engines or components, as such, the ICON Framework is extensible and adjustable to scenarios where a customization of

some components or even a customization of a full engine might be preferred to address the device or the environment where the ICON Node will be executed.

Altogether, the Decision Engine, the Network Engine and the Data Engine define the kernel of an ICON Node. Applications will use this kernel to be able to communicate using the ICON Framework network protocol in such a way that they do not need to worry about how data is transmitted between nodes. In fact, they do not even need to worry about who is the host of the data they ask for, or how it can be reached. With the routing configurations present in the forward information table, which map Name prefixes to Communication Interfaces, and that can be managed by the Decision Engine rule management component, the kernel earns the capacity to deliver, route and receive data whenever applications need to communicate with each other.

The ICON Framework imports other fundamental concepts from the CCNx architecture, such as the Profile, the Name, Name Prefix or Prefix and the Faces. All these concepts have their correspondent in the ICON Framework, which are:

- Profile – Naming rule, data format and ICON message semantics
- Name – Represents a hierarchical name on an ICON Interest or Content message
- Name Prefix or Prefix – Subset of a Name, where the right part of a Name can be excluded. Ex: ‘ccn4d:images’ is a prefix of ‘ccn4d:images/image1.jpg/1’
- Communication Interface – generalization of the concept of a communication interface

3.2 Protocol

The protocol defines how messages are exchanged between ICON Nodes. The first step is defining how the messages are formatted and encoded/decoded by the system, which is described in the Message format and encoding section. These messages transport information that must be identified, so the protocol defines the concept of a Name, which is used to identify the content being requested on an Interest message, and also to identify the data on a Content message, when the interest is being served. The protocol defines two types of messages which are the only messages used to establish communications between Nodes, these are the Interest message and the Content message. These messages flow on the network following a data

exchange pattern that starts when the receiver issues an Interest request to the network, and ends when it receives the corresponding Content message to satisfy the Interest that was sent.

3.2.1 Message format and encoding

Like the CCNx protocol an ICON message does not have a fixed-length of fields. But unlike the CCNx protocol where a custom binary format message was implemented, for the ICON Framework the option was use a message encoder who has already proven to be effective, efficient, reliable, extensible, easy to use and freely available, so the option was to use Protocol Buffers [23] as the message encoder for all messages transmitted between ICON nodes.

Protocol Buffers is Google's data interchange format and it is used by Google for almost all of its internal RPC protocols and file formats [25], and there are several implementations available for several languages and platforms.

As the ICON Framework target platform is Microsoft.NET, instead of implementing the protocol inside the ICON Framework, the choice was to use a current implementation of the protocol for the targeted environment. After looking for some of the current implementations available the choice was for the Protocol Buffers implementation implemented by Marc Gravell. Nevertheless, any other implementation could be used, as what effectively matters is what goes on the wire when the messages are transmitted, so as long as the protocol is correctly implemented, the choice behind any specific implementation is just a matter of preference based on the library usage.

The choice for Marc Gravell implementation was mainly because its usage is extremely easy and similar to the already know concepts present inside the Microsoft.NET framework. Rather than being "a protocol buffers implementation that happens to be on .NET", it is a ".NET serializer that happens to use protocol buffers", so the emphasis was to be familiar to Microsoft.NET users [26]. By creating a .NET serializer that uses protocol buffers, the library is consistent with the rest of .NET serializers and simplifies its usage. On the other hand, if a new serialization process based on the protocol buffers definition would have been implemented, its usage would be unnatural when compared the serialization mechanisms provided by the .NET Framework.

3.2.2 Content Identification

Inside the ICON Framework, Content is identified by its Name, no matter where it came from or which machines were used during its transport.

A Name is a hierarchical structure of components of any length that once combined identifies Content, part of Content or even a collection of data. A Name can be represented by an Uri, where a scheme and a sequence of segments are all that it is needed to define a Name.

A Name can be divided in three main components:

- A scheme
- A list of components that identify a location and / or a concrete content
- An optional content segment number

Each one of this components are expressed by their UTF8 binary serialization excluding any forward slash character "/" that might be present on any of its list of components that are used in the Uri representation of a Name.

The following table illustrates Name examples using their Uri representation that point to Content, a part of a Content or a collection of data:

Name	Name Meaning
ccn4d:Images/Desert.jpg	A Name that identifies the Desert.jpg Content on the Images location using the ccn4d scheme.
ccn4d:Images/Desert.jpg/1	A Name that identifies the first segment of the Desert.jpg Content on the Images location using the ccn4d scheme.
ccn4d:Images/	A Name that identifies any Content present on the Images location using the ccn4d scheme.

Table 1 – ICON Framework Name examples

3.2.3 Message Types

As mentioned previously, the ICON Framework requires only two types of messages to allow data interchange between its nodes, an Interest message and a Content message.

An Interest message is the message that a node sends to the network to manifest interest for some Content. The interest is expressed by the Name present on the Interest message, so as a Name can represent a specific Content, a chunk of it, or a collection of some data, when an

interest message is sent to the network the result that the node will get back will depend on the Name that was expressed on the Interest that was sent.

As the name implies, Content messages are messages were the requested data, expressed by an Interest message sent to the network, is transported back to the requester node. Like the Interest message, a Content message contains the Name of the data it transports, but extends it to contain the data in the form of a byte array, which carries the effective message content.

The current definition of these messages includes just the minimum structures to allow communication between ICON nodes. To further improve the protocol, other structures might be required in both types of messages. For example, if we need to provide additional information to allow an improved processing of Interest messages – filtering messages for instance – the Interest type message needs to be extended to support such additional processing. Another example of an area where the protocol might be extended relies on the protocol being able to attest that all communications are secured and not violated during transport, to support such scenario the Content message structure needs to be extended to support publisher information and a signature of the data it contains. Such improvements, which can be found in the CCNx protocol [27], can be added later on and extend the current message definitions.

3.2.4 Data exchange between nodes

Communications using the ICON Framework are controlled by the receiver. When transmitting data between nodes it is the consumer that initiates a communication by sending an Interest message to the network, the consumer also has the responsibility to retransmit the Interests if it is still interested in the data that has not arrived yet.

The ICON Framework simplifies the responsibility to relay Interests. On its Network Engine core there is functionality to automatically retransmit Interests during a configurable period of time, and within randomly generated time intervals. Applications using the ICON Framework only need to retransmit Interests if this period times-out, at which point the Network Engine will no longer issue Interest messages. This is an improvement in relation to what is implemented by the CCNx and Haggle projects, where such retransmission support does not exist and must be assured directly by the applications using these projects.

Any node that receives an Interest, and has the capacity to deliver the Content requested by the Interest, can send one Content message in response to a single Interest message. There is only one Content message delivered per Interest messages received, even if the node having the content can deliver more than one message for the requested Interest. Like in the CCNx protocol, this one-for-one mapping between Interest and Data messages maintains a flow balance that allows the receiver to control the rate at which data is transmitted from a sender, and avoids consuming bandwidth by not sending data to nodes when it is not wanted [27].

As such, when a consumer needs a set of data that requires multiple Content messages it must issue at least the same number of Interests to the Network. To improve data transmission, the consumer can send more than one Interest message in parallel even before receiving any Content message. This can only be done if the consumer knows how to specify the Name of the Contents that need to be asked to the network in the form of Interests. Once the messages are received, the consumer will use the Name and the Data of the Content message to rebuild the sets of data he asked for.

Sending in parallel multiple Interest requests to the network can be used in several scenarios. As an example, let us think of an application that displays images that are stored outside of the device where the application is running. Such application could issue in parallel several Interests, each representing a single image request. Another option could be to issue in parallel several Interests for different segments of an image, by specifying the segment number on the Name of the Interest messages sent to the network. And a third option would be the mixture of the two previous approaches, where the app could be asking for several images and several segments per image in parallel and process the Contents received accordingly.

As long as the network bandwidth does not get exhausted, such type of communication can greatly improve the data throughput between nodes. Such improvement in communications is especially noticed when some of the nodes can start to deliver Content messages based on data cached locally on their storage as a consequence of a previously similar Content message arrival on such nodes. The graphics presented in the Testing the ICON Framework section provide evidences of this result.

3.3 Node Model

As mentioned previously, modularity was always a key concept since the early design of the ICON Framework architecture. Therefore, inside the engines we can find components that further divide the responsibilities of each engine as illustrated on Figure 8.

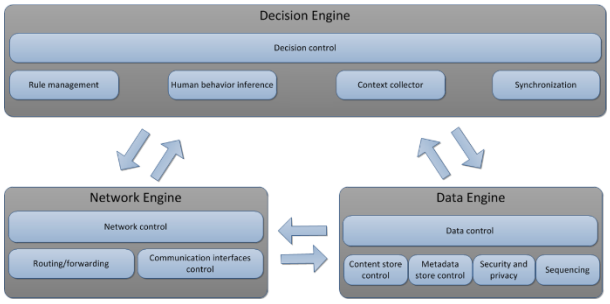


Figure 8 - ICON Framework architecture design

By having a kernel or Node that is composed by modular engines, it means that any of these engines can be customized or even completely replaced for different devices or runtime environments. The ICON Framework Implementation section, where details are given regarding the implementation of the ICON Framework, has details on how such modularity was supported and can be manipulated.

Besides the modularity found at the engines level, represented by the different framework engines, each of these engines contain a set of components to further refine their responsibility. This approach is similar with the architecture followed by the Haggle project, where a set of managers are responsible for specific tasks [28], but instead of having a bigger set of managers all interacting with a kernel, the ICON Framework kernel is in fact the composition of these three engines.

Engines also need to communicate with each other to provide the functionality expected by the content centric network node. Interactions between node engines are done in process using the ICON Framework API that provides access to properties, methods and functions on each of the components it defines. As every engine has access to the Node, and the Node has access to every engine, communication can be done from any engine to every engine.

3.3.1 Decision Engine

The Decision Engine component is meant to be the component responsible to take the major decisions that a Node needs to make. When compared to the CCNx protocol, the Decision

Engine plays a similar role as the Strategy Layer, but where extended functionalities can be found. The Decision Engine also includes behaviors that can be found in the Haggle Resource Manager. Here, the Context Collector can be used to collect information about the device, like power battery, available storage, bandwidth, etc., and then define rules using the Rule Management to change the behavior of the node and ensure that the resources are used appropriately and in an efficient way.

An implementation of the Decision Engine was out of scope of the current dissertation, in fact, as the Decision Engine plays an important role in the ICON Framework Node Model, this work is being further investigated in the SITILabs research center. In any case, the Decision Engine component should be able to:

- Manage rules to route / forward network messages
- Take decisions upon human behavior interactions
- Collect contextual information
- Synchronize data

By gathering and analyzing the data that flows on the Node, combined with the contextual information collected from the device where the system is running including the human interactivity, gives enough data to allow the Decision Engine to be capable to interact with the Network Engine to enable or disable communication interfaces, manipulate the forwarding information table and adjust the Node to better serve and receive data. Also, the Content Store control in the Data Engine, where the local cache is persisted and manipulated, can be controlled using synchronization policies defined in the Decision Engine to ensure a good usage of the local cache. For instance, a synchronization policy might determine that if the node could not store all Content messages of some data, then no Content messages should be stored for that data, and the Node storage should release all data already persisted for this content to provide more space to other contents. Another option would be to define least-recently-used or least-frequently-used policies to manage the local store.

The Decision Engine should be seen as the brain of the Node that is capable to control its communications via the Network Engine or the way it handles the data using the Data Engine.

3.3.2 Data Engine

The Data Engine responsibility is to handle the Node data. Not only this engine handles data related to the Content objects it receives and processes, using its Content Store Control, but it is also be responsible to manage two of the most important data sets of the ICON Framework architecture, the Forwarding Information Base or simply FIB using the FIB Manager, and the Pending Interest Table or simply PIT using the PIT Manager. It is also responsible to segment data for Contents who need to be divided or joined, and it uses the Sequencing component to handle such work. Finally, the Security and Privacy component is where data should be encrypted and/or signed to ensure the security and privacy of the data handle by the node.

3.3.2.1 Content Store Control

The Content Store Control responsibility is to manage all the Content messages that can be stored in a Node. Content messages are stored and indexed by their Name to speed up data retrieval when needed. A simple implementation of a Content Store Manager can be based on a memory dictionary to store Content messages indexed by their Name.

The policies to manage the persistence of data segments in the content store should be defined by the Decision Engine. This provides the capacity to have different behaviors based on the device, or even due to environment conditions that lead to changes in the data storage policies. The Handle Content Messages section where the pipeline to process Content messages is described, has a reference to the where the Decision Engine participation is made. Cache hit and misses are described in the Handle Interest Messages section. Essentially a cache hit will immediately return a Content message from the cache, while a cache miss will force the execution pipeline of the Interest message to proceed.

3.3.2.2 Metadata Store Control

The Metadata Store Control is composed by the PIT Manager and the FIB Manager, which are the repository of the pending interest requests and the forwarding information base tables of the Node.

3.3.2.2.1 PIT Manager – Pending Interest Table Manager

The PIT is essentially a dictionary of Names for the Interest requests who arrived at the Node. For each Name entry in the PIT there is the list of Communication Interfaces who ask for it, and to where the Content message once received or produced, should be sent to.

On the ICON Framework, the PIT Manager is responsible to manage this table. To do it, the PIT Manager needs to be able to interact and query the status of the PIT entries it manages. As such, the following operations must be executed by the PIT Manager:

- Add pending Interests – To add Interests and the Communication Interface to reply to, when the Interest arrives at the Node and it waits to be satisfied
- Remove pending Interests – To remove Interests when they are satisfied, or when the time available to satisfy them expires
- Get Communication Interfaces for pending Interest – List of Communication Interfaces who are waiting for the Interest to be satisfied

A simplified view of the PIT is shown below:

Interest Name	Expires	Communication Interfaces
ccn4d:Images/Desert.jpg/1	2012-12-23 19:00:04.23498273	Communication Interface 1 Communication Interface 2
ccn4d:Images/Desert.jpg/2	2012-12-23 19:00:04.33498273	Communication Interface 1
ccn4d:Images/Desert.jpg/3	2012-12-23 19:00:04.43498273	Communication Interface 2
ccn4d:Images/Cat.jpg/10	2012-12-23 19:00:06.23498273	Communication Interface 1 Communication Interface 2 Communication Interface 3
ccn4d:Images/Cat.jpg/11	2012-12-23 19:00:06.33498273	Communication Interface 2 Communication Interface 3

Table 2 – Simplified view of the PIT

3.3.2.2.2 FIB Manager – Forwarding Information Base Manager

The FIB is where a Node stores information regarding how it forwards Interests to other Nodes in order to receive the respective Content messages. Typically, FIB entries are defined in application settings and configured upon application startup, but the ICON Framework API also supports dynamic manipulation of the FIB, allowing applications to adjust or completely reconfigure the FIB. The Decision Engine can also manipulate the FIB by using a set of rules or in reaction to environment changes on the Node.

The FIB Manager role is to provide access to FIB table manipulation and support the following actions:

- Register routes to Communication Interfaces based on a Name – it can be a full Name, including the segment number, or just a Name Prefix
- Un-register route to a Communication Interface
- Get the Communication Interfaces for the given a Name
- Get the full list of registered routes given by Name and respective Communication Interfaces

3.3.2.3 Security and Privacy

Security and privacy are becoming increasingly important with regard to the transfer of data between network devices. Securing messages at the application level increases the security level when compared to the security level of current IP communications, which is mostly based in TLS/SSL and is done at the session level of the OSI network stack, which stays on top of the transport layer.

Applications using the ICON Framework have the capacity to secure their messages as data is added to Content messages. They should start by using an asymmetric key to send a symmetric key to be used later on for the data transmission. This provides an increased level of security, as applications will be able to encrypt the data segments that travel inside the Content messages. This is not a problem for the protocol since the protocol is completely agnostic to the data it carries, and it has no need to look at this information to read or change it. Due to the modularity of the ICON Framework, applications will be able to control if data encryption and/or signing is needed, by using or not the Security component.

An important point to mention is that all data stored locally on a Node has no information of their origin or destination, due to the fact that the data packets do not contain any kind of information that could identify the source or the destination of the data. This is achieved just because the protocol to transfer data is based on content centric networks where no host information relies on the data packets transferred between the network devices where the data flows, so even if the data is not secured, the data transfer protocol provides already

some degree of security and privacy due to its intrinsic nature of forwarding data through the network which does not identity the consumer and the producer of the data being transferred.

3.3.2.4 Sequencing

The Data Engine through its Sequencing component is also able to create Content message data segments, join and write them whenever needed. This is necessary whenever the node needs to transfer, or store data locally, that do not fit in the network packet size. The Sequencing component is also used to rebuild data from Content messages when data is received from several Content messages as a result of the data segmentation that was done when the data was produced.

The ICON Framework handles all the effort related to data segmentation on Content messages so that consumers of this framework do not need to have the burden of handling with such processing.

3.3.3 Network Engine

All network communications between distinct Nodes are done via the Network Engine. This means that nothing is sent or received from other Nodes without being processed by this component.

To provide such communication capabilities, the Network Engine allows communication interfaces to be registered within the interface communication manager reference exposed by the engine.

The communication interface manager will use the registered set of communication interfaces to send and receive Interest and Content messages.

