Design and Implementation of a Conversion Service for Content Centric Networking

Master Thesis

by

Elham Cheriki

under the guidance of

Prof. T. Braun
and
Prof. L. Mueller

Department of Electrical Engineering
Bern University of Applied Science
and
Institute of Computer Science and Applied Mathematics
University of Bern

January 2012

Contents

List of Figures

Chapter 1

Summary

Content Centric Networking (also content-based networking, data-oriented networking or named data networking) is an alternative approach to the Internet architecture of computer networks. Its founding principle is that a communication network should allow a user to focus on the data he or she needs, rather than having to reference a specific, physical location where that data is to be retrieved from. This stems from the fact that the vast majority of current Internet usage (a "high 90% level of traffic") consists of data being disseminated from a source to a number of users. [1]

Content Centric networking comes with potential for a wide range of benefits such as content caching to reduce traffic and improve delivery speed, simpler configuration of network devices, and building security into the network at the data level. However, the change of communication paradigm may pose problems for certain types of network activities, for instance for real-time multimedia applications, but recent research indicates these applications are feasible.[1]

While CCN strongly focuses on content retrieval, the Future Internet is expected to provide a more general support of services. Content delivery is merely one example of a service; other examples are content generation and manipulation as well as general processing services. A service-centric networking (SCN) scheme as an extension of CCN which considers both content and service as key design elements has been proposed in [11]. SCN could support a variety of services including file storage and retrieval, audio/video streaming and recording, processing of stored images and video, on-line shopping, location-based services, cloud computing, and tele-communication services. [11]

This master thesis, tries a new Internet architecture, Content Centric Networking and emphasizes on strengths of CCN. It is implementing a demonstration for SCN based on the open source software, CCNx as a proof of the concept. The example service supports image conversion . The demonstration will show the advantages of CCN/SCN in such an application scenario.

Chapter 2

About CCNx

2.1 Introduction

The following content is an overview of CCN. For more information refer to [2]. The current Internet was invented in the 1960's and 70's. The aim of the creation of the Internet was to share the resources because at that time resources were expensive. The idea of sharing resources led to the creation of a communication model based on connections between two hosts.

After 50 years of the creation of the Internet, the technology was evolving and computers and resources became cheaper. Nowadays the Internet serves many services to clients based on the communication model between two hosts. The need for data and content is increasing and the current Internet can not serve the increasing request of the clients for content efficiently.

Therefore the future Internet has to evolve to a communication model based on content. The current Internet is still speaking the language of connections between two machines. Most clients are concerned only about data and content and they do not care where the data is stored. They care about the data not the name of the host that it is stored on. The clients value the Internet because of what to get, from not where to get it.

Today's Internet has some limitations which is hoped in the future Internet these limitation would be solved.

Some of the limitation that the current Internet has:

- Scalability: The number of IP addresses is limited.
- Security: TCP/IP suffers from lack of security.
- Delay: In real time communication because of limitations in bandwidth and traffic, there is a delay.

A uniform solution to these limitations is to replace the current communication model, which is based on host names, to a communication model based on content names. The Content Centric Networking was introduced by Parc institute by V. Jacobson [9]. CCN brings some benefits compared to the current Internet, which are as following:

- Security: CCN signs the data itself and then sends it over the network
- Simplicity: it is simple and robust
- Fast: reduces traffic between servers
- Distributes the contents efficiently

2.2 CCN Components

Some of the most important CCN components are listed below:

2.2.1 CCNx

CCNx is a protocol for the communication model of CCN, which is built on named data. CCNx delivers named data content instead of connections from hosts to other hosts. Every packet will be cached at any CCNx router, which leads to a very efficient use of the network when a group of clients is interested in the same content. For example if one client requests a data packet for the first time, the data will be cached in all routers on the way to the client and if another client requests exactly the same content, the content will be delivered from the nearest router.

Now applications run the CCNx protocol on top of UDP to take advantage of the existing IP layer. CCNx supports a wide range of applications. CCNX is implemented for the communication between applications and it is intended to be integrated into the application layer rather than being a separate layer.

2.2.2 Name

In CCNx, names are very important, because the contents is looked up by the name, regardless of the address of any machine that is involved. Names in CCNx are human readable and CCNx uses the same principles as an IP address, like net, subnet, etc. to find the longest match in the network. Names in CCNx are hierarchically structured and do not have any fixed length.

2.2.3 Node

In the CCNx network, there is an entity called node that implements forwarding and buffering.

2.2.4 Type of Messages

There are two types of messages in CCNx:

- *Interest Message* is used to request data.

- *Content Object* is used to respond to the Interest message.

2.2.5 Data Structure

A node in CCNx contains three data structures [9]:

- *Content Store (CS)* is a buffer memory that keeps the retrieval Content Object by longest match lookup on names.
- *Pending Interest Table (PIT)* keeps track of Interests forwarded upstream toward content source so that returned data can be sent downstream to its requester.
- *Forwarding Information Base (FIB)* is used to forward Interest packets toward potential source of matching data.

2.3 Processing Interest Messages

An Interest message is processed according to the following procedure. First when a node receives an Interest message, it will look up the Interest message in the CS. If there is a Content Object matching the Interest message, the Content Object will be transmitted to the source of the Interest message. Second, If no match is found in the CS, the Interest message will be looked up in the PIT. If there is a matching Interest message in the PIT, it means that another Interest message exactly the same Interest has already been forwarded and is pending. Then the new Interest message is discarded. Third, if a match is found in the FIB, it means that an entry was created in the PIT and the Interest message is transmitted to different destinations to find the match for an Interest message. If no match is found in the CS, the PIT and the FIB, it means that there is no way to find the match for an Interest message and it will be discarded.

2.4 Processing Content Objects

A Content Object is processed according to the following procedure. The first lookup is executed in the CS. If a match is found in the CS, it means that the new Content Object is duplicated and the new Content Object will be discarded. The second lookup will be done in the PIT. If there is a match in the PIT, it means that the Content Object is transmitted to all clients that requested it. Third, if no match is found in the CS and the PIT, it means that the content is unsolicited data and it will be discarded.

2.5 An Example of Exchanging Messages

Suppose a client requests a content and sends an Interest message. The Interest message is forwarded to available nodes in the network. One of the nodes which has the match Content will respond to the Interest message and therefore satisfies the request.

For each Interest message there is just one Content Object sent back. If one node has two or more Content Objects which match the Interest message, only one of them will be sent as a response to the Interest message. The reason for sending only one is to keep the flow balanced. To select which Content Object has to be picked as a response to the Interest message, there is a component in the Interest message that determines which one has to be picked . This is called suppression mechanism.

CCNx is reliable in regards to the transport of messages. To provide reliable delivery, Interest messages that do not get any responses from other nodes, are resent after a period of time.

Chapter 3

Content and Service Centric Networking

3.1 Content Centric Networking (CCN)

In the current Internet architecture, the hosts and the network have very different roles. Hosts generate and consume packets; the network is in charge of delivering those packets. The Internet architecture has no inherent notion of "content". In the Internet, content resides in applications that themselves reside on specific hosts. In order to access content across the Internet, a host first needs to determine a host that holds (a copy of) the content of Interest and then it needs to obtain the specific IP address at which that hosts resides at the time. [8]

In Content Centric Networking, content becomes a first-order element. It is liberated from the shackles of Internet application silos, and the role of the network changes from transporting topologically addressed packets between hosts to delivering uniquely identifiable content to the hosts requesting it. With this approach, hosts no longer need to identify which other host stores a copy of the content of Interest; they simply request a named piece of content from the network and let the network to worry about where to retrieve it from. [8]

CCN offers these benefits compared to TCP/IP: [10]

- Optimal content distribution
- Painless mobility, wireless, virtualization, ...
- Same scalability and efficiency as TCP/IP
- Simple, secure, robust configuration
- An easy, incremental, evolutionary path
- Much better security

3.2 Service Centric Networking (SCN)

For extending Content Centric Networking, in paper [11] it has been proposed to support general services. Data can not be just retrieved, but also can be processed before being presented to clients. Names will not be used just for invoking content, but could be used to invoke services as well. Services and content are two different things and can be in different locations in network. This Service Content Networking (SCN) provides explicit addressing for both entities. SCN leverages the concept of the CCN infrastructure. A client sends an Interest message for the service to be invoked and the results from the service execution are returned in a Data message.

SCN brings several advantages and benefits compared to conventional services which are as following:

- In traditional service scenarios, services must be registered by the service provider at the registry and must be looked up by the client before the service is invoked. In SCN, the service registration is replaced by an announcing service, that is available in the underlying CCN infrastructure, i.e., in the CCN routing tables. It means that in SCN, services will be registered in CCN . As soon as an Interest message requests one of those services, CCN will forward the Interest to the service provid-er and there is no need for any server to be addressed. The service provider could be anywhere in the network and even in multiple places to more convince clients.
- As written above, SCN leverages the features of the underlying CCN infrastructure. For example, when one clients requests a service, data which is provided by the service can be cached in the routers and can be cached in different nodes. If some other client requests the same service, it will be loaded from the nearest router or node and the benefit of caching reduces the network traffic and response times specially for popular content and services.
- In a conventional service provider network a server must be involved. When a client requests a service, by contacting the server and considering the location of the clients, the closest server will be detected. In SCN, services that distribute routing appropriately can be deployed in multiple locations. Service requests are routed to the closest server for processing the request independent of the location of clients.
- In SCN, when taking advantage of CCN infrastructure, the request will be forwarded to the closest and most appropriate location. With this help, the distance between server and client will be optimized. When a service or a content has already been requested and retrieved, they can get cached for a faster second retreival by the local ccnd where is the closest distance between client and data content.

SCN could be useful for a variety of services such as file storage and retrieval, audio/video streaming and recording, processing of stored images and video, e-commerce applications like ticket ordering, e-banking, on-line shopping, location-based services (gas, food, travel, weather, events etc.), cloud computing, e.g., to instantiate virtual machines and databases as well as telecommunication services.

3.3 Project Vision

As we know services provide or execute processing of data. Hence services need data to process and need some place to store data. Therefore, SCN has to support both data and services. This means that names could use the same for invoking both data and service. With this method a client could request a data with the same name scheme as requesting for a service, e.g: */images.google.com/converting/*(service) or */ccnx.org/ccnx.pdf* (data).

This master thesis investigates the new Internet architecture, Content Centric Networking and capitalizes on strengths of CCN and is implementing a demonstration for SCN based on the open source software CCNx as a proof of concept. The demonstration will show the advantages of CCN/SCN in such an application scenario. This application provides an image conversion service for clients such that possible formats are available to its user clients. Data will be processed by the service provider before it is presented to clients. The conversion service will prove that names could be used to invoke data and services in the same way. [3]

The goal of this project is to design a method to invoke a service, which could be requested by CCN clients with the same name structure as CCN for retrieving data. This conversion service can provide its data by storing it itself. After this process, it will cache the data in the routers with help of the CCN infrastructure.

