Automatic Detection of Forwarding Opportunities in intermittently connected Content-Centric Networks

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Jürg Weber
2013

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

Contents

List of Figures

List of Tables

Acknowledgement

First of all I thanks my supervisor Carlos Anastasiades for sacrifice his valuable time, for his detailed and patient explanations and his fabulous care. Prof. Dr. Torsten Braun for the opportunity to write a thesis in a practice oriented research group. Further my thanks goes to my office mates Tobias Schmid and Alexander Striffeler, for interesting coffee breaks, excitatory mini table tennis sessions and mind expanding ”FüBi’s” (after work beer’s). To all my friends and family which I have annoyed with monologues over my bachelor thesis.

Abstract

In this thesis an overhearing mechanism for the content-centric network framework CCNx was implemented. In content-centric networks communication is based on content names instead of host identifiers. The implemented functionality overhears the network for multicast messages. Each of the overheard message contains a forwarding information in form of a name-prefix. A function extract this name-prefixes out of the messages and adds them to the local forwarding table. This approach provides an opportunity to request new available data sources.

Afterwards this overhearing functionality was evaluated in terms of processing and energy overhead compared to the CCNx version with disabled overhearing. The evaluation was done on resource constrained mesh nodes. In a first step the data throughput of CCNx was compared with the results of Van Jacobson et. al. [1]. When running CCNx on a resource constrained mesh node, the throughput is lower by factor 14.5 compared to the results of Van Jacobson et. al. In a second step the multicast wireless throughput was evaluated with varying parameters. By decreasing the message retransmission delay form 4 seconds to 0.25 seconds, the multicast data throughput can be increased of 731%. At last, the energy consumption of a mesh node with enabled and disabled overhearing was evaluated. Depending on the file size, the overhead of the energy consumption with enabled overhearing is between 0.74% and 5.22%.

Chapter 1

Introduction

1.1 Motivation

Computer networks were invented to enable access to shared resources. The location of the resources were known, i.e., at which computers they are stored or available . During the years, as the Internet became more popular, new services and content dissemination models appeared growing around this host-to-host communication model. Nowadays the circumstances have changed, the Internet has become a cloud for information and services. The average user is not interested where the machine is located. He/She is interested in specific content, such as youtube videos, flicker photos, news feeds, movie streams or music files. In his mind he asks the Internet for content and gets a response from the network without knowing the exact geographic location of the information. However, the architecture is still the same as in the 70's. For the future networking that is not really feasible, the proliferation of mobile devices such as netbooks, tablets and smart phones is increasing and there are more flexible approaches required to enable dynamic mobile networking. In fact, today's host-to-host communication is not suitable for the future. To address this problem, a new networking approach was introduced called content-centric networking. The CCNx project provides an open source implementation of this content-centric networking approach. Instead of host identifiers, communication is performed based on content names.

1.2 Tasks and Problem Formulation

At the moment, the CCNx open source framework is in its childhood. It provides only a few services. An example for a missing service is the routing configuration, where the network paths need to be set statically. Therefore, it is setup manually or by DHCP-like services configured by humans [2]. In dynamic or opportunistic environments with many devices, it is nearly impossible to do an appropriate set-up because communication partners will change frequently.

In CCNx, packets are routed based on their name-prefix and not based on host identifiers as in IP-networks. The main focus of the CCNx development points to wired Internet where name-prefixes do not change often. At the moment, the forwarding tables will be populated manually at start-up. During operation, no forwarding table updates are provided. This is not feasible for mobile networks where connections and available content may change often. Due to changing communication partners, FIB entries cannot be set up statically and messages need to be transmitted by multicast. Information about available content can be obtained by overhearing multicast communication. The tasks of this bachelor thesis are listed below:

- Implementation of an overhearing functionality to dynamically add new FIB entries based on available content.
- Extending ADAM images with CCNx.
- Evaluation of wireless CCNx communication performance on mesh nodes. Determination of optimal CCNx parameters.
- Evaluation of implemented overhearing functionality in terms of processing and energy overhead.

1.3 Outline

The remainder of this report is structured as follows: In Chapter 2 we describe the content-centric networking approach applied in the CCNx framework. Design and Implementation details of the overhearing functionality are described in chapter 3. Chapter 4 describes the testbed setup and presents our evaluation results. Finally, in chapter 5, we conclude our work and give an outlook to future work.

Chapter 2

Related Work

2.1 Networking Named Content

Content-centric networking (CCN)[1] is a new networking approach. In content-centric networks, routing is based on content names instead of host identifiers. There are two CCN packet types, namely Interest and Data.

2.1.1 Interest and ContentObject

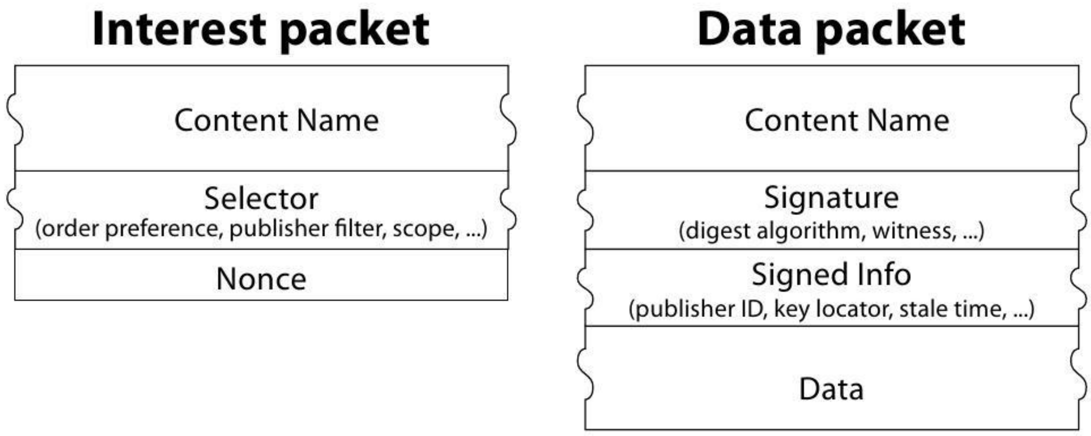
Interest packet
Content Name
Selector
(order preference, publisher filter, scope, ...)
Nonce
Data packet
Content Name
Signature
(digest algorithm, witness, ...)
Signed Info
(publisher ID, key locator, stale time, ...)
Data

Figure 2.1: Shows the two basic message types in CCN, namely Interest and Data packet. A consumer needs to express an Interest in a content name to receive corresponding data packets [1].

A consumer asks for content by broadcasting the Interests over all available faces. A face can be considered as an interface in an IP-network. Each node that hears the Interest and having data that satisfies it, can respond with a Data packet. Other nodes that do not have the requested content forward incoming Interests over an existing forwarding entry. In this case, a new entry will be included in the Pending Interest Table of the node. This table stores forwarded

Interests for which the node does not received a response yet and are therefore pending. Data is transmitted only in response to an Interest and consumes that Interest stored in the pending Interest table. Received ContentOjbects are stored temporarily in the Content Store serving as cache. With this solution no loops are possible, because each node without related pending Interests will discard the incoming content.

A data file consists of many ContentObjects, and each of these objects is identified by the segment number. This number describes the place of the object in the order of the requested data. So, if one gets lost, it is possible to request that specific ContentObject again.

2.1.2 CCN Forwarding

The CCN processing is based on three basic components:

- Pending Interest Table (PIT)
- Forwarding Interest Base (FIB)
- Content Store (CS)

If an Interest is received, the CS is checked fist, whether a matching object is available. If not, the PIT is considered if already similar Interest are pending. If there is already an entry in the PIT, the Interest can be discarded because an answer is already pending. If there is no entry in the PIT, the FIB holds information where to forward the Interest. If no information is available, the Interest is discarded.

In this bachelor thesis, we focus on entries in the FIB.

2.1.3 Naming

Names are composed of multiple components structured in an hierarchical way. An individual name is composed of a number of components. Each component is separated with a slash "/". This delimiters are not part of the name. Figure 2.2 shows a name in the human readable as well as the binary encoded from used in the framework. This strings are called name-prefixes.

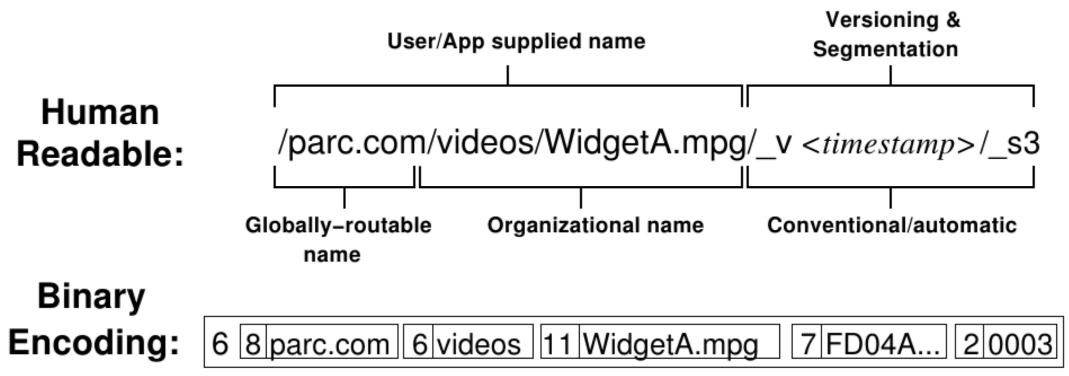
User/App supplied name
Versioning & Segmentation
Human Readable:
/parc.com/videos/WidgetA.mpg/_v <timestamp>/_s3
Globally-routable name
Organizational name
Conventional/automatic
Binary Encoding:
6 8 parc.com 6 videos 11 WidgetA.mpg 7 FD04A... 2 0003

Figure 2.2: Example Data name [1]

2.1.4 CCN Faces

In CCN, connections to other hosts or applications are called faces. In case of applications, the face corresponds to local Unix sockets, in case of hosts, it corresponds to UDP or TCP sockets. Therefore, multiple faces can be supported over the same interface.

The forwarding of Interests in CCNx is based on the FIB. Every entry in the FIB contains a prefix which directs Interests based on longest-prefix match on the specified faces. Every entry is only valid for the specified lifetime and is removed afterwards.

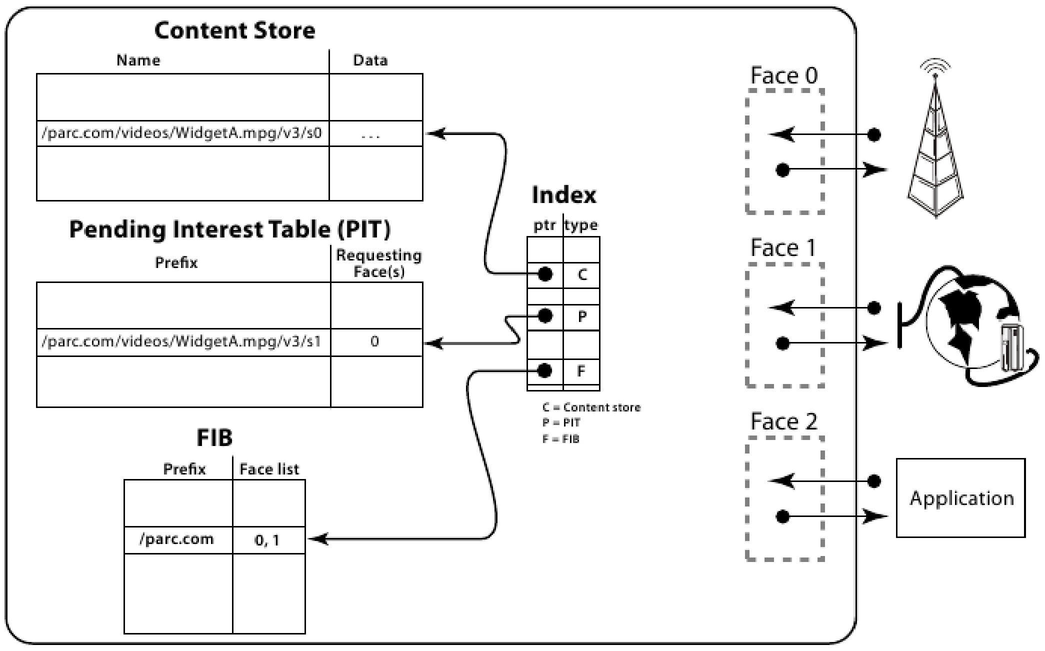

Figure 2.3: CCN forwarding engine model [1]

The forwarding engine model as shown in Figure 2.2 is handled by the CCNx daemon (CCND) (see 2.2.1).

2.2 Project CCNx

CCNx[3] is an open source project providing an implementation of the content-centric networking concept on top of a common IP network. This is the framework this thesis work with. The framework is written in C and in Java. There are some main functions which are used in this thesis which will be explained below.

2.2.1 CCNx Daemon

The CCNx daemon, called ccnd, is the central component of the CCNx implementation. It is running on every host and it performs all processing and forwarding of received messages. The typical configuration is to run one ccnd on each host. Applications running on the host will communicate via local faces with the ccnd, which may redirect Interests to attached networks based on the FIB configuration. The ccnd can be configured by environment variables. The ccnd application can be used to configure the FIB.

2.2.2 CCNDC

CCNDC is a utility to configure the FIB by adding or deleting static entries. FIB entries are added by specifying the prefix and face information. A face is defined by host address, port number, and transport protocol (UDP or TCP). In this work we only consider the UDP protocol.

2.2.3 CCNx key management

Integrity and authenticity in CCNx is ensured by signing every ContentObject with the private key of the publisher. For verifying a signature, the Public Key of the signer must be known. The public keys are stored under the prefix `ccnx:/ccnx.com/Users`. The first Interest of every request is requesting the key of a user stored under this name space. So it must be ensured that this prefix is stored as a forwarding entry in the FIB.

2.2.4 CCNx name-prefix entry

The `nameprefix_entry` is a struct which stores all the data which is related to an existing name-prefix such as the associated face or associated forwarding information. A `nameprefix_entry` is built as a linked list, each new entry is added as a child from another entry. All name-prefix entries are stored in a hash table which is keyed by name-prefix length. For more information, have a look at `nameprefix_entry` Struct Reference[4].

2.2.5 CCNx Flags

CCNx uses many different flags for internal face (see 2.1.4) and message processing. In principle the CCNx flags describes a state or a specific setting of an arbitrary CCNx functionality such as a face or a message. Below a short description of the flags which are used within this thesis:

- CCN_FACE_LOCAL
  Face is mapped to a local application.
- CCN_FACE_MCAST
  Defines whether a face is mapped to a multicast address.
- CCN_FORW_ACTIVE
  Activates the forwarding entry, so that messages can be propagated over this entry.

More information about specific flags can be found at the CCNx Face Management and Registration Protocol manpage[5]

2.2.6 Pipelining

The pipeline size defines the number of segments that can be concurrently requested.

Chapter 3

Design and Implementation

3.1 Design

In this work setup Forwarding Interest Base (FIB) entries dynamically by overhearing multicast content transmissions. This functionality is directly integrated in the CCND. Thus, we focus on the manipulation of the FIB.