The communication interface manager has a nuclear role in the Network Engine, besides providing support for interface communication registration, through him any Node is able to:

- Issue requests to the network – send Interest messages
- Handle and process all incoming messages
    - Interest messages
    - Content messages

3.3.3.1 Handle Interest Messages

When an Interest Message is received, the Network Engine, via its communication interface manager, will process the message as follows:

- Makes a Content search on the local store. If Content is found that satisfies the incoming Interest a Content message is returned containing the Content object found on the local store. If not, proceed to the next step.
- Check if the node's Data Engine is a content generator – typically only if it is a network leaf node. If it is, try to generate the Content, if succeeded, segment and store its segments in the local store to satisfy further Interests. Take the first segment of the Content produced and send a Content Message in response. If not, proceed to the next step.
- Do a lookup on the Pending Interest Table of the Data Engine for a matching pending Interest. If a pending Interest is found, add the Communication Interface to where the response should be sent to the list of Communication Interfaces of the PIT. If not, proceed to the next step.
- Do a lookup on the Forwarding Information Base of the Data Engine to see if the interest can be forward to another node. If the Interest can be forward, add a PIT entry containing the pending Interest and the Communication Interface where the response should be sent. If not, proceed to the next step.
- Ignore the Interest message as the node cannot process it.

Figure 9 shows how the Interest message is handled by the ICON Node.

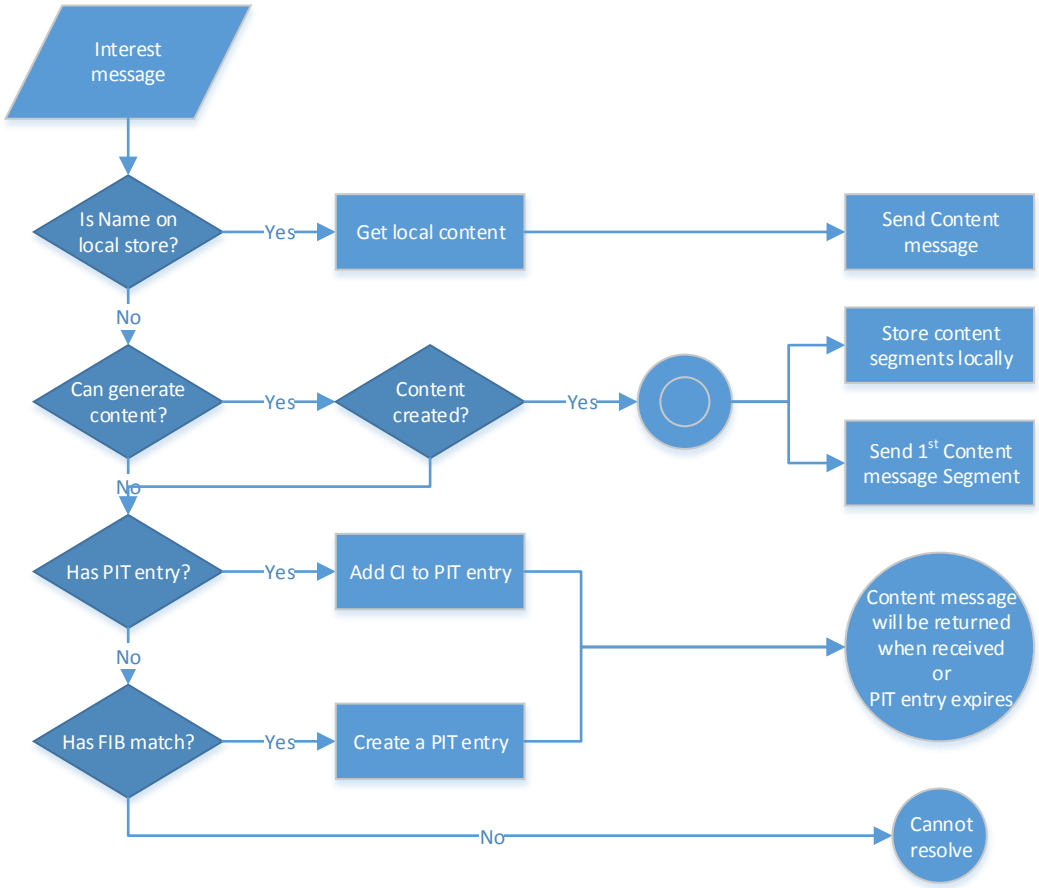


Figure 9 - Handle Interest message

3.3.3.2 Handle Content Messages

When a Content Message is received, the Network Engine via its communication interface manager will process the message as follows:

- Do a lookup on the Pending Interest Table of the Data Engine for an Interest whose Name matches the Content Name. If a matching PIT entry was found, remove it from the PIT, get all the Communication Interfaces registered with it, and send the Content Message to each one. Ask the Data Engine if the Content can be stored locally, use the content store manager to store the Content object if a positive response was received. If the PIT lookup was not successful, proceed to the next step.
- Ignore the Content message as the node was not waiting for it, and unsolicited Content messages must be ignored.

Figure 10 shows how the Content message is handled by the ICON Node.

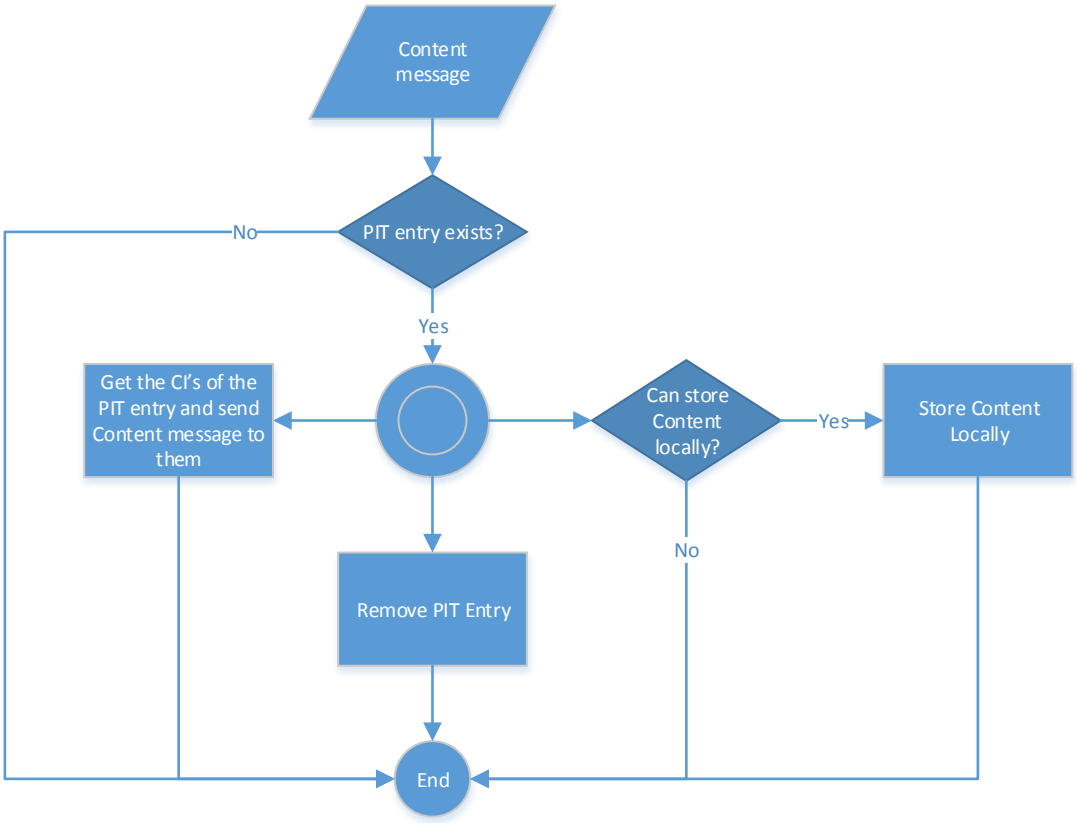


Figure 10 - Handle Content message

4 ICON Framework Implementation and Usage

The ICON Framework was developed using Microsoft Visual Studio 2010 and Microsoft Visual Studio 2012. The source code was written in Microsoft C# against the Microsoft.NET Framework 4.0 version ensuring Mono compatibility.

Early in the implementation phase, series of experiences were conducted to evaluate the capacity to produce code which could be executed on all of the proposed platforms:

- Windows using the Microsoft.NET Framework
- Linux using Mono
- Mac OS using Mono
- Android using Monodroid
- Windows Phone using the Microsoft.NET Framework
- iOS using MonoTouch
- Embedded platforms using Microsoft.NET Micro Framework

The experiences consisted in trying to create small code blocks that could prove the capacity of the platforms to use generics [29] [30], dependency injection [31], multithreaded support and sockets programming. The test results showed the need to reevaluate the target platforms for the prototype because some of the platforms did not provide the required functionality to allow the development of a solution on those targets. The Windows Phone operating system available at the time did not provide support for sockets, and this was a requirement since the Communication Interfaces implemented on the prototype were based on sockets using the UDP protocol. Microsoft.NET Micro Framework also has a lot of cuts when compared with the full Microsoft.NET Framework, for instance support for generics is missing, which would increase substantially the development time required to support dependency injection and as a consequence, modularity. As such, both of these platforms were excluded from the target platforms of the ICON Framework for the current prototype implementation. Android and iOS with Mono for Android and MonoTouch were also evaluated, but due to lack of equipment to run on those platforms – using emulators was not an alternative – the final solution was not tested against them, although the source code is compiled successfully against the Mono for Android platform. Despite the fact that the solution was not tested on those

platforms, the importance of being able to compile the source code against the Mono for Android means that there should be no problem to have the ICON Framework running on this platform.

4.1 Implementation

The prototype implementation of the ICON Framework built a full network Node, but with a subset of the full framework. For the prototype some shortcuts where made, most noticeable, the Decision Engine was implemented just to provide a basic support for the prototype usage, where the components that define its behavior where not implemented, or where just implemented to provide a basic functionality to support the prototype. On the Data Engine, the Security and Privacy component was not implemented also because it was not a fundamental component to demonstrate how the ICON Framework could be implemented and it is a component that can be added has the framework implementation evolves. Figure 11 shows the engines and components implemented, and the messages switched by the Nodes.

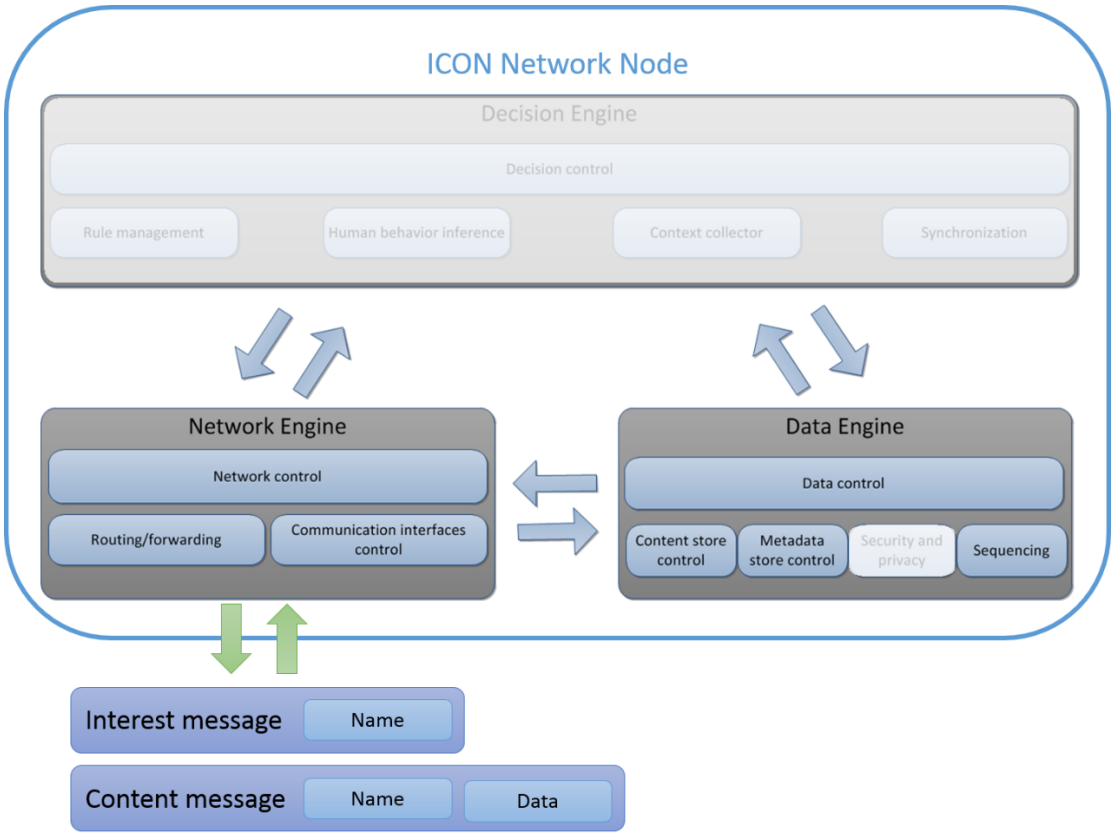


Figure 11 - ICON Framework prototype implementation

4.1.1 Decision Engine

As the Decision Engine components were not fully implemented, a basic Decision Engine was created just to provide a set of basic functionality to support the current implementation. As an example, there is the necessity to know if a Content message should be stored locally or not, to answer this, the implemented Decision Engine provides a function that allows all Content messages to be stored locally. Another example is the selection of Communication Interfaces of the Node to be used in communications, which, for the current implementation, will always use the first Communication Interface available.

4.1.2 Network Engine

The prototype delivered with this dissertation has implemented a fully functional Network Engine with a Communication Interface control manager that allows: registration and un-registration of Communication Interfaces; get the list of registered Communication Interfaces; and has the capacity to send interest requests to the network using the registered Communication Interfaces. The C# interface declaration that supports this functionality is presented on Figure 12.

```
/// <summary>
/// Implementors of the Communication Interface Manager Interface are responsible handling all the communications done by the node.
/// </summary>
3 references
public interface ICommunicationInterfaceManager
{
    /// <summary>
    /// Provides the list with the current registred communication interfaces
    /// </summary>
    /// <returns>List with the current registred communication interfaces</returns>
    2 references
    IEnumerable<ICommunicationInterface> GetsRegistredCommunicationInterfaces();

    /// <summary>
    /// Registers a Communication Interface to be able to listen for communications
    /// </summary>
    /// <param name="ci">The Communication Interface that will listen for communications</param>
    4 references
    void RegisterCommunicationInterface(ICommunicationInterface ci);

    /// <summary>
    /// Unregisters a Communication Interface so that messages are no longer processed by the communication interface
    /// </summary>
    /// <param name="ci">The Communication Interface that will be unregistred</param>
    1 reference
    void UnregisterCommunicationInterface(ICommunicationInterface ci);

    /// <summary>
    /// Requests an interest
    /// </summary>
    /// <param name="interest">The interest being requested</param>
    /// <param name="timeout">Specifies the time span to wait for the interest to be satisfied</param>
    /// <param name="contentArrivedCallback">The callback that will be invoked when the Content arrives</param>
    5 references
    void Request(Interest interest, TimeSpan timeout, Action<Content> contentArrivedCallback);
}
```

Figure 12 - C# interface definition of the Communication Interface control manager

The implemented Network Engine also controls network communications, where Interest and Content massages are sent and received. A detailed description on how these messages are processed was described previously on sections Handle Interest Messages and Handle Content Messages, where Figure 9 and Figure 10 were also presented showing the respective flowcharts.

The ICON Framework automatically resends Interest messages that are on the PIT waiting for the corresponding Content message to arrive. To support this, there is a permanent loop running on a worker thread of the Node process that is responsible to query the PIT for interests older that a configurable threshold. For the entries found, the network control will resend the Interest message to the network, removing such effort from the applications that use the ICON Framework. Figure 13 shows the code excerpt where the worker process is declared, and where we can see the Data Engine being used by asking the Pending Interest Table for the pending interests older than a certain amount of time, which can be configured at the application level.

```
// Worker thread to resend Interests that are pending in PIT for more than the ResendInterestOlderThan value...
ThreadPool.QueueUserWorkItem(_ =>
    {
        var rnd = new Random(DateTime.Now.Millisecond);

        while (true)
        {
            Thread.Sleep(rnd.Next(1, 10));    // Randomize the sleep to avoid CPU starvation

            var pendingInterests = Node.Current.DataEngine.PendingInterestTableManager.GetPendingInterestsOlderThan(ResendInterestOlderThan);
            foreach (var interest in pendingInterests.AsParallel())
            {
                // Randomize the sleep to avoid collisions
                Thread.Sleep(rnd.Next(Node.ConfigMinRandomSleepBeforeResendInterest, Node.ConfigMaxRandomSleepBeforeResendInterest));

                RequestInternal(interest, TimeSpan.Zero, null);
            }
        }
    });
```

Figure 13 - Worker thread to resend Interest messages

Routing and forwarding control is handled by the Network Engine using the Routing and Forwarding component. Here the FIB table, that exists on the Metadata Store control of the Data Engine, can be managed to: allow new routes to be registered given a Name and a pair of Communication Interfaces that define the source and destination of communications; unregister routes given the same set of information; get the route definition for a Name; and get a dictionary with a list of Names and routes defined for each one. In the current implementation, the FIB table was configured using a XML document where, along with the set of available interfaces, we can find the forwarding information between Names and Communication Interfaces. As already mentioned, future developments, where a Decision Engine has more

functionality than the provided, can avoid this manual configuration, as the Decision Engine will be able to define the information configured on this file. Figure 14 presents an example of a configuration file that can be used to configure the FIB using the API described in Figure 15.

