SyncHome - Synchronization Application for Mobile Content Retrieval

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Nina Mujkanovic
2015

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Contents

List of Figures

List of Tables

Listings

Acknowledgment

A huge thank you to all the people who have supported me on the path to the completion of this thesis. First and foremost, I am grateful to Professor Braun for granting me the chance to write the thesis within, as well as use the infrastructure of, the Communications and Distributed Systems Group. Next, I thank all my friends for their continuous advice concerning university administration. Finally, I thank my family, who have always been there for me, in spite of their frustrations concerning my answer to the question ”are you done yet?”.

Most of all, my thanks go out to Carlos Anastasiades, the best thesis advisor one could imagine. Without his encouragement, patience, sharing of knowledge and willingness to put up with ridiculous questions and perpetual sick leaves, this work would not have been possible.

Abstract

Content Centric Networking (CCN) is a new approach to routing content, meant to correct the pitfalls found in host based protocols. At its core is the idea that content should not be bound by host location, but rather by the name of the content itself. CCN is implemented in the open source CCNx, which allows for content retrieval as well as communication following content-centric principles.

This thesis centered on the CCNx framework. The main goal was to develop a synchronization protocol for wireless and mobile clients, as the current synchronization protocol is geared towards wired clients. The new protocol makes it possible to synchronize content between hosts that are not directly linked and introduces a functionality to stop the synchronization upon transfer completion in order to preserve energy on mobile devices. The bulk of these implementations is contained in the new `synchomeclient` and `synchomeserver` applications. Minor modifications were made to preexisting CCNx code.

In a second part, the feasability of the implementation was evaluated in terms of runtime and number of messages exchanged between client and server. The evaluation was performed on the University of Berne Linux Cluster (UBELIX) using ns3 Direct Code Execution. The `synchome` implementation was compared to the static synchronization protocol `ccnsyncslice`. We found that, while the difference in synchronization time between the applications is very small, `ccnsyncslice` generates up to 437% more messages when synchronizing intermittently compared to `synchome`.

Chapter 1

Introduction

1.1 Motivation

The Internet, as we know it today, is a gigantic mesh of interconnected networks. It functions based on the paradigm that everything is a point-to-point conversation. Similar to telephone conversations, establishing a conversation requires knowing the host locations. This, though, does not apply to todays standards of use, where, instead of communication between shared resources, content dissemination is the focal point of interest. Decreasing storage costs and ever increasing transfer speeds have brought a paradigm change, one in which ”content is king” and should be accessible without knowing its location.

In accordance with this, it is questionable whether host-based routing solutions still represent the best solution, as they have several problems in regard to content exchange, which include:

- point-to-point conversation being inefficient for content dissemination
- limited mobility; client location tied to an IP address
- lack of multicast support
- no security of data

The project we focus on attempts to avoid these pitfalls. CCNx, the open source Content-centric Network solution implemented by PARC, does not use IP addresses, instead making content directly addressable and routable. Hosts within the network communicate based on named data, which is retrieved via the exchange of content request messages (Interests) and content return messages (Data). The named data is location independent and routing decisions are made by the involved network elements. Additionally, instead of securing the communication pipe between the end hosts, the data itself is signed and secured.

At its core, CCN works as follows: a consumer issues a request for content by sending an Interest message containing the name of the content. The network routes the Interest based on the name using longest prefix matching. As it traverses the network, the Interest leaves states behind. Once a match is found, content is sent back as a Data message using the left over states.

1.2 Task Formulation

CCNx routes packets based on name prefixes. Content is maintained in so called repositories, which use the local file system for persistent storage of CCN Content Objects. In order to make new content available, it has to be added to the hosts' local repository. Various methods for content retrieval exist, such as issuing a specific request for desired content and retrieving it from the remote hosts repository, as well as methods for content synchronization, which constantly exchange Interest messages to compare the content available in the requesting and responding hosts' repositories, and retrieve content based on changes in one of the repositories.

The main problem is that these methods were designed with wired networks in mind, where hosts are constantly connected. In a wireless and mobile environment, where hosts may sporadically lose network connectivity, continuing to issue Interest messages is inefficient.

Thus, the goals of this thesis are:

- Study the current CCNx repository and synchronization protocols
- Implement a Synchronization Protocol for mobile hosts that initiates synchronization only when the host is connected to a network, and which consists of:
  - a client application, which connects to the home repository, starts and stops synchronization autonomously
  - a server application, which runs a home repository - a central repository containing content synchronized to it by participating hosts - and listens for requests to initialize synchronization
- Evaluate the implementation

1.3 Outline

The remaining report has the following structure: Chapter 2 describes the basics of Content-centric Networking. Chapter 3 gives the design and implementation details of our synchronization protocol for mobile hosts. Evaluation methods and results are discussed in chapter 4. In chapter 5 this work is concluded and future work is discussed.

Chapter 2

Related Work

2.1 Content-Centric Networking

The concepts of Content-Centric Networking were introduced in a paper by Van Jacobson[1]. The open source implementation of the concepts, entitled CCNx[2], was developed at Jacobson's research group at the Palo Alto Research Center (PARC). In the following, we describe the CCN concepts, as well as the CCNx synchronization protocol.

2.1.1 Packets

All communication in CCN is based upon the use of two packet types, Interest packets and Data or Content-Object packets. The exchange of packets between nodes is balanced; for each Interest packet, exactly one Data packet is returned. If an Interest packet is not answered by a Data packet, it may be retransmitted or removed from the Pending Interest Table, which will be explained in the Section 2.1.4 on Content Forwarding.

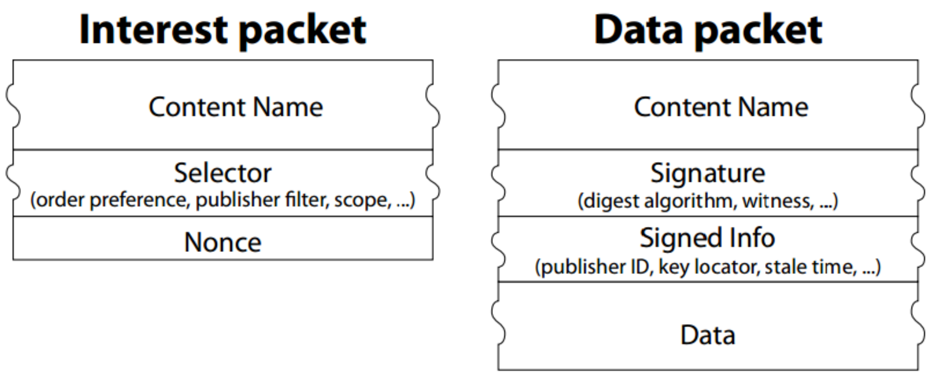
Interest packet
Content Name
Selector
(order preference, publisher filter, scope, ...)
Nonce
Data packet
Content Name
Signature
(digest algorithm, witness, ...)
Signed Info
(publisher ID, key locator, stale time, ...)
Data

Figure 2.1: CCN packet types[1]

As seen in Figure 2.1, Interest packets consist of the content name, which can either be a prefix or the full name of the Data object, as well as selectors concerning origin and matching of the Data packet. Data packets contain, next to the content name and data itself, a signature and signed info.

2.1.2 Naming

Content names in CCN are hierarchically structured and composed of multiple components, which may be encrypted for privacy. They are, like URIs, presented with / characters seperating the components.

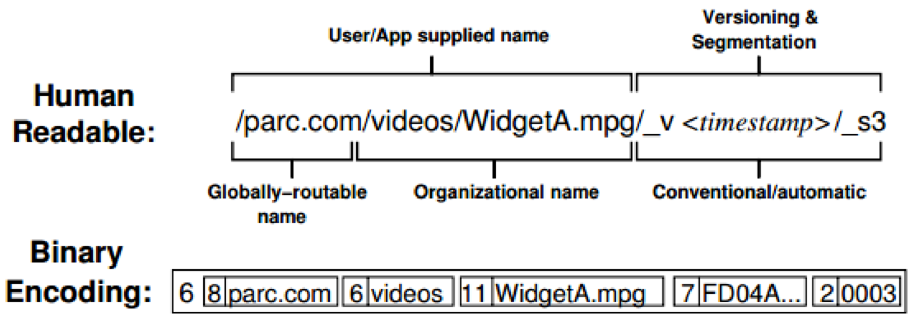
User/App supplied name
Versioning & Segmentation
Human Readable:
/parc.com/videos/WidgetA.mpg/_v <timestamp>/_s3
Globally-routable name
Organizational name
Conventional/automatic
Binary Encoding:
6 8 parc.com 6 videos 11 WidgetA.mpg 7 FD04A... 2 0003

Figure 2.2: CCN naming scheme

An example of the naming scheme is presented in Figure 2.2. As can be seen, the user supplied name is additionally extended by a version, as well as a segmentation number, as content may be split into several segments. The version number, indexed by the $\_v$ marker, is determined by a timestamp, after which another / and the segmentation number follow. The segmentation number is, likewise, indexed by the $\_s$ marker, after which the zero-based number of the segment is indicated.

2.1.3 Faces

Within CCN, a face is a kind of interface which allows for packets to be forwarded over hardware network interfaces as well as exchanged with application processes within a machine. Per default, hosts in a content-centric network forward Interest messages to their neighboring hosts only, which in turn forward them to their neighbors. Adding a distant remote host IP address as a face to a host makes it possible for the host to forward Interests to the remote host directly.

2.1.4 Content Forwarding

Content exchange is consumer driven: a node must transmit an Interest in order to retrieve data. The Interest then passes three main data structures:

FIB - Forwarding Information Base
The FIB is used to forward Interest packets to potential sources of matching Data. It

allows for a list of outgoing faces, which can be used to query multiple sources for data in parallel, as an Interest can be sent out through all faces.

PIT - Pending Interest Table
The PIT keeps track of Interest messages forwarded upstream towards content sources. Returned Data packets can be sent downstream to their requesters by backtracking the PIT entries. This process is comparable to leaving breadcrumbs: as Interest messages are propagated towards potential Data sources, they leave a trail of 'breadcrumbs' on every host as a PIT entry. Once a Data packet is found, it follows the 'breadcrumbs' back to the requester. On every host, the PIT entry for a given Interest is erased or consumed once a matching Data packet passes it. PIT entries for Interests that are not answered eventually time out.