3.1.1 Overhearing

Overhearing is the passive activity of a network participant listening to the communication between two other participants. Using a multicast Face is the only way to perform overhearing with CCNx.

To receive multicast messages, every node needs to register at least one multicast Face with a multicast IP address. This is required since the registration of a multicast Face opens an UDP socket that listens for messages from that address. We assume that all hosts use the same port for the multicast Face. At startup, no information about available prefixes is available and therefore, a prefix has to be registered to open the multicast socket.

When processing incoming content, additional functionality is required. First, the content names need to be processed and the prefixes need to be extracted. Second, the extracted prefixes need to be added to the FIB so that received Interest can be forwarded to the corresponding content source.

To ensure efficient processing, several issues need to be considered and are discussed below.

3.1.2 Messages

In this work, we only process overheard ContentObjects but no overheard Interest messages. ContentObjects can only be overheard if a content source is available. On the contrary, overheard Interests only indicate a local Interest which may not be necessarily satisfied.

3.1.3 Entry Lifetime

In mobile networks, hosts may move individually resulting in different connectivity patterns. Forwarding Interests based on outdated information may result in unnecessary transmissions. Therefore, overheard content information may only be valid within a certain lifetime. This lifetime may be short when receiving an individual object but can be increased with every received ContentObject.

3.1.4 Entry Name and FIB Size

In order to ensure efficient forwarding, the FIB size of dynamic added name-prefix entries should be limited. Otherwise, it may grow indefinitely. There are two strategies to fulfil this task:

1. Reduce the prefix length to receive a more general prefix

2. Limit the number of FIB entries of the FIB for prefixes which are added by overhearing

The procedure for reducing the prefix length works as follows:
Let us assume that the name-prefix *ccnx:/parc.com/data/local/test* is already registered at Face 2. A new incoming ContentObject with the name-prefix *ccnx:/parc.com/data/local/test2* arrives at Face 2. Both of them have the same prefix except for the last component. Instead of creating a new forwarding entry, we update the available prefix by replacing the old name-prefix with a new one which consists the common components of the incoming and the available prefix. The newly created prefix has the name *ccnx:/parc.com/data/local*.

FIB size limitation can be done by an accounting function which is implemented in the CCNx daemon. This function should not have an influence on user-configured forwarding rules. Thus, it must still be possible to add forwarding entries manually.

3.1.5 Processing Optimization

The number of data packets that are transmitted depends not only on the mobility but also on the content size. Large content files may require more data packets than small files. It may be inefficient and time consuming to perform the FIB update for every received ContentObject. Therefore, we apply this operation only to the *first* and every *nth* received data packet. By performing a modulo operation on the segment number, no additional state information needs to be remembered. The parameter *n* can be configured at start up. For example, if a node gets many ContentObjects from a source, it means that this connection is stable, thus, the parameter *n* can be increased.

3.1.6 Permanent Forwarding of Local Requests

Even in the presence of dynamic FIB population, some prefixes to data source may not be registered, because no overheard information is received. Without any additional measures, if a local application is looking for content, the Interest will not be forwarded and cannot be found until another host transmits this data. Therefore, to enable communication, Interests from local applications need to be forwarded, e.g., via the multicast Face. These Interests will pull data from neighbouring nodes if available.

3.1.7 Impact of Overhearing to Network Behaviour

This section describes overhearing in a CCNx network while using multicast. Below, we demonstrate a sample scenario where overhearing helps to forward Interests to another in six steps.

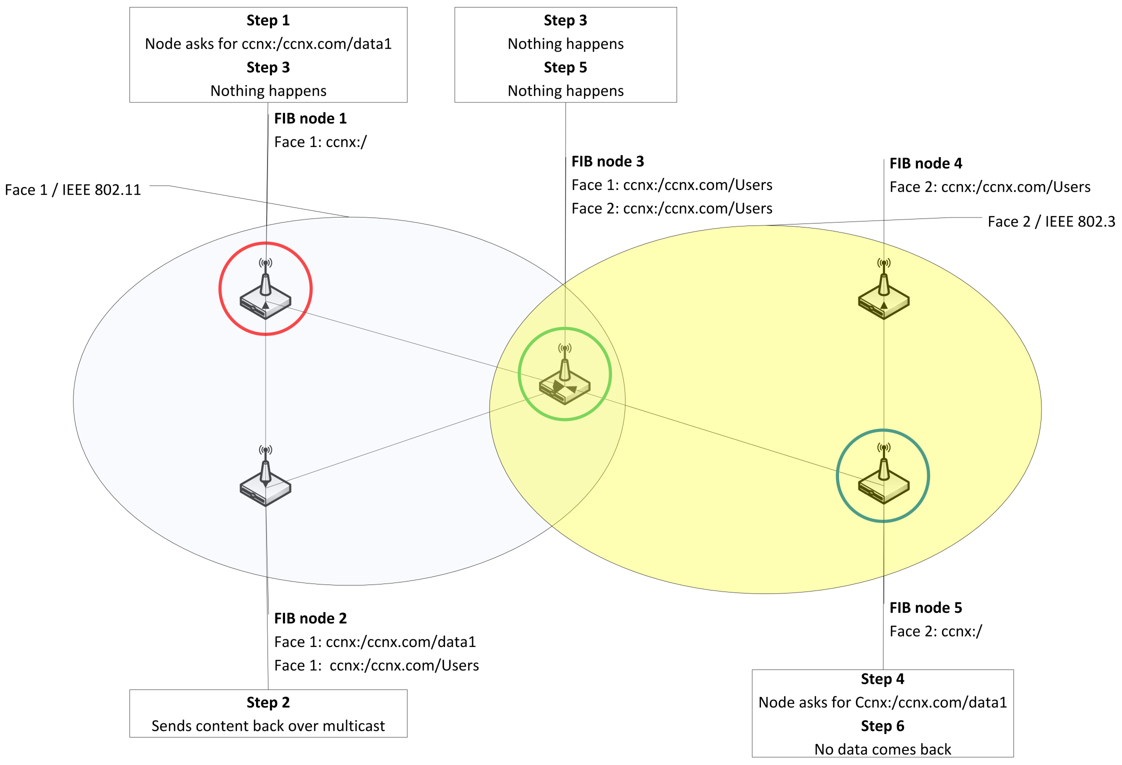
Step 1
Node asks for ccnx:/ccnx.com/data1
Step 3
Nothing happens
Step 3
Nothing happens
Step 5
Nothing happens
FIB node 1
Face 1: ccnx:/
FIB node 3
Face 1: ccnx:/ccnx.com/Users
Face 2: ccnx:/ccnx.com/Users
FIB node 4
Face 2: ccnx:/ccnx.com/Users
Face 1 / IEEE 802.11
Face 2 / IEEE 802.3
FIB node 2
Face 1: ccnx:/ccnx.com/data1
Face 1: ccnx:/ccnx.com/Users
Step 2
Sends content back over multicast
FIB node 5
Face 2: ccnx:/
Step 4
Node asks for Ccnx:/ccnx.com/data1
Step 6
No data comes back

Figure 3.1: Scenario without overhearing ContentObjects. Interest and ContentObjects are transmitted over Face 1 / IEEE 802.11a, respectively Face 2 / IEEE 802.2

Figure 3.1 illustrates content retrieval without the overhearing functionality i.e., as in original CCNx implementation would work. The figure contains five nodes. This sample network consists of two different network types. The left, blue ellipse indicates an IEEE 802.11n network and comprises nodes 1, 2 and 3. All nodes have prefixes registered for face 1. The yellow ellipse indicates an IEEE 802.3 network which is configured at nodes 3, 4, and 5 as face 2. Node 3 has an interface in both networks which serves as potential forwarder between those networks. Node 2 runs a repository with ContentObjects.

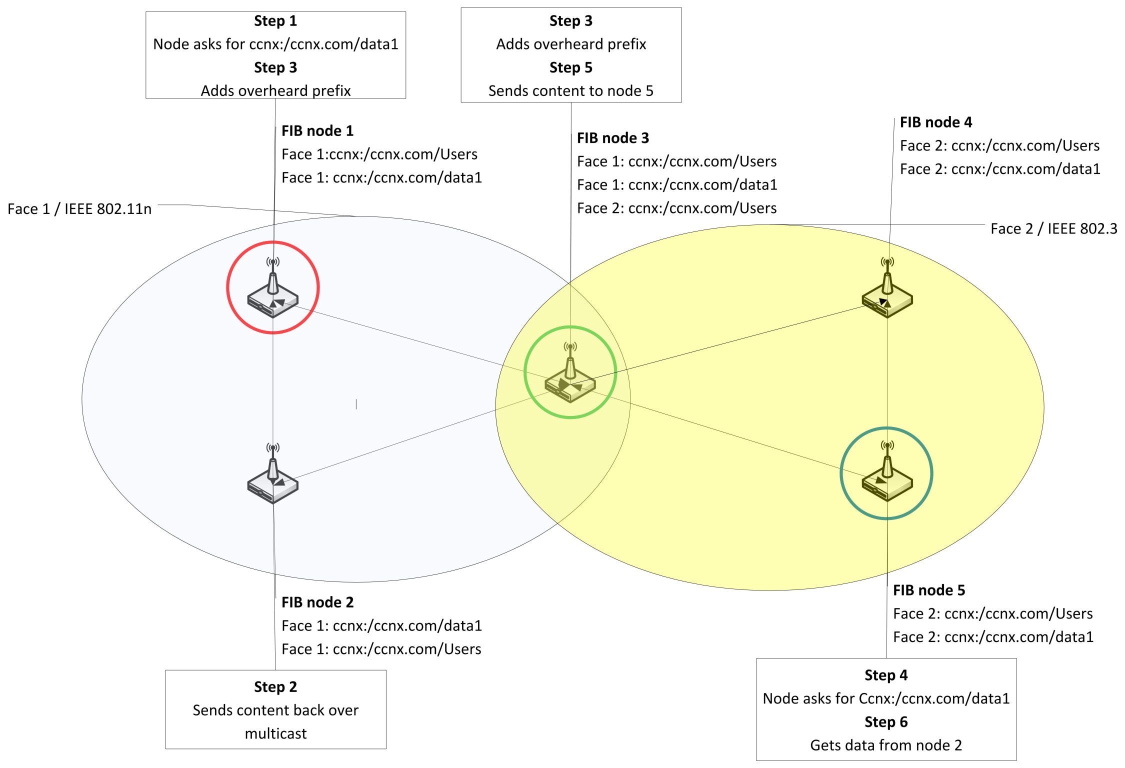
Step 1
Node asks for ccnx:/ccnx.com/data1
Step 3
Adds overheard prefix
Step 3
Adds overheard prefix
Step 5
Sends content to node 5
FIB node 1
Face 1:ccnx:/ccnx.com/Users
Face 1: ccnx:/ccnx.com/data1
FIB node 3
Face 1: ccnx:/ccnx.com/Users
Face 1: ccnx:/ccnx.com/data1
Face 2: ccnx:/ccnx.com/Users
FIB node 4
Face 2: ccnx:/ccnx.com/Users
Face 2: ccnx:/ccnx.com/data1
Face 1 / IEEE 802.11n
Face 2 / IEEE 802.3
FIB node 2
Face 1: ccnx:/ccnx.com/data1
Face 1: ccnx:/ccnx.com/Users
FIB node 5
Face 2: ccnx:/ccnx.com/Users
Face 2: ccnx:/ccnx.com/data1
Step 2
Sends content back over multicast
Step 4
Node asks for Ccnx:/ccnx.com/data1
Step 6
Gets data from node 2

Figure 3.2: Scenario with overhearing ContentObjects. Interest and ContentObjects were transmitted over IEEE 802.11a

Figure 3.2 illustrates content retrieval with enabled overhearing functionality. The topology is the same as in Figure 3.1. Furthermore, this Figure illustrates the additional functionality that is caused by overhearing. In the following example, this functionality is marked in italics.

- **Step 1:** Node 1 (red circle) asks for content `ccnx:/ccnx.com/data1` and sends Interests out over multicast Face 1. Node 2 and node 3 receives the Interests.
- **Step 2:** Because node 2 has the content for the request in his repo, it sends ContentObjects for data1 back over multicast Face 1.
- **Step 3:** Node 1 and node 3 (green circle) receive the ContentObjects. Node 3 stores the ContentObjects in the CS (content store) for a period of time. *If node 3 has enabled overhearing functionality, it adds the ContentObjects name-prefix ccnx:/ccnx.com/data1, to be reachable via face 1, to the FIB.*
- **Step 4:** After a while, node 4 (blue circle) asks for `ccnx:/ccnx.com/data1` which node 2 has in its repository. It sends out the Interests over multicast face 2 which node 3 and node 4 receive.
- **Step 5:** Meanwhile, node 3 may stored other ContentObjects in its CS. Thus, probably not every ContentObject from data1 will be in its CS or data becomes stale. That means

node 3 can not answer directly to nodes 5 request. *If overhearing is enabled, node 3 has the forwarding entry for this Interest in its FIB. Thus, it can forward the Interest into the blue network, otherwise, with disabled overhearing, it discards the incoming Interest and Node 3 gets no data.*

- **Step 6:** As mentioned in step 5, if node 3 has overhearing functionality enabled it forwards the incoming Interest of node 4 to the blue network. In this network, node 2 receives the Interest and forwards the requested data over multicast face 1. This data receives node 3 which finally forwards it to the yellow network where node 4 receives the requested data. *If overhearing is enabled, node 4 and node 5 do add the ContentObject name-prefix to its FIB.*

3.1.8 Implementation Layer

The whole processing should happen directly in the CCNx daemon. With that approach, we can simply extend the existing code by the overhearing capability. The overhearing functionality can be implemented at the same place where ContentObjects are already processed. The same applies for the processing of Interest messages from local applications. By that, it is possible to perform all the message processing at the same place exploiting already applied processing steps.

3.1.9 Conclusion

Enabling dynamic FIB entries require overhearing of ContentObjects. Multiple aspects need to be addressed such as the processing of incoming overheard ContentObjects and the forwarding of local Interests to support communication in case of lacking content transmissions. Additional properties are required to limit the processing and storage overhead. This includes the length of the name-prefix, the maximum FIB size, the entry lifetime and granularity of processing received ContentObjects.

3.2 Implementation

Since the design will be implemented on resource constrained mesh nodes, all functions of this thesis were written in C. For implementing the overhearing functions, the CCN daemon had to be supplemented by additional code.

This chapter gives an overview of the overhearing implementation. The existing functions `process_incoming_content`, to support overhearing, and `process_incoming_interest`, to guarantee the forwarding of Interests from local applications, were extended with a few lines of codes. All important functionality is included within these two functions. The implementations are explained by work flows. We refer to the steps in the flow chart by the corresponding numbers in brackets. We give an overview over the main components but more implementation details can be found in appendix 7.3. If a new function occurs in the explanation of the overhearing functionality, a reference guides to the section where it is explained.