The data will therefore remain in the cache of the routers and can be retrieved by other clients. Measuring time for retrieval, will prove the advantages of SCN with taking advantage of CCN.

Chapter 4

Evaluation of Different Design Approaches

In this chapter different approaches are described and evaluated. The most appropriate approach will be selected.

4.1 Design Issues

For the implementation of the service, it is better to consider these characteristics:

1. Deploy the code into an existing infrastructure without having any change in the current system which is already running on the nodes. For example, TCP/IP is stable and well-established. CCN has the same characteristics as it can run over any protocol, including IP. Using an existing model that is compatible with the conventional infrastructure brings some advantages regarding the ease of the implementation.
2. The behavior of the deployed service should not depend on the network state. For example, if a service needs a file in order to process its data, the file needs to be fetched, then the service can run, then start processing the data contained within. The service has to give correct results in any situation, independent of machine state.
3. The end users also need to be aware that the data can be modified by the service. This can occur when document format types are changed particularly with images that quality is important. For example, if a client requests a JPEG format of a file while the original file is BMP, the conversion service leads to loss in the quality of the image.

4.2 Options for Implementation

Regarding the properties of the implementation, there are three possibilities as following:

1. Implementation of service in the ccnd (could be applied to one node or different nodes)
2. Service at publisher side (e.g: file proxy server)
3. Service as separate application that could be deployed on any node with a running ccnd

The three different approaches will be outlined, along with their advantages and disadvantages in the next section.

4.3 Approach 1: Implementation of Service in ccnd

The ccnd is the software forwarder/router for CCNx and is required for normal CCNx protocol communication. The typical configuration is to run one ccnd on each host. The applications running on the host will communicate through the local ccnd, and it will communicate over attached networks. [2]

The implementation in the ccnd requires the ccnd to provide a conversion service to the users. It means that we could send an Interest to the local ccnd and the ccnd will modify the Interest message, which has been forwarded for an image with a specific format. See Figure 4.1.

For example, in Figure 4.1 a client requests a file *PIC.BMP* whereas the publisher only has *PIC.JPEG*. In this approach the conversion service would be transparent to the client. It means that the client is not aware of the service. It requests its file from its local ccnd. As one option to deploy the service in the ccnd is such that the ccnd removes the format of the file and just look for the file's name without format.

But this approach has its own advantage and disadvantage as mentioned below:

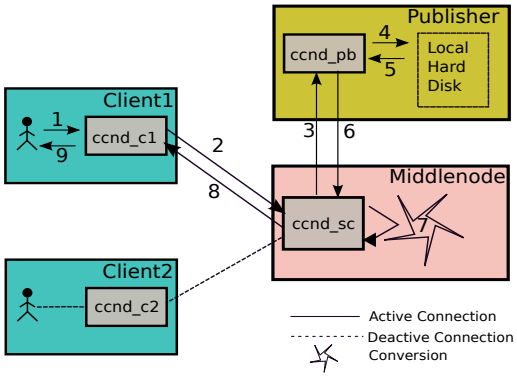

Figure 4.1: Approach 1: Implementation of Service in ccnd

4.3.1 Advantages

The service implemented by ccnd could be transparent to clients.

4.3.2 Disadvantages

1. To allow all the clients within the network to use the service, we must change all or most ccnd nodes. Since the Interest message will be forwarded to different ccnds, all of the ccnds have to be able to analyze and understand the Interest messages. This design will not follow the first design concern about not changing existing infrastructure.
2. There is no clear way to select an original file. Suppose the publisher has different files with different formats *PIC.JPEG* and *PIC.GIF*. With this solution the ccnd is not able to decide which file is the original file. It is also possible that these two files might have different content as they have different formats.
3. Suppose a client requests a file, *PIC.BMP*, and the the original file is another format like JPEG. After conversion the client receives the converted file. The client is not aware of any conversion that have occurred to the file. The user may be expecting a good original quality file but after conversion the quality of the original file, *PIC.BMP* is missing. The service will automatically convert another file such as *PIC.PNG* to *PIC.BMP* without informing anyone in the chain. The user may be unsatisfied with the converted image quality. But there might be a solution for this problem. One could also determine and request the quality of image as a parameter.
4. There are also legal issues involved with the process of changing original files. It is possible that the publisher does not want another copy of the original image in another format. This could play an important role in which method is chosen.

4.4 Approach 2: Service at Publisher Side

Using this approach, the conversion service is provided by the publisher. The publisher has to be modified such that the service code is implemented on the publisher. The publisher is also responsible for all the requests for all files on its local hard disk. Suppose a client requests *PIC.BMP* and the publisher only has *PIC.JPEG*. The publisher will announce to the client that the file *PIC.BMP* is available on its local hard disk. This file is selected and converted from *PIC.JPEG* to *PIC.BMP*. The publisher forwards the converted file to *ccnd_pb* then it will be forwarded from the *ccnd_pb* to *ccnd_c1*. See Figure 4.2.

Let's have a look what are the advantages and disadvantages of this approach:

4.4.1 Advantages

1. With this approach the publisher has full control over the conversion.
2. The publisher is responsible for the content data of the images.
3. The publisher is fully aware of all conversions that are done.
4. The files that are retrieved by the clients are signed by the publisher.

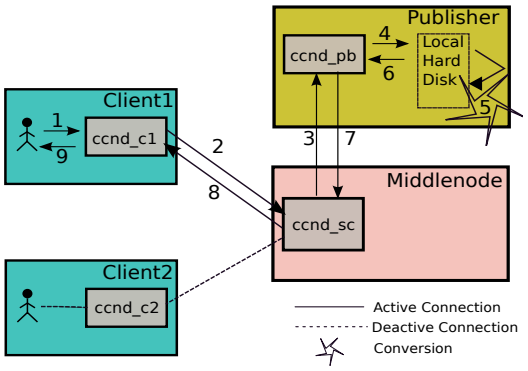

Figure 4.2: Approach 2: Service at Publisher Side

4.4.2 Disadvantages

1. The publisher has to be modified. There are many publishers in the network which are providing original files. With this approach, all publishers have to be modified to serve the conversion service to all clients, that is not a good solution.
2. Every time a client needs a different file format, the client has to ask again the publisher. For example, the client first asks for *PIC.BMP*. After that the client needs the same file in another format, so it asks for *PIC.GIF*. Again, the request will then be forwarded to the publisher. With this method, we cannot take advantage of the benefits of CCN. The amount of traffic would be neither optimized nor reduced.

4.5 Approach 3: Service as Separate Application

In this approach, there is a separate service which can be installed on any node. This service is responsible for everything regardless of its location. This approach will be explained in details in Chapter 5 and 6.

When this method is used, the client is requesting the original file in the same format as that the user is requesting. The client will call the conversion service, the name of the original file, and the required format are in one Interest message. This message will be forwarded to the conversion service. The conversion service will handle the Interest message, provides the original file, converts it, and delivers it to the client's ccnd. See Figure 4.3.

4.5.1 Advantages

1. In this approach the original file that has to be used for the conversion is clear. Because the client mentions which file has to be used to convert to the requested format.
2. The requested format is clear and the client demands the exact file required.

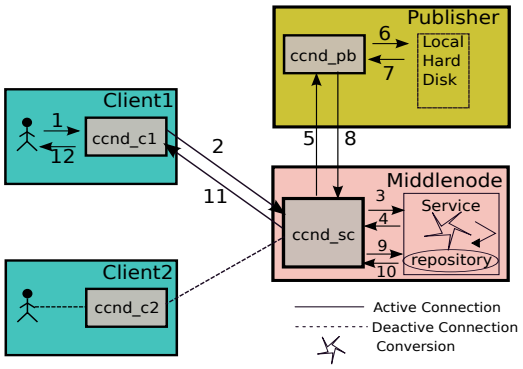

Figure 4.3: Approach 3: Service as Separate Application

3. The user will be aware of any conversions and any changing in the quality of image. There might not be any dissatisfaction because the client itself requested the conversion.
4. All nodes in the network can use the service without any modification on their sides. This service can be installed in one node or in multiple nodes.
5. All available sources on the network can be converted.
6. The existing infrastructure of the ccnd will not be modified. With this method we do not touch any existing infrastructure that is already running.
7. It is easy to add other applications to this service or to change to different services without affecting the service conversion application.
8. All the advantages of CCNx could be kept with this concept.
9. The source file will be cached by all nodes near the service.
10. Any converted files would also be cached in the network, automatically propagating the cached files.

4.5.2 Disadvantages

1. Only one client is processed at a time. It means if the two clients requests at the same time, just one of them will be handled and the other client's request will be pended. This limitation only exists because the current implementation is a proof of concept only. If required, t could be easily removed.
2. Clients will receive a file with a signature of the service conversion not with a signature of publisher.

4.6 Selected Design

When evaluating the advantages and disadvantages of each approach and design concern, it is clear that the third approach benefits from all the advantages of SCN and brings different possibilities. The most valuable aspect of this approach is the file

caching of different formats in the routers. This is done to avoid redirecting requests to the publisher, thereby reducing unnecessary traffic to the publisher. This allows the same name to be used to invoke services and data. The implementation of this approach to the existing infrastructure is detailed in Chapter 5 and 6.

Chapter 5

Implementation of the Conversion Service

5.1 Conversion Service Process

In this section, the different steps of the conversion service are described and shown in Figure 5.1. The applied changes in source code are also outlined.