CS - Content Store
The Content Store is the same as the buffer memory of an IP router, but with a different replacement policy. CCN packets are self-identifying and self-authenticating, thus each packet may be useful to many customers. This fact can be used to minimize upstream bandwidth demand and downstream latency by remembering arriving Data packets as long as possible, i.e. caching them.

Figure 2.3 depicts the schematics of content forwarding. The different data structures are accessed as follows: let us assume that the host issues an Interest packet named $/parc.com/videos/a\_video.mpg$. The Interest is first matched against the hosts' Content Store. In this case, the CS contains only the Data packet $/parc.com/videos/WidgetA.mpg/v3/s0$, which does not match the Interest packet. Next, the Interest packet is matched against the PIT entries. As the PIT does not contain a match, a new entry is added. If the PIT does contain the Interest, the new Interest and the PIT entry are consolidated, as the host is already waiting for the given Content. In a last step, the Interest prefix, in this case $/parc.com$, is matched against the FIB to find a face to forward it along. In this case, the face is found, but should it not exist, the Interest is discarded, as no routes along which the Interest can be satisfied are found.

Incoming Interest packets follow the reverse path. The Interest prefix is first matched against the host's FIB. If it is found, the Interest is matched against the PIT. If the PIT contains a matching entry, the CS is queried, otherwise, a PIT entry is added, and the Interest packet propagated to the next neighbor.

It is important to note that, since every action in CCN is initiated by issuing an Interest message, Interests can be routed to a remote host, or to the local host itself. A request to add content to the host repository is, for example, started by sending a `Start Write` command of the form `ccnx:(/<component>)*/%C1.R.sw/<nonce>` to the local CCN application[3]. Here `component` represents the name prefix of the content to be stored, and the `nonce` is a random binary value which ensures that the content name is unique. The repository, which will be defined in the next section, responds with a Data packet that acknowledges that the `Start Write` request was accepted and starts fetching and storing the content.

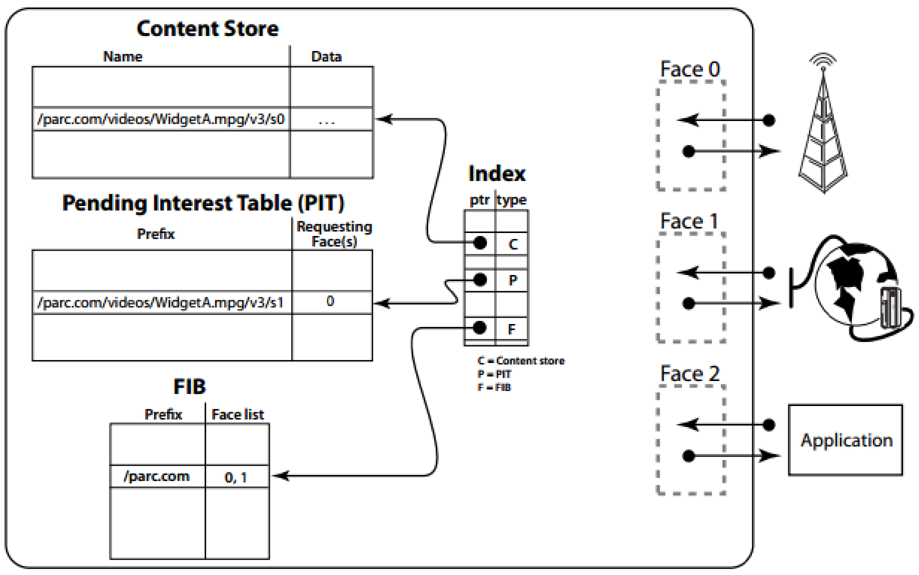

Figure 2.3: Forwarding engine model

2.1.5 Data Storage

Two types of content storage are employed in CCNx. They consist of the persistent CCNx repository, as well as the previously mentioned Content Store, which caches content.

The repository uses the local file system to persistently store content. Content to be shared must first be inserted in the repository. The insertion is initiated by issuing an Interest to the local CCN application. During the insertion process, the content is segmented, signed and encoded.

2.2 CCNx Project

The concepts in this section, such as the CCN daemon and synchronization protocol, are CCNx specific.

2.2.1 CCN Daemon

Within the CCNx implementation, all functionality is brought together and managed by the ccn daemon, `ccnd`. `ccnd` is the software forwarder/router and is required for normal protocol communication. Typically, one `ccnd` is run on each host. Applications running on a given host will communicate through the local `ccnd`, which in turn communicates over the attached networks.

2.2.2 Synchronization protocol

Synchronization is a CCNx funcionality that automatically exchanges content added to one hosts' repository accross all participating hosts repositories. In order to synchronize content, a slice, or subset of content sharing the same prefix, must be defined by participating hosts. As long as a host defines a given slice, content added to another hosts' repository, that the given host does not have, will be synchronized to it. The removal of content is not supported in CCNx and thus not synchronized. Synchronization is stopped by deleting or undefining the slice.

Each repository owns a synchronization agent that is responsible for keeping local collections up to date by detecting and acquiring new content objects that were added to remote hosts and are not contained in the local repository, as well as for responding to remote synchronization agents' inquiries about information concerning content in the local repository.

Collections are synchronized independently of each other. When a new collection is detected, the synchronization agent builds a synchronization tree for the collection representing its content and organization. As new content is added to a collection, the local synchronization agent updates the relevant synchronization tree.

An example of a synchronization tree is depicted in Figure 2.4. To synchronize all videos, for example, a user would declare the childnode `videos` as a slice. Once the slice is defined, synchronization starts. Every node in the synchronization tree stores a hash calculated over all its subnodes. The root nodes hash is called the roothash. When looking for new content, the synchronization agents of the hosts declaring the same slice first exchange their roothashes. If the hashes match, the synchronization trees are stable and there is nothing to synchronize. If they do not match, the childnodes' hashes are exchanged, until the childnode that contains differing content is found.

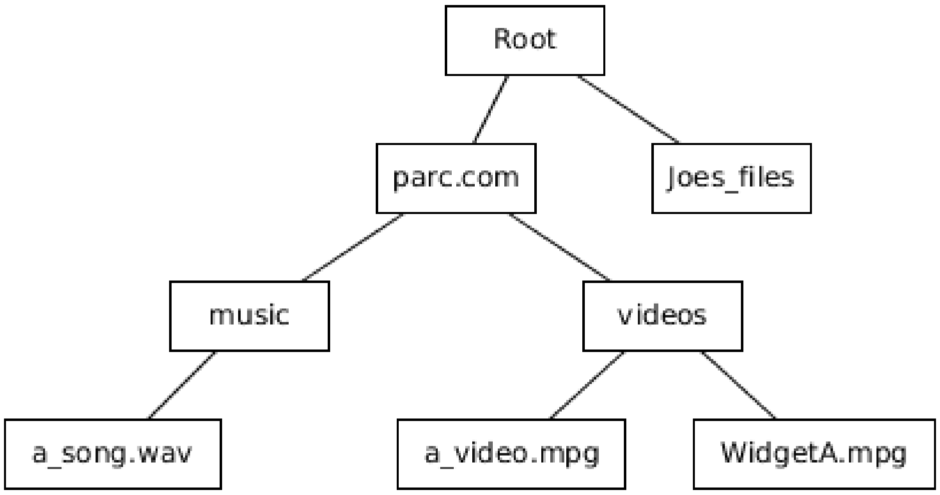

Figure 2.4: Example synchronization tree

For our exampe synchronization tree, the synchronization agents first exchange the roothash. In this case, the roothash is the hash of the `videos` childnode. If the roothashes do not match, the hashes of the childnodes are exchanged, in this case of `a_video.mpg` and `WidgetA.mpg`. If the hashes of `a_video.mpg` match, the content is not synchronized. If the hashes of `WidgetA.mpg` differ, the content is synchronized.

The exchange of hashes is handled by two types of Interest messages:

Root Advise Interest
The Root Advise Interest is used to determine if a remote collection contains names not available in the local collection. The synchronization agent expresses a Root Advise Interest for each local synchronization tree. The Interest contains the local synchronization trees root node hash. If a repository with a corresponding collection has the same roothash, no response is necessary. If the hashes differ, the synchronization agent must fetch the nodes of the remote synchronization tree that have differing hashes.

Node Fetch Interest
Node Fetch Interests are used to obtain a list of nodes of the remote synchronization tree that have hashes that differ from the local trees hashes. Node Fetch Interests can retrieve hashes from any part of the synchronization tree, while Root Advise Interests refer only to the node at the root of the tree.

Figure 2.5 illustrates an example of how the synchronization protocol operates[4]. First, a collection has to be defined, creating a slice. Once the slice has been established, the host's synchronization agent periodically sends out Root Advise Interests to ascertain that no new content is available in identical remote slices. As long as no other host has the same slice defined, no replies are returned. If a remote host has defined the same collection, and its Root Advice Interest reaches the first host's synchronization agent, their synchronization trees are compared and updated using Node Fetch Interests. After the synchronization trees have been updated, content is exchanged until the synchronization trees are stable again. After this, the transmission of Root Advice Interests begins again.