On Figure 14, three Communication Interfaces are defined, the first one is the listener where Interest and Content messages are received, and the other two are Communication Interfaces to other Nodes that are configured to be listening on the values provided in this configuration file. Forwarding information is also set, where we can see that all Interests with a Name 'Images/Desert.jpg' received on the first Communication Interface will be forwarded to the third Communication Interface, while all Interests with a Name Prefix 'Images' received on the first Communication Interface will be forward to the second one.

```
<Register>
  <CommunicationInterfaces>
    <CommunicationInterface key="1" host="192.168.227.130" port="8000" />
    <CommunicationInterface key="2" host="192.168.227.140" port="8000" />
    <CommunicationInterface key="3" host="192.168.227.150" port="8000" />
  </CommunicationInterfaces>

  <Listen>
    <CommunicationInterfaceKey value="1" />
  </Listen>

  <Forward>
    <Name uri="ccn4d:Images">
      <CommunicationInterfaces forwardTo="2" receiveOn="1"/>
    </Name>
    <Name uri="ccn4d:Images/Desert.jpg">
      <CommunicationInterfaces forwardTo="3" receiveOn="1"/>
    </Name>
  </Forward>
</Register>
```

Figure 14 - Configuration file to configure Communication Interfaces and the FIB table

```
/// <summary>
/// Implementors of the Forwarding Information Base Manager Interface are responsible for managing a Forwarding Information Base table.
/// </summary>
3 references
public interface IForwardingInformationBaseManager
{
    /// <summary>
    /// Registers a Communication Interface to be used when transfering information with the provided Name
    /// </summary>
    /// <param name="name">The Name to whom the route will be registered</param>
    /// <param name="communicationInterfacePair">The Communication Interface Pair to be registered</param>
    4 references
    void RegisterRoute(Name name, CommunicationInterfacePair communicationInterfacePair);

    /// <summary>
    /// Unregisters a Communication Interface that was previously registered
    /// </summary>
    /// <param name="name">The Name that was used to register the route</param>
    /// <param name="communicationInterfacePair">The Communication Interface Pair used during the registration process</param>
    1 reference
    void UnRegisterRoute(Name name, CommunicationInterfacePair communicationInterfacePair);


    /// <summary>
    /// Returns the list of Communication Interfaces that are registered for the provided Name
    /// </summary>
    /// <param name="name">The Name used to register the Communication Interface Pair being requested</param>
    /// <returns>An IEnumerable with the pair of Communication Interfaces found</returns>
    3 references
    IEnumerable<CommunicationInterfacePair> GetRegisteredCommunicationInterfaces(Name name);

    /// <summary>
    /// Gets the registered routes for each name
    /// </summary>
    /// <returns>List with the registered routes for each name</returns>
    2 references
    IDictionary<Name, List<CommunicationInterfacePair>> GetRegisteredRoutes();
}
```

Figure 15 - C# interface definition of the FIB manager

4.1.3 Data Engine

In terms of the Data Engine, all the major components where implemented except the Security and Privacy. Although most components where implemented, other might be added in the future as alternatives to the current ones. For instance, the Content Store control component that was implemented uses in memory storage that served the purpose of the prototype, but a new implementation that stores data on a solid state drive should be needed if a lot of data is to be stored on the Node. As the ICON Framework is modular, having more than one implementation of these components is good, because at the end, each application that is using the framework can choose the one to use, or, have rules defined on the Decision Engine to switch between components if that is the most appropriate option. Figure 16 shows the API used by the Content Store control component to manage the local store.

```
/// <summary>
/// Implementors of the Content Store Manager Interface are responsible to handle content data that is stored on the node.
/// </summary>
3 references
public interface IContentStoreManager
{
    /// <summary>
    /// Add content to the local store
    /// </summary>
    /// <param name="content">The Content to be stored localy</param>
    /// <param name="contentStoredCallback">Callback to be invoked after each content is stored on the local store</param>
    4 references
    void AddContent(Content content, Action<Content> contentStoredCallback);

    /// <summary>
    /// Adds all the Content objects provided in the IEnumerable to the local store
    /// </summary>
    /// <param name="contentList">The IEnumerable of Content objects to be stored on the local store</param>
    /// <param name="contentStoredCallback">Callback to be invoked after each content is stored on the local store</param>
    1 reference
    void AddContents(IEnumerable<Content> contentList, Action<Content> contentStoredCallback);

    /// <summary>
    /// Returns the Content for the specified Name if one exists
    /// </summary>
    /// <param name="name">The Name of the Content to be returned</param>
    /// <returns></returns>
    3 references
    Content GetContent(Name name);

    /// <summary>
    /// Get the Content objects identified by the specified filter
    /// </summary>
    /// <param name="filter">Filter predicate to identify the Content objects</param>
    /// <returns>List of Content objects that matched the filter</returns>
    3 references
    IEnumerable<Content> GetContents(Predicate<Name> filter);

    /// <summary>
    /// Removes from the local storage the Content with the specified Name
    /// </summary>
    /// <param name="name">The Name of the Content to be removed</param>
    1 reference
    void RemoveContent(Name name);
}
```

Figure 16 - C# interface definition of the Content Store component manager

The Data Engine, via the Metadata Store control component, is where the tables for storing pending Interests and forwarding information base are defined. Details on the FIB table manipulation where described on the previous section, during the configuration of Communication Interfaces and the management of the routing and forwarding information table. So, on this section we will address the management of the PIT, which is the other component of the Metadata Store control component.

```
public interface IPendingInterestTableManager
{
    /// <summary> ...
    2 references
    void AddPendingInterest(Interest interest, TimeSpan timeout, ICommunicationInterface ci, ICommunicationInterface ciWhereInterestWasForwarded);

    /// <summary> ...
    2 references
    void AddPendingInterest(Interest interest, TimeSpan timeout, ICommunicationInterface ci);

    /// <summary> ...
    2 references
    void AddPendingInterest(Interest interest, TimeSpan timeout, Action<Content> callback, ICommunicationInterface ciWhereInterestWasForwarded);

    /// <summary> ...
    2 references
    IEnumerable<ICommunicationInterface> GetPendingInterestCommunicationInterfaces(Interest interest, bool removePendingInterest);

    /// <summary> ...
    2 references
    IEnumerable<Action<Content>> GetPendingInterestCallbacks(Interest interest, bool removePendingInterest);

    /// <summary> ...
    1 reference
    void RemovePendingInterest(Interest interest);

    /// <summary> ...
    2 references
    bool HasPendingInterest(Interest interest, TimeSpan timeout, bool createIfNotExists);

    /// <summary> ...
    3 references
    IEnumerable<Interest> GetPendingInterestsOlderThan(TimeSpan timespan);

    /// <summary> ...
    2 references
    Dictionary<Interest, PITEntry> GetPendingInterestsDetails();

    /// <summary> ...
    2 references
    void UpdatePendingInterestRegistrationDateTime(Interest interest);
}
```

Figure 17 - C# interface definition of the PIT manager

Figure 17 shows the definition of the C# interface of the PIT manager. Here we can see methods to add, get, remove and check existence of Interests on the pending interest table. We can also see the function ‘GetPendingInterestCommunicationInterfaces’ which provides a list of Communication Interfaces of a PIT entry, and optionally removes them from the PIT. As seen in the Handle Content Messages section, in particular by reading the flowchart presented on Figure 10, if a PIT entry is found, the system will need to remove it to avoid other Content messages received for the same Interest to be processed, so this function atomically queries the PIT and optionally removes the entry if one is found, which simplifies concurrent read and write access to the PIT. Note that multiple Content messages for the same Interest can arrive to the Node, if for any reason more requests were sent for the same Interest before the

corresponding Content message has arrived. These multiple Interests requests for the same Interest can be done by the application using the ICON Framework, or more naturally, by the framework itself during the execution of the worker process that is constantly monitoring the PIT to resend Interests that were not satisfied in a timely manner, as specified by the application. The definition of this worker process, responsible for monitoring the PIT to resend Interest requests to the network if these are not satisfied in the expected interval, was presented in the previous section and its implementation was showed in Figure 13. Besides this worker process, there is another worker process that is responsible to remove from the PIT all Interests that were not satisfied at all, which means Interest requests that have expired even after the Node has made efforts to resend them. Figure 18 shows the implementation of this worker process, where we can see that the PIT is filtered to get all PIT entries for which the 'ExpiresAt' value is lower or equal to the current time. All the entries found will be removed. If the application still needs the Content messages for these Interests, it is then its responsibility to issue the Interest requests again.

```
// Worker thread to remove Interests from the PIT who are expired...
ThreadPool.QueueUserWorkItem(_ =>
{
    while (true)
    {
        Thread.Sleep(CheckTimedOutPendingInterestsInterval);

        foreach (var interest in PIT.Where(kvp => DateTime.Now >= kvp.Value.ExpiresAt).Select(kvp => kvp.Key).ToArray().AsParallel())
        {
            PIT.Remove(interest);
        }
    }
});
```

Figure 18 - Remove expired PIT entries worker process

The last component of the Data Engine that was required and implemented on the prototype was the Sequencing component. This component has two responsibilities:

- Given a content producer function, create Content messages that can be transmitted on the network, ensuring that the serialization size of the produced messages is guaranteed to be lower or equal to the network packed size defined by the application. A content producer function is a function that can generate content, for instance a function that reads a file from disk and expose the file content as a byte array, can be a content producer function;
- Given a set of Content messages, write their data to a stream from where data can be read and processed.

Figure 19 shows the C# interface definition of the Sequencing component. The 'CreateSegment' and 'WriteContents' functions provide the responsibilities of the component.

On the 'CreateSegment', the third argument is a pointer to a function that will get the Name extracted from the Interest provided on the first parameter, and an integer that represents the maximum data length in bytes for the data block of the Content message that is to be produced. This function should then produce a byte array containing the data for the Content message, where the length of the byte array might not exceed the size of the second parameter passed to the pointer function. The length of the byte array is determined based on the size of the Name and the network packet length, so the longer the Name the lower the size of the data block of the Content message.

The 'WriteContents' does the reverse operation, but from a set of Content messages. The set of Content messages must be provided with the right ordering sequence. Otherwise the stream will get unordered data, which would not produce the desired result. It is the application responsibility to provide the right order, because it is the application that has defined a Profile to transmit data, and as such, it is the application that knows how to order Content messages correctly. On this point, further work can be developed to try to use the segment number that a Name might have, and use this number to sort the Content messages if they are not sorted at the application level. This introduces more complexity, as there are scenarios where not all content messages can be received and sorted, before the application needs to start to process them. Live video streaming is one of those examples.

```
/// <summary>
/// Implementors of the Segmenter Interface are responsible for creating segments that fit into the transmission packets
/// </summary>
3 references
public interface ISequencing
{
    /// <summary>
    /// Creates a segment based on the Interest provided, func to produce data and func to determine if it's the last segment
    /// </summary>
    /// <param name="interest">The Interest from where the Content Name will be extracted</param>
    /// <param name="blockMaxLength">The network segment block maximim length</param>
    /// <param name="dataProducer">The function that will generate the Content Data, where Name is the Name of the Content being produced, i
    /// <param name="isLastSegment">The function that will be called to specify if the Content should be flaged as being the last segment</p
    /// <returns>A Content that can be sent to the network, because it's serializaed length will not be higer than the maximum network segme
    2 references
    Content CreateSegment(Interest interest, int blockMaxLength, Func<Name, int, byte[]> dataProducer, Func<Name, int, bool> isLastSegment);

    /// <summary>
    /// Writes the Contents Data to the provided stream. Contents will be sorted based on their segment number before writting.
    /// </summary>
    /// <param name="contentSegments">The segments from where tha Data will be extracted</param>
    /// <param name="writeToStream">The stream where the Data will be written</param>
    2 references
    void WriteContents(IEnumerable<Content> contentSegments, Stream writeToStream);
}
```

Figure 19 - C# interface definition of the Segmenting component

4.2 Source code and the project structure

The Figure 20 shows the source code of the C# project structure of the ICON Framework. In the project structure we can identify three major areas:

- Dependency Injection – Contains code used to register and resolve network modules
- Network – Defines the Node and includes the default Data Engine, Decision Engine and Network Engine modules. If needed, these can be replaced with custom assemblies to allow customizations
- Protocol – Where the types required to support the protocol are defined

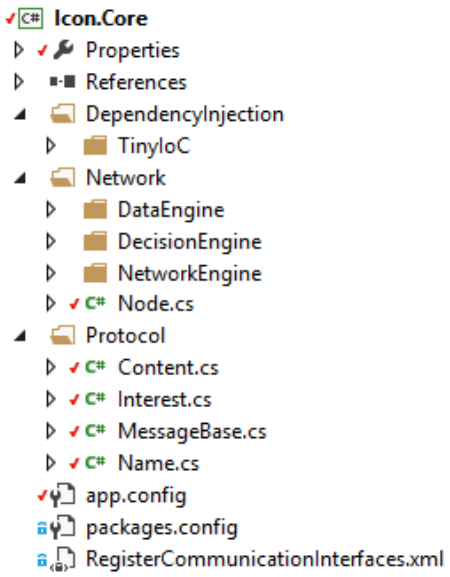


Figure 20 - ICON Framework project structure

4.2.1 Dependency Injection

In the web article Inversion of Control Containers and the Dependency Injection pattern [31], the author describes how the Dependency Injection works pointing out some of the advantages and disadvantages of it usage. To summarize the dependency injection pattern in a phrase, we can say it provides a way to inject dependencies on running code just by having the required types registered against contracts. The reason to use this pattern was to provide support for modularity in the ICON Framework.

In fact, there are lots of implementations of the pattern for the Microsoft.NET Framework, some more complex than others. The dependency injection requirements for the ICON Framework were quite simple but it is needed to run on the target platforms mentioned previously, so the choice was to look up for an implementation of a dependency injection solution that could be used across the platforms for which the ICON Framework is being targeted. The source code should also be available to allow any customization, if needed.

The choice was to use the TinyIoC source code, more specifically, the version that was available on March 13, 2012 at https://github.com/grumpydev/TinyIoC. TinyIoC is an Inversion of Control container and provides basic Dependency Injection capabilities.

The type registrations of the Node Modules are all done using the TinyIoC, which brought the capacity to register and dynamically load types. TinyIoC is also used internally to automatically register assembly types and inject them when required.

Consumers of the ICON Framework can use the application settings to configure the module types to be loaded, being the module types provided with the framework the default ones. Below is an example of registering the modules types in the application configuration file:

```
<!-- Node Modules Type Registration -->
<add key="DecisionEngineType" value="Icon.Core.Network.DecisionEngine.DecisionEngine, Icon.Core"/>
<add key="NetworkEngineType" value="Icon.Core.Network.NetworkEngine.NetworkEngine, Icon.Core"/>
<add key="DataEngineType" value="Icon.Core.Network.DataEngine.DataEngine, Icon.Core"/>
```

Figure 21 - Registering Module types

Here we can see that the types being registered are also from the ICON Framework, but others could have been specified to support specific functionalities if needed. This provides a great level of customization at a very low level, since consumers of the framework can replace specific parts in order to meet their needs. For instance, different platforms or even client applications will probably prefer to redesign the Decision Engine provided – since the provided implementation has just basic set of functionality. This can be done without the need to change the current implementation and just by registering a Decision Engine module from a different assembly. The same option is valid for the other two engines.

4.2.2 Protocol

In the Protocol we can find the definition of the types required for the ICON Framework network protocol. These types are also the types who will be serialized by the protocol-buffers serializer and consequently transferred on the wire whenever Interest and Content messages are sent over the network. The defined types are:

- Name – This class defines a Name to be used in an Interest or Content message and is used to identify the information being requested or sent.
- MessageBase – This is the base class for all the messages that are transferred by the protocol: Interest and Content
- Interest – Interests represent the information being requested over the network
- Content – Content objects are those who transport the information over the network

The following diagram is detailed view for each of types mentioned above:

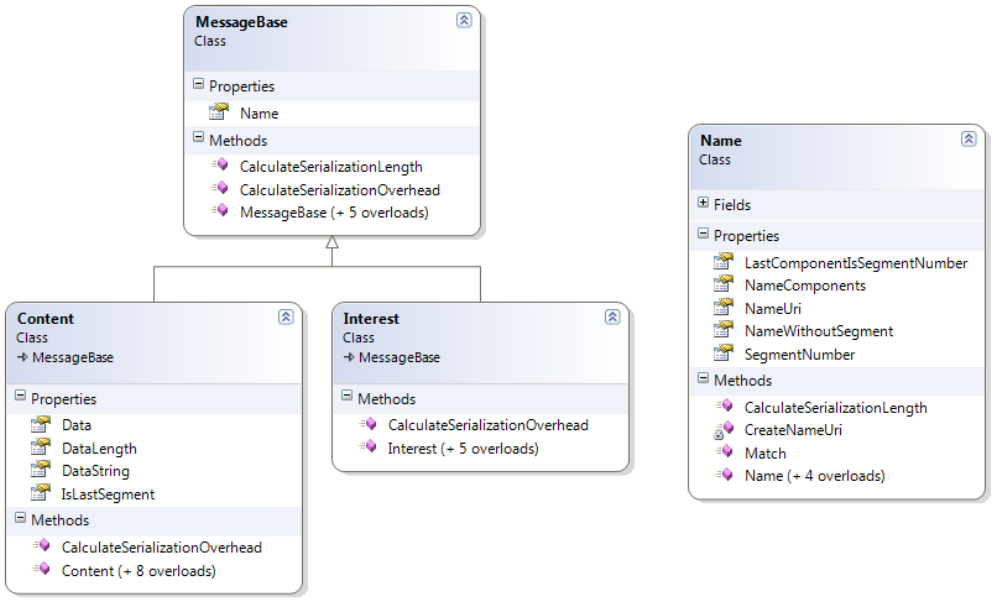


Figure 22 - Detailed view of the types defined under Protocol

4.2.3 Network

The Node type definition and the default engines – DataEngine, NetworkEngine, and DecisionEngine – are all defined under Network as showed in the project structure on Figure 20.