In this implementation, the client specifies or requests from its local ccnd, 3 entities in the Interest message:

- Requesting service conversion
- Required conversion format
- Original or source file with specific format

As an example of an Interest message:

`ccnx:/ccnx.org/sc/BMP/ccnx.org/files/PIC.JPEG/`

The following steps will be executed in conversion service:

- **Step 1**

  When a client sends this Interest message to its local ccnd, the ccnd forwards this Interest message to different ccnds in the network. Once this node with the conversion service receives this Interest message, it divides the Interest messages to different parts. The conversion service extracts the requested original file' name `/ccnx.org/files/PIC.JPEG` and the requested conversion format `/BMP/`.

```
1 // get original file
2 final String format = name.substring(13, name.indexOf("/", 16));
3 final String origName = name.substring(name.indexOf("/", 16)+1);
4 // create temporary file
5 final File tmp = File.createTempFile("wefwef", "wefwef");
6 final File tmpConverted = File.createTempFile("wefwef", "wefwef");
7 final OutputStream tmpStream = new FileOutputStream(tmp);
```

Listing 1: ExtractingFormat.java

- **Step 2**

  In this step, the conversion service requests the original file via the local ccnd. There are now two options regarding the original file: The local ccnd might already have a cached copy of the original file from a previous request or the ccnd does not have the file and must now retrieve the original file from the publisher. After that the ccnd of the conversion service delivers the original file to the conversion service.

```
1  // first retrieve file from source
2  System.out.println ("STARTING GETFILE " + new Date());
3  ContentName argName = ContentName.fromURI("ccnx://"+origName);
4  CCNHandle handle = CCNHandle.open();
5  InputStream stream = new CCNFileInputStream(argName, handle);
6  byte[] buffer = new byte[2048];
7  int len;
8  while ((len = stream.read(buffer)) >= 0)
9    tmpStream.write(buffer, 0, len);
10 stream.close();
11 tmpStream.flush();
12 tmpStream.close();
13 System.out.println ("FINISHED GETFILE " + new Date());
```

Listing 2: getFile.java

- **Step 3**

  The conversion service converts the original file to the requested format BMP specified in step 1.

```
1  // convert file
2  System.out.println ("STARTING CONVERT " + new Date());
3  BufferedImage image = ImageIO.read(new FileInputStream(tmp));
4  FileOutputStream sso = new FileOutputStream(tmpConverted);
5  ImageIO.write(image, format, sso);
6  sso.close();
7  System.out.println ("FINISHED CONVERT " + new Date());
```

Listing 3: ConvertFile.java

- **Step 4**

  The conversion service executes a ccnputfile (see appendix A) and calls this method to upload the converted file to its own repository. Calling this method, ccnputfile, causes an upload of the file via the ccnd to the repository of the conversion service. It means that the contents of the converted file first will be transfered to the ccnd of the conversion service, then to the repository.

```
1  System.out.println ("STARTING UPLOAD " + new Date());
2  ccnputfile put = new ccnputfile();
3  put.write(new String[] {interest.name().toString(), tmpConverted.
4  getAbsolutePath().toString()});
5  System.out.println ("FINISHED UPLOAD " + new Date());
```

Listing 4: UploadFileToRepository.java

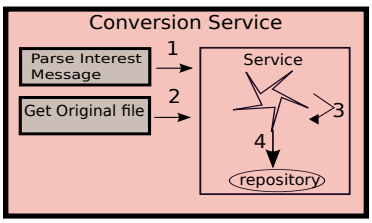

Figure 5.1: Service Conversion Operation

What is a file proxy?

The file proxy provides a read-only view of the directory hierarchy that we point it at, and generating the necessary content objects. It does not allow writing of files. The ccnFileProxy is a simple proxy making local files available via CCNx. For more information refer to appendix A.

What is a repository?

Generically a repository refers to a central place where data is stored and maintained. A repository can be a place where multiple databases or files are located for distribution over a network, or a repository can be a location that is directly accessible to the user without having to travel across a network. [4]

What is a file system?

Computers use particular kinds of file systems to store and organize data on media, such as a hard drive. Any place that a PC stores data is employing the use of some type of file system. A file system can be thought of as an index or database containing the physical location of every piece of data on a hard drive. [5]

5.2 Modification of CCNx to Implement the Conversion Service

In this project there are some implementations needed in the files *LogStructRepoStore.java* and *ccngetfile.java* of the open source CCNx. This new version of CCNx will be installed only in the node that has the task of the conversion service or on the service provider side. In the last page, all the modifications are deployed to *LogStructRepoStore.java*. This file has the default task of storing and reading files to and from the repository. Figure 5.2 shows the conventional process between ccnd and the repository. Figure 5.3 shows the modified version of process between the ccnd and the repository for implementation the conversion service.

There are three situations in which the conversion service is not necessary. In the *LogStructRepoStore.java* these three condition will be checked in case of receiving an

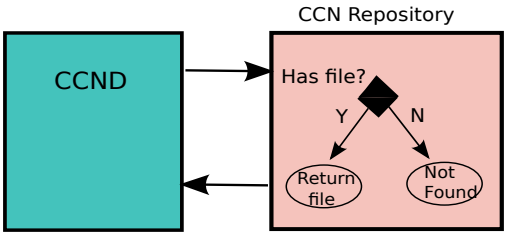

Figure 5.2: Conventional Repository Process

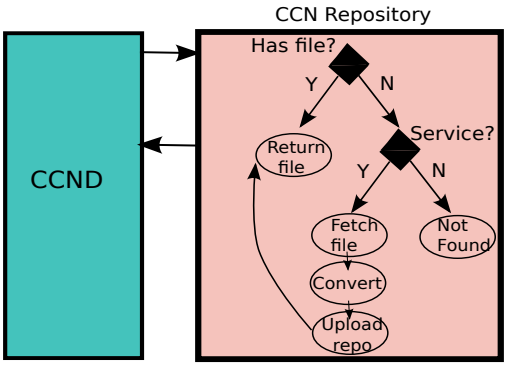

Figure 5.3: Modified Repository Process and Implemented Conversion Process

Interest message. If one of the conditions becomes true, the *LogStructRepoStore.java* will be run conventionally. As shown in Figure 5.2, no conversion service is needed. The three conditions which controls a conversion is required or not, are listed below (the conversion service is invoked if any of the conditions is true):

1. The conversion service is not requested.
2. There is a write command to the repository.
3. The requested file is cached in the repository.

If the first condition is met, then the Interest message is just a conventional Interest message for retrieving data without any services. For example, if a client requests ccnd to retrieve/store a file to/from the repository, then there is no need to execute the modified part of *LogStruRepoStore.java* .

The second condition is met when there is a command to write or upload a file to the repository. In this situation, there is no need for the conversion service to be run. In CCNx, the command for writing the data content to the repository is added at the end of the Interest message e.g: `/%C1.R.sw/`. It means that if an Interest message has at its end this command, data has to be written or saved in the repository. We need this condition because there is a *synchronized* block in the source code. In the synchronized block, only the first Interest is allowed to enter. Until the original file is retrieved and converted, the rest of the Interests will not enter to this block. Once the conversion

```
final String name = interest.name().toString();
System.out.println ("getContent(): " + name);
String base = name;
if (name.indexOf("/%") > 0)
base = name.substring(0, name.indexOf("/%"));
try{
final Object isCached = _index.get(Interest.constructInterest
(ContentName.fromURI(base), null, null, null, null, null), this);
if (!name.startsWith("/ccnx.org/sc/") || (name.contains("/\%C1.R.sw/"))
    || (isCached != null)) {
  // default handling
  System.out.println ("getContent(): default handling for: " + name);
  ContentObject co = _index.get(interest, this);
  if( Log.isLoggable(Log.FAC_REPO, Level.FINE) )
  Log.fine(Log.FAC_REPO, "Looking for: " + interest.name() +
        (co == null ? ": Didn't find it" : ": Found it"));
  return co;
          }
```

Listing 5: Conditions.java

service wants to upload the file into the repository, the uploading command which is `/%C1.R.sw/` will be added at the end of the Interest message. The source code is shown in Listing 7.

What is synchronized block?

The Java synchronized keyword is an essential tool in concurrent programming in Java. Its overall purpose is to only allow one thread at a time into a particular section of code thus allowing us to protect, for example, variables or data from being corrupted by simultaneous modifications from different threads. At its simplest level, a block of code that is marked as synchronized in Java tells the JVM: only let one thread in here at a time. [6]

The third condition is met when the converted file is already cached. For example, if the client requests a file and then a second client requests exactly the same file. In this situation, the converted file will be already in the cache of all ccnds and there is no need for the conversion service. Only the part of the source code needed to read the data from the repository will be executed, Figure 5.2, since the converted files will be stored in the repository after the conversion service is run.

5.3 Challenges and Solutions

There were some challenges and improvements that lead to faster performance and better results. The challenges and improvement are listed below.

- The first challenge involved the conversion service. It took a lot of time for large files to convert, and since the conversion time for larger files was larger than the timeout, the Interest message was discarded before the conversion was completed. On the client side, in *ccngetfile*, Listing 6, there is a change in the timeout variable since the default variable was set to 10 seconds in the CCNx code. The timeout variable is the maximum amount of time for which the Interest will be waiting to get an answer from the ccnd. After expiry, the Interest message will be discarded. Then the client can not retrieve any data contents. So the experimental timeout for first Interest is set from default value, that is 10 seconds to 60 seconds.

- On the publisher side, *ccnfileproxy* was first used (Appendix A), but ccnfileproxy creates the content objects from the file in the file system as requested. The signing takes place at this time with the user's key. But if files are stored in the repository, the retrieval time is much faster than using the fileproxy. The reasons for which it is faster is related to the signing requirements. In the repository, the contents of the file is only signed once. However, in ccnfileproxy, the file needs to be signed for each request, resulting in a slower process.

  The ccn repository stores content objects as they are received from a client, in the on-the-wire format, so the content has already been signed. This means that there is very little work to do when responding back with those content objects. The signatures are therefore those of the original client, not the repository. That's why using the files which are stored in the repository at the publisher is much faster than using *ccnfileproxy*. The evaluation test is done using the repository but we can not control the publisher. It is not our choice to take the file from the repository or via the *file proxy*, but the decision of the publisher.
- In the first implementation of the conversion service, the converted file was stored with the help of file proxy, but after getting aware of the poor performance, the conversion service now uses a repository to save the converted files. If another client asks for the same file, the requested file is already saved in the repository of the conversion service.

Chapter 6

Detailed Analysis of the Conversion Service Concept

In this chapter we discuss three scenarios which can show how the conversion service works in different situations. First, there is an example which shows conventional downloading of a file. After that we compare it with different scenarios of the conversion service.

6.1 An Example

To clarify how clients download a file with using CCNx protocol, it is better to have an example with the related command lines and figures to show clearly how the process of forwarding Interest messages and data contents works. See Figure 6.1.

The publisher first has to register to ccnd with this command and assign where ccnd can find files. Files are in the home documents directory and this directory will be registered as `ccnx:/ccnx.org/files/`.

```
./ccnfileproxy ~/Documents/ ccnx:/ccnx.org/files/
```

The client has to run the ccngetfile command to get the file *PIC.PNG* through CCNx with the following command and to save in another file named *PIC1.PNG*.

```
./ccngetfile ccnx:/ccnx.org/files/ PIC1.PNG
```

6.2 Scenarios

In this section, we define different scenarios. In the first scenario, we suppose none of the nodes have the requested file and the original file has to be loaded from the publisher. In the second scenario, we suppose that the original file is already cached

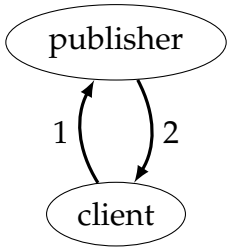

Figure 6.1: Conventional Downloading

in the ccnd of the conversion service and the client asks the same conversion service. In the last scenario, the client asks for another format of the same original file.