3.2.1 Overhearing ContentObject

The CCN daemon (see subsection 2.2.1) provides a function for processing incoming ContentObjects called `process_incoming_content`. This function processes incoming data and encodes it to a ContentObject. Therefore, we implement the overhearing functionality at the same place. Thus, no additional message parsing is necessary and the ContentObject library can be reused. Figure 3.3 shows the overhearing functionality of `process_incoming_content` as a flow chart. In the following, the steps will be explained.

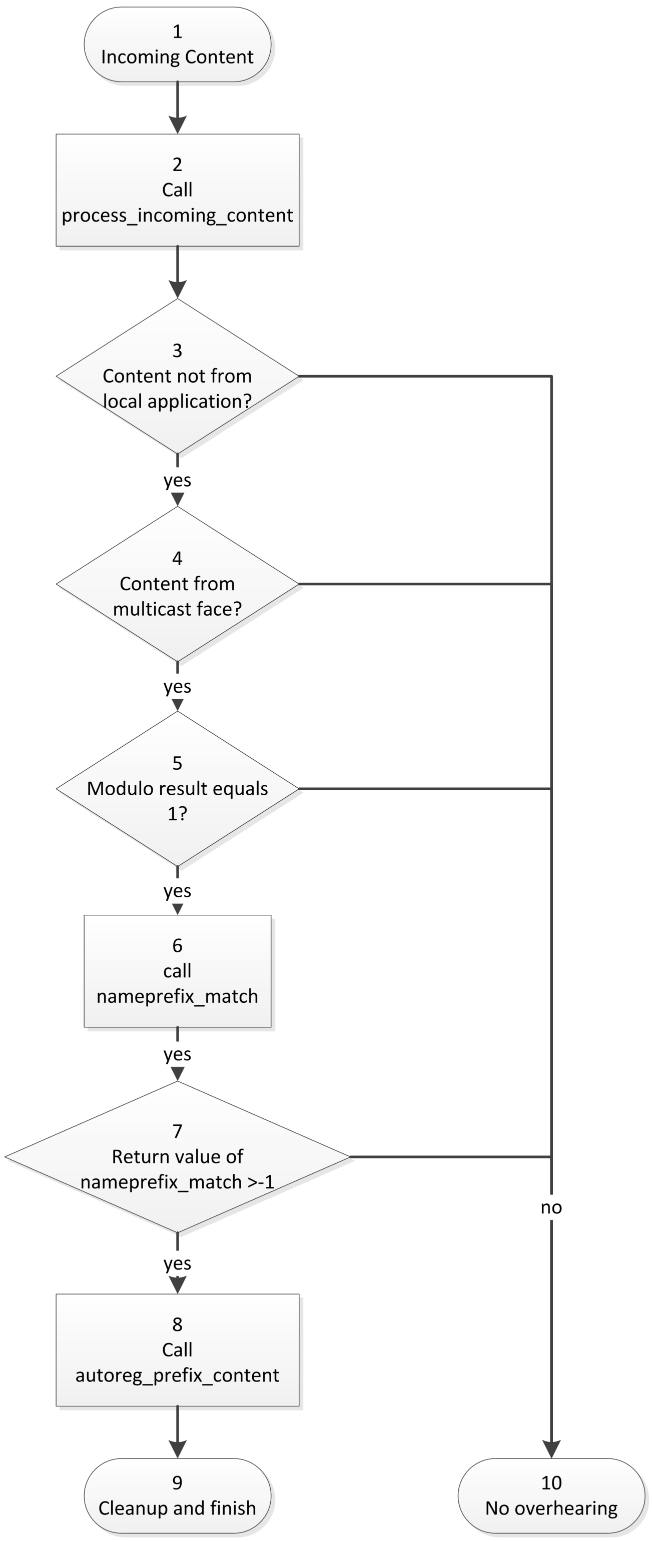

Figure 3.3: Flow chart of overhearing process from an incoming ContentObject in process_incoming_content

```
1 if((face->flags & CCN_FACE_LOCAL) == 0 && (face->flags &
     CCN_FACE_MCAST) > 0 && ((get_segment_nr(content))%
     CONTENT_ADD_FREQUENCY)==1)
2        {
3                struct register_data *reg_info= malloc(sizeof(*
                     reg_info));
4                if(nameprefix_match(h, reg_info, content->key, comps,
                      content->ncomps-1, face->faceid, 0)>-1){
5                        autoreg_prefix_content(h, reg_info, face->
                             faceid, content, comps);
6                }
7                free(reg_info);
8 }
```

Listing 3.1: overharing implementation in process_incoming_content

When a ContentObject arrives at a CCNx-node (step 1), the CCN daemon calls the `process_incoming_content` function (step 2). The incoming ContentObject is encoded in a message and, therefore, needs message parsing. The additional functionality developed within this thesis starts right after the parsing which allowing us to directly access all values of the ContentObject.

We perform the processing of incoming content only if certain conditions are met. Firstly, the content needs to be received from a non-local face since content from a local repository is already registered (step 3). Secondly, only multicast faces are considered (step 4), since to receive content over point-to-point links, hosts would need to transmit Interest to well-known hosts in first place. Thirdly, only the first and every nth segment is processed (step 5). This differentiation is performed within the function `get_segment_nr` (see appendix 7.7 for more information). Line number 1 in code listing 3.1 shows how the conditions are implemented. If all conditions are met, the CCN daemon creates a new `reg_info`. This struct holds the expiration and component information for registering a new name-prefix. We will use this struct in a later step. The complete struct definition can be found in appendix 7.2.3. After allocating that struct, the `nameprefix_match` function will be called (step 6). This is the main function required for the overhearing functionality. The function `nameprefix_match` checks if a forwarding entry is already registered in the FIB, processes name limitations such as prefix cut and the maximum number of FIB entries (see Section 3.1.4). Additionally, it updates the expiration time of a forwarding entry if it is already registered. More detailed information regarding the modification and implementation of `nameprefix_match` can be found in appendix 7.3.1. Depending on the return values of this function, different processing directions are selected (step 7).

- In case of -1, no overhearing needs to be performed since the name-prefix is already registered and no prefix cut is necessary (step 10).
- In case of 0, a new forwarding entry is registered using the registration data in the `reg_info` struct. In this case, the whole name-prefix of a overheard ContentObject is

registered.

- A positive return value n indicates the number of components that have to be registered in the `reg_info` struct.

Afterwards, the forwarding entry will be registered to a specific face. This will be done by calling `autoreg_prefix_content` (step 8). This function processes the information from the struct `reg_info` and registers the overheard name-prefix from an incoming ContentObject. More detailed information is available in appendix 7.3.3. After these steps, the new overheard name-prefix is registered as a new forwarding entry, enabling Interests to be transmitted over it. At last, we free allocate memory from `reg_info` (step 9).

Below we show a list of all new implemented functions and flags which must be used for overhearing ContentObjects:

- The `flags` which are required to decide if an incoming ContentObject should be processed, can found in Section 2.2.5.
- `nameprefix_match`, this function checks if a name-prefix entry already exists, performs prefix cuts and handles entry expirations. More details in appendix 7.3.1
- `get_segment_nr`, this function performs the selection of every nth segment for processing. More details in appendix 7.3.2
- `autoreg_prefix_content`, this function registers the overheard ContentObject name-prefix to the FIB. More details in appendix 7.3.3

3.2.2 Processing of Interests from Local Applications

The next task is to implement a way for adding Interests from local Applications to the FIB. In Section 3.1.6, we explained that all Interests coming from local applications must be propagated. Incoming Interests will be parsed and processed in the function `process_incoming_interest`. All received Interests have to pass this function being received either from remote hosts or local applications. This is where we implement the functionality of permanently forwarding of local Interests.

Figure 3.4 shows the flow chart for propagating all Interests from local applications. In the following explanation, each step from the flow chart is referenced by numbers in brackets.

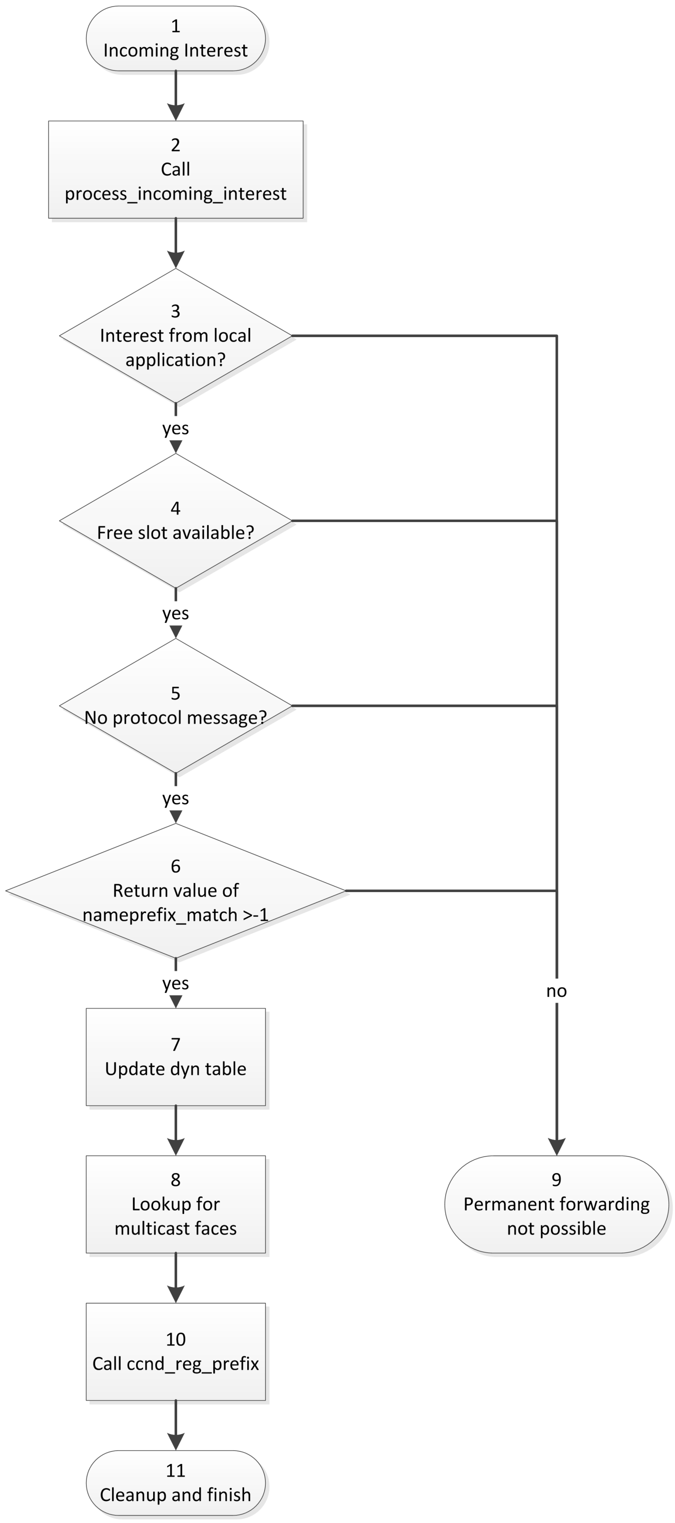

Figure 3.4: Flow chart of process overhearing of an incoming Interest

Listing 3.2 shows the new implemented part in `process_incoming_interest`. In the next paragraphs, the permanent forwarding of local Interests process will be explained by this listing and the flow chart.

```c
if(face->faceid>0 && (face->flags & CCN_FACE_LOCAL)!=0 && check_free_slots(h)>-1 && has_undesired_markers(msg, comps, pi->prefix_comps) > -1 &&  nameprefix_match(h, NULL, msg, comps, pi->prefix_comps, face->faceid, 1) >-1){
                struct face *test_face;
                update_dyntable(h, msg, comps, pi->prefix_comps);
                for (hashtb_start(h->faces_by_fd, e); e->data != NULL; hashtb_next(e)){
                        test_face = e->data;
                        if(test_face->flags & CCN_FACE_MCAST){
                                res = ccnd_reg_prefix(h, msg, comps, pi->prefix_comps, test_face->faceid, 0x3, SET_PREFIX_EXPIRATION);
                        }
                }
                hashtb_end(e);
                if(res<0){
                        ccnd_msg(h, "error could not registration prefix", res);
                }
        }
```

Listing 3.2: forwarding handling in process_incoming_interest

Upon the arrival of a received Interest (step 1), the CCN daemon calls `process_incoming_interest` (step 2). To permanently forwarding of local Interests, `process_incoming_interest` had to be extended by the code lines of listing 3.2. A FIB entry is automatically created if certain conditions are met. Firstly, we need to determine if an Interest comes from local application or from another host (step 3). This condition can be checked with the flag CCN_FACE_LOCAL as shown in the code listing 3.2. If this flag is set, the Interest comes from a local application and should be forwarded on the multicast face, if no other face is specified. Secondly, we have to check the function `check_free_slots` (step 4). This function limits the number of dynamically added forwarding entries to the FIB as explained in subsection 3.1.4. More detailed implementation descriptions of this function can be found in appendix 7.3.4. The maximum number of forwarding entries that can be added to the FIB based on overhearing is stored in a separate data structure called `dyn_add`. If the maximum number is reached, the overhearing functionality is disabled until there is new space in the `dyn_add` table, e.g. if an overheard forwarding entry is expired. Details about this struct can be found in appendix 7.2.2.

Additional to regular Interests for content, there are internal Interests that do not request content but are used for control and command purposes such as face creation or storing content to the local repository. A detailed list of used control messages can be found at the

CCNx documentation page[7]. The function `has_undesired_markers` checks for such Interests and returns a negative integer if this is the case. Such Interests must not be registered dynamically and are ignored (step 5). Implementation details of this function can be found in appendix 7.4.5)

Then, the Interest prefix is further processed in the `nameprefix_match` function which was already introduced in Section 3.2.1 (step 6). Called with an Interest, `nameprefix_match` has two possible return values:

- Zero, if no forwarding entry is available and the Interest name-prefix must be registered.
- Minus one, if a forwarding entry is already registered and no additional actions are required. (step 9).

As shown by the code listing 3.2, the if clause is not entered if `nameprefix_match` returns a negative value. If all conditions are met, the register process takes place. First, the `dyntable` is updated over the function `update_dyn_table` (step 7). The `dyntable` table contains all name-prefixes which were registered by overhearing and allows a faster access to the FIB. More detailed information about this struct is available in appendix 7.2.1. The function `update_dyn_table` adds the name-prefix information of the incoming Interest to the `dyntable`. A detailed explanation of this function is available in appendix 7.3.5. After updating the table, the incoming Interest will be registered on each available multicast face (step 8). A prefix cut is not necessary for registering an Interest name-prefix, because the entry will be added only for the lifetime of an Interest. If no data comes back, the entry expires. If data comes back, it will be processed as a ContentObject and the created FIB entry will be updated as described in Section 3.2.1. The Interest name-prefix is registered by the `ccnd_reg_prefix` function which is already provided by CCNx (step 10).

