DATA EXCHANGE IN INTERMITTENTLY CONNECTED CONTENT-CENTRIC NETWORKS

Bachelorarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Tobias Schmid
2013

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

Contents

List of Figures

List of Tables

Listings

Acknowledgement

On this page I would like to thank everybody who supported me to write this bachelor thesis.

First I would like to thank my coach Carlos Anastasiades. He supported me trough the whole process of writing this thesis, had an ear for discussing problems with the thesis and gave inputs when I was blocked with my work.

After that I would like to thank Prof. Dr. Torsten Braun who allowed me to write this thesis in his research group. I am also very grateful for the resources that were generously provided by the research group of Prof. Braun.

Last but not least my work colleagues and office mates Jürg Weber and Alexander Striffeler who have been there for solving problems, drinking coffee, killing time and fooling around.

Abstract

Content-Centric Networking (CCN) is another approach of network architecture and accessing data within networks. In contrast to host-based network architectures, content-centric networks have no need to know their neighbours and the whole network topology because data is requested based on a content specific identifier and not a host identifier. For this reason, content-centric networks are suitable for dynamic mobile networks with many moving devices.

The project CCNx has the goal to implement this approach as a open source framework. In this thesis the existing file transfer applications of the CCNx framework have been extended with a resume capability.

The evaluation of these extended applications showed that the resume capability enables a constant effective transfer time independent of the number of interrupts during a file transfer. Using a pipeline size of 16 and a block size of 4096 bytes enabled the fastest transfer rate. Nevertheless the results pointed out that the CPU of resource constrained devices limits the maximal reached transfer rate independent of the network connection.

Chapter 1

Introduction

When the Internet architecture was developed network devices and computers were not integrated in the environment the same way as today. Mobile computing was just a dream because the devices were too big to carry around. The Internet architecture has been developed to share resources and the communication was limited to wired connections. Since then technology evolved very fast and many constraints have disappeared. Nowadays, it is even possible to use the air to exchange data between mobile devices.

Today's network connections are used mostly to exchange data. It is important to receive a specific requested content but the real location of the content source is irrelevant. With the increasing dissemination of mobile devices such as smart phones or tablets, that fact is reinforced.

For this reason, different information-centric networking approaches have been developed. One prominent approach is Content-Centric Networking (CCN) [1]. The open source framework implementation CCNx [2] is the basis of this thesis.

1.1 Motivation and Task Formulation

1.1.1 Motivation

In content-centric networks (CCN) data is organized in segments. A user who is interested in the content can express Interest packets to receive the corresponding data segments in return. If a consumer has received all the segments, he can save the data to a file. In case of an interrupted transfer the received data is incomplete and useless. The transfer has to be restarted from the first segment. Depending on the mobility or contact time of a device, certain files can never be completely received. Requests with short connection interrupts or small data chunks may be provided by a local cache, however, if connection interrupts are long, the data may not be longer available there.

1.1.2 Task Formulation

The goal of this thesis is to develop an application that enables file downloads in intermittently connected networks based on the current CCNx framework [2].
The work comprises the following tasks:

- study the current CCNx file transfer applications
- identify requirements to extend these applications with a resume capability[1]
- implement the resume capability extension
- compile the current CCNx code on a current ADAM[3] image
- evaluate the resume applications on real mesh nodes in different scenarios

1.2 Outline

The remainder of this report is organized as follows. Chapter 2 presents the basics of content-centric networking. Chapter 3 describes the idea and implementation of the resume capability in CCNx. After that we evaluate the new application with different scenarios and parameters in Chapter 4. Finally, in Chapter 5, we discuss the developed approach and conclude our work.

[1]The resume capability enables storing of already received data so that in case of a disrupted file transfer obtained data does not have to be retransferred.