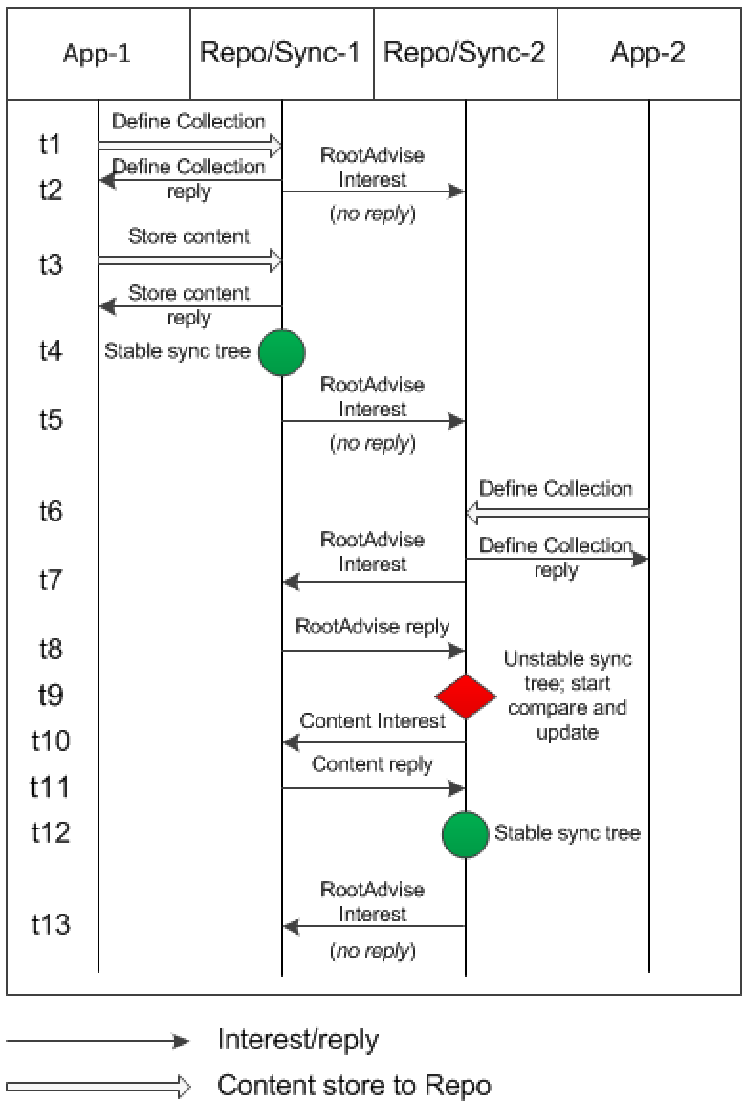

Figure 2.5: Illustration of the Synchronization Protocol[4]

The CCNx command used to create and delete slices is `ccnsyncslice`. The new mobile synchronization protocol implemented in this work is based on this command.

2.2.3 `ccnsyncwatch`

The `ccnsyncwatch` command is used for monitoring the operation of the synchronization agent of a repository. In this sense, it monitors the exchange of Root Advice and Node Fetch Interests between synchronization agents. Initially, we intended to use it to monitor whether a synchronization process was still active or had been stopped. This proved to be infeasible, as the exchange of Root Advice and Node Fetch Interests precedes the actual content transmission. Instead, we implemented a new function to track the synchronization status. The function, `synchomewatch`, is presented in the next chapter.

Chapter 3

Design and Implementation of a Synchronization Protocol for mobile hosts

3.1 Problem Description

The synchronization protocol's `ccnsyncslice` command allows for keeping up to date of collections defined accross multiple repositories. It faces a number of limitations, of which the following are important to us:

1. Synchronization only works with direct neighbors - nodes must be reachable in one hop. While this can be remedied by the user establishing a VPN tunnel, a solution integrated in CCNx would be prefereable.
2. The order in which names and content are synchronized is arbitrary - it is impossible to tell whether all segments have been synchronized or not.
3. Synchronization runs on infinitely - inefficient for mobile clients with network connectivity issues.

We propose a mobile synchronization protocol that allows for the synchronization of collections located on hosts without direct link. The protocol monitors the synchronization process and exits once the collections have been fully synchronized.

3.2 Design of the Synchronization Protocol for mobile hosts

The new mobile synchronization protocol is based on the CCNx synchronization protocol described in Section 2.2.2. As `ccnsyncslice` enables the synchronization of collections, we extended it with functionalities for mobile clients. The functionalities are divided between a client and a server application and consist of the following:

- Register the server's IP address as a face with the client
- Send the client's IP address to the server to be registered as a face

- Send all information necessary for synchronization to the server
- Start synchronization on both client and server
- Stop synchronization if no more content is to be exchanged in Data packets between client and server

To be able to detect whether the synchronization is still running between client and server, a third application to watch the message exchange between the hosts was implemented. The application exits when no more content is being exchanged in Data packets between client and server. The client is then responsible for discovering whether the synchronization has stopped because it finished, or if it was disrupted due to network connectivity problems. To do this, the client sends an Interest to the server. If the server answers the Interest, the synchronization has finished succesfully. If there is no answer, it is assumed that there is no link between client and server. In this case, the client times out before reattempting to synchronize.

3.2.1 Interest Exchange

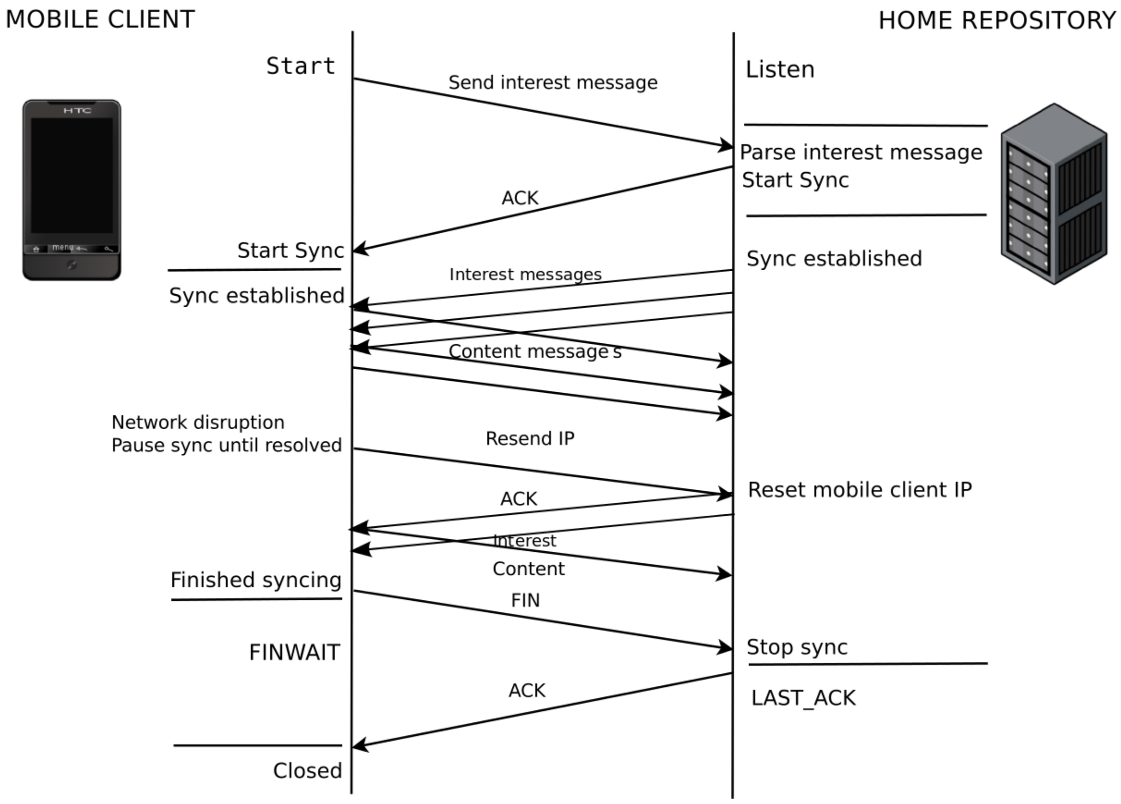

Figure 3.1: Interest exchange between client and server

The mobile synchronization protocol Interest exchange is depicted in Figure 3.1. The client application starts the synchronization between client and server by sending the server an Initial

Synchronization Interest, which contains the clients IP address and synchronization request (see Section 3.2.2 on Initial Synchronization Interest composition). The Initial Synchronization Interest can be sent even if client and server are not directly linked, as the client knows the server's IP address and can route its messages according to the FIB entry.

The server parses the Interest, defines the collections to synchronize and sends acknowledgements to the client. After receiving the acknowledgment, the client defines the collection and the synchronization protocol takes over by exchanging Root Advise Interests, Node Fetch Interests and finally the content. If the synchronization stops, the client ascertains whether the synchronization has been completed by sending an Interest message to the server. If the Interest is not answered, and the client's IP address has changed, the client attempts to send its new IP address to the server to register in the server's face list, after which synchronization is resumed. This process is repeated every time the synchronization stops.

Once the synchronization stops and the server answers the client's Interest, the synchronization is presumed completed. The client sends a finished synchronization message to the server and deletes its slices. The server deletes its own slices and acknowledges the client. At this point, the synchronization trees are stable and the client application exits.

3.2.2 Initial Synchronization Interest

The Initial Synchronization Interest is used to establish a link between client and server and start the synchronization of content. Its variables are listed in Table 3.1.

synchome client Interest variable	variable description
IP address	IP address of client
control bit	sets create or destroy slice
topo	topological prefix; generally ccnx:/
prefix	named collection prefix

Table 3.1: Initial Synchronization Interest variables

IP address

In order to be able to exchange Interests, client and server have to register the others' IP address in their FIB. This is done by adding the IP addresses as faces using the `ccndc` command. As the server functions as the home repository and is thus static, the user enters the server IP address manually when starting the client application. The client registers the entered IP address.

The client then retrieves its own IP address using a system call to the Linux command `ifconfig`, as the address may change with the client's mobility. The address is sent to the server in the Initial Synchronization Interest and registered in its FIB.

Control bit

A control bit determining the clients request is sent to the server application. The control bit may be set to create, delete, modify, or an integer between 100 and 999.

The create and delete options are passed onto the `ccnsyncslice` command, which accordingly creates or deletes a slice. The modify bit indicates that the clients' newly transmitted IP address should be registered as a face on the server. An integer between 100 and 999 is sent if an acknowledgement is required. This is done as Interests have a lifetime during which they are not resent.

Topological prefix

The topological prefix is used for routing Root Advise and Node Fetch Interests for the collection. Its purpose is to limit the number of hosts to which these Interests are routed. The prefix is registered in the hosts' FIB. We mainly used the default prefix `ccnx:/`, leaving the number of listening hosts unlimited.