Below is a list of all new implemented functions and flags which are used for permanently forwarding Interests from local applications:

- `nameprefix_match` checks for available forwarding entry and updates expiration time of FIB entries. For more information see appendix 7.3.1.
- `update_dyntable` saves the overheard name-prefix in the dyntable. For more information see appendix 7.3.5.
- `check_free_slots`, checks if it is possible to add one more name-prefix by overhearing. See details in appendix 7.3.4.
- `has_undesired_markers` checks if the incoming Interest is a protocol message from a local application. For more information see appendix 7.4.5.
- Flags: `CCN_FACE_LOCAL`, `CCN_FACE_MCAST`, these flags are used to indicate local or multicast faces. Section 2.2.5 explains it in detail.
- Values: `SET_PREFIX_EXPIRATION` which is used in `nameprefix_match`. For more information see appendix 7.1.

3.2.3 Summary

There are many factors which must be considered during the implementation of the overhearing capability. The implementation is divided into two parts. The first one is the handling of incoming ContentObjects, the second one is the handling of outgoing internal Interests. When overhearing ContentObjects, we have to ensure that only content from the multicast face is registered. Content from local repositories is registered independently by local registration operations. Overhearing of ContentObjects enables hosts to learn available content sources. Furthermore, the overhearing mechanism provides an option, to add selected ContentObjects by using the modulo function of the segment number of an ContentObject. Additionally, a prefix shortener has been implemented which compares existing prefixes with new incoming prefixes for common components. After this process optimisation part, we implemented a function which does register the new prefix entry to the FIB and setting additionally an expiration time. The expiration time will be automatically updated and increased, if the same data appears multiple time in the network. With this approach, we can ensure that ContentObject prefixes which are more frequently used stay longer in the FIB. Also forwarding entries which are not refreshed for a certain time period will expire automatically. Furthermore, the number of forwarding entries that can be added is limited. The `dyntable` stores the information whether the CCN daemon is allowed to register another entry to the FIB.

By overhearing Interests from local applications, we ensure that it can be forwarded, even if there is no available forwarding entry. With this approach, every application can always probe the environment to check the availability of content. This is important in situations without communication activity in which no content can be overheard. Of course, this works only if a multicast face is available since we do not know where to forward the Interest otherwise. In the unicast case, we know the other host and the FIB can be configured manually. Interest prefixes from local applications will be added for a short time into the FIB if there is no matching forwarding entry available. If a ContentObject returns back, the entry will be updated, if not, the entry will expire. There are some internal Interests which should not be forwarded, such as communication messages. We implemented a function which identifies and ignores these Interests in terms of overhearing.

Chapter 4

Evaluation

The chapter is structured into three sections. In Section 4.1, the hardware node configuration for both parts is explained. In Section 4.3, the topology and evaluation results of the performance tests are shown. In Section 4.4, we describe the power evaluations. All evaluation was done based on CCNx version 0.6, which was the latest version at time of the evaluation.

4.1 Hardware and Node Configuration

The implementation was evaluated on PCEngines Alix 3d nodes [8]. The nodes were running the ADAM operating system which is described below (see 4.2). For installing CCNx with ADAM, additional changes were necessary. The detailed manual to set up an Alix-Node with CCNx can be found in appendix 6.

The configuration of CCNx, such as the definition of forwarding entries, can be automated by the configuration file ccnx.conf. These entries are registered during the start of the CCN daemon `ccnd`. To support multicast overhearing, CCNx needs a face including the multicast address. Thus, a multicast face must be configured at start up. The CCNx multicast address assigned by IANA is `224.0.23.170`. Any port can be used for multicast except 9695 which is the unicast port. UDP is used as transport protocol since there is no need to build up sessions to specific hosts. At start-up, the ccnd.conf must contain the following command to add the multicast face:

```
add ccnx:/ccnx.org/Users udp 224.0.23.170 59695
```

The prefix `ccnx:/ccnx.org/Users` is required to find the keys of potential communication partners. Currently, the keys of all CCNx users are stored under the same prefix. Furthermore, it is required to register at least one multicast face in order to listen to traffic from that face.

4.2 Adam

ADAM[6] is an acronym for Administration and Deployment of Adhoc Mesh networks. Costly on-site node repairs in wireless mesh networks (WMNs) can be required due to misconfiguration, corrupt software updates, or unavailability during updates. ADAM is a management framework that guarantees accessibility of individual nodes in these situations. ADAM uses a decentralised distribution mechanism and self-healing mechanisms for safe configuration and software updates. In order to implement the ADAM management and self-healing mechanisms, an easy-to-learn and extendable build system for a small footprint embedded Linux distribution for WMNs has been developed. In this thesis, the self-healing and management function was not used.

4.3 Performance Evaluation

In this section, the topology and the evaluation results of the performance evaluation such as throughput and overhead are presented. Firstly in subsection 4.3.1, we introduce the topology. Afterwards we describe three performance test scenarios in subsections 4.3.3, 4.3.4 and 4.3.5. First, we want to evaluate the most appropriate parameters for the IEEE 802.11a unicast and multicast transmission on Mesh-Nodes (Alix 3d nodes). This is done in subsections 4.3.3 and 4.3.4. Afterwards, when the parameters are known, we evaluate the overhead between the original CCNx version 0.6 and the thesis implementation using CCNx version 0.6. This is shown in subsection 4.3.5.

4.3.1 Topology

The test topology was built with three Alix nodes and a workstation. The used devices are listed below.

- three Alix nodes running with CCNx version 0.6
- Linksys WRT54G switch
- Control Station

Figure 4.1 illustrates the topology structure. Furthermore it shows the assigned frequencies and the used IP address configuration. The used network interfaces are listed in table 4.1. For the evaluation RTS/CTS was disabled.

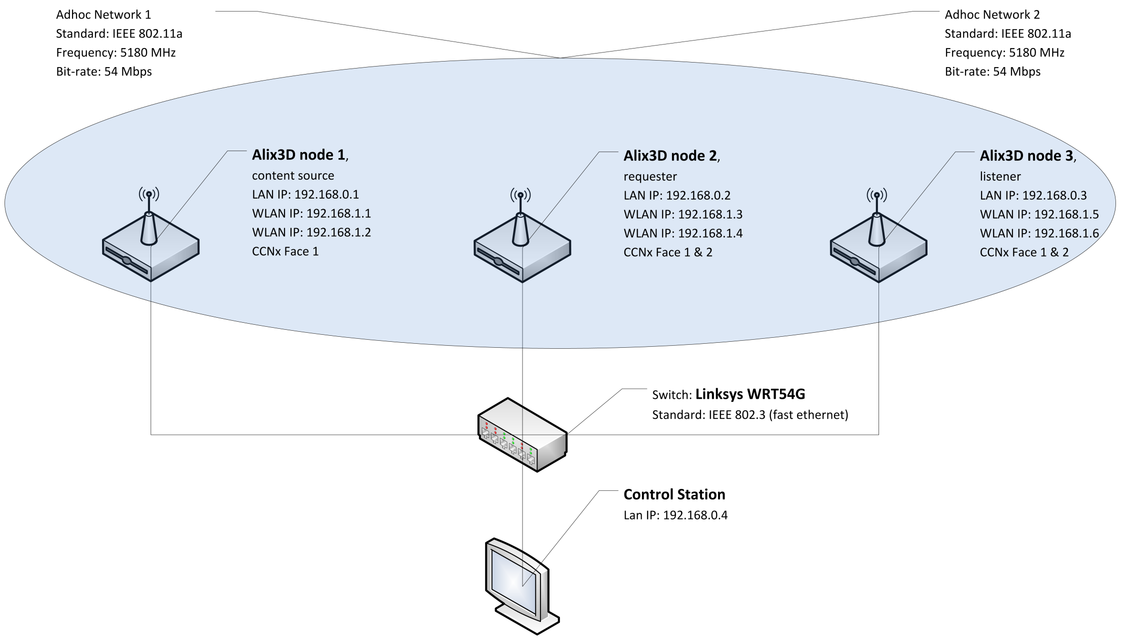

Figure 4.1: Testbed topology, performance tests

Network name	Standard
Adhoc network 1	IEEE 802.11a
Adhoc network 2	IEEE 802.11a
Fast Ethernet network 3	IEEE 802.3

Table 4.1: Network interfaces, test environment

4.3.2 Evaluation Results

The main focus was set on the following points:

- Comparing the throughput of Alix nodes with a high-performance workstation and the results of the initial CCNx experiments performed by Van Jacobson et. al. [1].
- Finding optimal transfer parameters such as pipeline size, segment size or Interest lifetime to increase wireless throughput.
- Measuring the processing overhead of the working overhearing functionality compared to the original CCNx version 0.6.

To evaluate the throughput, we used the function `ccncat` which is an available CCNx file transfer application. There were two different network technologies used, IEEE 802.3 which is used in the Van Jacobson et. al. paper and IEEE 802.11a. In order to automate the evaluations, a bash script was written. It is available in appendix 7.5.

All illustrated figures in this section use whisker bars which display minimum, 0.25 quantile, median, 0.75 quantile, and maximum.

4.3.3 Wired Unicast Transmission

The first scenario measures the IEEE 802.3 unicast throughput of an Alix-Node and a workstation. The test parameters are shown in table 4.2. The hardware specifications of the workstation are shown in table 4.3 and the specifications of the Alix-Node are shown in appendix 6 listed in table 6.1. The intent of this test was to evaluate how fast Alix nodes can process CCNx traffic. We tested the throughput for file sizes of 1MB, 2MB, 5MB, and 10MB, each using a segment size of 4096 bytes. We compared the gathered information with results from Van Jacobson et. al.[1]

Test case	Standard	Pipeline size	Segmentsize	Filesize	Test runs
unicast wired	IEEE 802.3	16	4096 bytes	1MB - 10MB	100

Table 4.2: IEEE 802.3 evaluation settings, for each file and segment size combination 100 test runs were done. The pipeline size was set to 16 according to the Van Jacobson et. al.[1] paper.

OS	CPU	RAM	Motherboard
Kubuntu 11.10	Intel i5-2400 (3.10 GHz)	8096MB DDR3	Asus P8Z68-VLX

Table 4.3: Hardware specification of the used workstation.

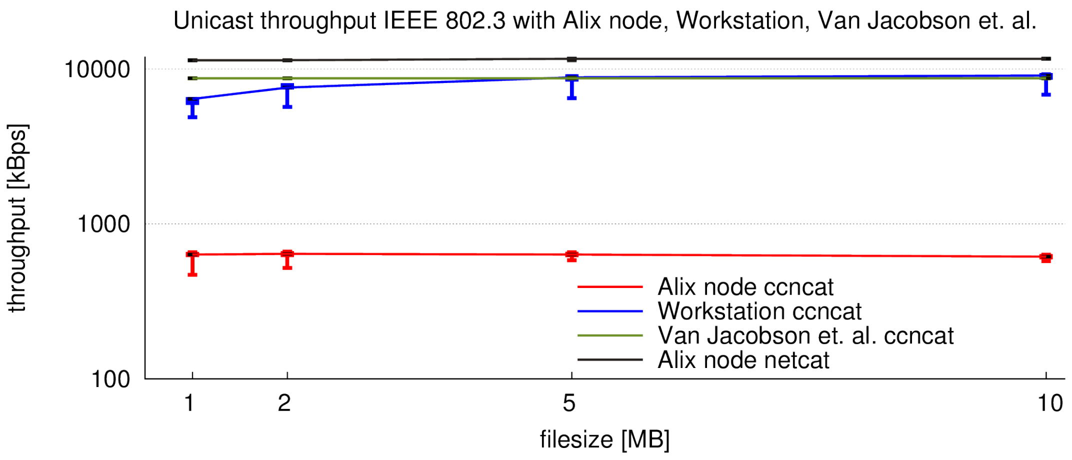

Figure 4.2: Throughput of CCNx version 0.6 for Alix node and workstation compared with the results from Van Jacobson et. al. paper. Van Jacobson et. al. tested only for transmissions of 6MB files.

The test results are shown in Figure 4.2. The y-axis shows the throughput in kilobytes per seconds (kBps) on a logarithmic scale. The x-axis displays the selected file size. As one can see, the throughput from Alix node to Alix node is at around 600 kBps with segment size of 4096 bytes. Van Jacobson et. al.[1] measured a throughput of around 8700 kBps. Thus, the throughput is lower by factor 14.5. The throughput of the workstation is for small file sizes around 6400 kBps and for bigger files approximates the throughput of about 8600 kBps. Thus, the results of the workstation are similar to results of Van Jacobson et. al.[1]. The Alix nodes are much slower because the processing power of the embedded processor CPU is limited and therefore, content processing and signature verification take more time. As comparison, we also measured the throughput of an Alix node for a TCP connection using netcat [9] instead of CCNx. The average throughput over all file sizes is around 11500 kBps. This illustrates the large processing overhead of CCNx. Mesh nodes with higher processing power or hardware-based signature verification may lead to higher CCNx throughputs.

4.3.4 Wireless Transmission with Varying Parameters

After learning the capabilities of the hardware, we observed the throughput of wireless communication. In this second test scenario, we measured the impact of increasing the pipeline size in unicast and multicast IEEE 802.11a transmissions. The selected parameters are listed in table 4.4. In the CCNx framework, the maximum pipeline size is limited to 16. Pipeline sizes larger than 16 do not result in any performance gains in IEEE 802.3 transmissions (see Van Jacobson et. al. [1]). For the IEEE 802.11a throughput measurements, we disabled this limitation and allowed larger pipeline sizes. The problem, especially with multicast IEEE 802.11a, is that with a pipeline size of 16 and the default parameters, the throughput reaches only values of around 45kBps. One reason are the additional delays for collision avoidance before transmitting Interests or sending ConentObjects back. To synchronize with multiple recipients, these delays are higher than in unicast. Therefore, it needs more time until a node receives an answer from an Interest which results in a bandwidth reduction. Van Jacobson et. al. [1] tested the throughput for IEEE 802.3. Thus, the used pipeline size may not be appropriate for IEEE 802.11a transmission.

Test case	**Standard**	**Pipeline size**	**Block size**	**File size**	**Test runs**
unicast wireless	IEEE 802.11a	4 - 1024	1024 - 4096 bytes	2MB	100
multicast wireless	IEEE 802.11a	4 - 16384	1024 - 4096 bytes	2MB	100

Table 4.4: IEEE 802.11a test case: figure out best pipeline size

In this test scenario, we transferred three test files of 2MB with segment size of 1024, 2048, 4096 bytes from one Alix node to another. The maximum segment size on the wireless interface was 2274 bytes. This means segment sizes of 2048 and 4096 bytes result in packet segmentations of the underlying IP layer. For multicast transmissions, we measured the throughput for pipeline sizes from 4 to 16384 and in the unicast scenario from 4 to 1024 pipelines. Each file and pipeline size combination was evaluated by 100 runs. In this scenario, we used the original CCNx version 0.6 framework.