6.2.1 Topology

Our topology consists of 4 types of node:

- Publisher
- Client1
- Client2
- Middlenode which has the conversion service

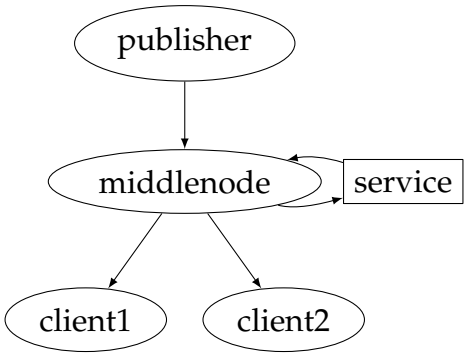

Figure 6.2: Topology

6.2.2 Scenario 1

In this scenario, Figure 6.3, client1 requests the original file, *PIC.JPEG* and then asks for the BMP format from its local ccnd. As outlined in the last chapter, this Interest message will be forwarded to the connected ccnds in the network. The node with the conversion service registered itself to ccnd under the name of `ccnx:/ccnx.org/sc/`. It will receive this Interest message and the conversion service parses this Interest message into different parts or names to specify different parameters of the conversion service. The first part, `ccnx:/ccnx.org/sc/`, is requesting a conversion services which helped to receive this Interest message by the conversion service. The second part, `/JPEG/`, specifies which format is requested. The third part, `ccnx.org/files/PIC.BMP`,

determines which file should be fetched as the original file.

```
./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP
```

The first Interest message will be received by ccnd of client1. If the ccnd of client1 does not have this file, it will forward the Interest message to the ccnd of the middlenode. The Interest message starts with `ccnx:/ccnx.org/sc/`, thus one of the If conditions, mentioned in section 5.2, is requesting for the conversion service.

In the conversion service, the Interest message, `ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP` is parsed into two parts: the first part, `ccnx:/ccnx.org/sc/JPEG/` is requesting the conversion services, which helped to receive the Interest message by the conversion service; The second part of Interest, `ccnx.org/files/PIC.BMP`, is not yet fulfilled since the service conversion does not yet have the file, PIC.BMP. The conversion service then forwards the second part of the Interest to the connected ccnd. As defined in section 5.1, the original file is available in the publisher under the the name of `/ccnx.org/files/PIC.BMP`. The Interest message will therefore be forwarded to the ccnd of the publisher. The Publisher will send the *PIC.BMP* from its local hard disk to its ccnd. The ccnd of the publisher will then send *PIC.BMP* to the ccnd of the middlenode. Once the conversion service fetches the complete file, the ccnd of the middlenode will send it to the conversion service. The conversion service will start to convert BMP to JPEG.

Following the completion of the conversion, the conversion service automatically runs ccnputfile. The ccnputfile makes a request to ccnd of the middlenode to put *PIC.JPEG* in the repository of the conversion services via the ccnd. During uploading of the file to the repository via ccnd, the ccnd of the middlenode retrieves contents of the converted file. The Interest messages are answered and ccnd starts to give the data to the client. Hence the client retrieves the converted version, the *PIC.JPEG* during upload.

These steps are shown below as well in Figure 6.3.

1. Client1 first sends an Interest message to his own local ccnd:

   ```
   ./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
   ```

2. The ccnd of client1 does not have this file and forwards it to the nearest ccnd, which is the ccnd of the middlenode.

   ```
   ./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
   ```

3. As the conversion service is registered with the name of `ccnx:/ccnx.org/sc/` in the ccnd, the Interest message will match to this name and will be forwarded to the conversion service.

   ```
   ./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
   ```

4. The conversion service receives the Interest message. It divides the Interest message into different parts. The first part requests the conversion service, which is already done and the second part is the requested format `/JPEG/`. the third part of the Interest message, which is `/ccnx.org/files/PIC.BMP` will be considered as the requested of the original file.The conversion service forwards to the ccnd of middlenode to get the original file.
5. The ccnd of the middlenode will search for a match among the nearest ccnd, which has the original file. As the publisher has registered its file under the name of `ccnx:/ccnx.org/files/PIC.BMP`, the ccnd of middlenode will send it to the ccnd of the publisher.
6. The ccnd of the publisher will send the Interest message to the publisher. The publisher could be a repository or a file proxy. As we defined in section 6.1, the original file is under the name of `ccnx:/ccnx.org/files/PIC.BMP`. The publisher retrieves the file from the underlying data store.
7. The publisher sends the original file to its ccnd.
8. The ccnd of the publisher will send *PIC.BMP* to the ccnd of the middlenode, because the conversion service requested this file from the ccnd of the middlenode.
9. The ccnd of the middlenode delivers the original file to the conversion service.
10. Once the conversion service gets the whole file, it starts to convert. After the conversion, the conversion service starts to upload the file via the ccnd of the middlenode.
11. The ccnd of the middlenode forwards the request of uploading to the repository and starts to upload the converted file to the repository.
12. During uploading to the repository, *ccnd* has partially the converted file, *PIC.JPEG* and starts giving the file to the ccnd of the client, because it is requesting this file and the Interests are pending for retrieval. The ccnd of middlenode will send the *PIC.JPEG* to client1.

Here it is necessary to mention that from the client, several Interest messages will be sent to the ccnd of the client and further to other available ccnds. The implementation of the conversion service will allow just the first Interest message to proceed into the conversion service. Other Interest messages will be pending until the conversion is finished and the upload starts. For more details refer to chapter 5.

Now after these steps we look at each node, which files it has:

- publisher: PIC.BMP
- ccnd_pb: PIC.BMP
- ccnd_sc: PIC.JPEG and PIC.BMP
- conversion service repository: PIC.JPEG
- ccnd_c1: PIC.JPEG
- client1: PIC.JPEG

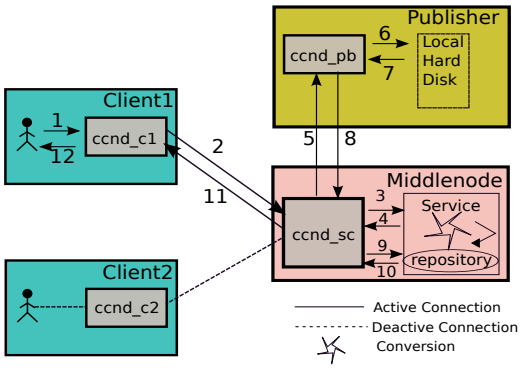

Figure 6.3: Scenario 1

6.2.3 Scenario 2a

Assumption: The state is continuing from scenario 1. In this scenario, client2 requests the same file, *PIC.JPEG* and sends this request to its ccnd. The following steps will happen. See Figure 6.4.

```
./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
```

Now we follow these steps again what happens in CCN:

1. Client2 first sends an Interest message to its own local ccnd:

   ```
   ./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
   ```

2. The ccnd of client2 does not have this file and it forwards it to nearest ccnd which is the ccnd of the middlenode.

   ```
   ./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
   ```

3. Since *PIC.JPEG* was cached through *ccnd_sc* in the last scenario, the converted file is in the cache of the ccnd of the middlenode, see Figure 6.4. So *ccnd_sc* will send back *PIC.JPEG* to the ccnd of client2.
4. The ccnd_c1 will send the converted file to client2.

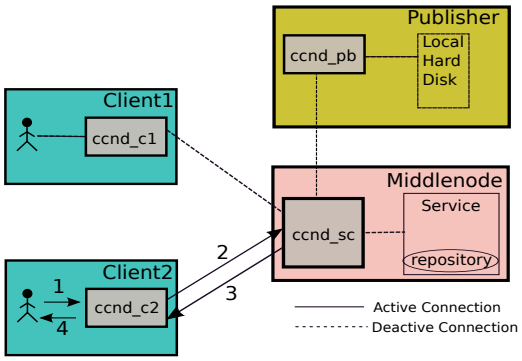

Figure 6.4: Scenario 2a

6.2.4 Scenario 2b

Assumption: The state is continuing from scenario 1. Client2 requests the same file, *PIC.JPEG* but the file is not anymore in the cache of *ccnd_sc*. The following steps will occur and are shown below. See Figure 6.5.

1. The client2 first sends an Interest message to its own local ccnd.

```
./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
```

2. The ccnd of client2 does not have this file and it forwards it to nearest ccnd which is the ccnd of the middlenode.

```
./ccngetfile ccnx:/ccnx.org/sc/JPEG/ccnx.org/files/PIC.BMP PIC.JPEG
```

3. The ccnd of the middlenode does not have the converted file then it sends the Interest Message to the conversion service.
4. The conversion service receives the Interest message. At first, the conversion service searches for the converted file which might already exist in its repository otherwise it will ask the publisher again. In this case, it has the converted version already, in the repository. It requests the converted file from its repository.
5. The conversion service will send it to the ccnd of the middlenode.
6. The ccnd of the middlenode forwards it to the ccnd of client2.
7. the ccnd of client2 gives the file to client2.

After the scenario 2a and 2b, nodes will have these files:

- ccnd_pb: PIC.BMP
- publisher: PIC.BMP
- ccnd_sc: PIC.JPEG and PIC.BMP
- conversion service repository: PIC.JPEG
- ccnd_c1: PIC.JPEG

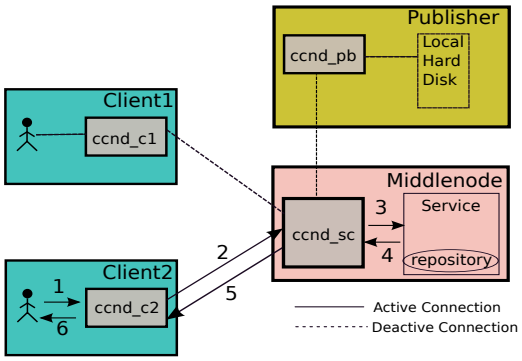

Figure 6.5: Scenario 2b

- client1: PIC.JPEG
- ccnd_c2: PIC.JPEG
- client2: PIC.JPEG

6.2.5 Scenario 3

Assumption: The state is continuing from scenario 2. In this scenario we consider that client2 asks for another format of *PIC.JPEG*, e.g: PNG. The process is shown below. See Figure 6.6.