Chapter 2

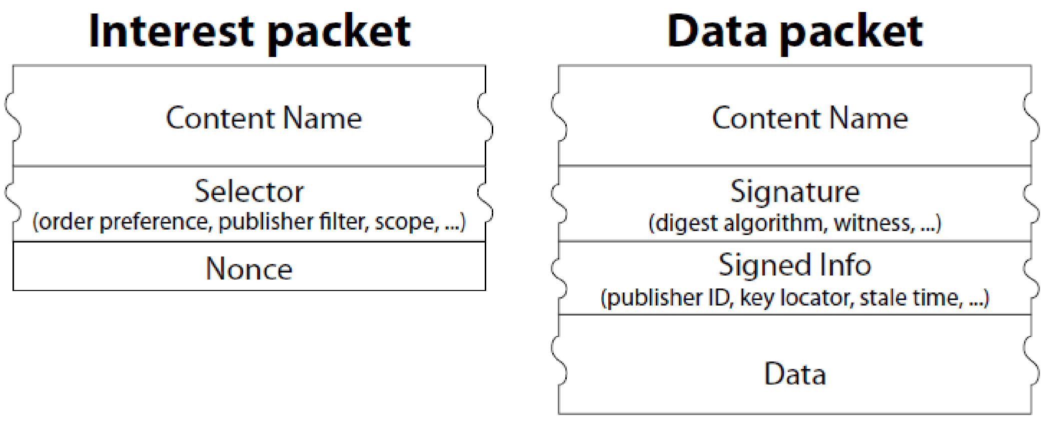
Related Work

2.1 Content-Centric Networks

Content-centric networking was introduced in a paper by Van Jacobson[1]. His research group at the Palo Alto Research Center (PARC) started to implement an open-source project called CCNx[2].

2.1.1 CCN Packets

In CCN, communication is based on two basic packet types: Interest and Data. Figure 2.1 shows the Interest packet for requesting content and the Data packet to deliver data which is called ContentObject in CCNx.



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

Figure 2.1: CCN packet types [1]

An Interest packet contains a prefix or the complete ContentName of the requested ContentObject and some selectors which allow to restrict the origin and matching of the Data packet.

A Data packet is defined as follows: It contains the ContentName, the Signature, the SignedInfo and the data itself.

There is an important rule for sending and receiving packets in CCNx; at most one Data packet is sent per Interest packet. If no Data packet matching an Interest arrives within a given time, a data is non-existent assumption will be made. This rule is a key behaviour for the implementation of the resume capability. Details are discussed later in chapter Design and Implementation of Resume Operations in CCNx.

2.1.2 Content Naming

Another essential part of content-centric networking is the content naming. Because content is requested by name a clever way of handling content names is necessary. A hierarchical name structure is suitable for this use case.

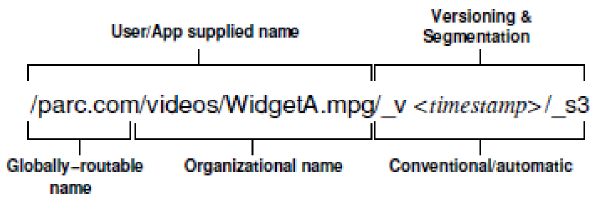


Figure 2.2: CCN naming

As shown in Figure 2.2 CCNx ContentNames can be represented by URIs which are easily read and remembered by humans. Every component between two slashes (/) represents a hierarchy level. Further every content has a version number and is split into several segments. To indicate the version and the segment markers are used. Versioning is implemented with timestamps. A *_v* marker followed by a timestamp represents the version of the packet. Similar the segment number is given after the *_s* marker. The first segment holds the segment number 0, for each additional segment the segment number will be incremented.

2.1.3 Data Storage in CCNx

In CCNx there are different ways of storing data. For persistent storing of data the CCNx repository is used. The ContentStore is used for caching data.

2.1.3.1 ContentStore (CS)

The ContentStore is like a cache for arrived Data packets. The received packets are stored in memory which allows a fast access time. If a packet arrives that is already included in the ContentStore it will be discarded. The life time of packets in the ContentStore is limited by the memory size and the *freshnessSecond* flag which is embedded in the Data packet header.

2.1.3.2 Repository

Repositories enable persistent storage of data on dedicated storage devices.