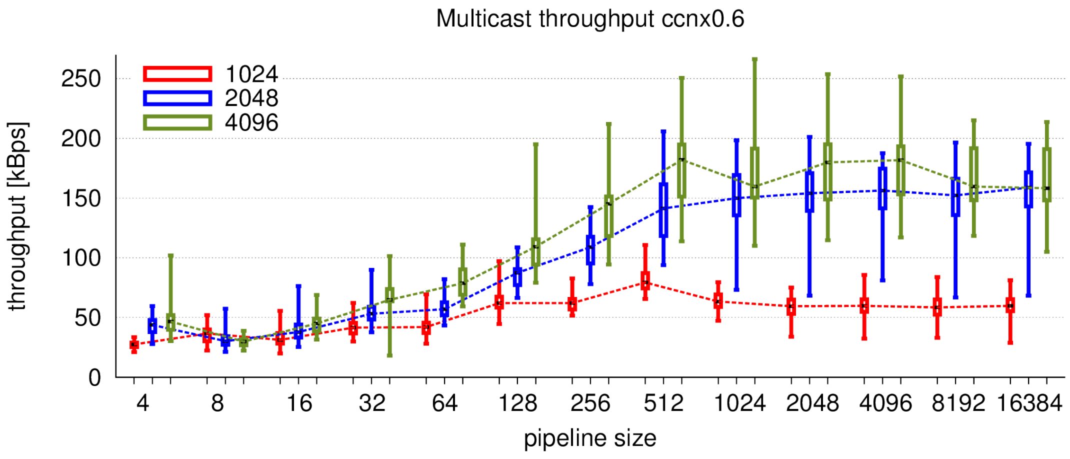

Figure 4.3: Throughput of 802.11a multicast ccnx0.6 with different pipeline sizes

Figure 4.3 illustrates the impact to multicast transmission of an Alix node communicating via 802.11a, when increasing the pipeline size. The y-axis shows the throughput in kBps. The x-axis shows the used pipeline size. The green line has a segment size of 4096 bytes, the blue one a segment size of 2048 bytes, and the red one a segment size of 1024 bytes. Bigger segment sizes, e.g., 4096 bytes, are more efficient for file transmissions than smaller e.g., 1024 bytes. This is because the overhead of the CCNx messages, including name-prefixes and signatures, is smaller for larger segment sizes and fewer packets are required. This results also in a lower processing overhead due to fewer signature verifications. Even if on lower layers, the packet must be fragmented, bigger segment sizes are better and result in higher throughput. Furthermore, Figure 4.3 illustrates, that until a pipeline size of 512 packets the throughput increases for all three segment sizes. For pipeline sizes larger than 512 we get no additional performance gain. If we compare the results with the recommended pipeline size of 16 for the IEEE 802.3 transmission, we can increase the throughput from 45kBps (16 pipelines) to 182kBps (512 pipelines) for a 2MB file.

To evaluate the disadvantages of large pipeline sizes, we measured the transmitted Interest and received ContentObjects at the requester. Figure 4.4 illustrates the transmitted Interests and received ContentObjects for 50 test runs to transfer a 5MB file with 4096 bytes segment size using different pipeline sizes. The y-axis shows the number of transmitted Interests and received ContentObjects. The x-axis shows the used pipeline size. As we can see, the number of transmitted Interests increases very fast with increasing the pipeline size. The received ContentObjects of the requester are constant up to a pipeline size of 256 but increases with pipeline sizes of 512 or more as well.

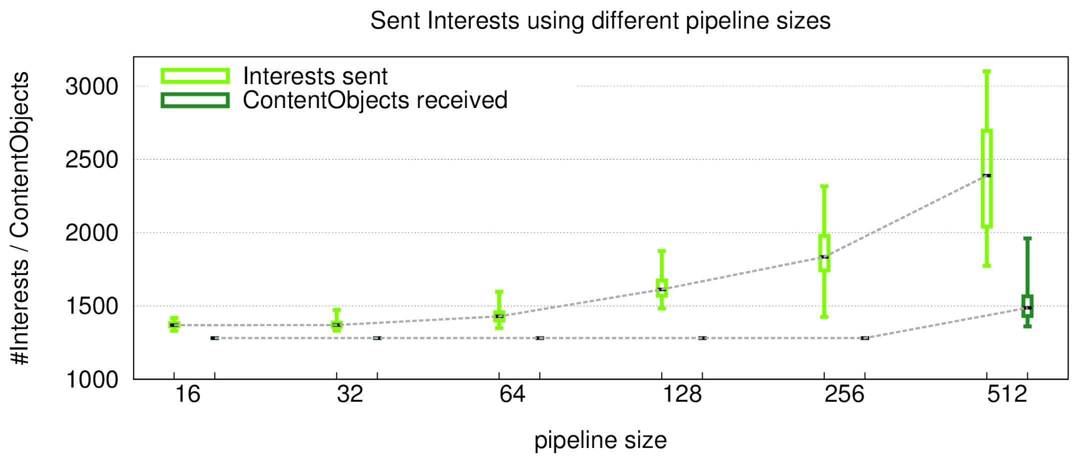

Figure 4.4: Transmitted Interests and received ContentObjects for different pipeline sizes.

Table 4.5 lists the number of transmitted Interests of a unicast and a multicast transmission using different pipeline sizes. When using the same pipeline size of 16 for both, unicast and multicast, the difference in transmitted Interests is minimal. To request a 5MB file, a multicast transmission requires approximatively 6% more Interests than a unicast transmission. This is because by using unicast, the retransmission is done on the MAC-layer without noticing by CCNx, in multicast it is done by CCNx on the application layer after an Interest timeout. Furthermore, the min-max deviation of the transmitted Interests is very small in unicast, this is another indication that the retransmission is done on the MACLayer. However, the multicast throughput that can be achieved with the same pipeline size of 16 and the same parameters is much lower than in unicast. Increasing the pipeline size to higher values increases the throughput but at the cost of more collisions and Interests retransmissions. If we compare multicast transmission with 16 pipelines to multicast transmission with 512 pipelines, the overhead of transmitted Interests is around 75% for the same file size. Although, this value is high, it still resulted in higher throughput but in scenarios with more nodes, the collision rate may increase resulting in more retransmissions and lower overall throughput.

	unicast 16	**multicast 16**	**multicast 512**
sent Interests	1295	1369	2390
relative deviation	0.10%	1.33%	15.56%
Kilo Bytes per second	606.64	44.96	182.04

Table 4.5: Number of transmitted Interests using unicast/multicast transmission, relative min-max deviation and throughput in kBps

Figure 4.5 shows the impact of increasing the pipeline size in the unicast case. In contrast to multicast, there is no performance gain by increasing the pipeline size to values larger than 16. With a pipeline size of 32 or larger, CCNx gets `seq_gap` error messages during the transmission indicating that packets were dropped. The larger the pipeline size the more often `seq_gap` problems occur and consequently, the throughput collapses. With multicast transmission, `seq_gap` never occurred in our evaluations.

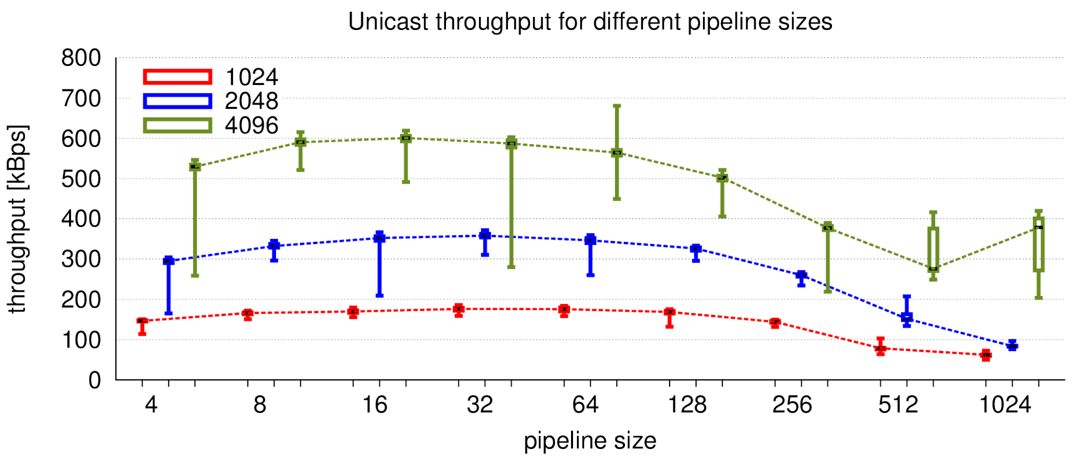

Figure 4.5: Throughput of 802.11a unicast ccnx0.6 with different pipeline sizes

As illustrated in Figure 4.3 and Figure 4.5 the min-max deviation of unicast is in most cases lower than the min-max deviation of multicast and the throughput is much higher. A reason is that IEE802.11a multicast has no MAC-acknowledgement (ACK) in contrast to unicast. Thus, a multicast transmitter does not know if the packet arrives to the requester or not. In unicast, the retransmission of collided packets is done on the MAC-layer and the higher layered CCNx does not recognize the packet as lost and does not schedule a retransmission. In CCNx, a receiver reexpresses an Interest after an Interest timeout. This retransmission is performed on the CCNLayer which is currently implemented on the application layer. Therefore, in multicast communication, the Interest retransmission takes more time compared to unicast MAC layer retransmissions. During the waiting time, no data can be transmitted resulting in longer transmission times. Additionally the backoff time of multicast is by default much longer, since it cannot be adapted based on collisions as with unicast.

Therefore, multicast transmission results in very low throughput. Increasing the pipeline to larger values increases the throughput but at the cost of large transmission overheads and collisions. Another strategy is to reduce the CCN delay until retransmitting an Interest.

Every Interest has an Interest lifetime, which defines the time an Interest is valid in the PIT. If the lifetime expires, the Interest can be re expressed. The Interest timeout has therefore an impact on the retransmission of Interests. In CCNx, the Interest lifetime is set by default to 4 seconds. In unicast transmission where the retransmission is done on the MAC-layer, 4 seconds may be a good choice. However, multicast transmissions are slowing down due to this large retransmission delay. After each collision, it gets a huge delay. Therefore, we evaluate different Interest timeouts lower than 4 seconds.

Figure 4.6 shows the behaviour of CCNx by adapting the Interest lifetime. The x-axis denotes the throughput in kBps. The y-axis shows the pipeline size. We evaluate a pipelined transfer of a 2MB file with four different Interest lifetimes of 4.0, 1.0, 0.5, and 0.25 seconds indicated by the four coloured lines. When using Interest lifetimes of 1 second or less, pipeline sizes larger than 16 result in no performance gain. This seems to be the limit for efficient CCN transmission as already observed for wired communication in Van Jacobson et. al. [1]. The black line of Figure 4.6 shows the throughput when using the default Interest lifetime of 4 seconds. Here, in contrast to the other lines, the throughput increases by using larger pipeline sizes than 16. The reason for this, is the higher number of Interests that can be transmitted until a timeout has been detected. These concurrent Interests may retrieve content that is then stored in the content store of the requester. Not all of them will result in a collision. After the timeout is detected, only the Interest that timed out needs to be retransmitted and subsequent Interests can then be satisfied from pre-fetched content in the requester's content store. This results in considerably higher throughput however at the expense of large transmission overheads. The performance gain due to decreasing the Interest lifetime is approximatively 79% or from 182.04 kBps with pipeline size of 512 and 4 seconds lifetime to 325.6kBps with pipeline size of 16 and 0.25s lifetime.

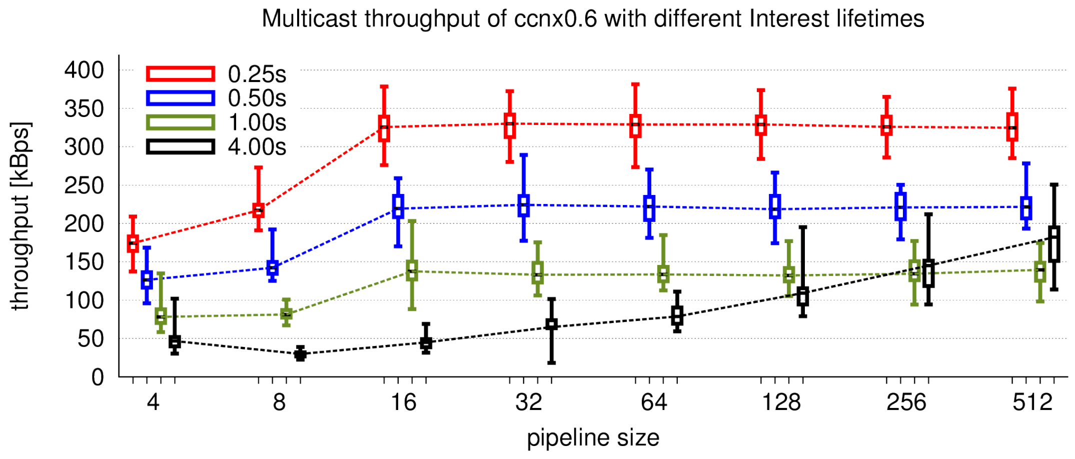

Figure 4.6: Throughput of a 2MB file, with different Interest lifetimes.

Our results showed that a higher throughput can be achieved for Interest lifetimes smaller than 1 seconds and a pipeline size of 16. Therefore, we evaluate the transmission overhead of all four Interest lifetimes using a pipeline size of 16 in the following.

Figure 4.7 illustrates the number of transmitted Interest and received ContentObjects and as well as the number of duplicated ContentObjects for different Interest lifetimes. The x-axis denotes the Interest lifetime in seconds. The left y-axis indicates the number of transmitted Interests and received ContentObjects. The right y-axis shows the number of received duplicated ContentObjects. If there are too many duplicated ContentObjects, the chosen Interest lifetime is too short since the CCN daemon will reexpress the Interest before the corresponding

ContentObject can be received. There are no duplicated ContentObjects for Interest lifetimes longer or equal to 0.5 seconds. When using Interest lifetimes of 0.25 seconds, we receive at most 30 duplicated ContentObjects. This value was only reached once in our evaluations and corresponds to an overhead of 2.34%. Therefore, in the remainder of this work, we use an Interest lifetime of 0.25 seconds since it results in a higher throughput of 48%, i.e. from 219.27kBps to 325.6kBps, compared to an Interest lifetime of 0.5 seconds. This is still about 45% lower than unicast throughput but already two simultaneous requesters will result in faster transmissions. Future implementations may use an adaptive mechanism to control the Interest lifetime based on the number of duplicated ContentObjects, number of transmitted Interests, and the throughput.