Collection prefix

The prefix, or collection prefix, is the common prefix for all names in a collection. If we wished to retrieve, for example, `ccnx:/parc.com/music/oldies/abba` and `ccnx:/parc.com/music/oldies/elvis`, we would use `ccnx:/parc.com/music/oldies` as the common prefix.

Note that multiple prefixes can be passed along in the Initial Synchronization Interest. The composed Initial Synchronization Interest is of the form:

```
/synchome/-<control_bit>-<client_IP_address>-<topological_prefix>-<slice_1>
    -<slice_2>-...
```

The client application is responsible for finding the client's IP address. All other variables (control bit, topological prefix and collection prefix) are entered manually by the user.

3.2.3 Synchome

In this section we describe the mobile synchronization protocol applications - named `synchome` applications for brevity - which consist of the client side `synchomeclient` and the server side `synchomeserver`. Additionally, the `synchomewatch` application, which is used to keep track of the synchronization, is described.

3.2.4 Synchome Client

The `synchomeclient` application flow is depicted in Figure 3.2. At startup, the client sends an Initial Synchronization Interest to the server. If the Interest is not acknowledged, the client resends the Interest after a timeout. If an acknowledgment is received, the client application proceeds to the synchronization stage.

The synchronization is initialized and monitored by the `synchomewatch` application, which is called by `synchomeclient` after a succesfull initial Interest exchange. When the `synchomewatch` application exits, `synchomeclient` analyzes whether the synchronization has been completed by sending an Interest to the server application. If the server responds, a termination Interest is sent to it. Otherwise, the Initial Synchronization Interest is resent after a timeout.

This process repeats until the synchronization trees are stable.

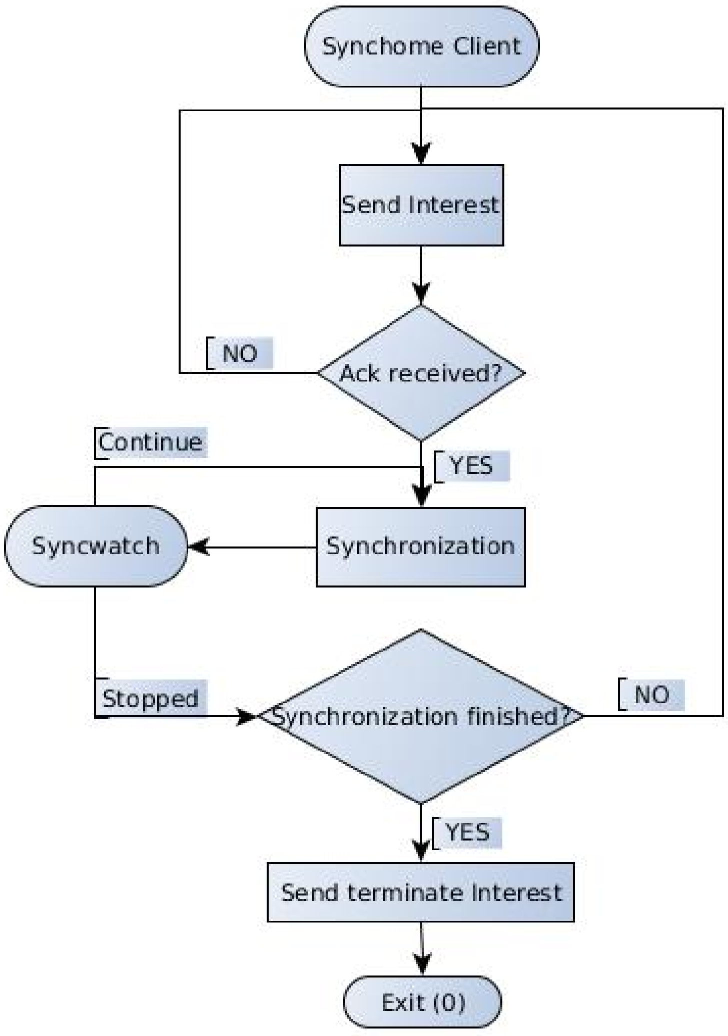

Figure 3.2: Synchome Client Flow

3.2.5 Synchome Server

On the server side, the `synchomeserver` application is continuously listening for new Synchronization Interests from clients. Upon receiving and parsing incoming Interest messages, the application proceeds according to the parsed control bit. Four actions may be taken, based on the control bit:

create
The clients IP address is added to the servers FIB and a synchronization slice is created.

modify
The clients new IP address is added to the servers FIB.

delete
The server's copy of the slice is deleted, the clients IP address is removed from the servers FIB.

number in range between 100 and 999
The server sends a reply without taking additional actions.

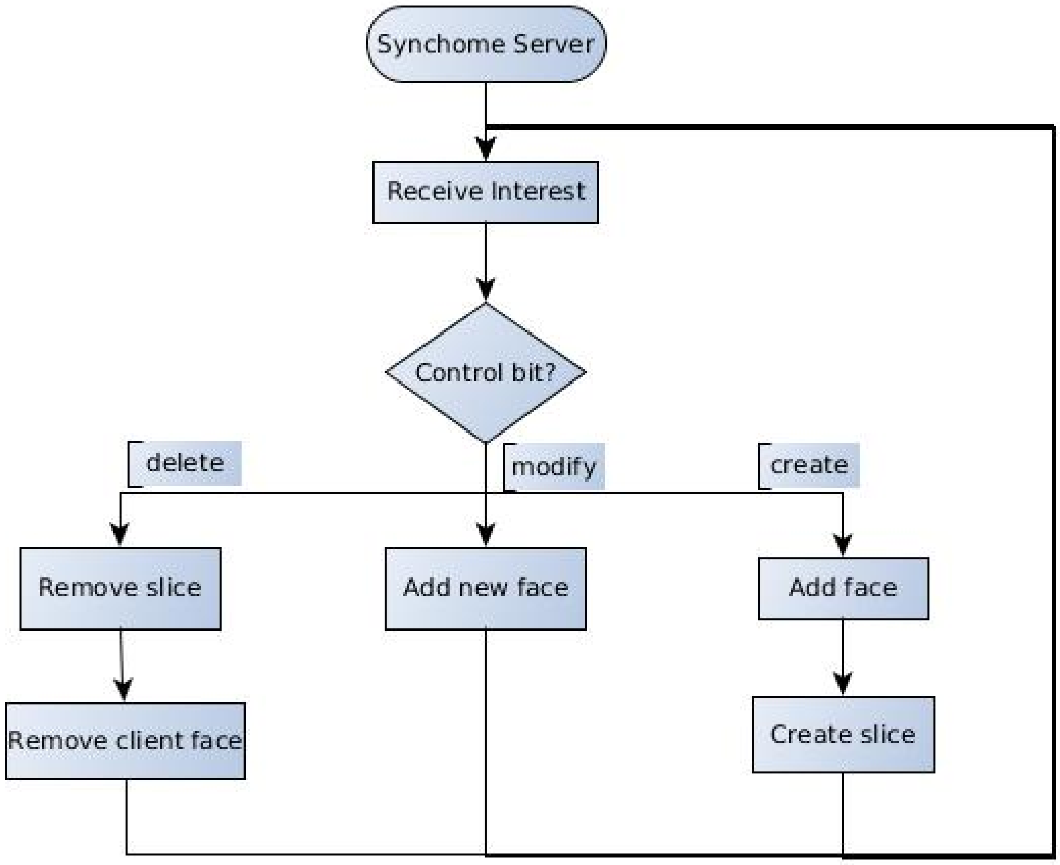

Figure 3.3: Synchome Server Flow

3.2.6 Synchome Watch

The `synchomewatch` application tracks the debug messages issued by the `ccnd` application at debug level 4. At this level, debugging messages concerning content are logged. These are mainly of the form `content_to` and `content_from`, with additional information concerning the face the content has arrived from, and the content name.

As long as the client is actively synchronizing content, `content_to` messages carrying requested content or synchronization protocol `Root Advise` and `Node Fetch` answers are exchanged. We track these messages in regular intervals as depicted in Figure 3.4. If no new messages are generated within the given interval, the synchronization has stopped.

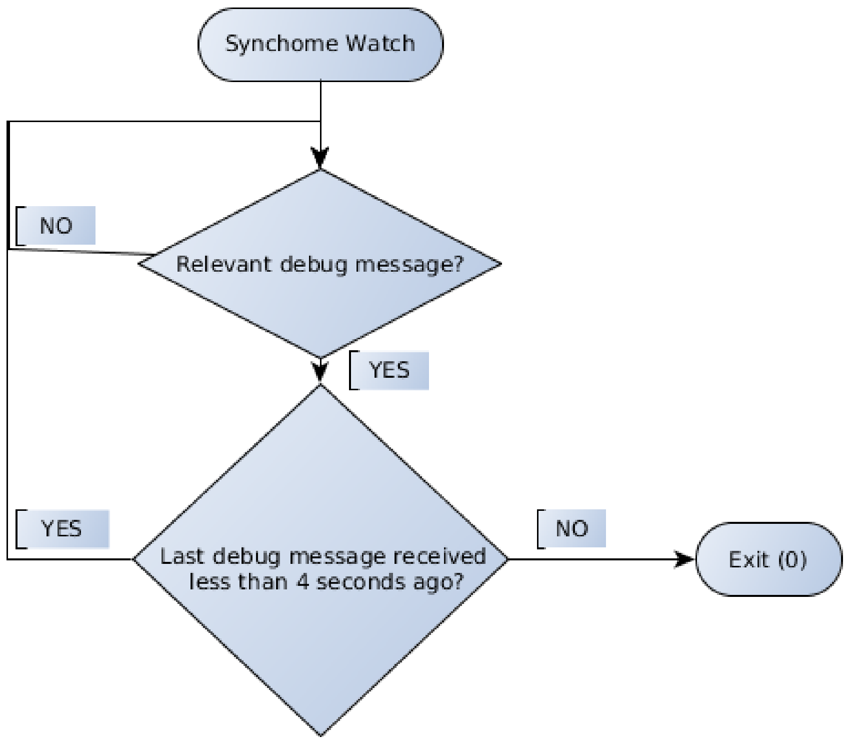

Figure 3.4: Syncwatch Flow

3.3 Implementation

This section contains implementation details concerning the `synchomeclient`, `synchomeserver` and `synchomewatch` applications. The CCNx project is implemented in C and Java. We worked on extending the synchronization protocol, a C implementation. All implementation details and code listings in this section concern the C implementation.