There are two different repository applications, one implemented in C the other in Java. The repository application written in C (called *ccnr*) has replaced the Java application (called *ccn_repo*) due to its better performance and integrated synchronization facility[4]. To share data over the repository they first have to be inserted. The inserting process includes segmentation, signing and encoding of the data which has only to be done once. After that inserted data is include in a file which can be stored persistent.

2.1.4 File Transfer in CCNx

The CCNx framework provides different applications to transfer files. Some of the applications offering the same functionalities are existing both in Java as well as in C. In this thesis we would only have a look at the C implementation because of the portability to embedded systems. Two command line transfer applications, *ccnsimplecat* and *ccncat* have been considered in this work and are presented in this section.

2.1.4.1 ccnsimplecat

The *ccnsimplecat* application provides sequential data transfer. The communication is performed segment by segment and only one data segment can be requested at a time. Figure 2.3 shows a sample request between a consumer and a content source. For obvious reasons this sort of file transfer is not very efficient but helps to understand the mechanisms of CCNx.

The application requests data with a given ContentName starting with segment number zero (*_s0*). After data retrieving the segment number is incremented by one. This step will be repeated until the final packet, indicated by a final block identifier, is received. Every Interest will time out if not answered within a defined period of time. Unanswered Interests are reexpressed infinitely often until the Data packet arrives or the application is terminated manually.

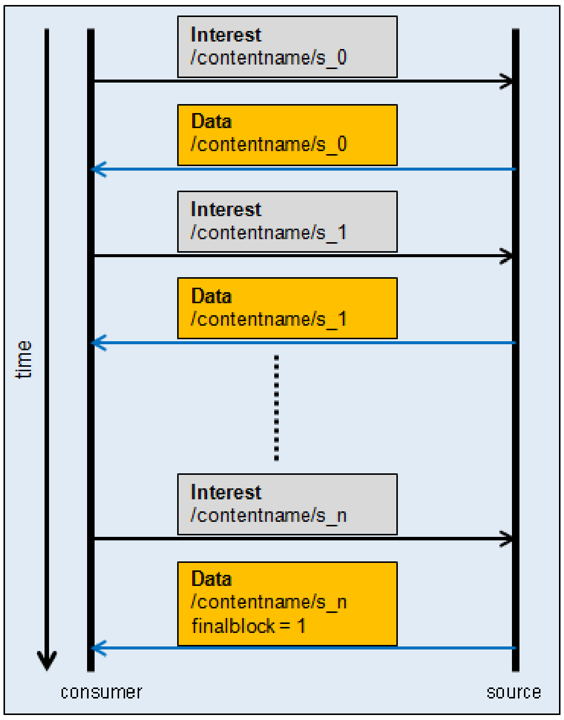


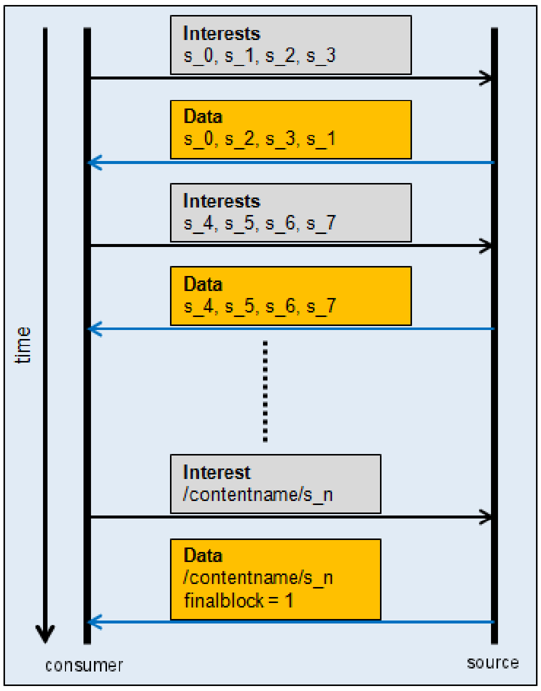
Figure 2.3: Sequential data transfer in CCNx