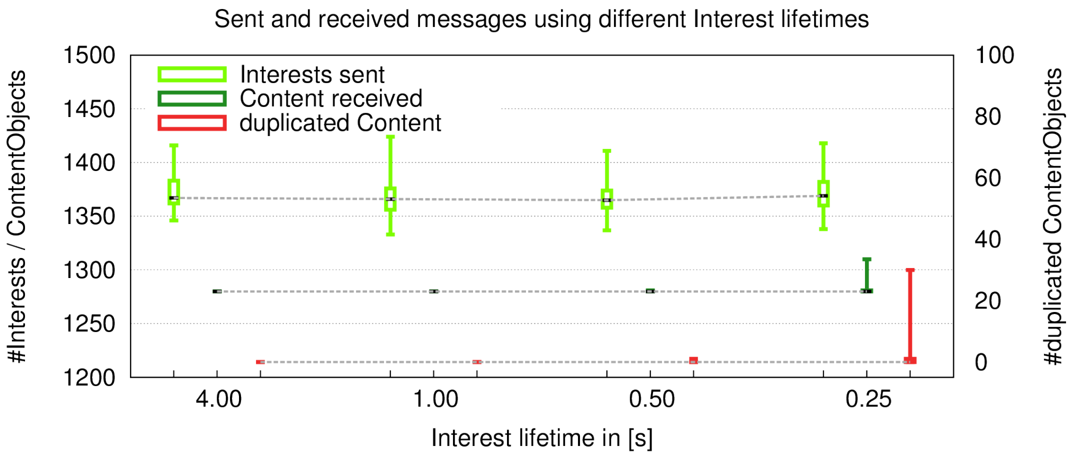

Figure 4.7

4.3.5 Overhead of Implementation

In this subsection, we evaluate the overhead of the working overhearing functionality compared to the original implementation. We set the pipeline size to a fixed value and compare the additional time the implementation needs for the transmission of different file sizes.

Test case	Interest lifetime	Pipeline size	Blocksize	Filesize	Test runs
multicast wireless	0.25s	16	4096 bytes	0.25MB - 10MB	50

Table 4.6: IEEE 802.11a test case: measuring the difference in throughput of original CCNx and thesis implementation with overhearing functionality

Table 4.6 shows the configuration of the test scenario. In this scenario, we used file sizes of 0.25MB, 0.5MB, 1MB, 2MB, 5MB, and 10MB each with a segment size of 4096 bytes. In subsection 3.1.5, we described a process optimization by using the modulo function. In this test scenario we processed every 20th ContentObject. We choose this low value as performance baseline. For higher modulo values, the overhearing overhead obviously becomes smaller.

Additionally, the maximum number of dynamically added forwarding entries was set to 10. However, during the tests the maximum number was never reached, because we transmitted only one file. The intent of this test was to investigate the processing overhead of the overhearing functionality by comparing the throughputs which decrease with increasing processing efforts.

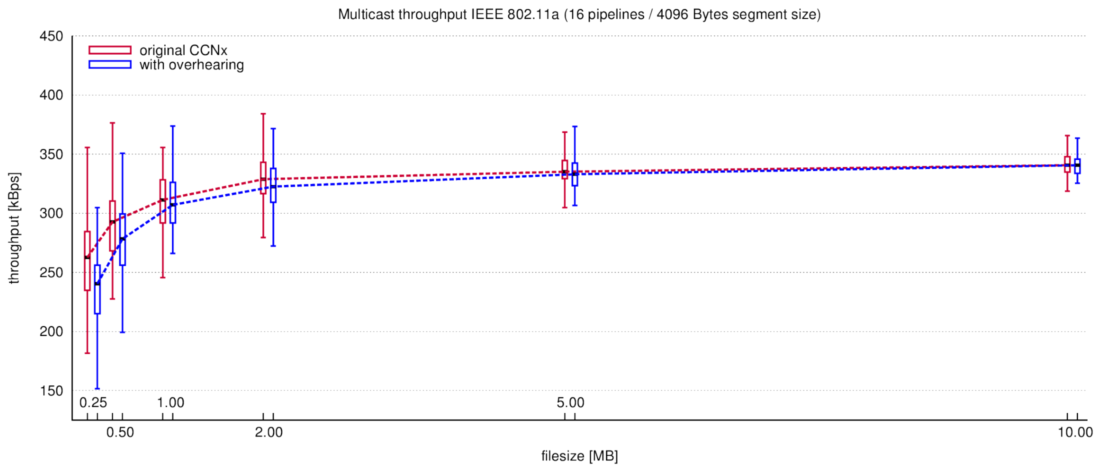

Figure 4.8: CCNx with enabled overhearing versus CCNx with disabled overhearing

Figure 4.8 shows the results when comparing the original CCNx implementation with the thesis implementation comprising the overhearing functionality. The transmission was performed by multicast communication. The y-axis shows the throughput in kBps, the x-axis shows the different file sizes. As expected, the throughput is lower with applied overhearing functionality compared to the original implementation due to a higher processing overhead. Table 4.7 shows the difference in percent. As one can see CCNx 0.6 with overhearing functionality is between 0.02% (10MB file size) and 9.23% (0.25MB file size) slower.

	4096 bytes
0.25MB	8.45%
0.5MB	4.89%
1.0MB	1.35%
2.0MB	1.89%
5.0MB	0.62%
10.0MB	0.02%

Table 4.7: Throughput overhead of CCNx 0.6 with enabled overhearing compared to CCNx 0.6 with no overhearing functionality for different file sizes with 4096 bytes segment size

4.4 Power Evaluation

In this section, we introduce the test topology and discuss the evaluation results of the power evaluation for an Alix node in different roles, i.e. content source, receiver, or listener. Firstly, in subsection 4.4.1, we introduce the topology. Secondly, in subsection 4.4.2, the evaluation results are discussed.

4.4.1 Power Evaluation Topology

Figure 4.9 illustrates the topology structure including the assigned frequencies and the used IP address configuration similar to Section 4.3. The power of the Alix nodes was measured using the Rigol digital multimeter [13]. The settings for the Rigol digital multimeter are listed in table 4.8.

Measure	Direct Current Intensity (DCI)
Range	10 Ampere
Sampling rate	100Sa/s
File size	5MB
Test runs	50

Table 4.8: Test settings for Rigol digital multimeter

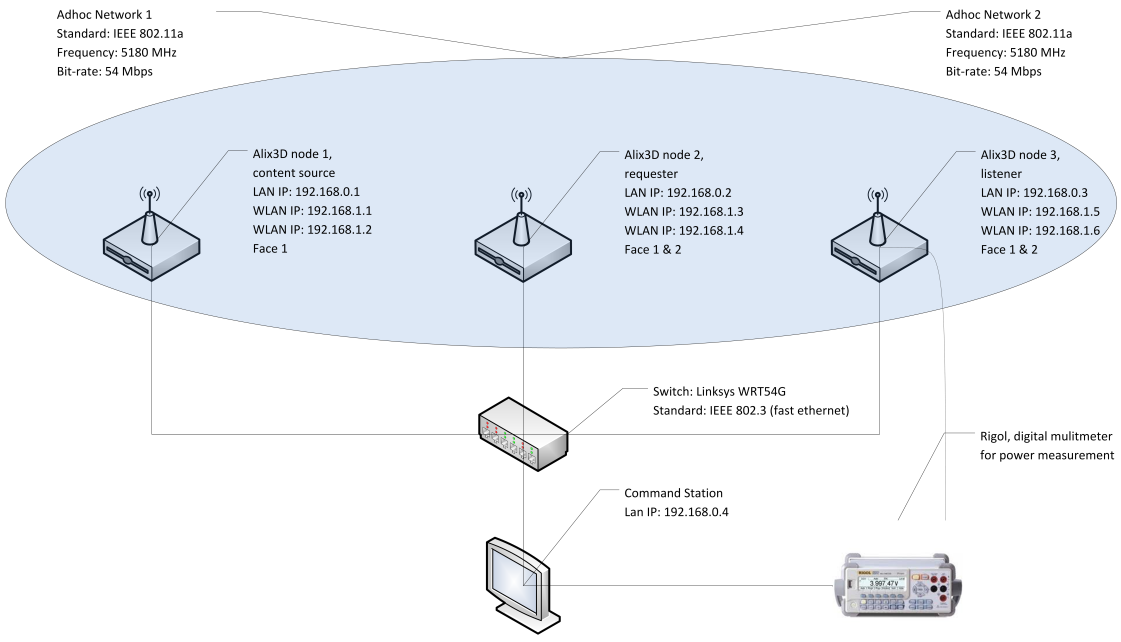

Figure 4.9: Test bed topology, power tests

4.4.2 Evaluation Results

In this subsection, we measure the power consumption of an Ali node running with CCNx using multicast communication. We compare the unmodified CCNx implementation with the implementation which has the additional overhearing functionality. Three node roles were defined:

1. **content source:** the power consumption in Watt to transmit data from the local repository is measured.

2. **requester:** the power consumption in Watt to request and receive data from a node is measured.

3. **listener:** the power consumption in Watt to overhear data from the environment without requesting it and if the overhearing functionality is enabled, adding the name-prefix entry as a new forwarding entry to the FIB.

For comparison, the power consumption of an Ali node with a running CCNx which does not receive or transmit any data was measured. This is called the idle mode.

Figure 4.10 shows the power measurements for every node role. The x-axis shows the four different modes, each of it for the original CCNx version 0.6 (red) and for the overhearing functionality implemented in the scope of this thesis (blue). The y-axis shows the power consumption in Watt. The black intervals on the top of each bar denote the standard deviation. In idle mode, both versions (original CCNx and CCNx with thesis implementation) have an equal power consumption. The content source has the highest energy consumption since it needs to fetch the already signed ContentObjects from the repository and transmits it over the wireless medium. The power consumption of the requester is only slightly lower, because it needs to transmit Interests and verifies the signatures of the received ContentObjects. A listener has a lower power consumption, because no Interests are transmitted and no signatures need to be verified. However, the listener still needs to process and parses all received content names.

Table 4.9 shows the median power consumption in Watt for the node roles. The power consumption of the original CCNx implementation was set as reference to 100%. The overhead for a requester is on average only 1.27% and increases for a passive listener to about 4.58%. The higher overhead of the passive listener compared to the requester is due to the new implemented overhearing mechanism. The functions which register the overheard data need processing capacities and this leads to higher energy consumption in contrast to the case without overhearing in which the listener only put the incoming ContentObjects temporary into the local content store.

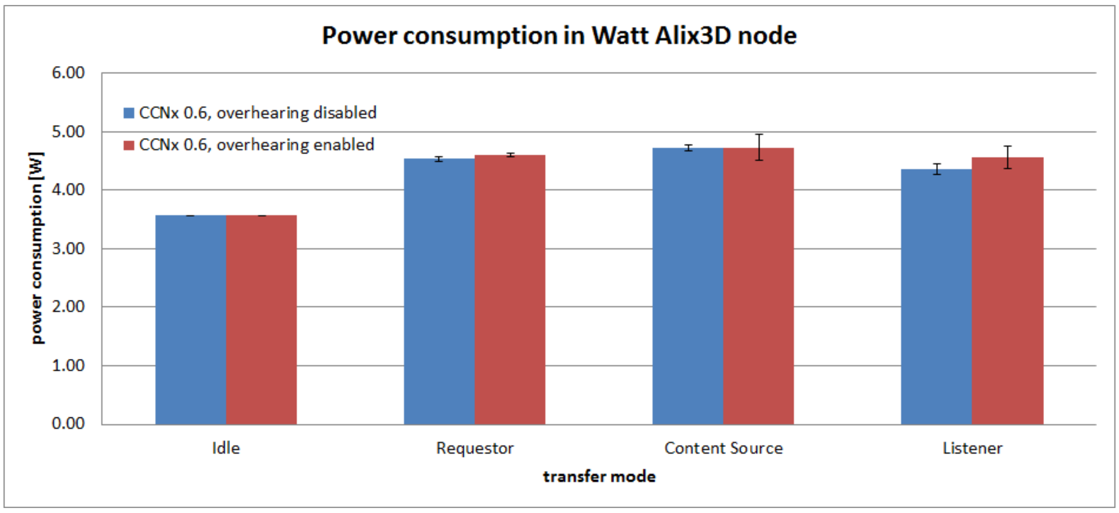

Figure 4.10: Power consumption of an Alix node with CCNx version 0.6 running during data transfer and idle. For the data transfer, IEEE 802.11a was used.

	Idle	**Requester**	**Content source**	**Listener**
CCNx 0.6 multicast,	3.57W	4.59W	4.73W	4.57W
overhearing enabled	100.00%	101.27%	100.13%	104.58%
CCNx 0.6 multicast,	3.57W	4.54W	4.72W	4.37W
overhearing disabled	100.00%	100.00%	100.00%	100.00%

Table 4.9: Median power consumption in Watt for different scenarios of CCNx 0.6 with thesis implementation and CCNx 0.6 original.

To evaluate the extra energy consumption in Joule to transfer an entire file, we took the results from the node roles defined above. As mentioned in subsection 4.4.1, these results were measured for a 5MB file. To get the energy consumption, we took the median value of the power consumption evaluation and multiplied it with the transmission time in seconds of the corresponding file transfer. Figure 4.11 illustrates the extra energy consumption for transferring a 5MB file in Joule using the CCNx version 0.6 with overhearing compared to the original CCNx version 0.6. The x-axis shows the nodes role. The y-axis shows the power consumption in Joule. The values correspond to the transmission of a 5MB file using multicast. A file transfer has with enabled overhearing a longer duration than with disabled, therefore, to have the same time interval, which allows a fair comparison, the faster node switch after a successful transmission into the idle mode. This additional energy consumption is added to the faster nodes total consumption. The content source consumes with enabled overhearing functionality approximative as much energy as with disabled overhearing functionality. This is because the content source does not process own ContentObjects, thus, it needs same energy as with disabled overhearing. The requester instead needs 1.88% more energy with enabled overhearing due to the additional processing and the registering of incoming ContentObjects. The listener has with 5.22% the largest difference between enabled and disabled overhearing functionality.

With enabled overhearing the listener changes from a passive to a more active role where it adds overheard ContentObjects to the FIB. Thus, the energy for the processing is much higher as with disabled overhearing where the listener only puts the incoming ContentObjects temporary into the content store.

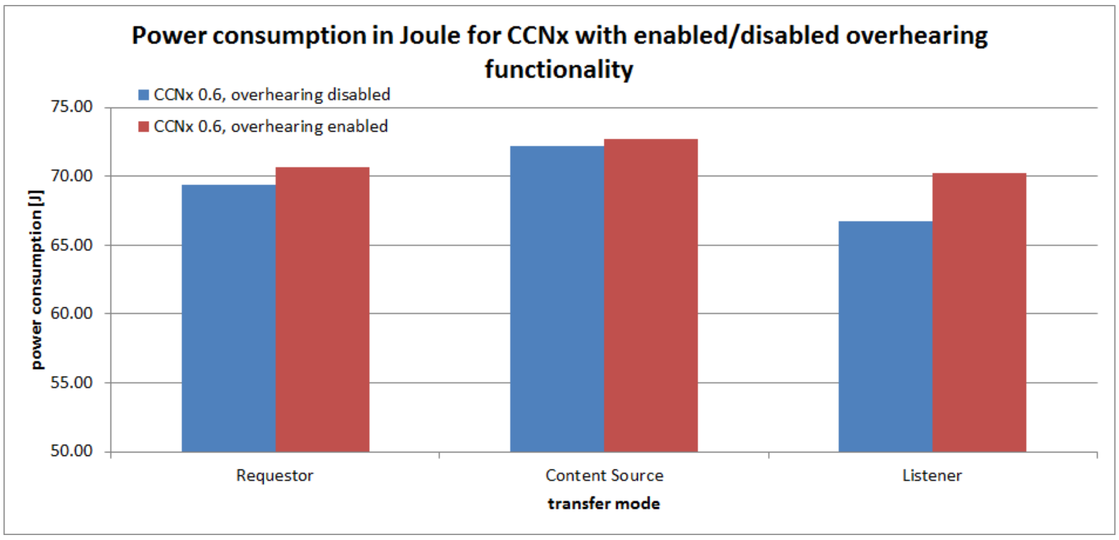

Figure 4.11: Extra power consumption of thesis implementation compared with original CCNx version 0.6 for transferring a 5MB file. The nodes change to idle mode after finishing transmission.