For the sake of brevity, only short excerpts of the implemented code are listed here.

3.3.1 Interest Message Handling

The Initial Synchronization Interest is composed on the client side by taking all variables and fusing them together into a single string, using a delimiter. This was done in order to be able to encode it into a single Interest message. The message is then sent to the server application.

```
//set together interest string.
//control bit can be: CRE - create slice; DEL - delete slice; REG - ip change
    ; 100-999 - no action
static char
*construct_interest_string(char *control_bit, char *ip_string, char
    *topo_uri, char *sync_uri, size_t sync_uri_length){
        int topo_uri_length = strlen(topo_uri);
    size_t ip_length = strlen(ip_string);

    //combine all arguments to interest string
    char *interest_string = calloc((topo_uri_length + sync_uri_length + 5),
        sizeof(char) + 1);

    char *replaced_sync_uri = replace(sync_uri, '/',"+");
    sprintf(interest_string, "%s%s%s", topo_uri, "~", replaced_sync_uri);
    free(replaced_sync_uri);

    //allocate memory for interest message
    char *msg = (char *) malloc(sizeof(char)* (ip_length + topo_uri_length +
        sync_uri_length + strlen(PREFIX) + 10));
    if(msg == NULL)
    {
        fprintf(stderr, "Can not allocate memory for URI\n");
        exit(1);
    }
    sprintf(msg, "%s%s%s%s", PREFIX, control_bit, ip_string, interest_string)
        ;
    free(interest_string);
    return msg;
}
```

Listing 3.1: Client function `construct_interest_string`

Listing 3.1 shows the `construct_interest_string` function responsible for handling the variables and parsing them together into an Interest message. As mentioned in Section 3.2.2, all variables except for the IP address are entered manually by the user. The IP address is retrieved using a system call as follows:

```
ifconfig | grep 'inet addr' | awk -F: '{print $2}'|awk '{print $1}'
```

The system call invokes the bash command `ifconfig`. The standard output of the command is then parsed using `awk` to retrieve the column containing the IP address.

The Interest string constructed using the `construct_interest_string` method is sent to the server using the `express_string` method, which handles the transmission of Interest packets, as well as the reception and extraction of the answer Interest packet.

On the server side, the message is parsed using the same delimiter as on the client side, and the strings are stored in their respective variables, which are then used to start the synchronization (refer to Listing 3.2 for the servers parsing function).

```
static struct parsed_interest_msg
parse_interest_msg(char *temp, char *delimiters){
    char *discard = strtok (temp, delimiters);
    char *control_bit = strtok (NULL, delimiters);
        char *ip = strtok (NULL, delimiters);
        char *topo_uri = strtok (NULL, delimiters);
        char *sync_uri = strtok (NULL, delimiters);
        char *sync_slices = replace(sync_uri,'+', "/");
    parsed_interest_msg p = {control_bit, ip, topo_uri, sync_slices};
        return p;
}
```

Listing 3.2: Server function `parse_interest_msg`

The server application employs two methods for handling the transfer of Interests. The `extract_interest_payload` method handles the receipt and extraction of incoming Interests, while `construct_interest_response` handles the sending of Interest packets.

3.3.2 Synchronization status

CCNx, as of yet, does not have a protocol that monitors whether a synchronization has finished, nor is it possible to track the arrival of content to the repository, as the synchronized content is exchanged in an arbitrary order. We determined tracking the `ccnd` debug output as the only feasible method to determine the synchronization status (whether it is running or has been finished).

In order to activate the CCNx debug output during runtime, we use `ccndlogging` with the `co` flag. `ccndlogging` calls a CCNx bash script that sets the environment variable CCND_DEBUG to the requested debug level, in this case "co", or debug level 4, at which debug messages concerning content objects are logged. The debug output of `ccnd` is then handled by the methods contained in `ccnd_msg.c`, which include the handler for Interest message debug, `ccnd_debug_ccnb`, as well as the handler for content message debug, `ccnd_debug_content`.

We inserted the logic shown in Listing 3.3 in the `ccnd_debug_content` method to gain access to content debug messages. Since CCNx handles requests by sending messages to the local or external `ccnd`, we exclude the local `ccnd` messages by querying for `content_to` messages leaving on any face except face 6, the default local `ccnd` face.

```
    if (strstr(ccn_charbuf_as_string(c), "content_to") != NULL && strstr(
        ccn_charbuf_as_string(c), "6 ccnx") == NULL)
    {
                FILE *f = fopen("/tmp/check_finished", "a");
                if (f == NULL)
                {
                    printf("Error opening file!\n");
                    exit(1);
                }
                fprintf(f, "%s\n", ccn_charbuf_as_string(c));
                fclose(f);
    }
```

Listing 3.3: Parse `ccnd_debug_content`

The relevant debug messages are then written to a text file located in the /tmp directory. The text file is monitored by `synchomewatch`. In order to determine whether new `content_to` messages are being generated, the current size of the file is compared to the previously logged size. If the file size has changed, the synchronization is still running. If not, it has stopped. This comparison is done every 4 seconds.

Note that, since `content_to` messages carry requested content as well as information regarding `Root Advise` and `Node Fetch` messages, the time necessary to synchronize content depends on the content transfer as well as the synchronization protocol information retrieval time. Also, `synchomewatch` is invoked sequentially on each slice in order to avoid interference, as all logged `content_to` messages are saved in the same output file.

After the synchronization of all slices has completed, debug logging is deactivated using `ccndlogging none`, and the text file is removed.

3.3.3 Issues of the proposed approach

3.3.3.1 Face entries and IP addresses

In the beginning stages we discovered that, even when registering an IP address with a node, it was not always ensured that the registered node could also be reached. At current, we are aware of two cases in which a node is not reachable and a global sync can not be maintained:

1. the requested node is behind a firewall
2. the requested node does not own a public IP address

The exception to both cases is when both nodes are located in the same private subnet. Additionally, if only one node is registered in the other's FIB, then information can flow only in one direction, from the registered node to the node that has registered it. Otherwise, when all participating nodes have registered with each other, Interest and Data packets are routed bidirectionally.

Consider the following figure for simplification:

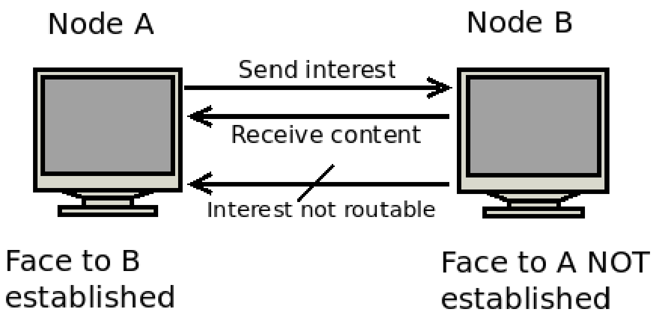

Figure 3.5: Impact of IP face registration on sync operation

3.3.3.2 Synchomewatch and debugging messages

The use of debugging messages has the downside of slowing down CCNx overall. In order to mitigate this, we activate debugging only during the actual synchronization. Otherwise, debugging is deactivated.

Similarly, the use of a text file to store the debugging messages is not optimal. The choice was made as the `ccnd` process variables had to be accessed during runtime by `synchomewatch`. Other inter-process communication methods such as shared memory or message queuing would have been preferable, but adding them to `ccnd_msg.c` was problematic, mostly due to the fact that allocated shared memory or message queues have to be destroyed when no longer required. The software architecture of `ccnd` did not readily provide an adequate location to perform this remove action.

Chapter 4

Evaluation

In this chapter we evaluate the `synchome` application and compare it to the regular `ccnsyncslice` application. In particular, we evaluate the processing and communication overhead by analysing the impact on file transmissions and the exchanged CCN messages.

4.1 CCN messages

For clarity, we again define the CCN messages which will be mentioned in subsequent sections:

CCN messages
CCN employs two types of messages, Interest messages and Data messages. Interest messages are used to query for content, while content is returned in a Data message.

Root Advise Interests
A Root Advise Interest is used by the synchronization protocol to determine if a remote collection contains names that are not in the local collection. The Root Advise is expressed as an Interest of the form:

```
ccnx:/<topo>/%C1.S.ra/<slicehash>/<roothash>
```

where `<topo>` is the topological prefix of the synchronization tree, `%C1.S.ra` is the command marker for the Root Advise Interest, the `<slicehash>` is a hash code that names the collection being synchronized and `<roothash>` is the combined hash of the root node for the synchronization tree.

The response to the Root Advise Interest is sent as a Data message and is of the form:

```
ccnx:/<topo>/%C1.S.ra/<slicehash>/<roothash>/<responsehash>/<
    version>/<segment>
```

where the `<responsehash>` is the hash for the responders root and `<version>` and `<segment>` refer to the version and segment number of the content being exchanged.

Node fetch Interests

When Root Hashes differ, the synchronization agent fetches the nodes of the remote synchronization tree that have hashes that differ from those in the local tree. To obtain these nodes, the synchronization agent issues Node Fetch Interests of the form:

```
        ccnx:/<topo>/%C1.S.nf/<slicehash>/<responsehash>
```

The response to the Node Fetch Interest is returned in a Data message and has the form:

```
        ccnx:/<topo>/%C1.S.nf/<slicehash>/<responsehash>/<version>/<
            segment>
```

The rest of the chapter is dedicated to the description of the scenarios, the hardware and software used, the test setup, and finally the evaluation results.

4.2 Hardware and Software

The `synchome` implementation was evaluated on the UBELIX[5] cluster using the ns-3 DCE[6] framework. The ns-3 DCE framework makes it possible to evaluate communication software by emulating it. This way, the communication software under evaluation can be run on any network setup with multiple nodes. The communication is simulated using ns-3.