2.1.4.2 ccncat

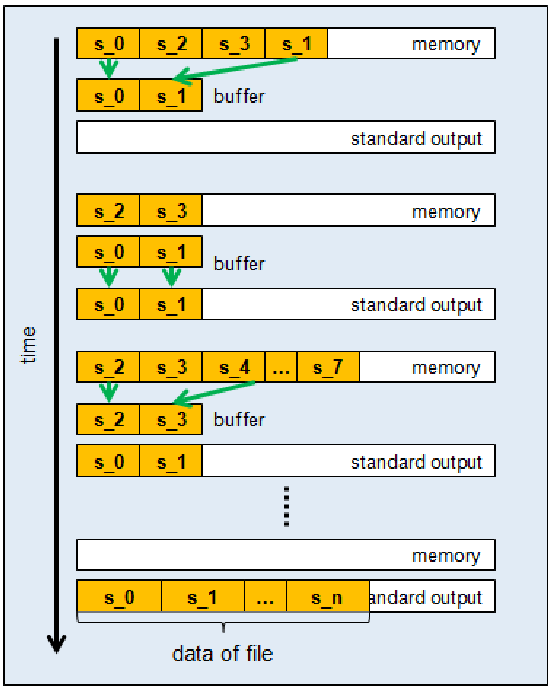
The *ccncat* file transfer application implements transfer pipelining. Pipelining means multiple Interest requests can be transmitted concurrently without receiving data in return. The maximum number of Interests that can remain unanswered at a time is defined by the pipeline size. Running *ccncat* with pipeline size 1 equals *ccnsimplecat*. The received data is buffered and will be put in the correct sequence afterwards.

Figure 2.4 shows a sample file transfer with a pipeline size of 4. The sequence diagram in Figure 2.4(a) shows that data packets sent by the source can arrive in an arbitrary order. Received packets are stored in the memory until it is possible to reassemble the data in the correct order (see Figure 2.4(b)). The data packets are reassembled in a buffer.
The *ccncat* application writes the data stored in the buffer to the standard output after that the buffer will be cleared. Checking if new data is in the buffer is implemented as a polling process which is discussed in detail in chapter Design and Implementation of Resume Operations in CCNx.

Like in *ccnsimplecat*, unanswered Interests are reexpressed in an infinite loop until either the data arrives or a stream error occurs which forces the application to terminate.



(a) Requests



(b) Data processing

Figure 2.4: Pipelined file transfer in CCNx

Chapter 3

Design and Implementation of Resume Operations in CCNx

In CCNx, segments are requested subsequently and are printed to the standard output. By piping the standard output stream to a file it is possible to save data. If the connection between a requester and a content source breaks before the completion of a file transfer, the transmission fails. No state information about already received segments is stored and therefore, the requester will request already received segments at the next try. Therefore, the complete file transmission has to be performed at once at the same time. In case of short disconnections, these segments may still be hold in the cache but in case of long disconnections or high traffic volumes and large file sizes, the segments may need to be retransmitted over the wireless medium. In this section we will describe the resume capability that is developed in this work. In Section 3.1 the problem addressed in this thesis is discussed. Section 3.2 introduces the meta information used for implementing the resume capability. Section 3.3 shows a visualization of the proposed approach. After that Section 3.4 introduces a new data structure for storing the proposed meta information. Finally Section 3.5 describes the implementation of the resume capability in both of the file transfer applications.

3.1 Problem Description

To be able to resume a transfer in CCNx, one has to know which segments already have been received and from which segment to start the resumed transfer. Currently CCNx provides variable and fixed segment lengths, therefore the size of the following segment is unknown. This leads to the problem that it is not possible to reconstruct the resume parameters out of the stored partial data. Because the segment block size in a file may vary, it is not possible to calculate the segment number of the last transferred segment. Hence, additional meta information is stored on the receiver side to support the resume capability.