4.4.3 Multi- vs Unicast Energy Consumption

We measured the power consumption of an unicast transmission and show the results in table 4.10. Table 4.11 shows the energy in Joule the multicast transmission requires compared to unicast transmission for different file sizes. Furthermore, this table lists relative additional power consumption of multicast compared to unicast in percent. The expected energy consumption for different file sizes was calculated based on the average transmission times in Section 4.3.5 and the power consumption in Watt for a multicast transmission listed in table 4.9 respectively for an unicast transmission listed in table 4.10. We observed in subsection 4.3.4 that the multicast throughput is lower than the unicast. Therefore, the unicast node switches, after finishing the transmission, into the idle mode until the multicast transmission is finished. For a fair evaluation, the energy consumption of the node within this time period was added to the total energy consumption of the unicast transmission.

As expected consumes a multicast transmission more energy than a unicast transmission. This is due to the lower throughput of multicast. The overhead for the requester is for all file sizes approximatively 10%, for the content source the overhead is for small file sizes 41% which decreases to 28% for large files. Thus, from the aspect of energy consumption, multicast has an advantage compared to unicast if there are at least 2 requester nodes for the same data. In multicast communication the overhead of listeners that are not participating in the communication is higher but they gain information of available content and can receive it without efforts.

	Idle	Requester	Content source	Listener
CCNx 0.6 unicast	3.57W	4.69W	3.76W	not possible

Table 4.10: Median power consumption of unicast in Watt for different scenarios of CCNx 0.6.

	Requester			**Content source**		
	Joule		**relative**	**Joule**		**relative**
	unicast	**multicast**	**in %**	**unicast**	**multicast**	**in %**
0.25MB	4.09	4.45	9%	3.59	5.05	41%
0.50MB	7.30	7.94	9%	6.42	8.69	35%
1.00MB	13.73	14.93	9%	12.06	15.78	31%
2.00MB	26.12	28.27	8%	22.86	29.99	31%
5.00MB	64.29	69.33	8%	56.11	72.64	29%
10.00MB	127.67	136.39	7%	110.58	142.03	28%

Table 4.11: Energy consumption in Joule using unicast/multicast transmission for transmitting different file sizes. After finishing transmission, the nodes change to the idle mode. The relative values show the energy overhead of a multicast transmission compared to unicast transmission for a given file size.

Chapter 5

Conclusion and Future Work

5.1 Conclusions

Configuring mobile networks with static unicast IP addresses is not feasible. A possible solution to avoid this static unicast IP addresses is to use a multicast address. Thus, mobile devices do not need to know other network participants' address to request data. This thesis has examined how to extend the forwarding table (FIB) of the CCNx framework with overhearing multicast traffic. The evaluation was done on Alix nodes running CCNx version 0.6, which was the latest version at the time of the evaluation. Firstly, to gain an idea how fast these nodes can process CCNx traffic, the reached IEEE 802.3 unicast throughput was compared with the result of Van Jacobson et. al [1]. It results that the transmission rates of Van Jacobson are higher by factor 14.5 compared to resource limited Alix nodes. This is due to the low CPU capacity of the Alix nodes. Secondly multicast transfer rates were tested. By increasing the pipeline sizes from 16 to 512, the rates reached approximatively 34% of the unicast transfer rates, however at the expense of 75% more Interest transmissions due to collisions and Interest timeouts. By decreasing the Interest lifetime from 4 seconds to 0.25 seconds, the multicast throughput can be increased of 731%. Compared to unicast, multicast reaches a throughput of approximatively 59% with same pipeline size and without increasing the number of transmitted Interests. The reasons for the slower transfer rates of multicast are i) the lack of acknowledgements and ii) the retransmission delay in case of collisions due to retransmissions on a higher layer and iii) the larger back off times. An adaptive mechanism to adjust the Interest timeout based on number of incoming duplicated ContentObjects and number of retransmissions could increase the transfer rates. Thirdly the overhearing overhead was evaluated. Compared with the CCNx version 0.6 the time overhead was between 0.02% and 9.23%. Finally, the power consumption of CCNx on Alix nodes was evaluated. Compared with disabled overhearing, an Alix nodes consumes, with enabled overhearing, between 0.74% (content source) and 5.22% (listener) more energy for transmitting a 5MB file. Compared to unicast communication a requester requires 7%-9% additional energy for multicast communication. A content source requires 28% - 41% additional power for multicast communication compared to unicast.

On one hand, the higher multicast transmission time and energy overhead compared to unicast communication seem to render multicast overhearing useless. However, if the transfer rates of multicast can be increased, e.g., by shorter Interest lifetime delays, the energy and time overhead can be reduced significantly. On the other hand, when two or more network participants need the same data, multicast has an advantage to unicast. A further advantage with multicast is, the other network participants get content information by overhearing which may be used by routing mechanisms. However, power saving mechanisms need to be developed to reduce the energy consumption of passive listeners.

5.2 Future Work

As discussed in chapter 4 multicast transmission over IEEE 802.11 is very slow. With some optimization we reached transfer rates of around 340kBps. For some applications e.g. video streaming, business software or video conferences, such transfer rates are too slow. Optimization of the the IEEE 802.11 MAC procedure for multicast transmissions could increase the multicast throughput. At the moment, the MAC procedure is optimized for unicast communication.

Another task is to reduce the processing overhead of the overhearing functionality. This could be done by allowing higher values for the modulo function that manages which next incoming ContentObjects will pass the overhearing process. An approach could be an adaptive mechanism based on the size of the incoming file, if this information is available.

At the moment the lifetime of a FIB entry depends on the number of processed overheard ContentObjects. For each overheard and processed ContentObject grows the entry lifetime of the corresponding prefix entry by the same static value. This may result in large expiration values for large files and small expire values for small files. However, the entry lifetime should be based on content availability and not content size. This is inaccurate in case of mobility and e.g., if the large file is transmitted only once the related entry stays for a long time in the FIB.

Adapting the Interest lifetime is a good way to increase the multicast transmission speed. At the moment the Interest lifetime has to be set manually in the source code. Implementing an adaptive mechanism which decides the Interest lifetime based on the ContentObject response time and the received duplicated ContentObjects may increase the throughput in dynamic environments.

A further open issue is the prefix-cut. Let us assume that a node has configured two faces. The FIB contains the forwarding entry `ccnx:/ccnx.org/example` for face 1 and face 2. On face 1 the name-prefix `ccnx:/ccnx.org/example` was formed by the name-prefix cut between the prefixes `ccnx:/ccnx.org/example/ex1` and `ccnx:/ccnx.org/example/ex2`. On face 2 the same name-prefix was formed by the name-prefix cut between the prefixes `ccnx:/ccnx.org/example/ex3` and `ccnx:/ccnx.org/example/ex4`. If that node requests the data source ex2, it sends the Interests over face 1 and face 2. Without the prefix-cut, the Interests would have been sent only over face 1. Thus, a more sophisticated approach for the prefix cut is needed.

Chapter 6

Technical Appendix

6.1 Overview

Firstly follows a ALIX3D2 hardware specification. Those boards were using in the evaluation. Afterwards an overview over the necessary changes of the image-builder, which are indispensable for a successful installation of the CCNx framework on the board.

6.2 Hardware

Table 6.1 shows the specifications of the ALIX3D2 board.

Features	**Specification**
CPU	500 MHz AMD Geode LX800
DRAM	256 MB DDR DRAM
Storage	CompactFlash socket
Power	DC jack or passive POE, min. 7V to max. 20V
LED	three
Expansion	2 miniPCI slots, LPC bus
Connectivity	1 Ethernet channel (Via VT6105M 10/100)
I/O	DB9 seral port, dual USB
Board size	100 x 160 mm - same as WRAP.2E
Firmware	tiny BIOS

Table 6.1: Specifications and features of ALIX3D2 board

Additionally on every miniPCI slot is a Wistron DNMA92 802.11 a/b/g/n miniPCI radio.

6.3 Setup ALIX-Node

There are some necessary changes and updates for installing CCNx on an ADAM[6] system. A short overview for using CCNx with ADAM is given below. The svn-common packet must be installed on the computer to check out the corresponding repository. First of all checkout the image-builder[14] and switch to the newly created image-builder folder. Next, the following changes have to be done:

- Adding the buildscript ”ccnx.sh” which is needed for building the ccnx framework on the node. It is possible to take the existing source code from the standard repository. The better way is to download the latest of CCNx version from the CCNx [3] website. Change to the location of the downloaded CCNx version and build a new distribution (with make distfile) e.g.,

  make distfile VERSION=0.1beta

  That command crates a new CCNx package named ccnx-0.1beta.tar.gz. Do not forget calculating the sha1sum from the new generated packet. After that put the CCNx package into the desired source location of your image-builder and rename the necessary values (VERSION, SHA1SUM, URL) in the ccnx.sh build script (see listing below).

```
#!/bin/bash

##################################################
. ${BUILDDIR}/buildscripts/functions

VERSION="0.4b-webj"
SHA1SUM="75e38e647e74cf06232a5041bc869a82b43d9073"
URL="http://ba.darky.ch/webj"
FALLBACK="http://ba.darky.ch/webj"
BUILD_DEPS="toolchain"
##################################################

download_gz ccnx &&

cd ${BUILDDIR} &&
tar -xzvf ${SRCDIR}/ccnx-${VERSION}.tar.gz &&
cd ccnx-${VERSION}/csrc &&
export INSTALL_BASE="${INSTALLDIR}"
CC="${CC} -Os -fPIC" ./configure &&
make &&
make DESTDIR=${INSTALLDIR} install &&

cd ${BUILDDIR} &&
rm -rf ccnx-${VERSION}
```

- CCNx needs openssl rc2 support. In the trunk version, the openssl.sh script ignores the rc2 support. Without rc2 support, the CCNx security mechanisms (encryption an decryption) will not work. Thus, it will be not possible to start the CCN-daemon on the node. Removing no-rc2 string in the openssl.sh script is the only thing to do for having the support.

```
if [ "${BOARDARCH}" = "arm" ] ; then
        ./Configure linux-generic32 -DL_ENDIAN --prefix=/ --openssldir=/etc/ssl no-idea no-md2 no-mdc2 no-rc2 no-rc5 no-sha0 no-smime no-rmd160 no-aes192 no-ripemd no-camellia no-ans1 no-krb5 no-ec no-err no-hw shared zlib-dynamic no-engines no-sse2 no-perlasm
else
        ./Configure linux-embedded-${BOARDARCH} --prefix=/ --openssldir=/etc/ssl no-idea no-md2 no-mdc2 no-rc2 no-rc5 no-sha0 no-smime no-rmd160 no-aes192 no-ripemd no-camellia no-ans1 no-krb5 no-ec no-err no-hw shared zlib-dynamic no-engines no-sse2 no-perlasm
fi
```

Listing 6.1: oppenssl.sh without CCNx

```
if [ "${BOARDARCH}" = "arm" ] ; then
        ./Configure linux-generic32 -DL_ENDIAN --prefix=/ --openssldir=/etc/ssl no-idea no-md2 no-mdc2 no-rc2 no-rc5 no-sha0 no-smime no-rmd160 no-aes192 no-ripemd no-camellia no-ans1 no-krb5 no-ec no-err no-hw shared zlib-dynamic no-engines no-sse2 no-perlasm
else
        ./Configure linux-embedded-${BOARDARCH} --prefix=/ --openssldir=/etc/ssl no-idea no-md2 no-mdc2 no-rc5 no-sha0 no-smime no-rmd160 no-aes192 no-ripemd no-camellia no-ans1 no-krb5 no-ec no-err no-hw shared zlib-dynamic no-engines no-sse2 no-perlasm
fi
```

Listing 6.2: oppenssl.sh with CCNx

- For using CCNx on the node, install expad as well. It must be added to the buildprofile. The expad buildscript is already available.
- For using IEEE 802.11[15] 5 GHz frequencies it needs a pached version of the linux.sh packet. See linux-3.2-pached.sh in /buildscripts/packages for further information.

Once this changes are done, it is easy to setup the alix-node with CCNx support. Follow the instructions of README.built and README.alix where is explained how to flash the image.

6.4 Start CCNx on the Node

There are two ways to connect with a running the Alix-Node:

- **SSH**: to set up a SSH connection the node has to be connected with an Ethernet cable to a network. Afterwards the command interface of the node can be invoked by the following command: ssh root@ip-address.

- **Minicom**: minicom is a connection software which allows to access an Alix-Node without a working IP-configuration. The node is connected by an USB-Cable to a workstation. To connect the Alix-Node over minicom the following steps are todo:
  - Install minicom with `apt-get install minicom`
  - Configure minicom with `minicom -s` and choose `Serial port setup`
  - Choose `A` and exchange `/dev/tty8` through `/dev/ttyUSB0`
  - Save changes and restart minicom

After successfully accessing the node, the following steps are to do:

- Start the `ccninitkeystore` command. This command will create the keys for the security mechanisms of ccnx. If there is no rc2 support on the node, it will fail and CCNx will not work.
- To add multicast communication support, it is necessary to add a new IP route to the routing table. For using multicast with CCNx, type

  ```
  ip route add multicast address/subnet dev device
  ```

  e.g.

  ```
  ip route add 224.0.23.170/32 dev wlan0
  ```

Note, without this configuration the CCN daemon will not be able to add multicast faces.

Chapter 7

Appendix

7.1 CCNx values

For the thesis implementation some new values were created:

- CONTENT_ADD_FREQUENCY
  It is a environment variable which describes how frequent a ContentObject will be processed.
- MAX_PREFIXES
  This is the value which contains the maximum name-prefixes which can be added with overhearing as forwarding entry into the FIB. Default value is 10. It is an environment variable.
- SET_PREFIX_EXPIRATION
  This constant contains the number of additional seconds a forwarding entry is valid, if an new content arrives over an existing forwarding entry.

7.2 Structs

This section gives a short overview of the structs which were used during the implementation of the overhearing functionality.

```
struct register_data{
        int expire;
        int newcomps;
};
```

Listing 7.1: struct register_data

7.2.1 Struct dyn_add

This struct holds every necessary information to each dynamically added forwarding entry. Furthermore `dyn_add` stores information over left free space for adding more forwarding entries dynamically. The value `next_slot` points to the next free `dyn_entry` 7.3. This struct is used for a faster access to the FIB.

```
struct dyn_add{
        int next_slot;
        struct dyn_entry **dyn;
};
```

Listing 7.2: struct dyn_add

7.2.2 Struct dyn_entry

The struct `dyn_entry` contains information about one specific name-prefix. The struct contains the following variables:

- **key**: the ccnb-encoded name-prefix
- **length**: the number of letters the ccnb-encoded name-prefix has.
- **ncomps**: the number of components the stored name-prefix has.

```
struct dyn_entry {
        unsigned char *key;
        int length;
        int ncomps;
};
```

Listing 7.3: struct dyn_entry

7.2.3 Struct register_data

This struct is relevant for the updating process after a prefix cut 7.1 was necessary. It contains two values (see listing 7.4):

- **expire**: the time in seconds which a forwarding entry has left until it expires.
- **newcomps**: the number of new components which has to be registered.

```
struct register_data{
        int expire;
        int newcomps;
};
```

Listing 7.4: struct register_data

7.3 Functions

This section describes the functions which are mentioned in the sections 3.2.1 and 3.2.2. The intent of each functions is described in a separated subsection.