For the evaluations, DCE-1.4 running ns-3-21 was employed.

4.2.1 DCE limitations

DCE has a number of limitations concerning its network module implementations as well as system and function calls. To circumvent these, changes had to be made to the mobile synchronization application source code for the simulations. The following DCE limitations had to be mitigated:

Call to `system()` not supported

The call to the `system()` command is not supported within DCE. We use `system()` in our application to invoke `ccnsyncslice`, `ccnsyncwatch` and `ccndc`.

In order to be able to create synchronization slices during the evaluation, we incorporated a function named `call_syncslice` in all `synchome` applications. The function consists of the code found in `ccnsyncslice`, with some minor changes, such as the removal of commandline argument input and assertion.

The same was done for `synchomewatch`. The code was incorporated in the `synchomeclient` application as the function `syncwatch`.

The `ccndc` function, which handles routing by adding or removing faces for remote hosts in the FIB, was not incorporated into the `synchome` application. Instead, we call `ccndc` within the ns-3-DCE simulation script before the synchronization starts, to add the remote hosts' face to the FIB, and after the synchronization ends, to remove the remote hosts' face from the FIB.

Call to `popen()` not supported

`popen()` is used by the `synchomeclient` to parse the output of `ifconfig` in order to retrieve the client's IP address. Like `system()`, it is not supported by ns3-DCE. As there is no DHCP support in ns3-DCE, we chose to remove this method from the application for the evaluations and instead enter the client's IP address statically.

No DHCP

The DCE wireless module does not have DHCP support. Wireless handoff between access points is only supported if the client keeps the same IP address.

4.3 Scenarios

We consider two scenarios:

Single synchronization

We synchronize one or multiple collections simultaneously. The scenario is comparable to a user creating new content, adding it to the repository, synchronizing it and then putting away the device with no synchronization applications left running.

Continuous synchronization

In this scenario, the user creates new content in intervals and pushes it to the repository into the same collection. If the `ccnsyncslice` application is used to synchronize the content, the collection is created once and the synchronization agent left to run until the user decides to remove the collection. If the `synchome` application is used, the user has to start it for every new content added, as `synchomeclient` shuts down after successful content synchronization.

For both scenarios, we use a topology with a static client. Figure 4.1 shows the topology. There is one client that connects via access point to a home repository. In our scenarios, the client remains in connectivity range of the access point to perform the synchronization operation. The client is connected to the access point using IEEE 802.1g. The access point is a bridged wireless access point/CSMA node. The bridge is necessary to merge the wireless and CSMA subnets into one network. The CSMA link between access point and home repository supports a data rate of 1000Mbps and has a delay of 6560ns.

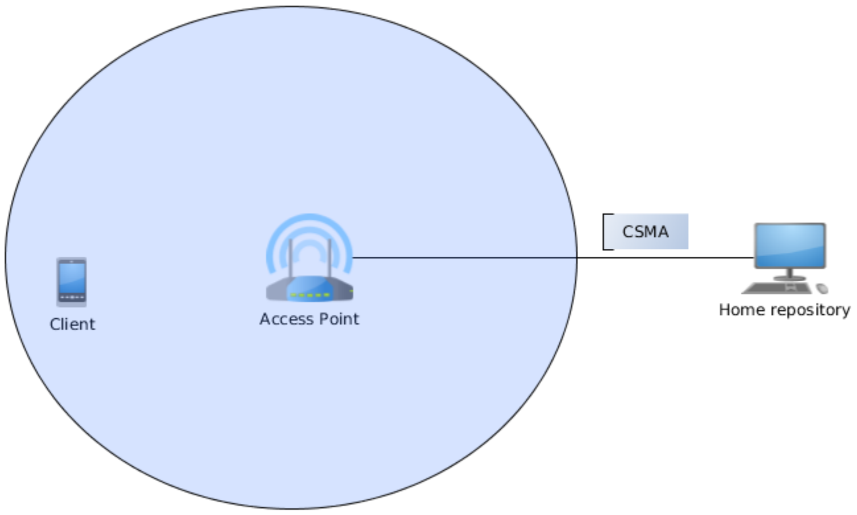

Figure 4.1: Network topology with static client

4.4 Performance Evaluation

We will first discuss the test setup and then show the evaluation results.

4.4.1 Test setup

We run the simulations by executing the following steps in the ns3-DCE script:

1. Create the necessary folders and files on the client and home repository nodes, such as the `REPO` folders, in which the repository is initiated, as well as the file to be pushed to the repository, which is stored in the client's \tmp directory and named `bigone`.

2. Start `ccnd` on client and server.

3. Initiate repository with `ccnr` on both the client and home repository.

4. Once the repository is running, the client pushes the generated `bigone` file to the repository by issuing the command `ccnseqwriter` as follows:

   ```
   ccnseqwriter -r <ccnname> bigone
   ```

   The command `ccnseqwriter` creates new ccn content by using stdin as the source of data. The `-r` option generates a start-write request to the repository. Without it, the content would not be saved to the repository. `ccnname` is the uri name of the content.

5. Next `ccndc` is invoked to register client and home repository with each other using the static IP addresses allocated to the nodes.

6. In this step, the actual synchronization is performed. For the evaluation of `ccnsyncslice`, `ccnsyncslice` is called from the `synchomewatch` application in the first scenario, as we need to monitor the synchronization completion time. In the second scenario, `ccnsyncslice` itself is used to synchronize the content.

   For the evaluation of `synchome`, in both scenarios `synchomeserver` is initiated on the home repository and `synchomeclient` on the client.

7. Once the given scenario has completed, `ccnd` is stopped using `ccndsmoketest kill`, and the simulation exits.

4.4.2 Single synchronization

In this scenario, the client performs one synchronization of one or multiple collections. The file added to the collections is of size 1, 5 or 10MB. It is stored in 1, 2 or 3 collections containing 5 files. We used this scenario to evaluate the overhead of the `synchome` application compared to `ccnsyncslice`.

Figure 4.2 shows the comparison of the `synchome` and `ccnsyncslice` applications when synchronizing 2 collections containing 5 files sized 1, 5 or 10MB. The x-axis shows the filesize of the content. The y-axis shows the runtime of the applications in seconds. We used `synchomewatch` during the simulation to monitor `ccnsyncslice`. The runtime is calculated by adding together the synchronization times for each collection as reported by ns3-DCE.

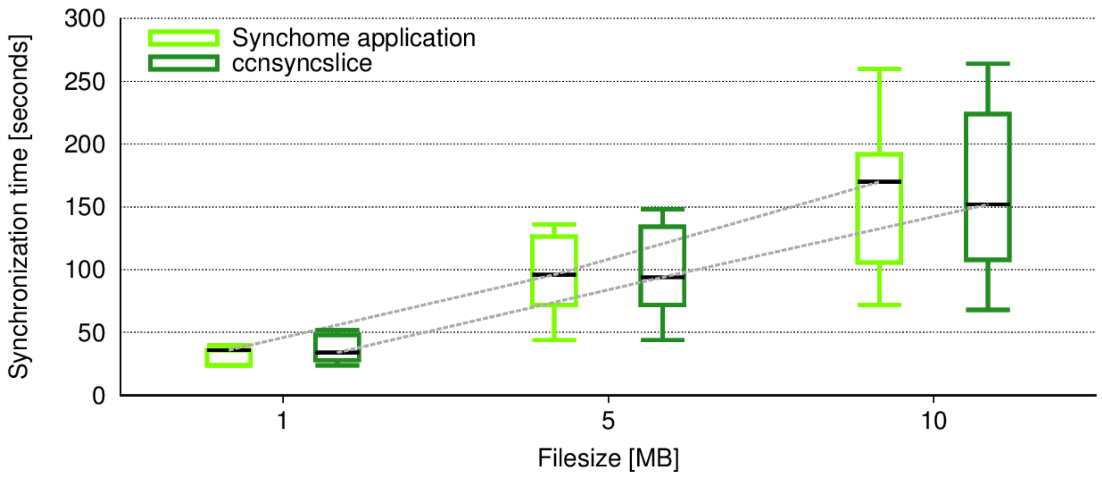

Figure 4.2: Results of synchronizing collections containing 5 files of size 1MB

It can be observed that:

1. The mean runtimes of both `synchome` and `ccnsyncslice` are nearly equal for each collection number. `ccnsyncslice` has a slightly better mean performance. This is due

to `synchomes'` additional overhead caused by registering the IP addresses, etc.

2. For both `ccnsyncslice` and `synchome`, there is an increase in variance between maximum and minimum runtime as the filesize increases. This is due to the way the synchronization protocol handles Root Advise, Node Fetch and content messages.

   The synchronization protocol issues a Root Advise Interest every 20 seconds if no new content is found. If new content is found and needs to be synchronized, the frequency of Root Advise Interests increases to multiple Interests per second until the synchronization finishes. Only a small part of all these Interests are answered. If a Root Advise Interest is answered, and the root hash differs from the local synchronization trees root hash, Node Fetch Interests are exchanged.

   Since Node Fetch Interest replies are Data messages, the transfer of the content Data messages is stalled until the synchronization protocol has finished exchanging hashes. This may incur significant overhead, depending on how often Root Advise Interests are successfully exchanged and Node Fetches initialized. The replies to Node Fetch Interests are sized between 886 and 4133 bytes when exchanging files of 1MB; for larger files, the range in Node Fetch reply size becomes broader. This means that the number of Node Fetch replies exchanged is highly variable.

   As larger and more content is transferred, the time during which Root Advise messages can be exchanged and start Node Fetch exchanges increases. The resulting variance in the maximum and minimum runtime is caused by the high variability in the number of exchanged Node Fetch Interests, since these stall content exchange.

3. The variance in runtime is more pronounced for `ccnsyncslice` for each filesize. We explored this in the next experiment.