The Node type is what represents the kernel or Node in the ICON Framework, and as such, it is the main component used by applications to communicate with each other using the ICON Framework protocol. The Current static property of the Node type contains an instance of it, and is the preferred way for applications to gain access to a Node. From here, applications using the framework can have access to the engines using the DataEngine, DecisionEngine and NetworkEngine properties. The types for each of these engines are resolved by dependency injection using the registration process described in Figure 21. So, to have an ICON node operational, applications using the ICON framework just need to access the Current static property of the Node type to automatically initialize the ICON Framework. The initialization process will use dependency injection to resolve the required types and instantiate the DataEngine, DecisionEngine and NetworkEngine properties.

To get information from an ICON Node regarding its pending interests, its forwarding information table, or the data it has stored locally, the Node initialization process creates and listens on an http endpoint that can be used to get all this information. The choice for an http endpoint was just to simplify these readings, and use a simple web browser to get this data from the Node. Samples of the information provided by this endpoint are provided in a following section.

The source code of the prototype contains implementations for the three engines, however the provided implementation of the Decision Engine does not include a full implementation of the components described in the ICON Framework definition. This means that the current prototype does not define the Rule Management, the Human Behavior Interface, the Context Collector nor the Synchronization components, which means, that for the current prototype the Decision Engine implementation only defines a basic module that does not take into account all these components. As such, the current implementation simply selects the first available communication interface to transmit data, and stores locally all data that it receives. This is not an issue or impediment to use the current prototype, because as we are using dependency injection to resolve types at runtime, consumers of this prototype can define their own Decision Engine implementation and use the registration process described in Figure 21 to replace the current implementation, taking advantage of the simplicity provided by the ICON Framework design to replace modules as needed. The Network Engine has been fully implemented using an UDP Communication Interfaces that support communications using UDP protocol as the transport layer. Other protocols could have been chosen, TCP for instance,

but the reliability provided by the TCP protocol was not needed and the lower overhead and reduced latency provided by UDP was preferred. The UDP protocol is fast, simple to use, and provides all the requirements needed by the ICON Framework to exchange data. To support this, an UDP Communication Interface was implemented and tested to ensure the best possible results. Nevertheless, other Communication Interfaces could have been implemented to support other transport protocols, for instance a Bluetooth Communication Interface. The Data Engine provides access to the implementations of the PIT Manager and the FIB Manager, both forming the Metadata Store control, and to the Content Store Manager and the Sequencing components. The figure below provides more details for each of the classes and interfaces defined under Network.

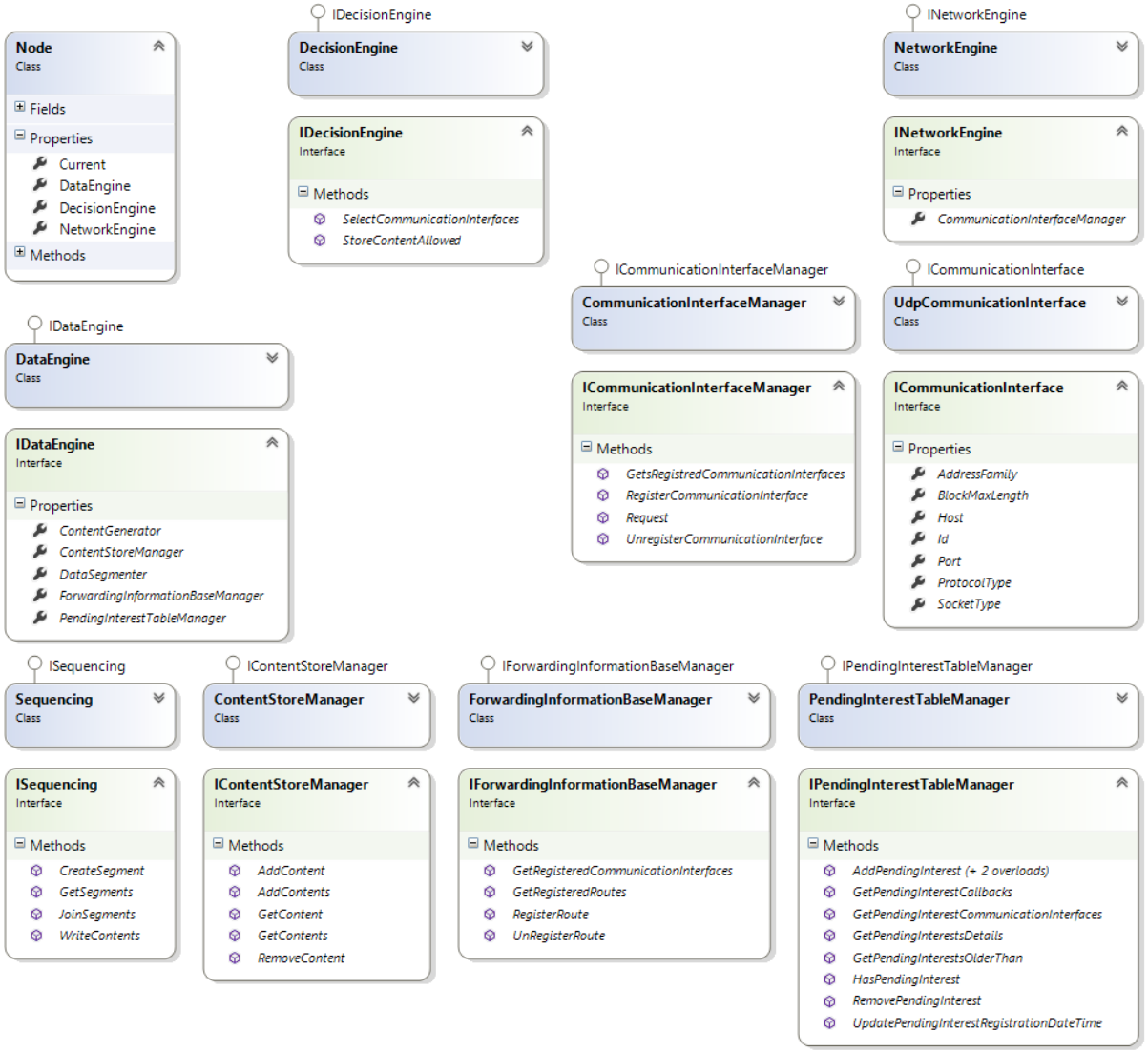


Figure 23 - Interfaces and Classes defined under Network

As we can see from the previous image, every class, except the Node class, implements an interface. By defining interfaces for each module and their respective components, we are abstracting each one of these concepts behind the interface, which means we can have different class implementations for the same interface, with different behaviors in it, and choose the best implementation for each application that uses the ICON Framework. This supports abstraction and modularity, which was always a top priority in the implementation of the ICON Framework. The abstraction simplifies the modularity concepts behind the framework, because we can replace some, or even all, of the ICON Framework provided modules on a specific Node just by doing the registration configuration process like the one showed on Figure 21. For example, if we need to register a different Decision Engine for a specific Node, all we need is to provide a custom implementation of this engine, using the provided interface declared on the ICON Framework, and then register this engine implementation in the application configuration file, again, using the configuration process described in Figure 21. By doing this, and because we are using dependency injection, when the Node starts it will use the custom engine implementation instead of using the one provided by the framework. The current prototype implementation requires a Node reboot if we need to change any of its engines, future implementations can monitor changes on the configuration file and re-inject the modified engines to avoid the reboot. This gives a great level of flexibility in terms of modularity for applications that use the ICON Framework.

4.3 Using the ICON Framework

The first step to use the ICON Framework on an application developed with Microsoft Visual Studio 2010, 2012 or with MonoDevelop, is to add a reference to the assembly of the ICON Framework. A reference is a link between the application and the ICON Framework compiled code. The assembly, is the file containing the Intermediate Language compiled code were the ICON Framework declares all of its types to define their engines and components. An assembly was generated when the ICON Framework prototype was compiled from the C# source code file to the Intermediate Language. The sources and the compiled assemblies of the framework, along with all the applications built to test the prototype are also provided with this dissertation.

The second step, more conceptual, depends on what kind of app is being developed. The framework is ready to be used under three different scenarios, or a mixture of them:

- Producer – Node is primarily a Content producer
- Router – Node does not produce any Content and its main responsibility is to forward Interest and Content messages between other Nodes
- Consumer – Node is primarily a Content consumer

Any of the above combinations are valid. In fact a Node can have the three roles mentioned previously, so it can be a Content producer, a message forwarder, and a Content consumer. For instance, consuming the ICON Framework on an opportunistic network scenario is a great example of such a Node type. In such environment the app might use the available Communication Interfaces to communicate with other Nodes, produce and consume data, and at the same time be a vehicle for other Nodes to interact with each other.

Once the scope of the app is defined, different actions need to be taken to be able to interact with the framework. The simplest case is the one where the app is just a passive Node on the network, which means it will not request directly for Interests nor produce Contents, it is merely a forwarder of those messages. In this case all it needs to be done is to configure the Communication Interfaces and define how to forward messages to other Nodes based on the Interest Names it can forward. Figure 24 displays an example of such configuration. This configuration was required on the prototype implementation, because the Decision Engine module that was created with the prototype did not implement the required components to determine this information, components like the rule management and the context collector should be able to get this information automatically and without recurring to this configuration file. Having done this, the app just needs to access the static Node.Current property to bootstrap the Node's initialization process, and the ICON framework does all the rest.

```
<Register>
  <CommunicationInterfaces>
    <CommunicationInterface key="1" host="192.168.227.128" port="8100" />
    <CommunicationInterface key="2" host="192.168.227.128" port="8200" />
  </CommunicationInterfaces>

  <Listen>
    <CommunicationInterfaceKey value="1" />
  </Listen>

  <Forward>
    <Name uri="ccn4d:Images">
      <CommunicationInterfaces forwardTo="2" receiveOn="1"/>
    </Name>
  </Forward>
</Register>
```

Figure 24 - Sample of Communication Interfaces Node configuration

To be a Content producer, a Node just has to initialize the ContentGenerator property of the DataEngine property, which is accessible through the Current static property of the Node instance. Another option is to store Contents directly in the Node's local store. Below is an

example on how to initialize the ContentGenerator property of the DataEngine on a Node instance, to allow it to send the content of the files identified by the received Interests.

```
Node.Current.DataEngine.ContentGenerator = (receivedInterest, blockMaxLength) =>
{
    var fileName = new Uri(receivedInterest.Name.NameWithoutSegment).AbsolutePath;
    if (File.Exists(fileName))
    {
        using (var file = File.OpenRead(fileName))
        {
            return Node.Current.DataEngine.DataSegmenter.CreateSegment(receivedInterest,
                blockMaxLength,
                (name, dataBlockSize) =>
                {
                    var buffer = new byte[dataBlockSize];
                    file.Seek((name.SegmentNumber - 1) * dataBlockSize, SeekOrigin.Begin);
                    file.Read(buffer, 0, dataBlockSize);
                    return buffer;
                },
                (name, dataBlockSize) =>
                {
                    var fileInfo = new FileInfo(fileName);
                    return (name.SegmentNumber - 1) * dataBlockSize >= fileInfo.Length;
                }
            );
        }
    }
    return null;
};
```

Figure 25 - Sample code to have a Node as a Content producer

Finally, to consume Content by issuing Interests to the network, the app will use the Request method available on the CommunicationInterfaceManager of the NetworkEngine via the Current static property of the Node type. This method requires the Interest, a timeout and a handler to process the Content message when received.

```
var xDoc = XDocument.Load("RequestInterestNames.xml");
var interestNames = xDoc.Descendants("InterestName").Select(xElem => xElem.Attribute("value").Value).ToList();
Console.WriteLine(string.Format("[{0}]:Requesting {1} interests to the network.", DateTime.Now, interestNames.Count()));
foreach (var interestName in interestNames.AsParallel())
{
    var nameRequest = new CitySense.Core.Protocol.Name(interestName);
    var interest = new Interest(nameRequest);
    Node.Current.NetworkEngine.CommunicationInterfaceManager.Request(interest, InterestTimeout, RequestHandler);
}
```

Figure 26 - Sample code to have a Node as a Content consumer

In short, if a Node consumes or generates data, by requesting Interests to the network or by producing Content messages for the received Interests, if it forwards Interests and Content messages also, all it has to do is add a reference to the ICON Framework and reproduce steps similar to those described above. By doing it, it will be ready to use a content centric based protocol which abstracts completely the notion of hosts and the lower level transport protocols; while at the same time it will gain an efficient communication layer even on disruptive environments where network stability between hosts is not always granted. In fact, every

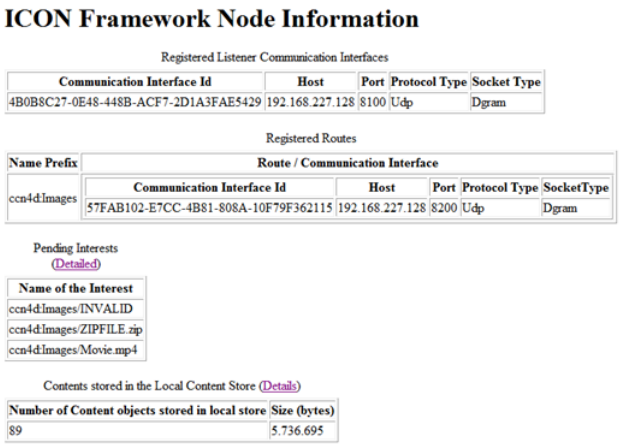
application will benefit from the inherited capacity of the ICON Framework to store locally Content messages once they are received, avoiding the need to transfer data directly from the producers every time the same information is requested by several Nodes. This will also increase the data throughput as demonstrated by the measures taken on some of the realized tests.

4.4 Node information available at runtime

Getting data about the data structures stored on an ICON Node can provide information about the current Node behavior. To provide such information the prototype implementation has added support to get such data using simple HTTP Get commands, so every Node has an HTTP listener, which by default will listen on the local host on port 7999, unless otherwise specified in the application configuration file using the HttpListenerPrefix setting. As an alternative to the HTTP listener an application can use the ICON Framework network protocol using a specific Profile to retrieve the same type of information. The result would be similar, except that the client interface would be a custom application instead of a simple web browser. Once an HTTP Get command is issued, the Node will respond with an HTML response containing information about the Node. The Node provides three types of information:

- General Node information – available on the root address of the configured Http prefix
    - Registered Communication Interfaces where the Node is listening for Interests
    - Registered routes – reflects the content of the FIB Table
    - Pending Interests – reflects the content of the PIT Table
    - Number of Content messages stored locally and the size they are taking
- PIT contents – available on the PITDetailed address of the configured Http prefix
- Locally stored contents – available on the LocalStoreDetails address of the configured Http prefix

Any HTTP client can be used to get information about a Node. Any browser can be used to issue the HTTP requests on the Node's configured HTTP listener prefix. The following figures provide samples of the information that can be retrieved when the HTTP Get commands are issued on the configured HTTP port of the Node.