1. Client2 first sends an Interest message to its own local *ccnd*.

```
./ccngetfile ccnx:/ccnx.org/sc/PNG/ccnx.org/files/PIC.BMP PIC.PNG
```

2. The ccnd of client2 does not have any of the files, neither *PIC.BMP* nor *PIC.PNG*. The ccnd of client2 forwards the Interest message to the nearest ccnd which is the ccnd of the middlenode.

```
./ccngetfile ccnx:/ccnx.org/sc/PNG/ccnx.org/files/PIC.BMP PIC.PNG
```

3. From the last scenario, we have the *PIC.JPEG* in the repository and the original file *PIC.BMP* is already cached in *ccnd_sc*. So the ccnd of the middlenode will forward the Interest message to the conversion service.
4. The conversion service receives the Interest message and will look for the requested files, *PIC.BMP* and *PIC.PNG* in its own repository. Since none of these files are there, the same procedure is undertaken as in scenario 1. The Interest message will be forwarded from the repository to the conversion service. The conversion service will send the request for the original file to the ccnd of the middlenode.
5. The ccnd of the middlenode has the original file from the first scenario already. Otherwise, it will fetch the original file from the publisher again, but the ccnd

will keep the file in its own cache for a while. The ccnd delivers *PIC.JPEG* to the conversion service.

6. Once the conversion service received the whole file, it starts to convert *PIC.BMP* to *PIC.PNG*. After the conversion, the conversion service starts to upload the file via the *ccnd* of the middlenode to its own repository. The conversion service makes another request to *ccnd* to put the file to the repository of conversion service.
7. During uploading to the repository, the *ccnd* has partially the file, *PIC.PNG* and starts to give the file to the *ccnd* of client2.
8. The ccnd of client2 will send the *PIC.PNG* to client2.

After this scenario, nodes will have these files:

- ccnd_pb: PIC.BMP
- publisher: PIC.BMP
- ccnd_sc: PIC.JPEG and PIC.BMP and PIC.PNG
- conversion service repository: PIC.JPEG and PIC.PNG
- ccnd_c1: PIC.JPEG
- client1: PIC.JPEG
- ccnd_c2: PIC.JPEG and PIC.PNG
- client2: PIC.JPEG and PIC.PNG

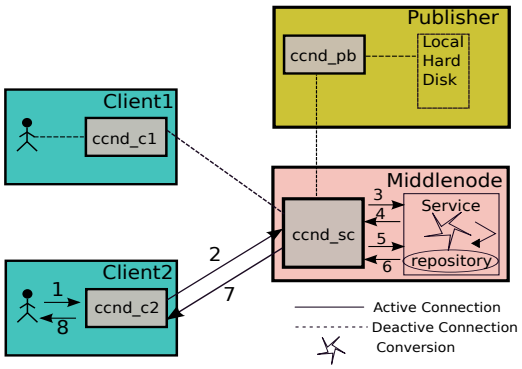

Figure 6.6: Scenario 3

Chapter 7

Experimental Evaluation

We attempt to measure the time required to cache the files with different scenarios in order to evaluate the performance of the selected approaches. The first section contains a setup for running CCNx and the conversion service on the network.

7.1 Setup

7.1.1 Topology

The topology consists of 4 nodes which are:

- Publisher
- Client1
- Client2
- Middlenode which has the conversion service

The conversion service is installed in the middlenode, but in reality it can be installed in any node or even on nodes in the network.

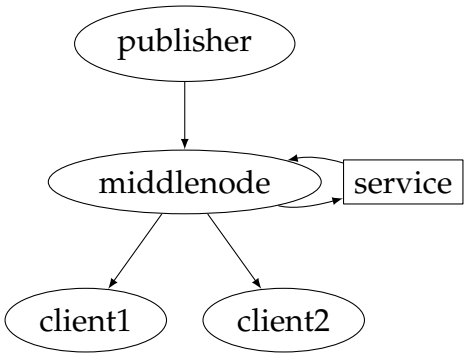

Figure 7.1: Topology

There are some steps required for the setup:

- **Step 1:**

  Starting ccnd on each node.

  ```
  cd  /ccn/bin/ ./ccndstart
  ```

- **Step 2:** Configuration of routing table on each node. Each machine has its own IP address.

  Publisher : 130.92.70.61
  Client1 : 130.92.70.62
  Client2 : 130.92.70.63
  Middlenode: 130.92.70.64

  The publisher is connected to the middlenode:

  ```
  ./ccndc add ccnx:/ccnx.org/ udp 130.92.70.64 9695
  ```

  The middlenode is connected to the client1, the client2, and the publisher:

  ```
  ./ccndc add ccnx:/ccnx.org/ udp 130.92.70.61 9695
  ./ccndc add ccnx:/ccnx.org/ udp 130.92.70.62 9695
  ./ccndc add ccnx:/ccnx.org/ udp 130.92.70.63 9695
  ```

  The client1, just connects to the middlenode:

  ```
  ./ccndc add ccnx:/ccnx.org/ udp 130.92.70.64 9695
  ```

  The client2, just connects to the middlenode:

  ```
  ./ccndc add ccnx:/ccnx.org/ udp 130.92.70.64 9695
  ```

- **Step 3:**

  The publisher is the node which provides the original file. Normally there is no control from the client over the publisher. The client can not ask the publisher to put the files in the repository or run the ccnfileproxy. As this topology is built by us, we decided that the publisher puts the file in its own repository. It would be possible to use ccnfileproxy as well. The command below starts the repository with the name of "ccnx_rep". Note that the directory does not need to exist already.

  ```
  bin/ccn_repo  /ccn_repo
  ```

This command means that the file, *PIC.JPEG* is available in the ccnd, under the name `ccnx:/ccnx.org/files/PIC.JPEG`. If a request comes with this name `ccnx:/ccnx.org/files/PIC.JPEG` then it will read the underlying local directory, which *PIC.JPEG* is stored.

```
./ccnputfile ccnx:/ccnx.org/files/PIC.JPEG PIC.JPEG
```

- **Step 4:**

  Starting the repository in the middlenode. Be aware that CCNx which is installed in the middlenode contains the conversion service and is a modified version of CCNx.

```
bin/ccn_repo  tmp ccn_repo
```

7.2 Different Experiments

This section shows the measured time in the different experiments. There are different cases with different scenarios. In the table 7.1 different scenarios are defined. These scenarios start from top, E1, which takes the shortest time, to E5, which takes longest. The invocation flow is indicated by the columns from the left to the right. It means that the converted file first will be looked up in the ccnd of the client and if it is existed, the converted file will be received from there otherwise it will be looked up in the ccnd of the conversion service. After that if the converted file is not existed in the ccnd of the conversion service, it will be looked up in the repository of the conversion service otherwise the original file and conversion are needed.

In the first scenario, E1, the converted file is received from the ccnd of client. In the second scenario, E2, the converted file is received from the ccnd of conversion service to the ccnd of client. In the third scenario, E3, the converted file is received from the repository of conversion service. In the fourth scenario, E4, the ccnd of conversion service has the original file and the conversion service is required. In the fifth scenario, the original file is only at the publisher and the conversion service is required. These scenarios are used in the test cases. The last three scenarios, E3, E4, E5 are shown below. See Figures 7.2, 7.3 and 7.4 respectively.

The abbreviations used in 7.1 are the following:

- sn: Scenario
- con: Converted file
- org: Original file
- NT: Network Transfer

- SC: Service
- Pub: Publisher
- Y: Yes
- N: No

No	ccnd_c		ccnd_sc		service		Publisher	Action
sn	org	con	org	con	converted	required	Invoked	
E1	-	Y	-	-	-	-	-	Local
E2	-	N	-	Y	-	-	-	NT: SC ->ccnd_c
E3	-	N	-	N	Y	-	-	NT: SC ->ccnd_c
E4	-	N	Y	N	N	Y	-	NT: SC ->ccnd_c+ Conversion
E5	-	N	N	N	N	Y	Y	NT: SC ->ccnd_c+ Conversion+ Pub ->SC

Table 7.1: Defining Different Scenarios

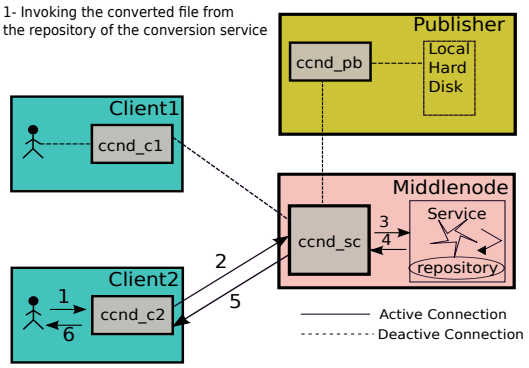

Figure 7.2: Scenario E3

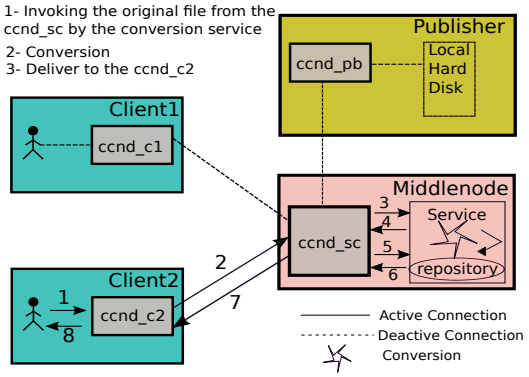

Figure 7.3: Scenario E4

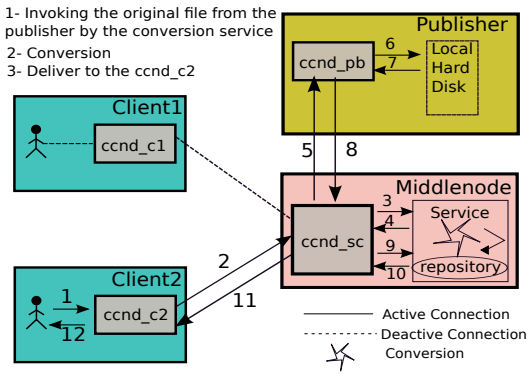

Figure 7.4: Scenario E5

In table 7.2 different test cases are listed. For example if the original file is large and the converted file is small or vice versa. There are three different sizes (small, medium, large) for the source file and also the converted file has different sizes (small, medium, large), There are 9 test cases in total. In this section just some of these test cases are selected as an experiment.