Figure 4.3 shows the resulting runtimes of multiple runs synchronizing 3 collections containing files of size 10MB. The x-axis shows the experiment number, while the y-axis shows the runtime in seconds. We observe that:

1. The minimal runtime for both applications is 100±2 seconds for all repetitions of the scenario.

2. The mean runtimes for both applications range between 255 and 300 seconds. In run 1, both applications have similar runtimes, while in run 2 `ccnsyncslice` has the better mean runtime and in run 3 `synchome` has the better mean runtime.

   This inconsistency in performance is due to the previously mentioned variability in the number of successfully exchanged Root Advise messages and consequently initialized Node Fetch exchanges. As previously stated, when more content is exchanged, Root Advise Interests have a higher chance to be answered, and in this experiment 50MB are transferred during each run.

3. The maximum runtime of the `ccnsyncslice` application is slightly higher than the maximum runtime of `synchome` for every repetition of the experiment. An analysis of the log files showed that `ccnsyncslice` tends to exchange more Node Fetch Interests than `synchome`. The only difference between the applications message exchange is that `synchome` generates additional messages routed under the `synchome` prefix. It seems that these messages have priority over Root Advise messages, thus cutting down the number of Node Fetch exchanges.

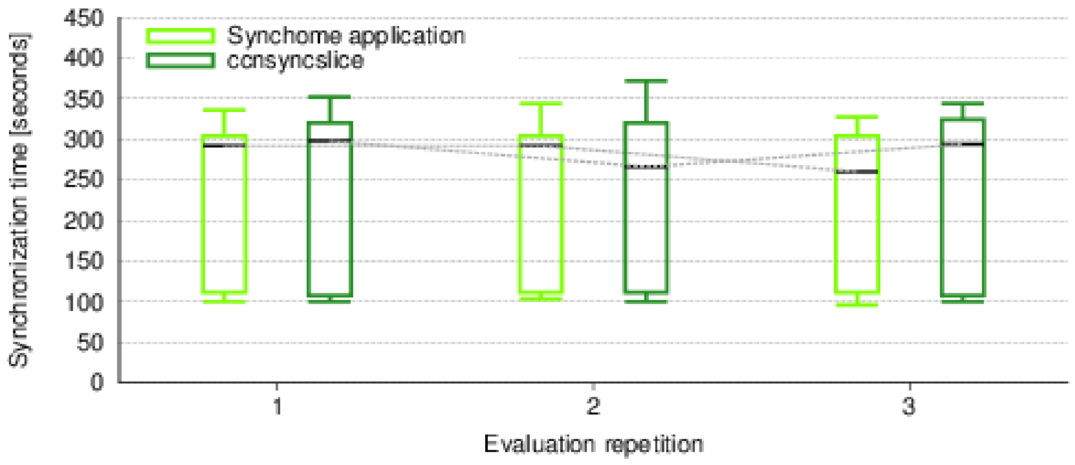

Figure 4.3: Results of multiple runs synchronizing 3 collections 10MB

4.4.2.1 Conclusions

We anticipated that the `synchome` application would run slightly longer than `ccnsyncslice`. Comparisons showed that which application had a better mean run-time was variable. Through all runs, minimum runtimes were nearly the same for both applications. `ccnsyncslice` had a slightly higher variance between maximum and minimum runtime and often a higher maximum runtime.

The inconsistency in performance between the applications is caused by the exchange of Node Fetch Interests, which are often exchanged even when the actual exchange of content has concluded. The messages generated by `synchome` apparently have a slight influence on how often Node Fetch Interests are exchanged thus minimizing the maximum runtime of the `synchome` application when compared to `ccnsyncslice`.

4.4.3 Continuous synchronization

In the second scenario, we simulated a client generating new content periodically for 24 hours. The generated content is one file of size 1 or 2MB added every 900, 1800 or 3600 seconds. This is equal to 96, 48 or 24 files, respectively, being synchronized over the 24 hour period. When synchronizing the content using `ccnsyncslice`, one collection slice is created for the 24 hours duration. Since `synchomeclient` exits upon successful synchronization, it is called at every interval to synchronize the new content.

We evaluated the effect of periods without active synchronization on the number of messages received and transmitted by the client. While messages received/transmitted by both client and home repository were evaluated, we analyze only the clients' messages. This is because CCNx maintains flow balance between Interest and Data messages. A party must transmit at most one content object in response to a single received Interest message. Thus, for example, `interest_from` messages received by the client map to `content_to` messages relayed to the home repository.

As `ccnd` generates a new face for every CCNx application call - be it `ccndc`, `ccnr`, `ccnseqwriter` or `synchomeclient` - we parsed only the messages relayed between the client and home repository `ccnd`, excluding internal faces. The following graphs display only the messages transmitted to the home repository over the external face generated by `ccndc add ccnx:/ udp <home repository ip>`.

Figure 4.4 shows the comparison of the `interest_to` messages transmitted by `synchome` and `ccnsyncslice`. The x-axis shows the interval between new content. The y-axis shows the number of issued messages. These messages consist of Root Advise Interests issued by the client.

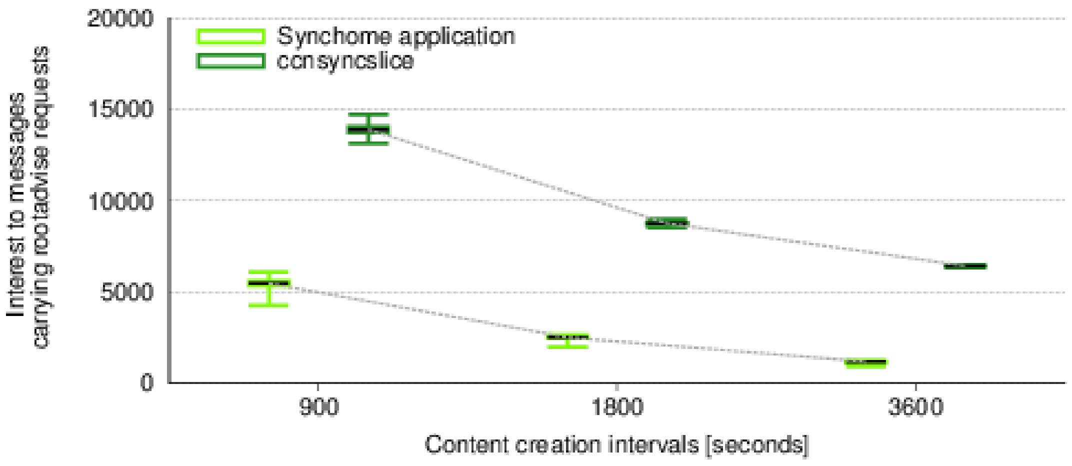

Figure 4.4: Interest_to messages generated by client

It can be observed that:

1. The `ccnsyncslice` application generates significantly more Root Advise Interests than `synchome`. The larger the interval between new content, the larger the gap between generated messages becomes. When content is generated every

900 seconds, `ccnsyncslice` generates 153% more `interest_to` messages than `ccnsyncslice`. At 3600 second intervals, the increase is 437% more messages. This is due to the fact that `ccnsyncslice` exchanges Root Advise Interests every 20 seconds while not synchronizing. `synchome` does not exchange any messages when not actively synchronizing, as it removes the slices.

2. As the interval between new content generation decreases, the number of Interest messages converges. This behavior is expected, as `synchome` has to start the synchronization process more often, thus also increasing the number of Root Advise Interests transmitted.

3. The variance between maximum and minimum value increases for decreasing intervals. As mentioned in the previous section, Root Advise Interests are relayed every 20 seconds when not synchronizing content and multiple times per second when synchronizing. As the interval between new content additions decreases, the applications spend more time synchronizing, thus increasing the number of exchanged Root Advise Interests.

The graph displayed in Figure 4.5 shows the difference in generated `content_to` messages. The x-axis label shows the content generation interval, the y-axis the number of sent content messages.

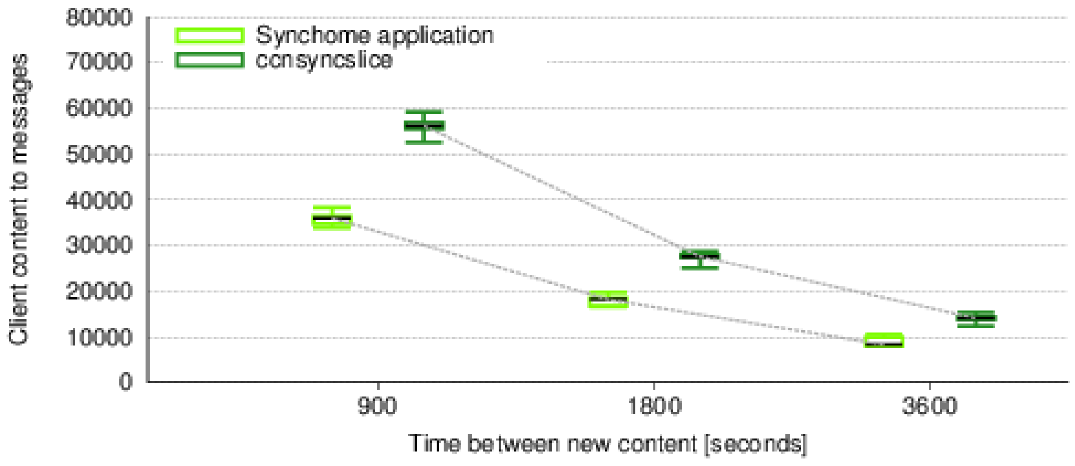

Figure 4.5: Content_to messages generated by client

As it can be seen, the difference in the number of transmitted content messages is significant. The `ccnsyncslice` application sends out around 50% more messages at every time interval than the `synchome` application. Further probing showed that the `content_to` messages are not only composed of messages relaying requested content, but also of Data messages relaying information for the synchronization protocol, such as Root Advise and Node Fetch Interest answers. Extracting the the Root Advise and Node Fetch Interests resulted in the following graphs.

Figure 4.6 shows the Data messages carrying Root Advise Interest answers. The x-axis label shows the content generation interval in seconds, the y-axis the number of sent Root

Advise Interest answers.