ICON Framework Node Information

Registered Listener Communication Interfaces

Communication Interface Id	Host	Port	Protocol Type	Socket Type
4B0B8C27-0E48-448B-ACF7-2D1A3FAE5429	192.168.227.128	8100	Udp	Dgram

Registered Routes

Name Prefix	Route / Communication Interface				
	Communication Interface Id	Host	Port	Protocol Type	SocketType
ccn4d:Images	57FAB102-E7CC-4B81-808A-10F79F362115	192.168.227.128	8200	Udp	Dgram

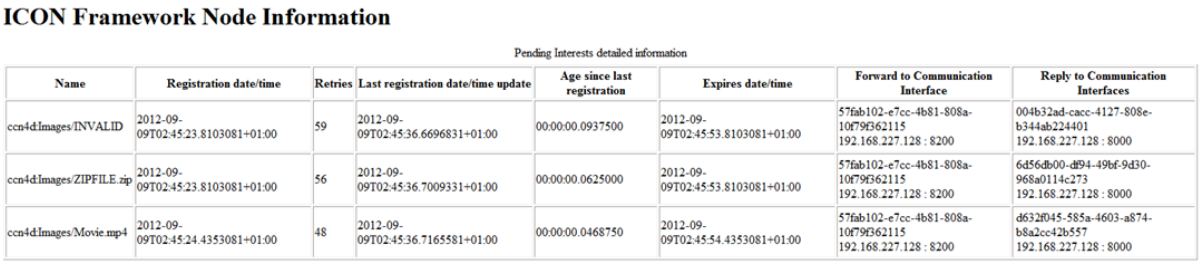
Pending Interests
(Detailed)

Name of the Interest
ccn4d:Images/INVALID
ccn4d:Images/ZIPFILE.zip
ccn4d:Images/Movie.mp4

Contents stored in the Local Content Store (Details)

Number of Content objects stored in local store	Size (bytes)
89	5.736.695

Figure 27 - HTTP response sample containing general Node information

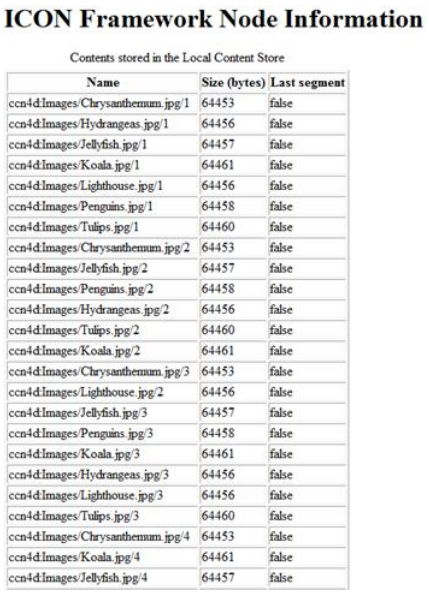


ICON Framework Node Information

Pending Interests detailed information

Name	Registration date/time	Retries	Last registration date/time update	Age since last registration	Expires date/time	Forward to Communication Interface	Reply to Communication Interfaces
ccn4d:Images/INVALID	2012-09-09T02:45:23.8103081+01:00	59	2012-09-09T02:45:36.6696831+01:00	00:00:00.0937500	2012-09-09T02:45:53.8103081+01:00	57fab102-e7cc-4b81-808a-10f79f362115 192.168.227.128 : 8200	004b32ad-cacc-4127-808e-b344ab224401 192.168.227.128 : 8000
ccn4d:Images/ZIPFILE.zip	2012-09-09T02:45:23.8103081+01:00	56	2012-09-09T02:45:36.7009331+01:00	00:00:00.0625000	2012-09-09T02:45:53.8103081+01:00	57fab102-e7cc-4b81-808a-10f79f362115 192.168.227.128 : 8200	6d56db00-df94-49bf-9d30-968a0114c273 192.168.227.128 : 8000
ccn4d:Images/Movie.mp4	2012-09-09T02:45:24.4353081+01:00	48	2012-09-09T02:45:36.7165581+01:00	00:00:00.0468750	2012-09-09T02:45:54.4353081+01:00	57fab102-e7cc-4b81-808a-10f79f362115 192.168.227.128 : 8200	d632f045-585a-4603-a874-b8a2cc42b557 192.168.227.128 : 8000

Figure 28 - HTTP response sample containing detailed PIT information



ICON Framework Node Information

Contents stored in the Local Content Store

Name	Size (bytes)	Last segment
ccn4d:Images/Chrysanthemum.jpg/1	64453	false
ccn4d:Images/Hydrangeas.jpg/1	64456	false
ccn4d:Images/Jellyfish.jpg/1	64457	false
ccn4d:Images/Koala.jpg/1	64461	false
ccn4d:Images/Lighthouse.jpg/1	64456	false
ccn4d:Images/Penguins.jpg/1	64458	false
ccn4d:Images/Tulips.jpg/1	64460	false
ccn4d:Images/Chrysanthemum.jpg/2	64453	false
ccn4d:Images/Jellyfish.jpg/2	64457	false
ccn4d:Images/Penguins.jpg/2	64458	false
ccn4d:Images/Hydrangeas.jpg/2	64456	false
ccn4d:Images/Tulips.jpg/2	64460	false
ccn4d:Images/Koala.jpg/2	64461	false
ccn4d:Images/Chrysanthemum.jpg/3	64453	false
ccn4d:Images/Lighthouse.jpg/2	64456	false
ccn4d:Images/Jellyfish.jpg/3	64457	false
ccn4d:Images/Penguins.jpg/3	64458	false
ccn4d:Images/Koala.jpg/3	64461	false
ccn4d:Images/Hydrangeas.jpg/3	64456	false
ccn4d:Images/Lighthouse.jpg/3	64456	false
ccn4d:Images/Tulips.jpg/3	64460	false
ccn4d:Images/Chrysanthemum.jpg/4	64453	false
ccn4d:Images/Koala.jpg/4	64461	false
ccn4d:Images/Jellyfish.jpg/4	64457	false

Figure 29 - HTTP response sample containing locally stored Content details

5 Testing the ICON Framework

The ICON Framework was designed and implemented to be able to run under multiple platforms and environments and with the capacity to exchange data between them without requiring changes the source or compiled code. As a result, once the source code is compiled and the MSIL – Microsoft Intermediate Language, code is produced as the output of the compilation, it can be used to be executed on the following platforms:

- Any system with Microsoft.NET Framework 4.0 or greater installed
- Any system with Mono 2.10 or greater installed
- Any Android system with Mono for Android installed

During the development phase, most of the code was implemented and executed on a system running Microsoft Windows 7 operating system and with the Microsoft.NET Framework 4.0 as the target platform. This approach was excellent to provide quick results, useful on the initial implementation stage, but as as the development stage was evolving there was the need to start targeting and testing the ICON Framework under different platforms and environments, so the system was tested on the following systems:

- Windows 8 using Microsoft.NET 4.5
- Windows 7 using Microsoft.NET 4.0
- Ubuntu 11.04 using Mono 2.10
- Mac OS X 10.6 Snow Leopard using Mono 2.10

5.1 Test configuration

Unfortunately, testing the ICON Framework on an Android system was not possible, but, all the source code of the ICON Framework compiles successfully against the Mono for Android platform, which means, at least in theory, that Android systems with this platform installed should be able to use the ICON Framework and interact with other systems running the ICON Framework too.

The network configuration used to test the system consists of four systems, running on different environments and platforms. Due to some resource constraints, one of the systems was based on a virtual machine hosted in one of the other configured environments.

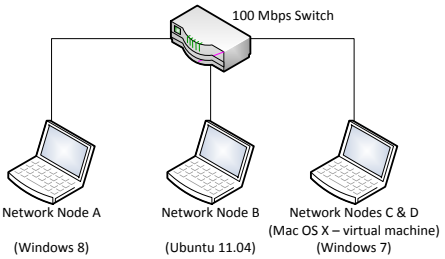


Figure 30 - Network Nodes setup

Figure 30 presents the network configuration for all the nodes configured to test the solution. Node C communicates with the network through Node D, using the virtual machine network configuration NAT – Network Address Translation – option. Further details with each Node characteristics are presented on Table 3 below.

Node	Characteristics
Node A	Windows 8 Pro (x64) Intel Pentium Dual-Core CPU T4300 @ 2.1 GHz 4 GB RAM 100 Mbps network card Microsoft.NET Framework 4.5 Runs the Consumer application (will issue interest requests to the network) Consumer' is a second instance of the Consumer application running on Node A
Node B	Ubuntu 11.04 (natty) – Kernel Linux 2.6.38-15-generic Intel Pentium M processor 1600MHz 520 MB RAM 100 Mbps network card Mono 2.10 Runs the Router application (acts as router between Nodes A, C and D)
Node C	Mac OS X 10.6 Snow Leopard - running on a virtual machine hosted by Node D 4 virtual processor cores (2 processors with 2 cores each) 2 GB RAM 100 Mbps network card Mono 2.10 Runs the Producer application (acts as a content provider on the network)
Node D	Windows 7 Professional(x64) - hosts Mac OS X virtual machine running Node C Intel Core i7 CPU M640 @ 2.8 GHz 8 GB RAM 100 Mbps network card Microsoft.NET Framework 4.0 Runs the Consumer application (will issue interest requests to the network)

Table 3 – Nodes configuration for running tests

5.2 Test description

Three types of applications, using the ICON Framework to transfer information between them, were developed and installed in all the network nodes described previously.

The purpose was to test the ICON Framework while transferring data between two or more network nodes. To achieve that, three different apps were developed, with each one consuming the ICON Framework to establish the required communications to transfer data between each other. The first piece of software acts as a content provider, as such, it should be capable to provide content to the requests sent to the network whenever they reach the Node, and the Node has the capacity to produce the content for the requested data. A second type of application should be able to act as a router and transfer data between Nodes just by looking to the interests it receives and using the Node communication interfaces with the forwarding information that was configured. The third and final element required for the test was the application that could send Interests to the network, wait for their Content, and do something with it.

All these pieces of software were built to test the functionality of the ICON Framework, so three applications were built to represent each of the targeted scenarios:

- Consumer – is an application that uses the ICON Framework to issue Interests to the network. Each content message that it receives from the ICON Framework, upon sending an Interest, is stored in memory until all the Content messages are received. Once the last message is received the cached Contents are flushed to disk, making the content available to be consumed by the operating system or other apps. During the tests, this application was executed on the Network Node A with one or two instances, the latter being referred as Consumer'. A second instance was added to have another Consumer app running for some tests where the same set of interests were issued but, with some delta. In this scenario, the expected result was to have faster Content messages on the instance that was issuing the Interests requests with a delay, because it should benefit from the cached data present on the Router app running on Node B, as a consequence of a first request done by the first Consumer app instance. Network Node D also launches this application during the execution of some tests where multiple Network Nodes, A and D, where simultaneously asking for Interests using this app.

- Router – is an application that does nothing but execute the ICON Framework initialization process. Its purpose is to be able to receive Interests from other Nodes and forward them to Nodes who can provide the requested Interest. It is also able to receive Content messages for each Interest it sends, store those Contents in memory, and finally forward them to the Nodes who have asked for it. This is all done by the ICON Framework, so the Router application is just a container for it, with the proper communication interface configurations set up. During tests, the Router application was executed on the Network Node B.
- Producer – is the application that is responsible to produce content for the Interests that will be requested during the tests. It listens for messages requesting Interests, processes them to check if it can produce the Content messages for those Interests, and finally send those messages back to the Nodes who asked for it. Network Node C is where this application was executed during the tests that were performed.

The following figure gives a view of the test environment, showing the Network Nodes and all the applications running on each one. The dashed line between the Network Nodes C & D means that the Network Node C is a virtual machine hosted by Network Node D, so the connection to the network of Node C is made through Node D using NAT.

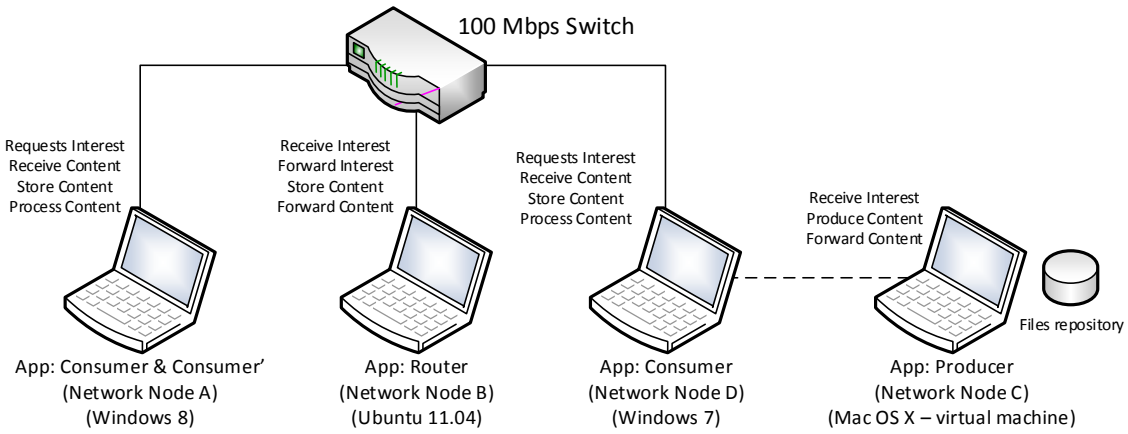


Figure 31 - Network Nodes & applications running on each one

The test was based on data transfers between nodes, so the Producer app running on Network Node C accesses a repository of files that will be requested by Network Nodes A or D via the Consumer app, either directly or indirectly through the Router app running on Network Node B. The files are sent as several blocks of Content messages when the Producer

app receives Interests and generates Content messages for those Interests. The Producer app has also configured a communication interface where it will listen for the Interest requests.

The Consumer app uses a XML file where it has specified all the interests it will issue to the network, and for which, it will wait for and process the received Content messages. Besides this, the Consumer app also configures the communication interfaces required to route Interests to the network based on Interest prefixes.

The Router app will not issue any Interests directly, it will only forward Interests it receives from Network Node A via the Consumer and Consumer' app instances, or through Network Node D via the Consumer app. All of them also forward Interests directly to Network Node C where the Producer app is running. To support this, it only needs to have the communication interfaces configured to listen and forward Interests to the network.

The tests performed using the Consumer, Router and Producer applications were based on file transfers, but the ICON Framework running under it is just dealing with Interest messages and Content messages, so anything can be transferred, live data for instance, to stream video was also a scenario where the framework was tested.

5.3 Profiling code and optimizations

During the development of the ICON Framework several code analysis were done to evaluate the performance of the algorithms, response times and bottlenecks.

Code profiling, while running the tests, was one of the most important ways to identify implementation problems and measure code quality. Presenting here all the analysis that was done would not be possible or desired, but a simple concrete example will be shown to demonstrate what was generally done.

Figure 32 shows the hot path of code running on the Producer application while the tests were running. This is the app that responds with Content messages to the Interests it receives. The analysis of the hot path shows that the methods that took longer were methods of the ICON Framework API, which is not strange because the app itself is delegating most of its work to the API. However, what is strange, is the functions that are doing most individual work and the percentage they take to do it. System.Threading.Monitor.Enter is related to a threading

lock being acquired, and System.Console.WrileLine is due to informational messages being written to the console. These functions should not be the ones where the code should be spending more time, so the code hot path was analyzed.

Figure 32 - Hot paths and functions with most work before code optimizations

Starting the analysis on the ListenerHandler function identified on the hot path, we can see that inside of it, most of the time is spent on the ProcessInterestMessage and the WrileLine functions – see Figure 33. Further analysis of the ProcessInterestMessage on Figure 34, identifies GetContent AddContent and WrileLine as the functions doing most work. A deeper analysis on the GetContent function presented on Figure 35 identifies the bottleneck – the lock being acquired to support multithreaded access to the local content store of the Node.

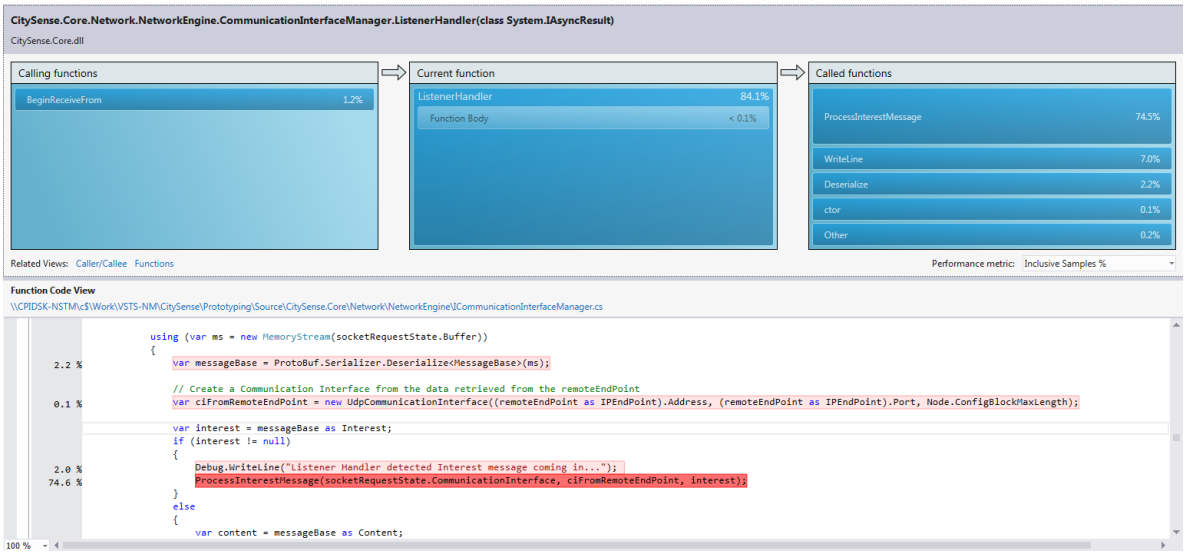


Figure 33 - ListenerHandler function analysis before code optimizations

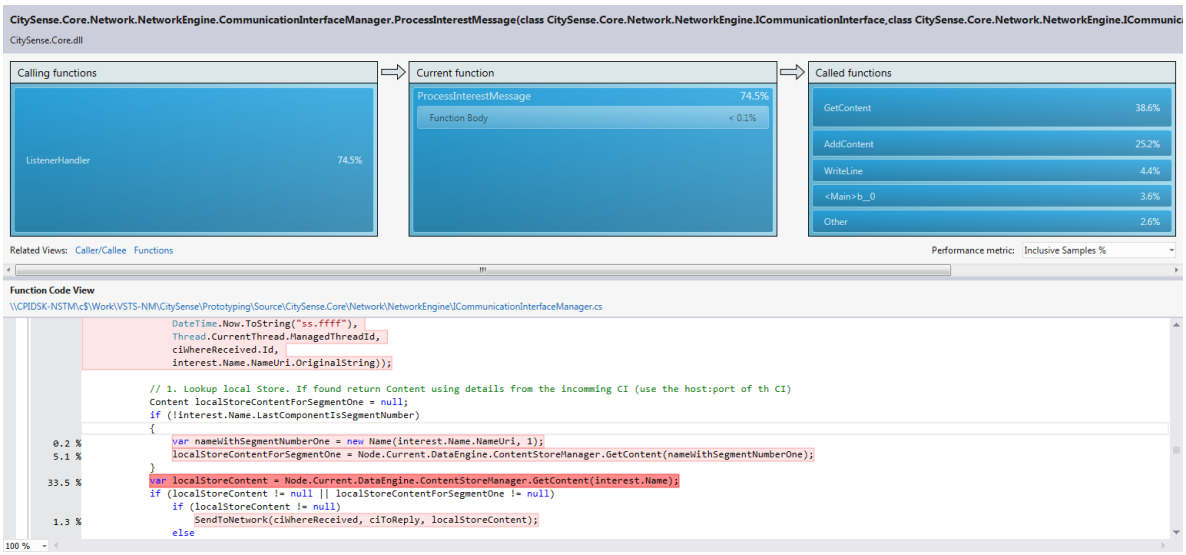


Figure 34 - ProcessInterestMessage function analysis before code optimizations

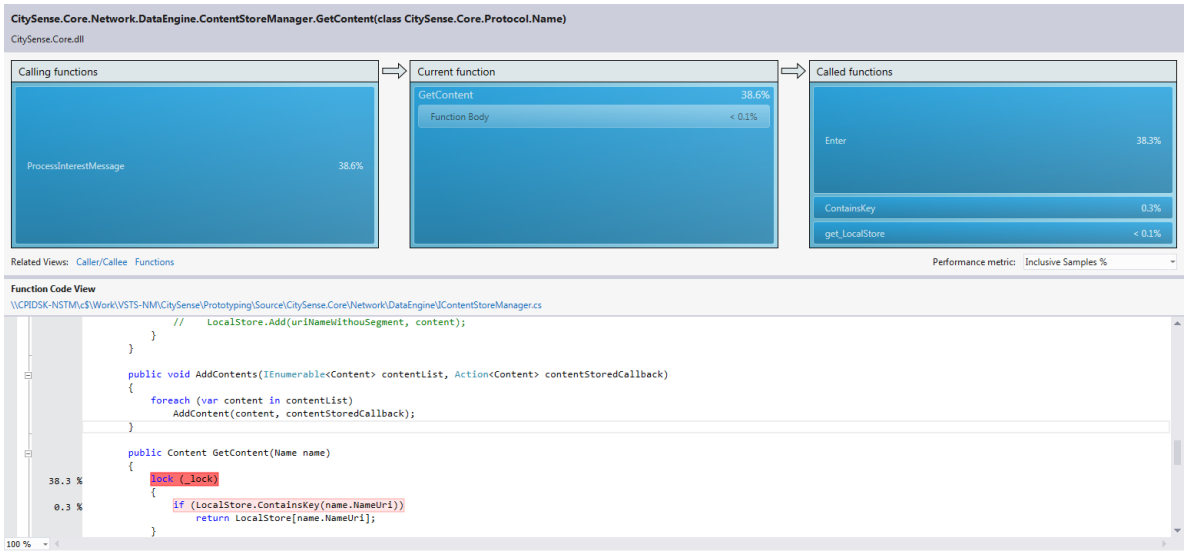


Figure 35 - GetContent function analysis before code optimizations

It happens that acquiring the lock was taking longer than expected due to the fact that executing Console.WriteLine takes a long time to run and most of the calls to it were done using the same thread that was accessing the local storage collection. In this case the improvement was simple, and it only required to replace all Console.WriteLine calls to Debug.WrileLine calls so that this negative impact was only noticed during debug builds.