File	Original1	Original2	Original3
Convert1	L-L	M-L	S-L
Convert2	L-M	M-M	S-M
Convert3	L-S	M-S	S-S

Table 7.2: Defining Different Test Cases

The abbreviations used in 7.2 are the following:

- L: Large ->36 Mbytes
- M: Medium ->6 Mbytes
- S: Small ->536 kbytes

7.2.1 Test Case 1- A

This test case is based on conversion from large source file to a small convert file, see below:

Source: PIC.BMP, size: 36 Mbytes
Convert to: PIC.JPEG, size: 536 kbytes

- **Step 1:** Client1 requests PIC.BMP. See Figure 7.5.
  Time to download= 3min: 06s: 0ms

  The contents of the nodes after step 1:

- ccnd_pb: PIC.BMP
- publisher: PIC.BMP
- ccnd_sc: PIC.BMP
- ccnd_c1: PIC.JPEG
- client1: PIC.JPEG

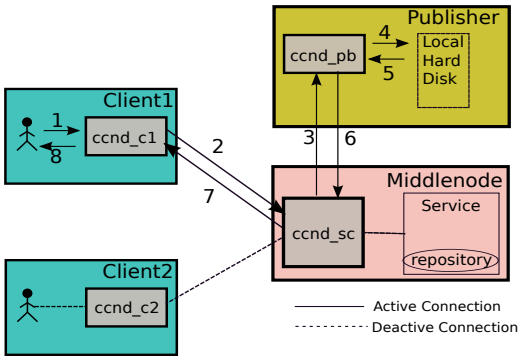

Figure 7.5: Test Case 1- A- Step 1

- **Step 2:**
  Client2 requests the same file as in step 1. See Figure 7.6.
  The contents of the nodes after step 2:
  - ccnd_pb: PIC.BMP
  - publisher: PIC.BMP
  - ccnd_sc: PIC.BMP
  - ccnd_c1: PIC.BMP
  - client1: PIC.BMP
  - ccnd_c2: PIC.BMP
  - client2: PIC.BMP

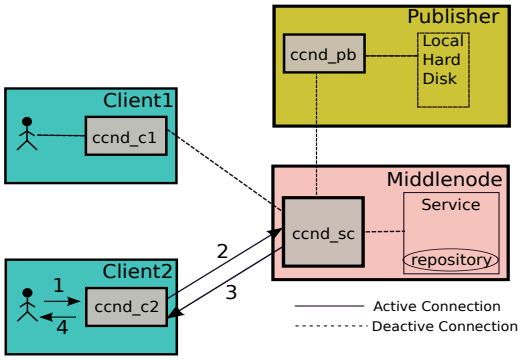

Figure 7.6: Test Case 1- A- Step 2

- **Step 3:**

  Client2 asks again for the second time the same file. See Figure 7.7.
  Time to download= 0min: 26s: 0ms

  The contents of the nodes after step 3 is the same as step 2.

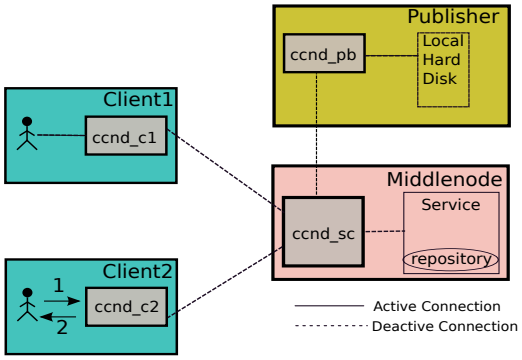

Figure 7.7: Test Case 1- A- Step 3

- **Step 4:**

  Now client2 requests the *JPG* format of *PIC.BMP*. The original file existed in the ccnd of the conversion service (scenario E4). See Figure 7.8.
  Time to download= 0min: 31s: 06ms

  The contents of the nodes after step 4:

  - ccnd_pb: PIC.BMP
  - publisher: PIC.BMP
  - ccnd_sc: PIC.BMP and PIC.JPEG
  - conversion service repository: PIC.JPEG
  - ccnd_c1: PIC.BMP
  - client1: PIC.BMP
  - ccnd_c2: PIC.BMP and PIC.JPEG
  - client2: PIC.BMP and PIC.JPEG

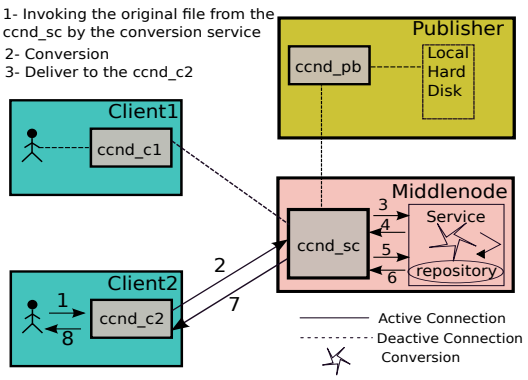

Figure 7.8: Test Case 1- A- Step 4

- **Step 5:**

  In this step we can see the advantage of CCNx. Client1 now requests the converted version of the file again. The requested file is *PIC.JPG* (scenario E2). See Figure 7.9.
  Time to download = 0min: 4s: 2ms
  The contents of the nodes after step 5:
  - ccnd_pb: PIC.BMP
  - publisher: PIC.BMP
  - ccnd_sc: PIC.BMP and PIC.JPEG
  - conversion service repository: PIC.JPEG
  - ccnd_c1: PIC.BMP and PIC.JPEG
  - client1: PIC.BMP and PIC.JPEG
  - ccnd_c2: PIC.BMP and PIC.JPEG
  - client2: PIC.BMP and PIC.JPEG

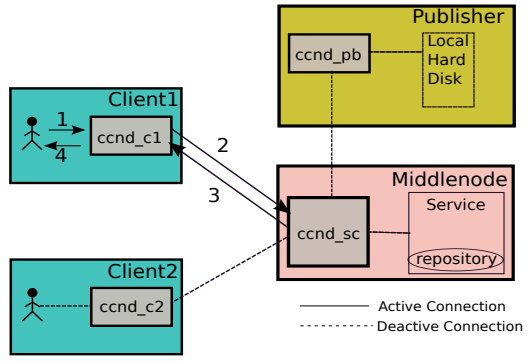

Figure 7.9: Test Case 1- A- Step 5

7.2.2 Test Case 1- B

- **Step 1:**

  First we clear *ccnd_sc*, *ccnd_c1* and *ccnd_c2*. So they do not have *PIC.BMP* in their cache anymore. Client1 requests from *ccnd_c1*, to provide *PIC.JPEG*. This time, the requested file, *PIC.JPG* is already stored in the repository of the conversion service. See Figure 7.10.
  Time to download= 0min: 04s: 73ms

  The contents of the nodes after step 1:

  - ccnd_pb: PIC.BMP
  - publisher: PIC.BMP
  - ccnd_sc: PIC.JPEG
  - conversion service repository: PIC.JPEG
  - ccnd_c1: PIC.JPEG
  - client1: PIC.JPEG

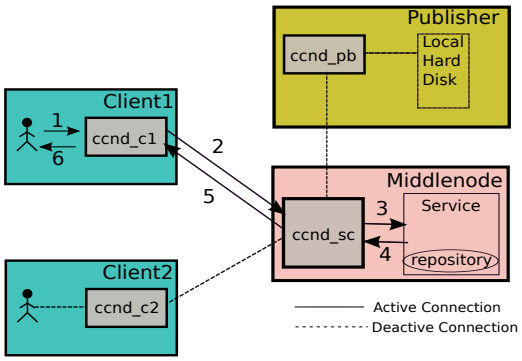

Figure 7.10: Test Case 1- B- Step 1

- **Step 2:**

  Now we want to know how long does it take to get the file from the local *ccnd* of client1. So client1 requests the same file, *PIC.JPEG*, again. We saw from step 1, that *ccnd_c1* has already cached this file. Therefore the retrieval time of the file, *PIC.JPEG* is shorter than the retrieval time of the file from the ccnd of another node (scenario E1). See Figure 7.11.
  Time to download= 0min: 02s: 71m

  The contents of the nodes after step 2 are the same as step 1.

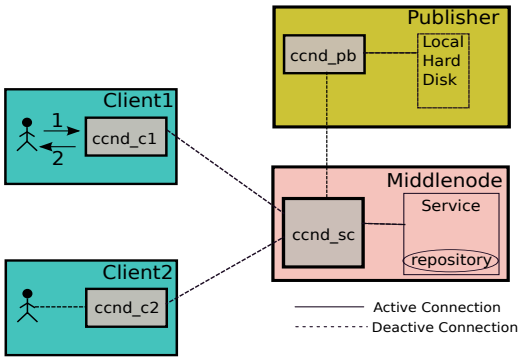

Figure 7.11: Test Case 1- B- Step 2

- **Step 3:**

  If client2 requests the same file, *PIC.JPEG*, this file will be cached in the local *ccnd* and the time has to be almost the same as in step 5 (scenario E2). See Figure 7.12. Time to download= 0min: 03s: 87ms

  The contents of the nodes are listed below.

  - ccnd_pb: PIC.BMP
  - publisher: PIC.BMP
  - ccnd_sc: PIC.JPEG
  - conversion service repository: PIC.JPEG
  - ccnd_c1: PIC.JPEG
  - client1: PIC.JPEG
  - ccnd_c2: PIC.JPEG
  - client2: PIC.JPEG

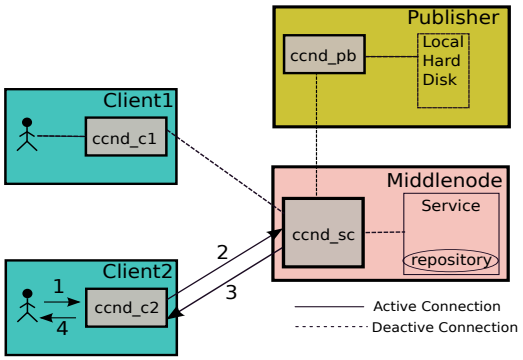

Figure 7.12: Test Case 1- B- Step 3

- **Step 4:**

  To prove again the correct time for retrieving the file from local *ccnd*, client2 requests the same file again. It requests the file, *PIC.JPEG* from its local *ccnd_c2*. As in step 2 the *ccnd* has cached this file already and downloading it is very fast.

The resulting time has to be almost the same as in step 1 (scenario E1). See Figure 7.13.
Time to download= 0min: 2s: 54ms

The contents of the nodes are the same as in the last step.