7.3.1 nameprefix_match

```
int nameprefix_match(struct ccnd_handle *h, struct register_data *
    reg_info, const unsigned char *msg, struct ccn_indexbuf *comps,
    int newcomps, int incom_face, int is_interest)
```

Listing 7.5: header of nameprefix_match

The function `nameprefix_match` is the key function for registering overheard name-prefixes. It manages the whole registering process of the overhearing process.

First `nameprefix_match` checks if in the FIB a forwarding rule already exists for an incoming name-prefix. As well it updates the expiration time 3.1.3 of a forwarding rule if it is below an certain value. For new registrations the function writes all the needed values into the struct `register_data` 7.4. This struct contains the information which will be used for registering an new forwarding entry into the FIB. Furthermore `nameprefix_match` handles the prefix cut functionality which was described in section 3.1.4. For reusing code, the function is constructed in two parts, the first one handles Interest messages, the second part ContentObjects.

Matching ContentObjects in nameprefix_match

Overheard ContentObjects are processed in `project_incoming_content`. If the overhearing functionality is implemented, the `nameprefix_match` function will be called with a message (Interest or ContentObject). In this case it is a ContentObject. The first thing `nameprefix_match` checks, is whether the struct `dyn_add` 7.2.2 contains already the name-prefix (longest prefix match) of the incoming ContentObject. It compares the name-prefix components from the incoming ContentObject with the name-prefix components of the already dynamically added entries. If it gets a match, `nameprefix_match` stores the number of matching components and the number of components from the existing forwarding entry as an integer in the values `newcomps` respectively `oldcomps` (see listing 7.6 line number 13 and 14).

The intent of `newcomps` and `oldcomps` is explained in the next paragraph.

```
1       for(i=0;i<MAX_PREFIXES;i++){
2           res=1;
3           if(h->dyn_add==NULL){
4                   ccnd_msg(h, "error dyn_add is not inizialized");
5           }
6           if(h->dyn_add->dyn[i]==NULL)
```

```
                continue;
        compmsg = h->dyn_add->dyn[i]->key;
        if(compmsg != NULL){
                        for(j=newcomps;j>1;j--){
                                res = memcmp((msg+base), compmsg,
                                    comps->buf[j]-base);
                                if(res==0){
                                        newcomps=j;
                                        oldcomps=h->dyn_add->dyn[i]->
                                            ncomps;
                                        break;
                                }
                        }
        }
```

Listing 7.6: check for a matching name-prefix in function nameprefix_match

The function `nameprefix_match` looks, after getting a name-prefix match, for the related `nameprefix_entry` (npe) 2.2.4 in the name-prefix table. The name-prefix table is a hash table which stores all npes.

Once we figured out the npe, a name-prefix cut has to be done (the code is available in the appendix 7.4.1). In a CCNx network there are many data objects which having common components, such as `ccnx:/parc.com/data/file1` and `ccnx:/parc.com/data/file2`. Instead of storing both of them as a forwarding entry, simply combine the first and the second to `ccnx:/parc.com/data`. With this possibility a lot of entries are not necessary and that will increases the routing performance. On the other hand it decreases accuracy of the routing because it may route a Interest over a face which will not have the related content.

Figure 7.1 illustrates the work flow of a prefix-cut. Firstly the function `nameprefix_match` determines the size of the two temporary variables `oldcomps` and `newcomps`. Secondly `nameprefix_match` compares the size of this two variables. If the value of `oldcomps` greater than the value of `newcomps` is this an indication that the new incoming content name-prefix has common components with an available forwarding entry and it is possible to merge this two prefixes to a shorter one. In that case, the common name components will be stored in the `register_data` struct. The already stored longer name-prefix will be replaced through the new shorter name-prefix. If the value of `oldcomps`is shorter than the value of `newcomps`, the entry lifetime of the already existing name-prefix will be updated. If the incoming name-prefix has no matching forwarding entry, it will be registered as a new entry into the FIB. Thus, no prefix cut processing is necessary.

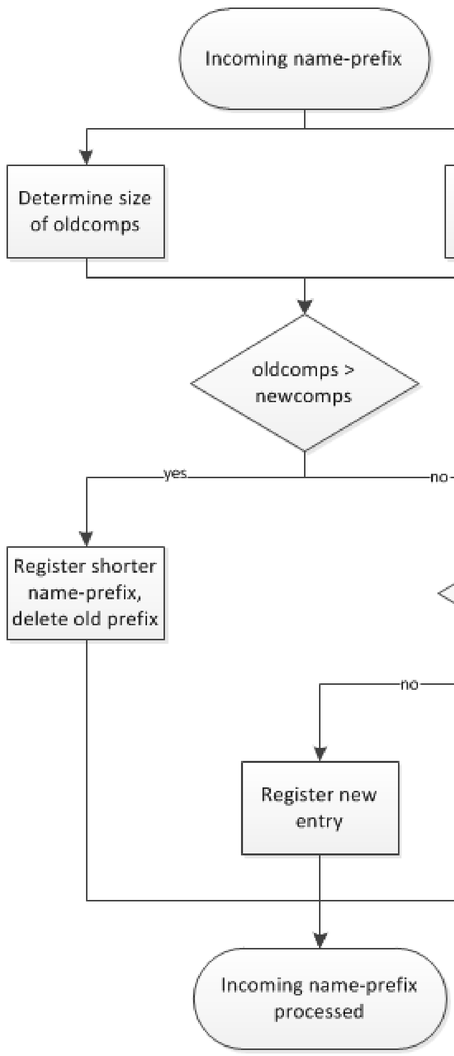
Incoming name-prefix
Determine size of oldcomps
Determine size of new comps
oldcomps > newcomps
yes
no
Register shorter name-prefix, delete old prefix
Name-prefix already registred?
no
yes
Register new entry
Update available name-prefix
Incoming name-prefix processed

Figure 7.1: Flow chart of prefix cut, it shows how a name-prefix of an incoming ContentObject will be shortened in the `nameprefix_match` function. The value `oldcomps` contains the number of components which the already registered matching name-prefix has, the value `newcomps` contains the number of component matches which has the incoming name-prefix with the already registered name-prefix.

After name-prefix cut, the function `nameprefix_match` has three possible return values:

- -1, if nothing has to be done, means `nameprefix_match` has done the name-prefix entry (npe) update, or no free slot is available 7.3.4.
- 0, if a new name-prefix must be registered. The necessary registration values are stored in the struct `register_data` 7.2.3.
- greater than 0, if a name-prefix cut has to be registered. As well the necessary update values are stored in the struct `register_data`.

Matching Interests in nameprefix_match

If the function `nameprefix_match` is called with an Interest, the work flow is nearly the same as for a ContentObject. The difference is that the name-prefix cut is not necessary for an Interest. We simply register a new entry for a short period of time. So it is possible to forward Interests. Thus there are only two possible return values:

- -1, if nothing has to be done, means the forwarding update was done by `name_prefix_match`
- 0, if a new entry has to be registered.

7.3.2 Function get_segment_number

This function figures out the segment number 2.1.1 of a given ContentObject. As explained in Section 3.1.5 it may be beneficial not to process every ContentObject in the same way. Therefore, we use the extracted segment number and calculate the modulo.

```c
int get_segment_nr(struct content_entry *content){

        int i;
        char *pos;
        struct ccn_charbuf *c;
        int start;
        int stop;
        int length;
        int res;
        c = ccn_charbuf_create();

        ccn_uri_append(c, content->key, content->size, 1);

        if(res<0){
                return 0;
        }
        pos = strstr(ccn_charbuf_as_string(c), "%00%");

        if(pos==NULL)
                return 0;

        start = pos - ccn_charbuf_as_string(c)+4;
        stop = start;
        for(i=0; c->buf[start+i]!='/';i++){
                stop++;
                if(i>=c->length){
                        return -1;
                }
        }
        length = stop-start;

32          if(stop==start){
33                  ccn_charbuf_destroy(&c);
34                  return 0;
35          }
36
37          char segment[length+2];
38          memset(segment,'\0',sizeof(segment));
39
40          for(i=0;i<length;i++){
41                  segment[i]=c->buf[start+i];
42          }
43          res = strtol(segment, NULL, 16);
44          ccn_charbuf_destroy(&c);
45          return res;
46  }
```

Listing 7.7: listing for function nameprefix_match

For understanding how `get_segment_number` extracting the segment_number out of an ContentObject we must understand some key-points. We know from the Section Related Work, that a messages naming is built hierarchically. Each backslash defines a new component. However each component is ccnb-encoded, thus it is not possible to read out a specific part from a single component. CCNB is a custom binary encoding format for XML to meet specific needs of CCNx. So, first the components must be parsed into an ascii representation. The `ccn_uri_append` function from CCNx (line 12 in listing 7.7) , provides exactly that functionality.

To get out the segment-number, decoding ccnb was the only tricky part to do. Behind that, the result is stored in an `ccn_charbuf` in which the name-prefix is represented in ascii (see `ccn_charbuf *c` in listing 7.7, line 12). With this representation, each segment number follows after an `%00%` sign. For finding this marker, some string operations are needed. Eventually `get_segment_nr` returns the segment number which is related to the ContentObject.

7.3.3 Function autoreg_prefix_content

The function `autoreg_prefix_content` connects the registering function from CCNx with the gained data from the overhearing functionality.

The implementation of `autoreg_prefix_content` is visible in appendix 7.4.3. First follows the initialisation, in this part the needed flags will be setted up. The default flags are `CCN_FORW_ACTIVE` and `CCN_FORW_CHILD_INHERIT`. Thereafter `autoreg_prefix_content` distinguish between register a totally new name-prefix or register shorter version of an existing prefix. The data for the registration process are available in the struct `reg_info` (see 7.2.3). With this information a name-prefix can be registered. The return values are:

- zero, if everything is great
- minus one, if something failed

7.3.4 Function check_free_slot

As described in Section 3.1.4 it is not good to register too many overheard name-prefixes to the FIB. The struct `dyn_add` 7.2.2 contains all dynamically added name-prefixes. It has also a value which points to the next free slot where a new registered name-prefix can be stored. The function `check_free_slot` finds the next free slot in the `dyn_add` struct and stores the array index of it in the `next_slot` integer from `dyn_add` struct. Code is available in appendix 7.4.4.

7.3.5 Function update_dyntable

```
void update_dyntable(struct ccnd_handle *h, unsigned char *key,
    struct ccn_indexbuf *comps, int ncomps)
}
```

Listing 7.8: update_dyntable function

As described in Section 3.1.4, it is bad to have a to big FIB. So `update_dyntable` 7.8 updates after each successful registration the `dyn_add` 7.2.2 table. Which means it creates a new name-prefix entry and sets the necessary variables.

There are two name-prefix cases to distinguish:

1. A specific name-prefix length is given, this case occurs when a successfully prefix cut was done.
2. A new name-prefix was registered.

In the first of this two cases, only some components of the name-prefix will be registered. In the second version all components of the name-prefix will be registered. The registration process itself is the same.

Each name-prefix which was dynamically added to the FIB occupies a free slot. For the name-prefixes which were added with overhearing, only a limited number of slots available (see 3.1.4). The value `next_slot` (see listing 7.4.2) stores an integer which points to the next available array index (slot) of the `dyn_add` table 7.2. Each of this slots contains a `dyn_entry` 7.2.2. So for adding a new name-prefix `update_dyntable` writes all available information into the specific slot.

7.4 Code Listings

With respect to the environment, the code listings are only available in electronic format. In the following subsections the edited and newly implemented functions are declared.

7.4.1 Function nameprefix_match

`/src/ccnx/csrc/ccnd/ccnd.c`

7.4.2 Function update_dyn_table

`/src/ccnx/csrc/ccnd/ccnd.c`

7.4.3 Function autoreg_prefix_content

`/src/ccnx/csrc/ccnd/ccnd.c`

7.4.4 Function check_free_slot

`/src/ccnx/csrc/ccnd/ccnd.c`

7.4.5 Function has_undesired_markers

`/src/ccnx/csrc/ccnd/ccnd.c`

7.4.6 Structs

`/src/ccnx/csrc/ccnd/ccnd_private.h`

7.5 Scripts

In this section we declare the scripts we used for the node setup and the evaluation.

7.5.1 Evaluation Scripts

`/experiments/scripts/evaluation/`

7.5.2 Scripts to Initialize Alix node

`/experiments/scripts/setup/`

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] "Ndn dhcp." [Online]. Available: https://github.com/NDN-Routing/ccnx-dhcp

[3] "Project ccnx," September 2009. [Online]. Available: http://www.ccnx.org

[4] *nameprefix_entry Struct Reference*. [Online]. Available: http://www.ccnx.org/releases/latest/doc/ccode/html/structnameprefix__entry.html

[5] *CCNx Face Management and Registration Protocol*, August 2012. [Online]. Available: http://www.ccnx.org/releases/latest/doc/technical/Registration.html

[6] T. Staub, S. Morgenthaler, D. Balsiger, P. Goode, and T. Braun, "Adam: Administration and deployment of adhoc mesh networks," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a*, june 2011, pp. 1 –6.

[7] http://www.ccnx.org/releases/latest/doc/technical/NameConventions.html, August 2012.

[8] [Online]. Available: http://www.pcengines.ch/

[9] [Online]. Available: http://netcat.sourceforge.net/

[10] L. E. P. N. Somaya Arianfar, JÃ¶rg Ott and W. Wong, "A transport protocol for content-centric networks," *18th IEEE ICNP*, October 2010.

[11] G. Carofiglio, M. Gallo, and L. Muscariello, "Icp: Design and evaluation of an interest control protocol for content-centric networking." in *INFOCOM Workshops*. IEEE, 2012, pp. 304–309. [Online]. Available: http://dblp.uni-trier.de/db/conf/infocom/infocom2012w.html#CarofiglioGM12

[12] N. Rozhnova and S. Fdida, "An effective hop-by-hop interest shaping mechanism for ccn communications," in *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, march 2012, pp. 322 –327.

[13] *Rigol DM3058*. [Online]. Available: http://www.rigolna.com/products/digital-multimeters/dm3000/dm3058/

[14] S. T. Staub, T., “Adam: image-builder ccnx,” 2012. [Online]. Available: https://subversion.cnds.unibe.ch/svn/adam/branches/ccnx/image-builder/

[15] “Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications,” *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, pp. 1 –2793, 29 2012.