As soon as that change was done, the bottleneck caused when the lock is being acquired was removed. Figure 36 shows that most of the time is now spent on the clr.dll

assembly, which is part of the Microsft.NET Framework. By analyzing the test results before and after the code optimization, we can see performance improvements in every scenario where a concurrency situation occurs, and this is the expected result because now the code locks are not taking valuable time anymore.

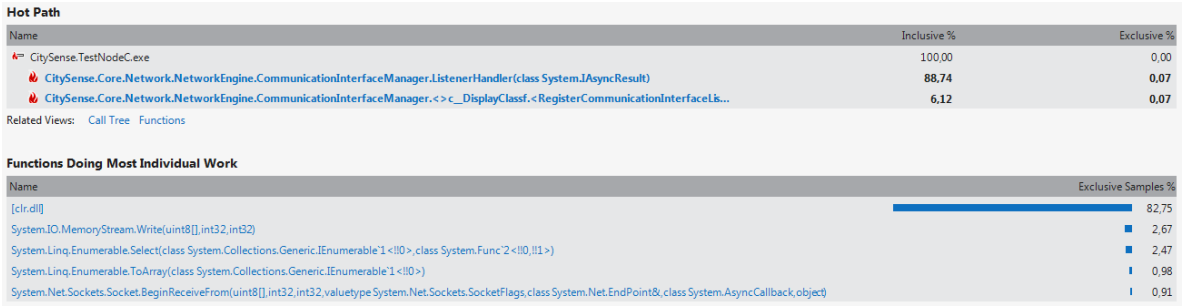


Figure 36 - Hot paths and functions with most work after code optimizations

5.4 Performance tests

While testing the ICON Framework there were several types of tests executed to evaluate the framework capacity to transfer data on the network. Table 4, presented on Tests summary section, shows some of the performance tests that were executed. The table includes the test results before and after the code optimizations mentioned on the previous section, where improvements on resource locking were done to support the multithreaded environment that exists inside the ICON Framework.

The first and primary test was to see if the data could effectively be transferred between Nodes and without information being lost. The ensure this, the first analysis was checking if the files stored on Network Nodes A & D contained the same information from where they were transferred. To check this, Interests requested to the network from Network Nodes A & D should produce a file containing the same information that exists in Network Node C. All the executed tests produced successful results, which means, all Interests requested by the Consumer app either from Network Node A or D, always generated the expected files on the file system of these Nodes, and the consistency observed on all tests provided a great level of confidence on the system. To further test the ICON Framework, several sets of tests were executed to validate its efficiency and robustness. The following sections summarize the types of tests realized.

5.4.1 Tests summary

This section shows on Table 4 the summary of the tests performed, and the following sections give a detailed description and analysis for each of these tests.

The legend below provides information on how the tests are presented, and should be used to read the results:

- Interest(s) – Name of the Interest or Interests identification issued to the network. A blank cell means that the test is part of the test described on the previous row
- Size (bytes) – Total size, in bytes, of the original content for the Interests requested. A blank cell means that the test is part of the test described on the previous row
- Description – Brief description of the test performed. A blank cell means that the test is part of the test described on the previous row
- Nodes running their apps (first column) – Identification of the Node and the applications that each Node is running
    - A = Node A running the Consumer app
    - A' = Node A running the Consumer' app (second instance)
    - B = Node B running Router app
    - C = Node C running Producer app
    - D = Node D running Consumer app
    - > = Interest message sent or forwarded
    - » = Content message sent or forwarded
- Nodes running their apps (second column) – The same as above, but required to provide on two rows metrics for each one of the Consumer app that was running
- Duration – The total time taken to receive the total content after the first Interest message was sent
- MB/seg – The megabytes per second ratio

Interest(s)	Size (bytes)	Description	Nodes running their apps		Duration	MB/seg
		Test Set 1 (1 client)				
Movie.mp4	132.646.801	First request	A>B>C»B»A	A>B>C»B»A	00:00:50	2,53
Movie.mp4	**132.646.801**	**First request (after code optimization)**	**A>B>C»B»A**	**A>B>C»B»A**	**00:00:45**	**2,81**
Movie.mp4	132.646.801	Second request	A>B»A	A>B»A	00:00:21	6,02
Movie.mp4	**132.646.801**	**Second request (after code optimization)**	**A>B»A**	**A>B»A**	**00:00:21**	**6,02**
Movie.mp4	132.646.801	Second request	D>B»D	D>B»D	00:00:21	6,02
		Test Set 2 (2 clients, A & A' request simultaneously)				
Movie.mp4	132.646.801	First request	A+A'>B>C»B»A+A'	A>B>C»B»A	00:00:52	2,43
				A'>B»A'	00:00:51	2,48
Movie.mp4	**132.646.801**	**First request (after code optimization)**	**A+A'>B>C»B»A+A'**	**A>B>C»B»A**	**00:00:51**	**2,48**
				A'>B»A'	**00:00:51**	**2,48**
Movie.mp4	132.646.801	Second request	A+A'>B»A+A'	A>B»A	00:00:27	4,69
				A'>B»A'	00:00:27	4,69
Movie.mp4	**132.646.801**	**Second request (after code optimization)**	**A+A'>B»A+A'**	**A>B»A**	**00:00:27**	**4,69**
				A'>B»A'	**00:00:27**	**4,69**
		Test Set 3 (2 clients, A & D request simultaneously)				
Movie.mp4	132.646.801	First request	A+D>B>C»B»A+D	A>B>C»B»A	00:00:51	2,48
				D>B»D	00:00:50	2,53
Movie.mp4	132.646.801	Second request	A+D>B»A+D	A>B»A	00:00:30	4,22
				D>B»D	00:00:29	4,36
Movie.mp4	**132.646.801**	**Second request (after code optimization)**	**A+D>B»A+D**	**A>B»A**	**00:00:29**	**4,36**
				D>B»D	**00:00:27**	**4,69**
		Test Set 4 (2 clients, A & A' or A & D request with ~10 sec. delta)				
Movie.mp4	132.646.801	First request	A+A'>B>C»B»A+A'	A>B>C»B»A	00:00:51	2,48
				A'>B»A'	00:00:41	3,09
Movie.mp4	**132.646.801**	**First request (after code optimization)**	**A+A'>B>C»B»A+A'**	**A>B>C»B»A**	**00:00:50**	**2,53**
				A'>B»A'	**00:00:39**	**3,24**
Movie.mp4	132.646.801	First request	A+D>B>C»B»A+D	A>B>C»B»A	00:00:51	2,48
				D>B»D	00:00:41	3,09

		Test Set 5 (2 clients, A & A' or A & D request with ~20 sec. delta)				
Movie.mp4	132.646.801	First request	A+A'>B>C»B»A+A'	A>B>C»B»A	00:00:52	2,43
				A'>B»A'	00:00:40	3,16
Movie.mp4	**132.646.801**	**First request (after code optimization)**	**A+A'>B>C»B»A+A'**	**A>B>C»B»A**	**00:00:51**	**2,48**
				A'>B»A'	**00:00:30**	**4,22**
Movie.mp4	132.646.801	First request	A+D>B>C»B»A+D	A>B>C»B»A	00:00:52	2,43
				D>B»D	00:00:32	3,95
		Test Set 6 (1 client, 7 files of 23.129.260 bytes each)				
Zipfile_1.zip to Zipfile_7.zip	161.904.820	First request	A>B>C»B»A	A>B>C»B»A	00:00:36	4,29
Zipfile_1.zip to Zipfile_7.zip	**161.904.820**	**First request (after code optimization)**	**A>B>C»B»A**	**A>B>C»B»A**	**00:00:20**	**7,72**
Zipfile_1.zip to Zipfile_7.zip	161.904.820	Second request	A>B»A	A>B»A	00:00:18	8,58
Zipfile_1.zip to Zipfile_7.zip	**161.904.820**	**Second request (after code optimization)**	**A>B»A**	**A>B»A**	**00:00:16**	**9,65**
		Test Set 7 (1 client, 175 files of 879.394 bytes each)				
Img(1).zip to Img(175).zip	153.893.950	First request	A>B>C»B»A	A>B>C»B»A	00:00:40	3,67
Img(1).zip to Img(175).zip	**153.893.950**	**First request (after code optimization)**	**A>B>C»B»A**	**A>B>C»B»A**	**00:00:20**	**7,34**
Img(1).zip to Img(175).zip	153.893.950	Second request	A>B»A	A>B»A	00:00:22	6,67
Img(1).zip to Img(175).zip	**153.893.950**	**Second request (after code optimization)**	**A>B»A**	**A>B»A**	**00:00:20**	**7,34**
Img(1).zip to Img(175).zip	153.893.950	Second request	A>C»A	A>C»A	00:00:21	6,99
Img(1).zip to Img(175).zip	**153.893.950**	**Second request (after code optimization)**	**A>C»A**	**A>C»A**	**00:00:20**	**7,34**
		Test Set 8 (1 client,8 files with sizes between 561.276 and 879.394)				
8 files with different sizes	5.837.151	First request	A>B>C»B»A	A>B>C»B»A	00:00:03	1,86
8 files with different sizes	**5.837.151**	**First request (after code optimization)**	**A>B>C»B»A**	**A>B>C»B»A**	**00:00:01**	**5,57**
8 files with different sizes	5.837.151	Second request	A>B»A	A>B»A	00:00:02	2,78
8 files with different sizes	**5.837.151**	**Second request (after code optimization)**	**A>B»A**	**A>B»A**	**00:00:01**	**5,57**

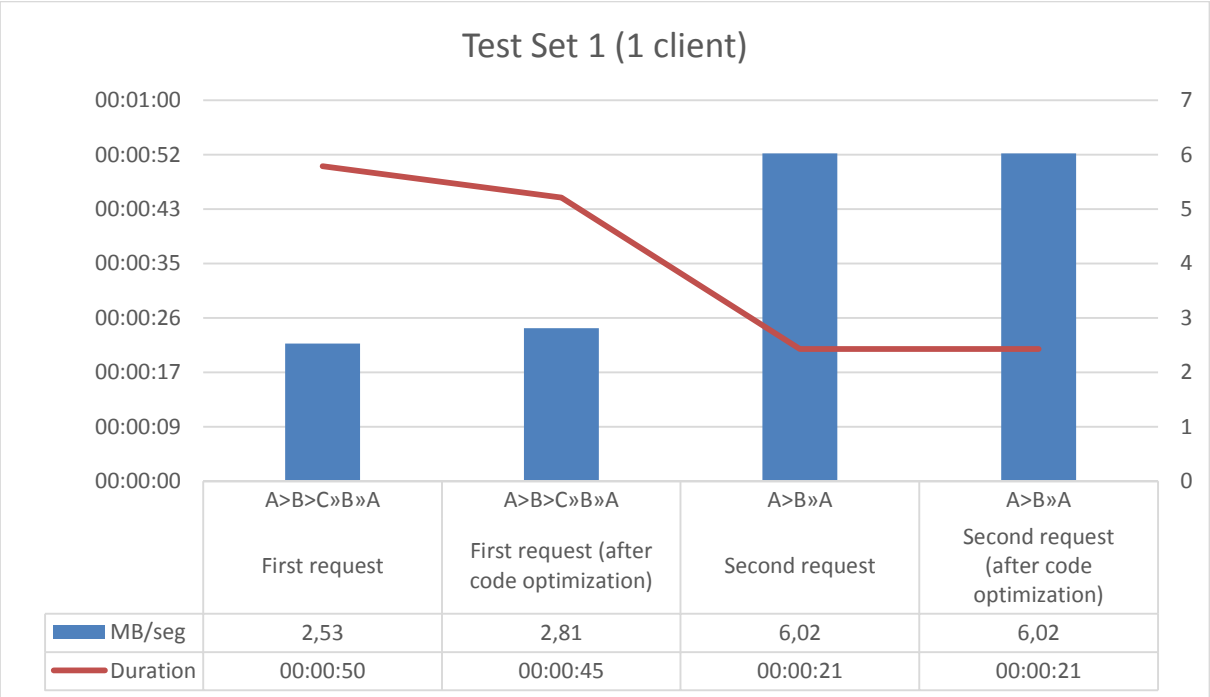
Table 4 – Test results

5.4.2 Test Set 1

5.4.2.1 Test description

- A Consumer app requests an Interest to the network where the Interest is routed via Network Node B
- Interests are sent sequentially and upon Content retrieval until the last Content message is received
- Measures were taken for first and second requests
- For the second request Network Node B no longer forwards the Interest to Network Node C because it has already cached, in the local storage, the Content for the Interests requested after the first run

5.4.2.2 Test results



	A>B>C»B»A First request	A>B>C»B»A First request (after code optimization)	A>B»A Second request	A>B»A Second request (after code optimization)
MB/seg	2,53	2,81	6,02	6,02
Duration	00:00:50	00:00:45	00:00:21	00:00:21

Figure 37 - Graphical analysis for test set 1

5.4.2.3 Test analysis

- A slight improvement for the first request test, noticed when the test was executed with the code optimized. This was the expected result, since the code improvements were done to improve parallel execution. Although this was not the focus of this test, Node B still has some parallel execution to process Interest requests from Node A and Content messages from Node C
- No improvements on the second request, when we compare the before and after code optimization. This was the expected result, since Node B is now processing just the Interests requests from Node A, which are sequential. As there is no parallelization occurring on Node B, the optimizations done on the code have no effect, so there is no improvement on the test results
- An improvement, 114% looking at the values of the optimized code, between the first and second requests. This was the expected result, since the second request can now retrieve the contents from Node B, because the contents were cached on the first request

5.4.3 Test Set 2 and 3

5.4.3.1 Test description

- Multiple Consumer apps running on Network Node A only or in Network Node A and D, which request an Interest to the network where the Interest is routed via Network Node B
- Interests are sent sequentially and upon Content retrieval until the last Content message is received
- Measures were taken for first and second requests
- For the first request, clients A' and D will get content messages directly from Network Node B, which no longer forwards Interests to Network Node C because it has already cached, in the local storage, the Content for the Interests requested from the client A
- For the second request Network Node B no longer forwards the Interest to Network Node C because it has already cached, in the local storage, the Content for the Interests requested after the first run
- Interest requests were issued at the same time on both application Nodes