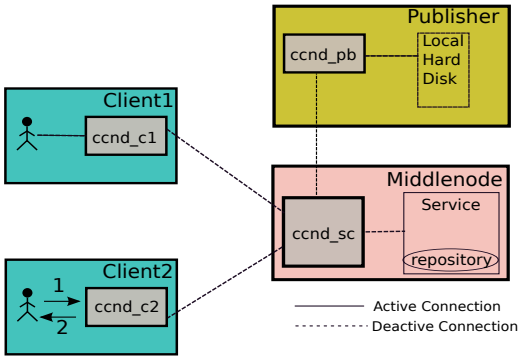

Figure 7.13: Test Case 1- B- Step 4

- **Step 5:**

  For another test we clear the repository of the conversion service, *ccnd_c1* and *ccnd_c2*. But we do not clear the ccnd of the conversion service. So we have *PIC.JPEG* in the *ccnd_sc* and from the last steps we have *PIC.JPEG* in *ccnd_sc*. The *ccnd_sc* does not have the original file, *PIC.BMP* anymore. Then, client1 asks for *PIC.JPEG*. In fact the ccnd has this file and it takes almost the same result as in step 5 in test case 1-A and in step 3 in test case 1-B. Client1 will get *PIC.JPEG* from the ccnd of the conversion service (scenario E2), see Figure 7.14. But something else happens. The Interest message is forwarded to the conversion service and starts to fetch the original file, *PIC.BMP* from the publisher, converts it and uploads it to its own repository. While client1 already got the file, the process of getting the file from the publisher, converting and uploading is continuing. The reason for this action is that the ccnd of the conversion service do check for the latest version of the file. Afterwards we have again the original file, *PIC.BMP* and the converted version, *PIC.JPEG* in the repository of the conversion service. See Figure 7.15.
  Time to download from ccnd= 0min: 04s: 0ms
  Time to download from publisher, conversion, uploading to repository= 02min: 24s: 0ms

  The contents of the nodes are listed below.

  - ccnd_pb: PIC.BMP
  - publisher: PIC.BMP
  - ccnd_sc: PIC.JPEG and PIC.BMP
  - conversion service repository: PIC.JPEG
  - ccnd_c1: PIC.JPEG
  - client1: PIC.JPEG

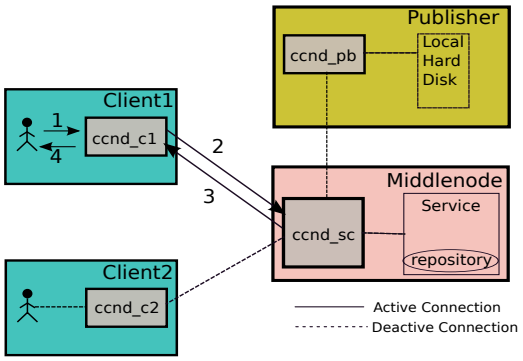

Figure 7.14: Test Case 1- B- Step 5-1

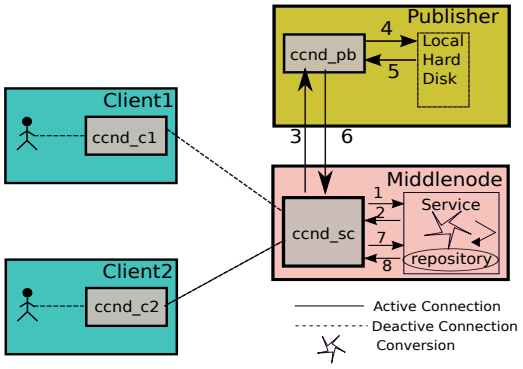

Figure 7.15: Test Case 1- B- Step 5-2

7.2.3 Test Case 2

Now in this test case, the medium source file is converted to the large file, see below:
Source: PIC.JPEG, size: 6 Mbytes
Convert to: PIC.BMP, size: 36 Mbytes

- The client requests *PIC.BMP* and this file is not cached anywhere.
  Time to download= 01min: 56s: 26ms
- The client requests again the same file.
  Time to download= 0min: 26s: 33ms
- Now *PIC.BMP* exists just in the repository of the conversion service and is not cached in the ccnd.
  Time to download= 02min: 14s: 26ms
- In this step *PIC.BMP* is also retrieved from the ccnd of the conversion service and another client requests the same file, *PIC.BMP*.
  Time to download= 01min: 53s: 81ms

For the first time when the file is cached, the converted file will be fetched from the ccnd of the conversion service, because during uploading the converted file to the repository of the conversion service, the ccnd of the conversion service will cached the

file and then the client can retrieves the converted file from the ccnd of the conversion service. That's why for the first time the retrieval time is shorter but when the file is fetched for the second time, it will be fetched from the repository of the conversion service. Because the converted file will remain in the repository for long time and the ccnd of the conversion service does not have in its cache anymore.

7.2.4 Test Case 3

Another test case is converting the small source file to the medium file.
Source: PIC.JPEG, size: 536kbytes
Convert to: PIC.BMP, size: 6Mbytes

- The client requests the *PIC.BMP* and this file is not cached anywhere.
  Time to download= 0min: 11s: 71ms
- The client requests again the same file.
  Time to download= 0min: 5s: 15ms
- Now the *PIC.BMP* just exists in the repository of the conversion service and it is not cached in the ccnd.
  Time to download= 0min: 12s: 19ms
- In this step *PIC.BMP* is also retrieved from the ccnd of the conversion service and another client requests the same file, *PIC.BMP*.
  Time to download= 0min: 10s: 38ms.

Chapter 8

Conclusions and Outlook

8.1 Conclusions

This thesis investigated Content Centric networking. This project emphasized on the strengths of CCN in terms of supporting services such as image transfer. In addition it was shown that data is not just being retrieved, but can be processed before being presented to the user. The Interest names are not just used for retrieving data, but also for invoking the service.

The project was tested with different scenarios and test cases to show the performance and advantages of CCN and SCN. The content distribution to different nodes helped to reduce and optimize traffic from the publishers. Clients retrieved the requested data and services much faster than before.

On the other hand, by taking advantage of CCN, we saw that the services also can be invoked with the same name structure as they are used in CCN.

Also, by deploying services to the different nodes, the traffic to the services can be reduced, regardless of the location of the services.

The implemented conversion service is able to retrieve the necessary data contents from the original source from the nearest node in the network.

One of the important advantages of SCN is, that different services can be implemented easily. For example, we could add different conversion mechanisms with different file formats by just calling the service with different parameters.

We saw that using Content Centric Networking could be a future mechanism n the Internet, that brings many benefits compared to the current network infrastructure.

8.2 Outlook

Much future work remains to be done. As already outlined earlier, the implementation presented in this master thesis is a proof of concept only.

In our approach, we had to increase the timeout on the client side for the first Interest. A future improvement would be to try to find another approach to handle the timeout

problem and avoid changing anything on the client side. With this improvement, all clients would be able to use the service without any modification of their timeout values.

Another improvement would be to expand the service to allow multiple clients to be processed at the same time. With the current design, only one client can be handled at a time. Because only the first Interest message is allowed to be forwarded and is being processed and the rest of Interstate messages will be hold unlit the process of the conversion is finished.

The current upload method to the repository of the conversion service via ccnd is ccnputfile. This method is not a clean approach and does not perform optimal as well. In future version the file should uploaded differently.

Appendix A

Manuals

For more details refer to the reference [7].

A.1 ccngetfile

NAME:

The ccngetfile retrieve a file published as CCNx content and save it to a local file. A command-line utility for pulling files out of ccnd or a repository.

SYNOPSIS:

```
ccngetfile [-unversioned] [-timeout millis] [-as pathToKeystore] [-ac] ccnxname
filename
```

DESCRIPTION:

The ccngetfile utility retrieves content published under the ccnxname and writes it to the local file filename. The content must be published as a collection of CCNx Data in accordance with the naming conventions for segmented streams or files, optionally unversioned. For the default case of versioned content, ccngetfile will retrieve the latest version available. The ccnxname must be specified using the CCNx URI encoding syntax. For simple cases of ASCII name components this is just pathname syntax with / delimiters.

A.2 ccnputfile

NAME:

ccnputfile - publish a file as CCNx content.

SYNOPSIS:

```
ccnputfile [-v] [-raw] [-unversioned] [-local] [-timeout millis] [-log LEVEL] [-as
pathToKeystore] [-ac] ccnxname filename|url
```

DESCRIPTION:

The ccnputfile utility publishes a local file filename or url as content with the ccnxname. The content is published as a collection of CCNx Data in accordance with the naming conventions for segmented streams or files, optionally unversioned. For the default case of versioned content, ccnputfile will publish content with the version based on the local machine time. The ccnxname must be specified using the CCNx URI encoding syntax. For simple cases of ASCII name components this is just pathname syntax with / delimiters. The filename must be the pathname of a local file that will published under the ccnxname. The url must be a valid url to be published under a ccnxname.

A.3 ccnfileproxy

DESCRIPTION:

The ccnfileproxy is a file system proxy that makes files on the local system available over the CCNx network.

For example, if you have a directory `/foo` in the file system, with the following contents: `/foo/bar.txt` and `/foo/baz/box.txt` and you call CCNFileProxy `/foo ccnx:/testprefix`. Then asking for `ccnx:/testprefix/bar.txt` would return the file *bar.txt* (segmented appropriately), and asking for `ccnx:/testprefix/baz/box.txt` would return *box.txt*. The version for each file is set using the last modified information available from the file system for the real file (but the file is re-signed every time you ask for it from this server, so will result in slightly different pieces of content with different signatures). The default prefix is `ccnx:/`, which means asking for `ccnx:/bar.txt` would get you *bar.txt*.

A.4 ccn_repo

NAME:

The ccn_repo is a utility to start, stop and signal Java application CCNx repositories.

SYNOPSIS:

```
ccn_repo [-memory memory] [-debug portno] [-output output_file] [-debug-daemon
portno] [-profile profile_info] [-suspend] [-noshare] [-debug-flags flags]
<repository_directory> [-log loglevel] [-prefix prefix] [-global <global_pre-
fix>] [-local <localname> ] | interactive <repository_directory> | stop <pi-
d> | stopall | signal <name> <pid>]
```

DESCRIPTION:

The ccn_repo utility starts, stops and signals CCNx Java content object repositories. This call requires a backend repository_directory to store the content objects to the file system. Only one repository should run on a single repository_directory at any time.

Appendix B

Source Code

B.1 Request a file

See Listing 6.

B.2 Repository

See Listing 7.