We observe that:

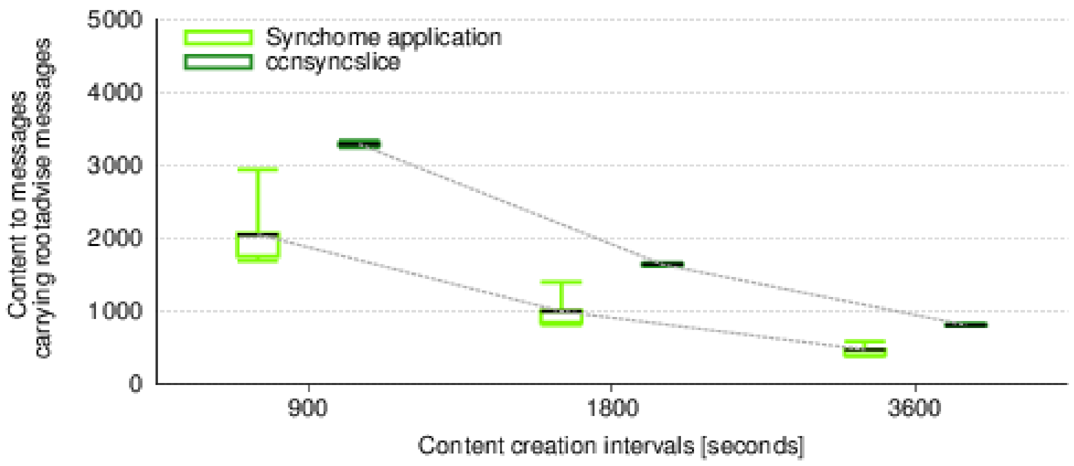

Figure 4.6: Content_to messages carrying Root Advise Interest replies

1. The number of sent Root Advise Interest answers shows nearly no variance from the mean number of messages at each interval for `ccnsyncslice`. This is likely due to the fact that the collection is being constantly synchronized, thus leaving the same window of opportunity open for Root Advise Interests to be answered during each run.

2. `synchome` shows an increase in variation between maximum and minimum messages sent as the content creation interval decreases. As the interval decreases, more content is being synchronized, thus leaving more time for Root Advise Interests to be sent. Additionally, as more Root Advise Interests are answered, Node Fetch exchanges are initialized more often, prolonging the synchronization time and again increasing the window of opportunity for new Root Advise Interests. As multiple Root Advise Interests are transmitted every second during active synchronization, slight increases in synchronization time may cause significant increases in the overall number of Root Advise replies

3. For both applications, the number of sent messages doubles as the interval decreases from 3600 seconds to 900 seconds.

The last graph, Figure 4.7, shows the number of Node Fetch answers transmitted at every interval. The x-axis shows the content generation interval in seconds, the y-label the number of content messages transmitted by the client carrying Node Fetch Interest answers.

We observe that:

1. The `synchome` application shows nearly no variance from the mean number of relayed Node Fetch replies, while `ccnsyncslice` shows an increase of the variance between

maximum and minimum messages sent as we decrease the intervals from 3600 seconds to 900 seconds. Taking into consideration the results concerning the transmission of Root Advise replies, this would indicate that `synchome` has a higher variance in the number of sent Root Advise Interests because they are answered less often, which in turn causes more Root Advise Interests to be sent.

Additionally, when `ccnsyncslice` is running constantly, Root Advise Interests are occasionally answered multiple seconds after the synchronization has finished. As `synchome` stops the synchronization within 4 seconds of the last content exchange, it cuts off these additional Node Fetch exchanges.

2. `ccnsyncslice` shows a message increase of 192% at the 3600 second interval and 268% at the 900 second interval compared to `synchome`.

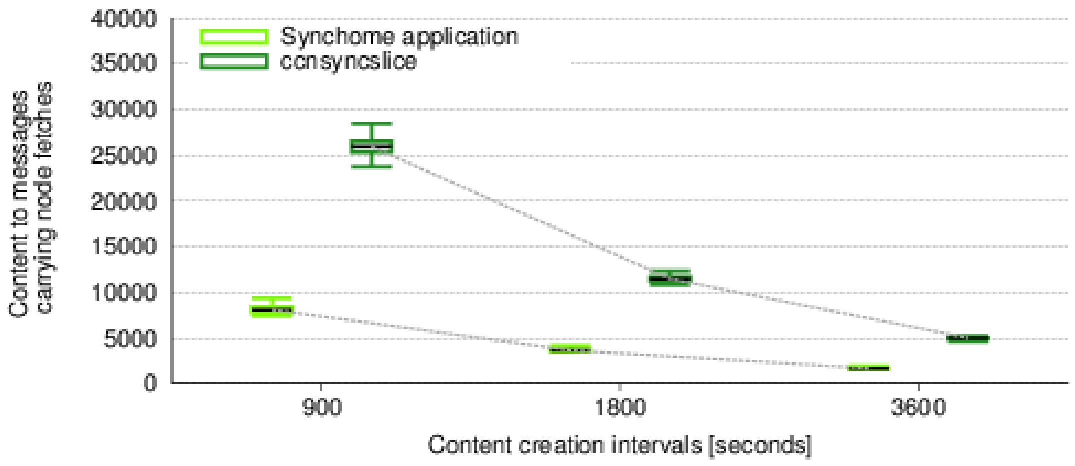

Figure 4.7: Content_to messages carrying Node Fetch Interest replies

4.4.3.1 Conclusions

The results of the evaluation show, as expected, that `synchome` generates significantly less CCN messages than `ccnsyncslice`. They also show that, as content is synchronized more often, the number of Root Advise Interests transmitted converges for the two applications, while the number of Node Fetch Interest replies diverges, although this divergence would dissapear if there were no periods without active synchronization.

The convergence of the Root Advise messages is due to the fact that, as content is being synchronized more often, the amount of time during which Root Advise messages can be exchanged increases. If `synchome` would constantly synchronize content without pauses, it would essentially be identical to `ccnsyncslice` in functionality, as it is built on top of it.

The reason for the divergence is not as clear. An analysis of the log files indicates that the Root Advise Interests sent by `ccnsyncslice` resolve more often in a Node Fetch exchange than the ones sent by `synchome`. As mentioned in the discussion of the application runtimes, the only difference concerning the message exchange of `synchome` and `ccnsyncslice` is that `synchome` generates additional messages, which are routed under its own prefix.

Chapter 5

Conclusions

5.1 Summary

In this thesis, we developed a synchronization protocol for mobile clients. The protocol extends the CCNx synchronization protocol by adding the capability to synchronize content with any node, regardless of location, as well as being self-regulatory. The user needs to only start the `ccnd` and the repository. The application takes care of registering the remote nodes face to its own `ccnd`, as well as its own face to the remote node. Next, it synchronizes the content, stops synchronization when completed, removes the registered faces and exits.

In addition to the synchronization protocol for mobile clients, we implemented an application called `synchomewatch` to monitor whether the synchronization of content has stopped. This was required since the current synchronization protocol could not differentiate network interruptions from successfully completed synchronizations. In both cases, Root Advise messages, which are monitored by the `ccnsyncwatch` application, would have resulted in no response.

We evaluated the implemented mobile synchronization protocol and compared it to the existing synchronization protocol. The evaluation was performed with ns3-DCE, which allowed for the evaluation of the implemented code under the ns3 network simulator without the need to change the application source code.

5.2 Conclusions of results

During the evaluation, we found that the differences in synchronization time between our implementation and the synchronization protocol when running a single synchronization of one or multiple collections was nearly negligible for small files, and converged for larger files. We also found that, the larger the size of the content to be transferred is, the more the synchronization time deviated from the mean, due to the exchange of synchronization protocol messages and the implementation of our monitoring method, which also takes the synchronization protocol messages into account.

We also found that our mobile synchronization protocol lowered the number of generated content and Interest messages carrying Root Advise and Node Fetch messages significantly compared to the synchronization protocol when synchronizing content intermittently. During the 24 hour synchronization evaluation with content generation every 3600 seconds, the home repository transmitted 192% more Node Fetch replies and 437% more Root Advise messages when running the synchronization protocol, compared to our protocol.

5.3 Future Work

Our `synchomewatch` application monitors the synchronization process by analyzing `ccnd` debug messages stored in a text file. As the logic to query and store relevant debug messages is stored within the `ccnd` code, all `synchomewatch` instances called on top of the same `ccnd` instance access the same text file. It would be advisable to examine the case in which a client invokes multiple `synchomeclients` for concurrency issues and perhaps introduce a way to seperate `synchomewatch` access to the log file into sessions.

Additionally, functionality could be added to the home repository to keep track of what content a client possesses. This would make it possible for the home repository to inform other clients, which have subscribed for content updates, about new content available.

Chapter 6

Appendix

6.1 Code Listings

In the following, all edited code files are declared. The source code itself can be found on the attached physical medium

6.1.1 Newly Created Source Code

The following files have been newly created for this thesis:

`src/ccnx/csrc/cmd/synchomeclient.c`
The mobile client application `synchomeclient` was implemented in this file.

`src/ccnx/csrc/cmd/synchomeserver.c`
The synchome server application `synchomeserver` was implemented in this file.

`src/ccnx/csrc/cmd/synchomewatch.c`
The function responsible for tracking the sync was implemented in this file.

6.1.2 Modified Source Code

The following files have been modified for this thesis:

`src/ccnx/csrc/ccnd/ccnd_msg.c`
Logic to extract debug messages was implemented in this file.

Bibliography

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '09. New York, NY, USA: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1658939.1658941

[2] (2015) Ccnx project. [Online]. Available: http://www.ccnx.org/

[3] (2015) Ccnx repository protocols. [Online]. Available: https://www.ccnx.org/releases/latest/doc/technical/RepoProtocol.html

[4] (2015) Ccnx synchronization protocol. [Online]. Available: https://www.ccnx.org/releases/latest/doc/technical/SynchronizationProtocol.html

[5] (2015) Ubelix - university of bern linux cluster. [Online]. Available: https://www.id.unibe.ch/content/services/ubelix/overview/index_ger.html

[6] (2015) Direct code execution. [Online]. Available: https://www.nsnam.org/overview/projects/direct-code-execution/