5.4.3.2 Test results

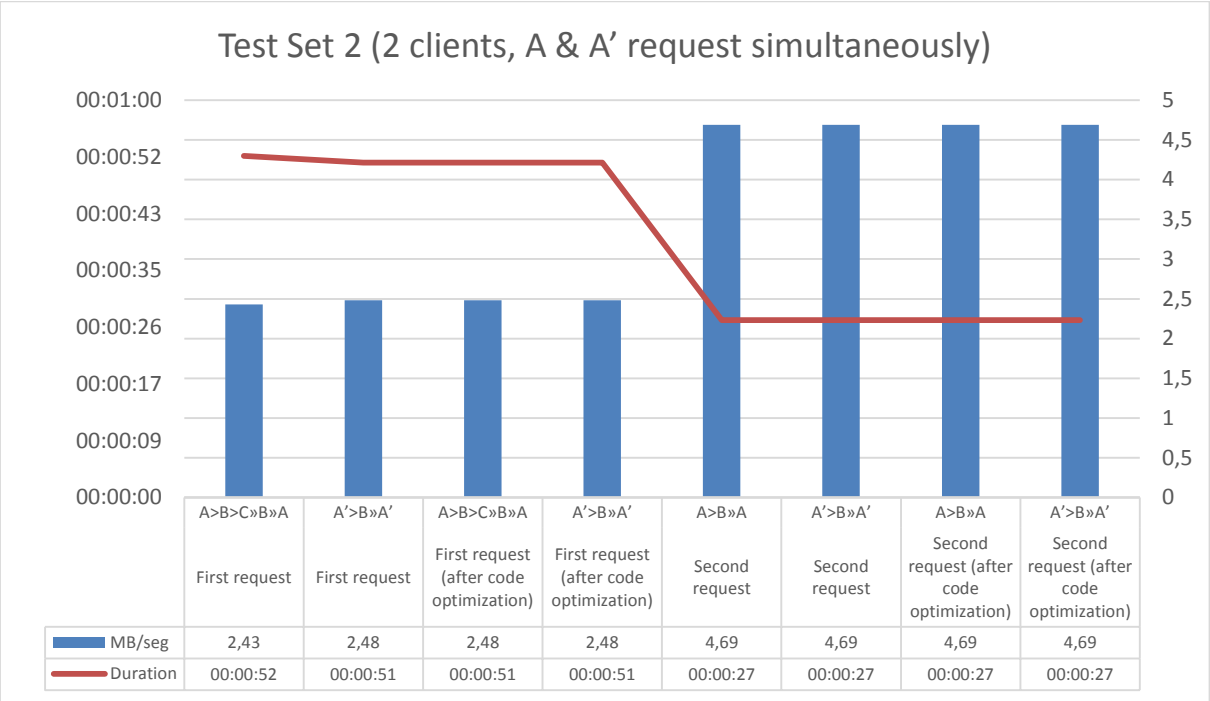


Figure 38 - Graphical analysis for test set 2

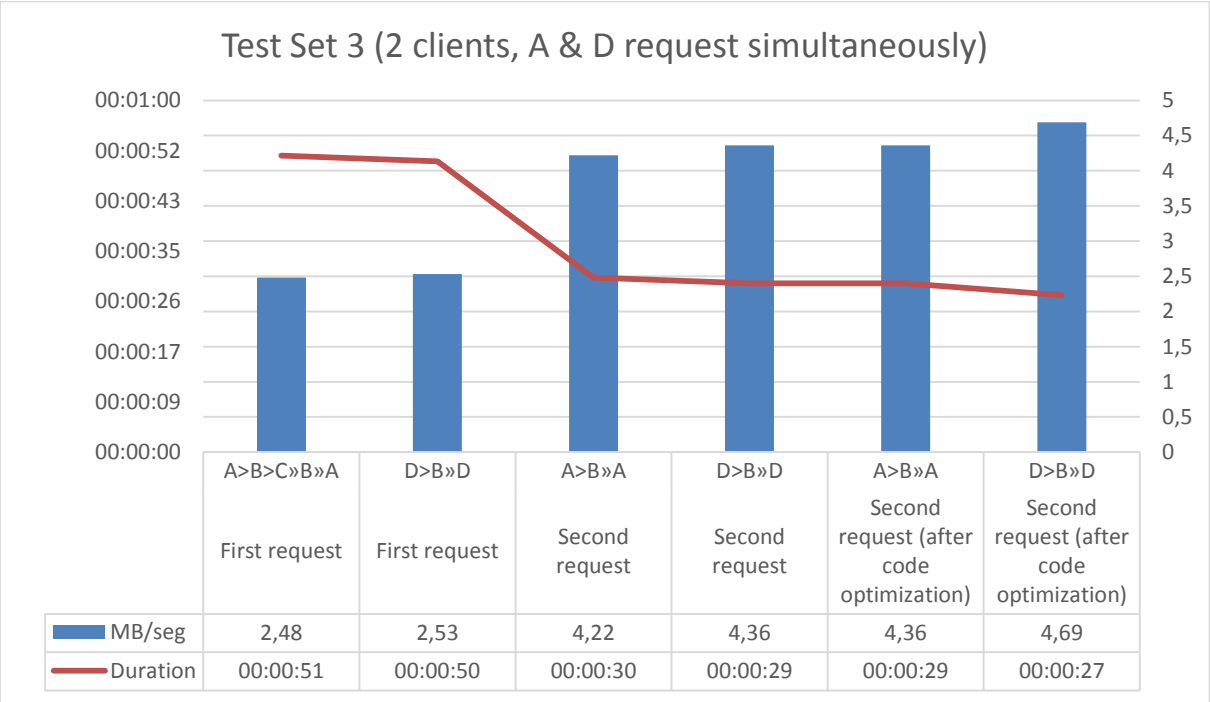


Figure 39 - Graphical analysis for test set 3

5.4.3.3 Test analysis

- A slight improvement for the first request test when the A>B>C»B»A + A'>B»A' and A>B>C»B»A + D>B»D tests were executed. This occurs because clients A' and D will get content messages directly from Network Node B, which no longer forwards Interests to Network Node C because it has already cached, in the local storage, the Content for the Interests requested from the client A. Test sets 4 and 5 will show better results on this point, since a delay for clients A' and D to start issuing Interests will be introduced
- Almost no improvements, or no improvements at all on the second request, when we compare the before and after code optimization. This was the expected result, since Node B is now processing the Interests requests from Node A and Node D, which are few clients to cause a great level of concurrent operations. As there is almost no parallelization occurring on Node B, the optimizations done on the code have no effect, so there is no improvement on the test results
- An improvement, 89% if we look at the values of the optimized code, between the first and second requests. This was the expected result, since the second request can now retrieve the contents from Node B, because the contents were cached on the first request.
- Comparing test sets 2 and 3 we can observe similar results, which means we have similar performance indicators whether we run a second instance of the Consumer application on the same network Node, or in different network Nodes

5.4.4 Test Set 4 and 5

5.4.4.1 Test description

- The same type of test that was described in Test Set 2 and 3, with the exceptions below
- Measures were taken for first request only
- In Test Set 4 and 5, the second application will issue Interest requests with a delta of 10 and 20 seconds respectively
- The delayed requests done by the consumer applications A' or D will benefit from the cache of Content messages on Node B, as a result of the first requests being done by the consumer application A

5.4.4.2 Test results

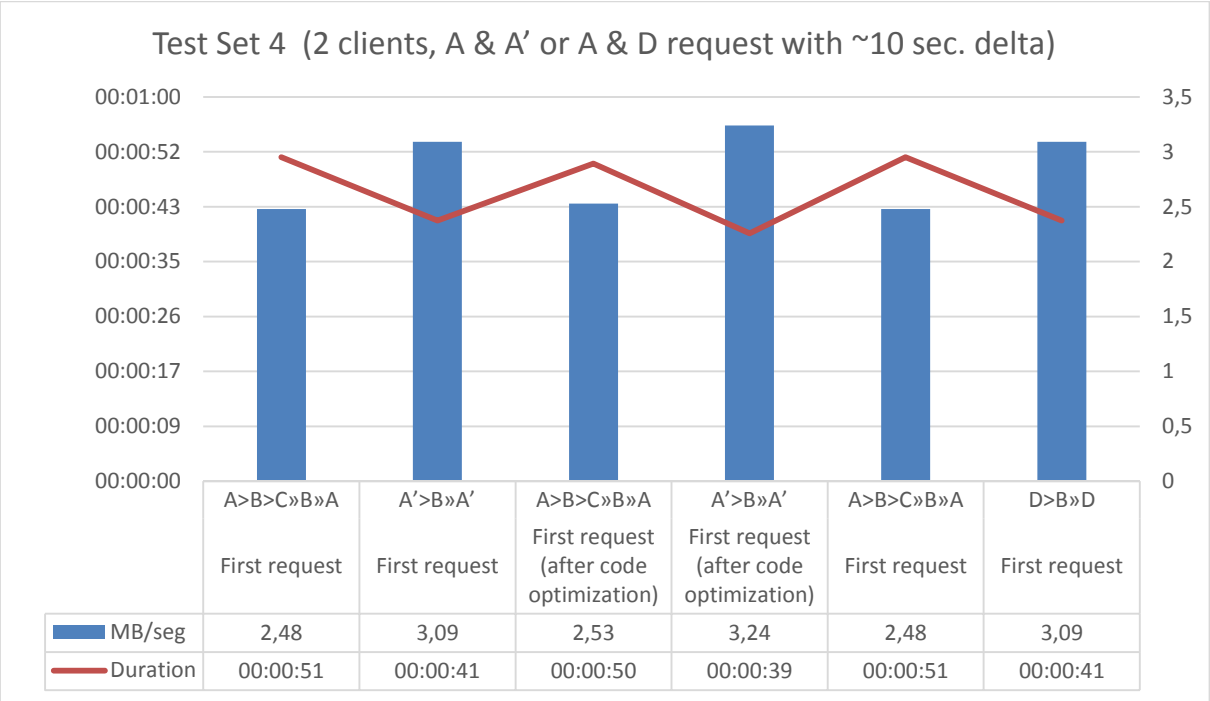


Figure 40 - Graphical analysis for test set 4

Figure 41 - Graphical analysis for test set 5

5.4.4.3 Test analysis

- We can see a significant improvement of the first requests on the Consumer applications that issued the requests with the 10 and 20 second delay. This improvement occurs because these applications benefit the fact that a previous application, Consumer application A, has already asked for the same contents, which are now cached on network Node B, and ready to be delivered to any consumer application that asks for them
- As expected, the improvement is higher on the optimized code where we can see an improvement of 70% for the 20 second delay. In this scenario there is much more parallelized code running on network Node B. This Node is now processing interests from the consumer application A and forwarding them to network Node C. At the same time, and after the defined delay, it starts to receive interests from the consumer application A' or D, and these will be served directly from its cache. While this process occurs, it is also receiving Content messages for the interests forwarded to network Node C
- The higher the delay the higher the improvement. This is normal and expected since a lower delay will be faster to try to get Content messages that are not in cache yet, and from that point on, the Node will be in the same scenario that was demonstrated in Test Set 2 and 3. If we compare the A'>B»A' tests of the optimized code between Test Set 4 and 5, we can see an improvement of 30%, and if we compare the result of Test Set 5 with the same test result obtained on Test Set 2, we see an increase of 70%, which corroborates the percentage referred on the second point of this analysis

5.4.5 Test Set 6

5.4.5.1 Test description

- A unique Consumer app running on Node A requests 7 Interests to the network where the Interests are routed via Network Node B
- Interests are sent sequentially and upon Content retrieval until the last Content message is received, but since multiple different Interests were requested initially this causes multiple threads to request Interests simultaneously to the network

- Measures were taken for first and second requests
- For the second request Network Node B no longer forwards the Interest to Network Node C because it has already cached, in the local storage, the Content for the Interests requested after the first run

5.4.5.2 Test results

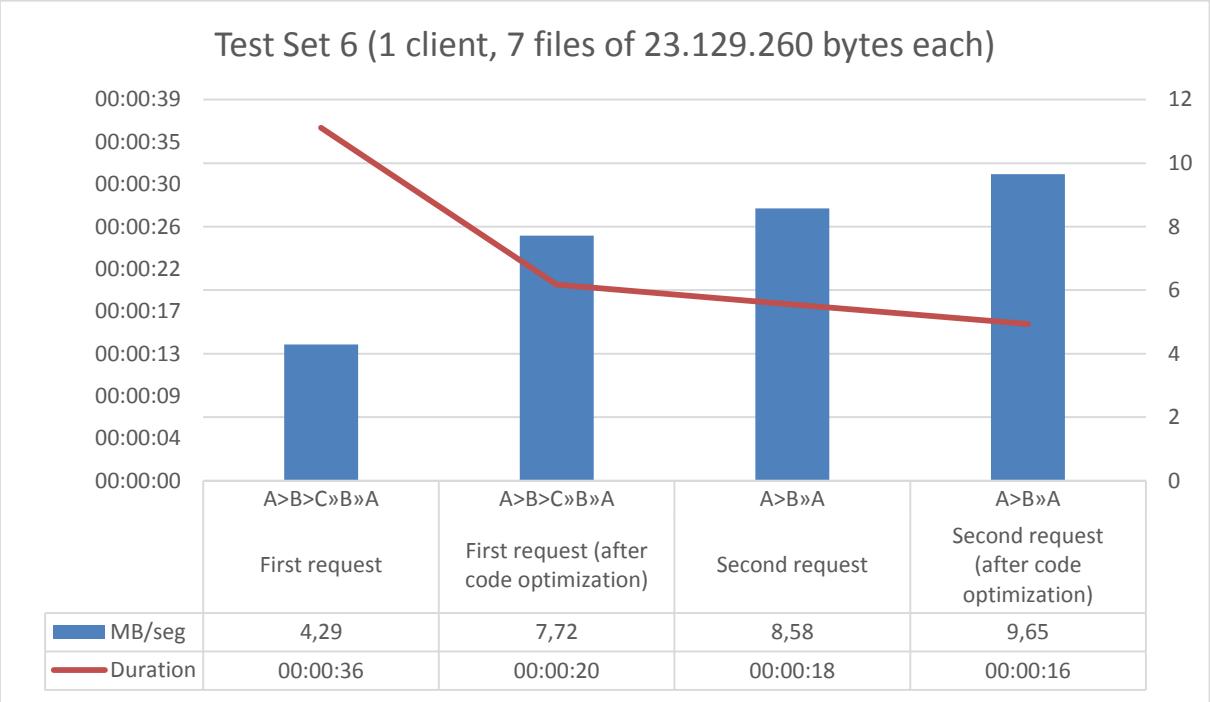


Figure 42 - Graphical analysis for test set 6

5.4.5.3 Test analysis

- This test shows the Node behavior while handling concurrent requests, and demonstrates how the code optimizations produced an improvement when several input and output operations are processed by the Node
- For the first request we can see an improvement of 80%. The improvement was expected since the code optimizations provided a better handling of parallelized code to any kind of Node – Consumer, Router or Producer
- Improvements between the second requests are also seen, although in a lower percentage – 12%.
- The results seen on previous tests continue to be present here, as we can see an improvement of 25% between the first and second requests on the optimized code. Once

again, in this situation the results are due to the Node cache. The increase here is not as high as we have seen in other tests, because the first request has already improved significantly. This occurs because multiple concurrent interests are being sent to the network, and the code was optimized to handle these scenarios efficiently

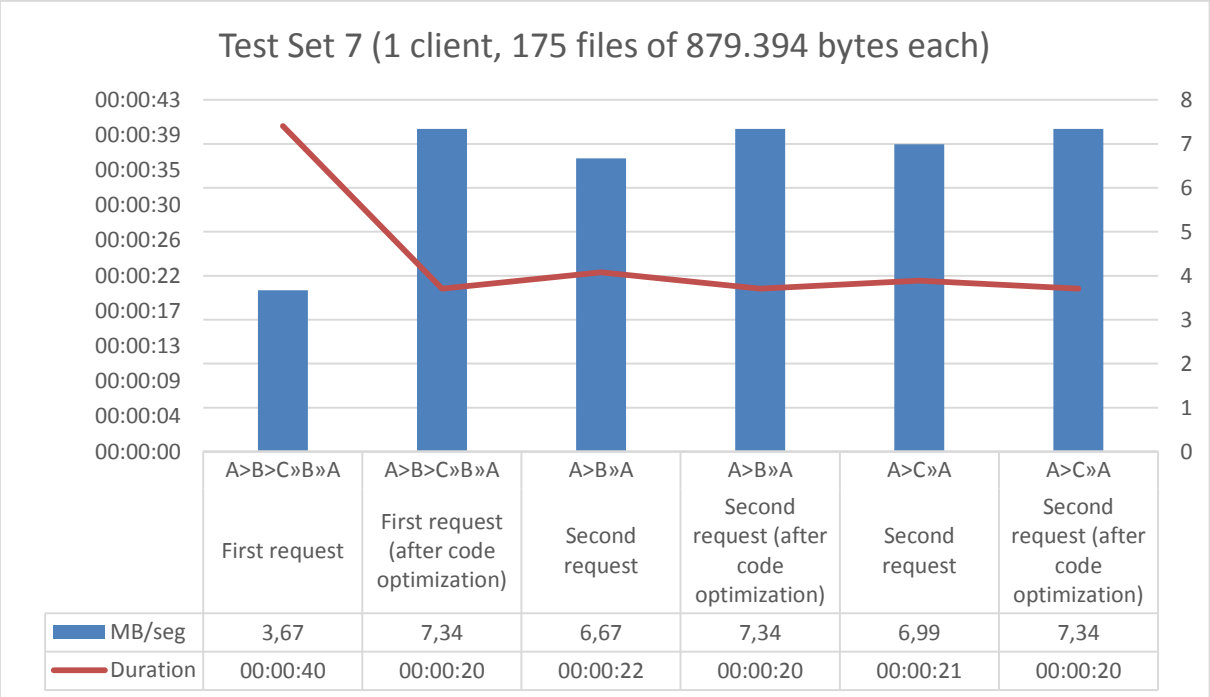
- This test also reached the highest throughput 9,65 MB/sec, using 77% of the network bandwidth. The Node cache and the capacity of a Node to process requests concurrently contribute to such result

5.4.6 Test Set 7

5.4.6.1 Test description

- The same type of test that was described in Test Set 6, but with the exceptions below
- The consumer application request 175 Interests instead of 7
- Since Network Node B is a slower machine, measures were also taken between Network Nodes A and C without going through Node B