B.3 Putfile

See Listing 8.

```
1  package org.ccnx.ccn.utils;
2
3  import java.io.File;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6
7  import org.ccnx.ccn.CCNHandle;
8  import org.ccnx.ccn.config.ConfigurationException;
9  import org.ccnx.ccn.io.CCNFileInputStream;
10 import org.ccnx.ccn.io.CCNInputStream;
11 import org.ccnx.ccn.protocol.ContentName;
12 import org.ccnx.ccn.protocol.MalformedContentNameStringException;
13
14 public class ccngetfile implements Usage {
15   static Usage u = new ccngetfile();
16   public static void main(String[] args) {
17
18     for (int i = 0; i < args.length - 2; i++) {
19       if (!CommonArguments.parseArguments(args, i, u)) {
20         u.usage();
21         System.exit(1);
22       }
23       if (CommonParameters.startArg > (i + 1))
24         i = CommonParameters.startArg - 1;
25     }
26
27     if (args.length < CommonParameters.startArg + 2) {
28       u.usage();
29       System.exit(1);
30     }
31
32     try {
33       int readsize = 1024;
34       ContentName argName = ContentName.fromURI(args[CommonParameters.startArg]);
35
36       CCNHandle handle = CCNHandle.open();
37
38       File theFile = new File(args[CommonParameters.startArg + 1]);
39       if (theFile.exists()) {
40         System.out.println("Overwriting␣file:␣" + args[CommonParameters.startArg + 1]);
41       }
42       FileOutputStream output = new FileOutputStream(theFile);
43
44       long starttime = System.currentTimeMillis();
45       CCNInputStream input;
46       if (CommonParameters.unversioned)
47         input = new CCNInputStream(argName, handle);
48       else
49         input = new CCNFileInputStream(argName, handle);
50
51           ((CCNFileInputStream)input).allowFirstSlower = true;
52
53       input.setTimeout(10000);
54
55       byte [] buffer = new byte[readsize];
56
57       int readcount = 0;
58       long readtotal = 0;
59       //while (!input.eof()) {
60       while ((readcount = input.read(buffer)) != -1){
61         //readcount = input.read(buffer);
62         readtotal += readcount;
63         output.write(buffer, 0, readcount);
64         output.flush();
65       }
66       if (CommonParameters.verbose)
67         System.out.println("ccngetfile␣took:␣"+(System.currentTimeMillis() - starttime)+"ms");
68       System.out.println("Retrieved␣content␣" + args[CommonParameters.startArg + 1] + "␣got␣" + readtotal + "␣bytes.");
69       System.exit(0);
70
71     } catch (ConfigurationException e) {
72       System.out.println("Configuration␣exception␣in␣ccngetfile:␣" + e.getMessage());
73       e.printStackTrace();
74     } catch (MalformedContentNameStringException e) {
75       System.out.println("Malformed␣name:␣" + args[CommonParameters.startArg] + "␣" + e.getMessage());
76       e.printStackTrace();
77     } catch (IOException e) {
78       System.out.println("Cannot␣write␣file␣or␣read␣content.␣" + e.getMessage());
79       e.printStackTrace();
80     }
81     System.exit(1);
82   }
83
84   public void usage() {
85     System.out.println("usage:␣ccngetfile␣[-unversioned]␣[-timeout␣millis]␣[-as␣pathToKeystore]␣[-ac␣(access␣control)]
86 ␣<ccnname>␣<filename>");
87   }
88
89 }
```

Listing 6: ccngetfile.java

```
public class LogStructRepoStore extends RepositoryStoreBase implements RepositoryStore, ContentTree.ContentGetter {


  public boolean conversion = false;
  public Object lock = new Object();
  public ContentObject getContent(Interest interest)
      throws RepositoryException {

    final String name = interest.name().toString();
    System.out.println ("###getContent(): " + name);
    String base = name;
    if (name.indexOf("/%") > 0)
      base = name.substring(0, name.indexOf("/%"));

    try {
    final Object isCached = _index.get(Interest.constructInterest(ContentName.fromURI(base), null, null, null, null, null),
                this);
    if (!name.startsWith("/ccnx.org/sc/") || (name.contains("/%C1.R.sw/")) || (isCached != null)) {
      // default handling
      System.out.println ("###getContent(): default handling for: " + name);
      ContentObject co =  _index.get(interest, this);
      if( Log.isLoggable(Log.FAC_REPO, Level.FINE) )
        Log.fine(Log.FAC_REPO, "Looking for: " + interest.name() + (co == null ? ": Didn't find it" : ": Found it"));
      return co;

    } else {

      if (conversion) {
        System.out.println ("DROPPING REQUESTS");
        return null;
      }

      try {
        synchronized (lock) {


          // special handling (conversion service)
          System.out.println ("getContent(): CONVERSION SERVICE!!! for: " + name);

          // return already converted files
          try {
            if (_index.get(Interest.constructInterest(ContentName.fromURI(base), null, null, null, null, null), this)
                                                    == null) {
              System.out.println ("###getContent(): not yet converted (!!! CONVERSION STARTING !!!): " +name);
              // get original file
              final String format = name.substring(13, name.indexOf("/", 16));
              final String origName = name.substring(name.indexOf("/", 16)+1);
              // create temporary file
              final File tmp = File.createTempFile("wefwef", "wefwef");
              final File tmpConverted = File.createTempFile("wefwef", "wefwef");
              final OutputStream tmpStream = new FileOutputStream(tmp);
              // first retrieve file from source
              System.out.println ("STARTING GETFILE " + new Date());
              ContentName argName = ContentName.fromURI("ccnx://"+origName);
              CCNHandle handle = CCNHandle.open();
              InputStream stream = new CCNFileInputStream(argName, handle);
              byte[] buffer = new byte[2048];
              int len;
              while ((len = stream.read(buffer)) >= 0)
                tmpStream.write(buffer, 0, len);
              stream.close();
              tmpStream.flush();
              tmpStream.close();
              System.out.println ("FINISHED GETFILE " + new Date());
              // convert file
              System.out.println ("FINISHED CONVERT " + new Date());
              BufferedImage image = ImageIO.read(new FileInputStream(tmp));
              FileOutputStream sso = new FileOutputStream(tmpConverted);
              ImageIO.write(image, format, sso);
              sso.close();
              System.out.println ("FINISHED CONVERT " + new Date());
              // put file back to repository
              System.out.println ("STARTING UPLOAD " + new Date());
              ccnputfile put = new ccnputfile();
              put.write(new String[] {interest.name().toString(), tmpConverted.getAbsolutePath().toString()});
              System.out.println ("FINISHED UPLOAD " + new Date());
              // delete files
              tmp.delete();
              tmpConverted.delete();

            }
          } catch (Exception e) {
            throw new RuntimeException(e);
          }

          ContentObject result = _index.get(interest, this);
          System.out.println ("### getContent(): RESULT: " + (result != null));
          return result;
        }
      } finally {
        conversion = false;
      }

    }
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
```


Listing 7: LogStructRepoStore.java

```
1  package org.ccnx.ccn.utils;
2
3  import java.io.IOException;
4  import java.security.InvalidKeyException;
5  import java.util.logging.Level;
6
7  import org.ccnx.ccn.CCNHandle;
8  import org.ccnx.ccn.config.ConfigurationException;
9  import org.ccnx.ccn.impl.support.Log;
10 import org.ccnx.ccn.protocol.ContentName;
11 import org.ccnx.ccn.protocol.MalformedContentNameStringException;
12 public class ccnputfile extends CommonOutput implements Usage {
13   static ccnputfile ccnputfile = new ccnputfile();
14   public void write(String[] args) {
15    Log.setDefaultLevel(Level.WARNING);
16
17    for (int i = 0; i < args.length - 2; i++) {
18      if (args[i].equals("-local")) {
19        CommonParameters.local = true;
20      } else if (args[i].equals(("-raw"))) {
21        CommonParameters.rawMode = true;
22      } else {
23        if (!CommonArguments.parseArguments(args, i, ccnputfile)) {
24          usage();
25        }
26        if (CommonParameters.startArg > i + 1)
27          i = CommonParameters.startArg - 1;
28      }
29      if (CommonParameters.startArg <= i)
30        CommonParameters.startArg = i + 1;
31    }
32
33    if (args.length < CommonParameters.startArg + 2) {
34      usage();
35    }
36
37    long starttime = System.currentTimeMillis();
38    try {
39      ContentName argName = ContentName.fromURI(args[CommonParameters.startArg]);
40
41      CCNHandle handle = CCNHandle.open();
42
43      if (args.length == (CommonParameters.startArg + 2)) {
44        if (CommonParameters.verbose)
45          Log.info("ccnputfile: putting file " + args[CommonParameters.startArg + 1]);
46
47        doPut(handle, args[CommonParameters.startArg + 1], argName);
48        System.out.println("Inserted file " + args[CommonParameters.startArg + 1] + ".");
49        if (CommonParameters.verbose)
50          System.out.println("ccnputfile took: "+(System.currentTimeMillis() - starttime)+" ms");
51
52        return;
53      } else {
54        for (int i=CommonParameters.startArg + 1; i < args.length; ++i) {
55          ContentName nodeName = ContentName.fromURI(argName, args[i]);
56          doPut(handle, args[i], nodeName);
57          System.out.println("Inserted file " + args[i] + ".");
58        }
59        if (CommonParameters.verbose)
60          System.out.println("ccnputfile took: "+(System.currentTimeMillis() - starttime)+" ms");
61
62        return;
63      }
64    } catch (ConfigurationException e) {
65      System.out.println("Configuration exception in put: " + e.getMessage());
66      e.printStackTrace();
67    } catch (MalformedContentNameStringException e) {
68      System.out.println("Malformed name: " + args[CommonParameters.startArg] + " " + e.getMessage());
69      e.printStackTrace();
70    } catch (IOException e) {
71      System.out.println("Cannot put file. " + e.getMessage());
72      e.printStackTrace();
73    } catch (InvalidKeyException e) {
74      System.out.println("Cannot publish invalid key: " + e.getMessage());
75      e.printStackTrace();
76    }
77
78    System.exit(1);
79  }
80  public void usage() {
81    System.exit(1);
82  }
83
84  public static void main(String[] args) {
85    ccnputfile.write(args);
86  }
87 }
```

Listing 8: ccnputfile.java

Bibliography

[1] http://en.wikipedia.org/wiki/Content-centric_networking.

[2] http://www.ccnx.org/releases/latest/doc/technical/CCNxProtocol.html.

[3] http://named-data.org/.

[4] http://www.webopedia.com/TERM/R/repository.html.

[5] http://pcsupport.about.com/od/termsf/g/filesystem.htm.

[6] http://www.javamex.com/tutorials/synchronization_concurrency_synchronized1.shtml.

[7] http://www.ccnx.org/releases/latest/doc/manpages.

[8] S. Arianfar. A transport protocol for content-centric networks. Technical report, Aalto University, 2010. http://eggert.org/papers/2010-icnp-con-trans.pdf.

[9] V. Jacobson. Networking named content. Technical report, Palo Alto Research Center, 2009. http://pages.cs.wisc.edu/~akella/CS838/F09/838-Papers/ccn.pdf.

[10] V. Jacobson. Content centric networking. Technical report, Palo Alto Research Center, 2010. https://wiki.tools.isoc.org/@api/deki/files/2634//=1.vj.isoc.mar10.pdf.

[11] M. Hofmann I. Rimac M. Steine M. Varvello T.Braun, V. Hilt. Service-Centric Networking. Technical report, ICC'11 Workshop on FutureNet IV, Kyoto, Japan, 2011. http://www.planethofmann.com/markus/publications/.