5.4.6.2 Test results



	A>B>C»B»A First request	A>B>C»B»A First request (after code optimization)	A>B»A Second request	A>B»A Second request (after code optimization)	A>C»A Second request	A>C»A Second request (after code optimization)
MB/seg	3,67	7,34	6,67	7,34	6,99	7,34
Duration	00:00:40	00:00:20	00:00:22	00:00:20	00:00:21	00:00:20

Figure 43 - Graphical analysis for test set 7

5.4.6.3 Test analysis

- With more files being requested, the difference on the first request after code optimizations reached 100%
- For the first time, the first and second requests on the optimized code reached the same values. This can be justified because of the significant increase in performance that was seen on the first request, which was only possible because of the large support for parallelization that was placed on the ICON Framework
- Network Node B was the slowest machine and the one with less RAM available. By having the same results between tests A>B»A and A>C»A we conclude that the ICON Framework can produce similar results even when executed in systems with very different hardware configurations

5.4.7 Test Set 8

5.4.7.1 Test description

- A unique Consumer app running on Node A requests 8 Interests to the network where the Interests are routed via Network Node B
- Interests are sent sequentially and upon Content retrieval until the last Content message is received, but since multiple different Interests were requested initially this causes multiple threads to request Interests simultaneously to the network
- Measures were taken for first and second requests
- For the second request Network Node B no longer forwards the Interest to Network Node C because it has already cached, in the local storage, the Content for the Interests requested after the first run
- The full size of the Content for each requested Interest is different

5.4.7.2 Test results

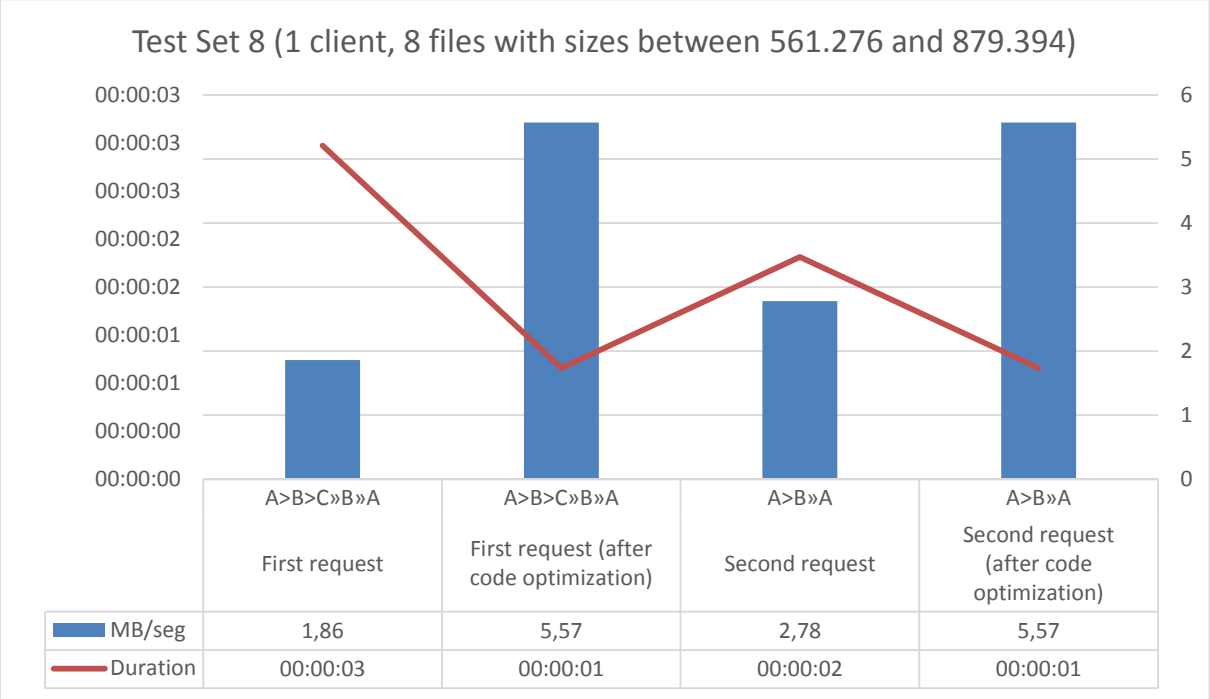


Figure 44 - Graphical analysis for test set 8

5.4.7.3 Test analysis

- The analysis of the results of this test show the same tendency of the ones described previously, particularly the ones for Test Set 7
- In this test the gain for the first request on the optimized code reached 199%
- For the second request, the code optimization produced a gain of 100%
- And again, the first and second requests reached the same values on the optimized code, but in this case the files had different sizes and were only 8 files.

5.4.8 Test analysis summary

The analysis of the tests reveals the expected behavior and the gains brought by the fact of Nodes being able to locally store Content. This can be seen by the fact that on all tests, the second request always takes less time to get the content, and this happens because the content is retrieved directly from the Node B where the Router app is caching Content messages. This cache behavior can also be seen when multiple Consumer instances are asking for the same Interests, but with some delay – in the case of test 4 and 5. In these cases, the

second instance always receives the Content much faster, because once again, it benefits from the fact that the first instance had already asked for the Interests, for which the Content messages were then stored locally on Node B that is running the Router app, that also provides Content messages to the second instance of the Consumer app. This appends because the Content messages stored in the local Node cache are valid to be send once the Node receives an Interest that can be satisfied by this data, avoiding unnecessary requests to the Content producers. This shows that a network environment with more Nodes similar to the Router application could bring benefits the overall data flow. This will give data producers less work, since they will not have to provide the same Content so often, and at the same time it will keep data closer to the consumers bringing better response times to the data requested. The content centric network architecture played a key role to achieve these results. It is the nature of this network, which brings the possibility to route data instead of machines or IP addresses, combined with the cached Contents on each Node that produces such gains.

The implementation also demonstrates the ICON Framework implementation efficiency when it was able to process up to 175 concurrent requests without losing Content packets and by keeping the data transfer rate at an acceptable level.

The improvements done based on the code profiling were also quite noticeable, as we can see, almost all of the tests performed after the code optimization provided better results, and especially those where concurrency exists, if we look at the test sets 6, 7 and 8, where multiple concurrent requests are done because the tests are requesting Interests in parallel, the improvement is evident, on test set 7 the first request dropped from 40 to 20 seconds. These improvements were done continuously and through all the implementation phase of ICON Framework, so the differences measured here were just a simple example of the gains brought to the system just by doing code optimizations based on code execution analysis. The gains shown here are more noticeable on tests where concurrency exists, because the code was optimized to reduce significantly the time to acquire a lock during execution of parallelized code.

5.5 Other tests performed

Besides the tests described in the previous sections, other tests were realized to further validate the architecture and the implementation of the solution.

One of those tests was seeing how the proposed solution would handle communications under scenarios where network failures occur frequently or have a highly variable or extreme performance, and thus prove is robustness. This type of test was of major importance, since opportunistic networks emerge and live in such environments, so ensuring that the ICON Framework was able to work efficiently under those circumstances need to be proven. A video with these tests was published and can be viewed or downloaded using any of the following addresses:

- http://www.youtube.com/watch?v=ExZ_wxLK1lY
- https://skydrive.live.com/redir?resid=30C3E79A52875CB2!1065&authkey=!ACD6ZQKBL5UjIAA

Another test was targeted to live data, for instance capturing video from a web cam and stream it live with a producer app was also an important point to validate. Live streaming normally requires a relatively constant data transfer bit rate to ensure some quality, so putting the ICON Framework under such test and checking the results was also a major goal.

With these two types of tests in mind two apps were created. A first app capable of connect to the PC web cam and capture video frames from it, will act as the Content producer. While doing it, the app also stores approximately and consistently four seconds of frames on the local storage of the ICON Framework Node it contains. This app also listens for Interests requiring live streaming under the Name prefix ccn4d:liveStream, which in turn will be satisfied by the Contents stored in the local Node's memory storage.

A second application would play the Consumer role. As such, it will send Interest requests to the network using the Name ccn4d:liveStream/<CurrentDateTime.Ticks>, where CurrentDateTime.Ticks is a number that represents the current date and time. When the Content messages for the requested Interests are received, which contain the captured video frames, the app will show them in sequence on a window, reproducing the video captured by the camera on the producer Node.

During this data transmission network failures were forced which caused the live stream to stop, but as soon as the network connection between Nodes was restored, the video stream continued at the time where the network interruption was fixed.

These interruption tests were also performed on the tests described earlier in this chapter, and also produced the expected results. In fact the video prepared for the CCNxCON 2012 Conference, where the ICON Framework was presented demonstrates this situation, but these were essentially preliminary tests that revealed a stable solution, capable to address the targeted scenarios and environments, so a deeper analysis should be made to quantify and measure these results.

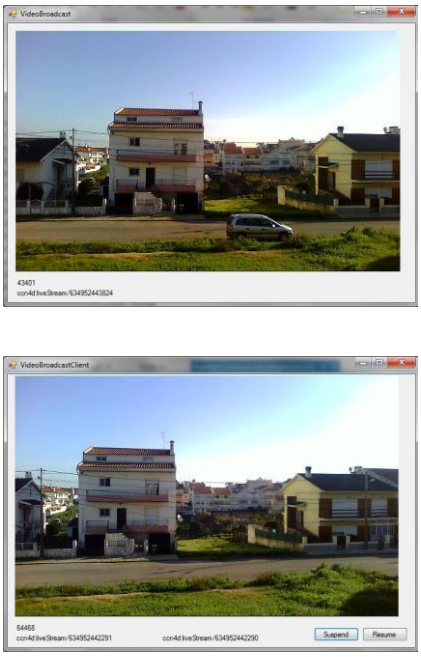


Figure 45 - Video streaming apps

6 Conclusions

Since the early days of computer networks, the world has gone through tremendous changes. The architecture of these networks has not changed much during the past half century, but the way we access digital information today is way too different from how we have done it on those early days.

Information Centric Networks are on the current agenda, its concepts can ensure that the use we make of networks can scale better to achieve the challenges of tomorrow. The CCNx project architecture proposes a solution based on hierarchical names to route data, retaining the simplicity and scalability if IP networks but offering much better delivery efficiency and disruption tolerance.

With the increasingly trend to access information using mobile devices, where communications typically are slower and unstable, Information Centric Networks open new perspectives of data retrieval, transport and delivery. Opportunistic networks built on top of such networks will benefit from their hop-by-hop data transfer nature, where data can be progressively carried without the need to flow through all the way again, when packets are missed or the device changes to another network.

The main objective of this dissertation was centered on studying, exploring and experimenting with Information Centric Networks and building a prototype of such a network which could be used on opportunistic networks under several platforms. Most of these goals were achieved and the tests results demonstrated that data transmissions could be more efficient and tolerant to network failures by the fact that the principles behind Information Centric Networks were used.

The produced prototype only addresses the basic concepts of Information Centric Networks, which means, data is transferred based on the requested information and any host details were left out of the protocol definition. Content for those requests is returned by traversing the reverse path using the breadcrumbs left on each Node whenever an Interest arrives on a communication interface. Data is persisted on the Nodes, which improved the efficiency and fault tolerance of communications.

The sample apps, used to test the prototype were simple to develop and at no point of code required to mention references to host of any kind. The apps only needed to ask for the Data they needed regarding where it was. These apps, that were built using the ICON Framework, were presented on September 12, 2012 during the CCNxCON 2012 Conference at INRIA Sophia Antipolis, France. The conference served to show the architecture of the ICON Framework in public for the first time, where some of the similarities and differences with the CCNx platform where discussed, being the cross-platform and modularity nature behind its design the points that showed most interest from the audience. The ICON Framework was presented with a poster, a video and the sample applications. The video is available on two locations for visualization or download:

- https://skydrive.live.com/redir?resid=30C3E79A52875CB2!1065&authkey=!ACD6ZQKBL5UjIAA
- http://www.youtube.com/watch?v=ExZ_wxLK1lY

Nevertheless, the work developed is just a prototype, a proof of concept we may say, which means there is a lot space for improvements. Security for instance, was left off for simplicity but it will not be hard to add it to the solution. Besides all performance tests and code analysis realized which greatly improved to final result, different usage scenarios will definitely find room for more improvements on some of the implemented algorithms. The future implementation of a Decision Engine module, which effectively could make decisions based on data gathered by the Node's statistics, will definitely bring major improvements to the solution.

Finally, the success of this dissertation can be resumed in the next six items:

- Brought knowledge about Information Centric Networks and Opportunistic Networks;
- Proposed an Information Centric Architecture called ICON Framework;
- Delivered code running for the implementation of the ICON Framework;
- Provided sample apps built on top of the ICON Framework;
- The ICON efficiency, as demonstrated by the tests performed;
- ICON Framework presented at the CCNxCON annual conference on September 2012

7 Bibliography

[1] Palo Alto Research Center - PARC, "CCNx," [Online]. Available: http://www.ccnx.org/. [Accessed 22 September 2012].

[2] University of California at Los Angeles (UCLA), "Named Data Networking," [Online]. Available: http://www.named-data.net/. [Accessed 17 August 2012].

[3] J. Scott, P. Hui, J. Crowcroft and C. Diot, "Haggle: a Networking Architecture Designed Around Mobile Users," in *Third Annual IFIP Conference on Wireless On-Demand Network Systems and Services*, 2006.

[4] University of Paderborn, "NetInf," [Online]. Available: http://www.netinf.org/home/home/. [Accessed 13 September 2012].

[5] F. Warthman, "Delay-Tolerat Networks (DTNs)," Warthman Associates, 2003.

[6] W. Moreira and P. Mendes, "Routing in Opportunistic Networks," in *Social-aware Opportunistic Routing: The new trend*, Springer, August, 2013.

[7] W. Moreira, P. Mendes and S. Sargento, "Assessment Model for Opportunistic Routing," *IEEE Latin America Transactions*, vol. 10, no. 3, April 2012.

[8] W. Moreira, P. Mendes and S. Sargento, "Opportunistic Routing based on daily routines," *IEEE AOC 2012*, June 2012.

[9] W. Moreira, M. de Sousa, P. Mendes and S. Sargento, "Study on the Effect of Network Dynamics on Opportunistic Routing," in *adhoc Now*, Belgrad, Servia, July 2012.

[10] T. Hossmann, F. Legendre and T. Spyropoulos, "From Contacts to Graphs: Pitfalls in Using Complex Network Analysis for DTN Routing".

[11] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall and H. Weiss, "Delay tolerant network architecture," IETF, 2007.

[12] N. Morais, B. Batista and P. Mendes, "CCN support for Information-Centric Oportunistic Networking," in *CCNxCon2012*, Sophia Antipolis, France, September 2012.

[13] Xamarin, "Mono," [Online]. Available: http://www.mono-project.com/Main_Page. [Accessed 1 August 2012].

[14] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, k. claffy, D. Krioukov, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley and E. Yeh, "Named Data Networking (NDN) Project," October, 2010.

[15] CCNx Conference, "CCNx 2012 Conference Program," 13 September 2012. [Online]. Available: http://www.ccnx.org/ccnxcon2012/program/. [Accessed 7 October 2012].

[16] B. Batista and P. Mendes, "ICON - Information and Context Oriented Networking," in *CCNcCon*, Palo Alto, USA, Sept. 2013.

[17] M. Chawla, T. Koponen, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker and I. Stoica, "A Data-Oriented (and Beyond) Network Architecture," *SIGCOMM'07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, p. 181–192, 2007.

[18] M. Gritter and D. R. Cheriton, "TRIAD: A New Next-Generation Internet Architecture," Stanford University, 2000.

[19] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs and R. L. Braynard, "Networking Named Content," ACM, Rome, Italy, 2009.

[20] Palo Alto Research Center, "About CCNx," [Online]. Available: http://www.ccnx.org/about/. [Accessed 15 September 2011].

[21] Z. Lixia, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, K. Claffy, D. Krioukov, D. Massey, C. Papadopoulos, T. Adbelzaher, L. Wang, P. Crowley and E. Yeh, "Named Data Networking (NDN) Project," Named Data Networking, October, 2010.

[22] Wikipedia, "Event-Driven Messaging," 7 June 2011. [Online]. Available: http://en.wikipedia.org/wiki/Event-Driven_Messaging. [Accessed 13 October 2010].

[23] Google, "Protocol Buffers," April 2012. [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed 28 October 2012].

[24] W. Moreira and P. Mendes, "Survey on Opportunistic Routing for Delay/Disruption Tolerant Networks," SITI, Lisbon, 2010.

[25] Google Project Hosting, "protobuf - Protocol Buffers - Google's data interchange format - Google project hosting," September 2012. [Online]. Available: http://code.google.com/p/protobuf/. [Accessed 13 September 2012].

[26] M. Gravell, "protobuf-net - Fast, portable, binary serialization for .NET - Google Project Hosting," September 2012. [Online]. Available: http://code.google.com/p/protobuf-net/. [Accessed 28 October 2012].

[27] CCNx, "CCNx Protocol," November 2011. [Online]. Available: http://www.ccnx.org/releases/ccnx-0.2.0/doc/technical/CCNxProtocol.html. [Accessed 13 September 2012].

[28] E. Nordström, P. Gunningberg and C. Rohner, "A Search-based Network Architecture for Mobile Devices," Uppsala University, 2009.

[29] A. Kennedy and D. Syme, "Design and Implementation of Generics for the .NET Common Language Runtime," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, USA, 2001.

[30] Wikipedia, "Generic programming," 13 10 2013. [Online]. Available: http://en.wikipedia.org/wiki/Generic_programming.

[31] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," 23 January 2004. [Online]. Available: http://martinfowler.com/articles/injection.html. [Accessed 12 January 2013].