3.2 Meta Information

To resume an aborted transfer at least the following meta information needs to be stored:

name of the ContentObject
To remember which data chunks of a ContentObject already have been received the ContentName needs to be known.

version of the ContentObject
Consumers usually request data without knowledge of the delivered version but when they try to resume a transfer there may already be other versions with the same ContentName available. To avoid inconsistencies the same content version has to be retrieved. In CCNx the version is encoded as a timestamp and has to be stored in the meta information.

next segment to receive
To have an entry point for a future resume one has to know from which segment to start. For this reason the incremented segment number of the last received segment is stored in the meta information.

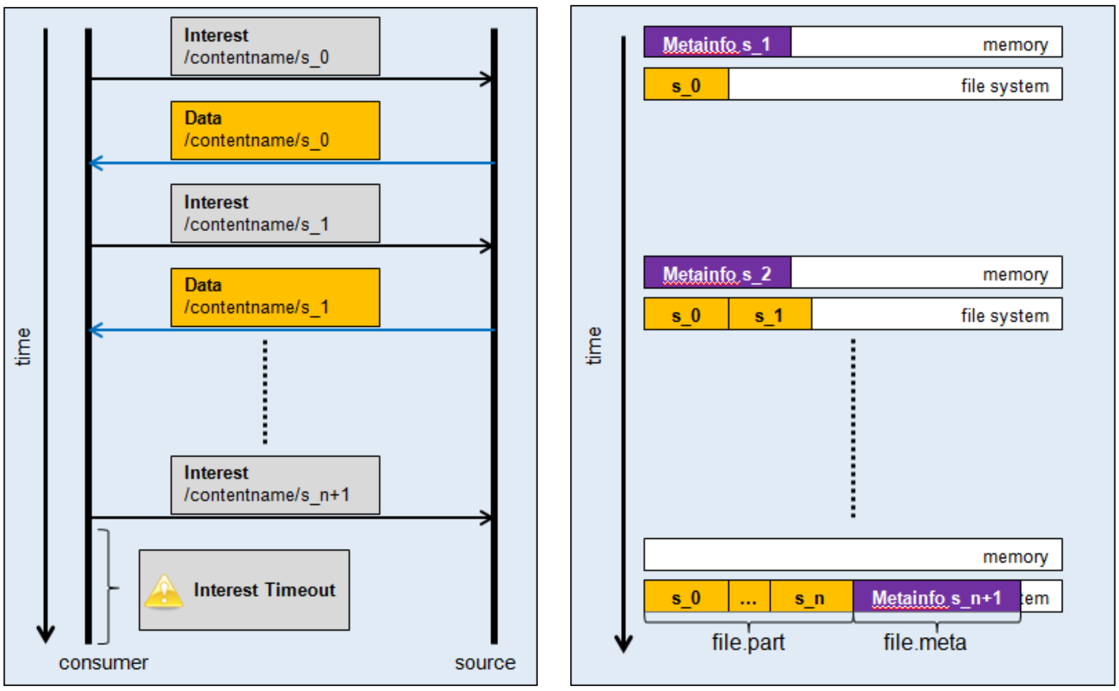
position of the partial data file
To ensure that the partial data file has not been modified outside of the CCNx file transfer application the file position is saved after every inserted data chunk.

Some guidelines are needed so that the application is able to find already stored data if the transfer will be resumed. The received partial data is written to the temporary file filename**.part** and the meta information is stored in filename**.meta**. The file name consists of the human readable representation of the ContentName without any markers. These files are kept on the device and are used to resume the transfer in case of regained connectivity. If a resume leads to a successful transfer the partial file is renamed to the filename without special extension and the meta information file will be removed from the storage device.

3.3 Resume Capability in CCNx Transfer Processes

This design idea will be implemented for the *ccnsimplecat* and the *ccncat* file transfer applications which have been introduced in Section 2.1.4.1 and 2.1.4.2. The main behavior of these applications is kept and resume capability is added. Hence, the application names are modified by appending *_resume*.

3.3.1 ccnsimplecat_resume

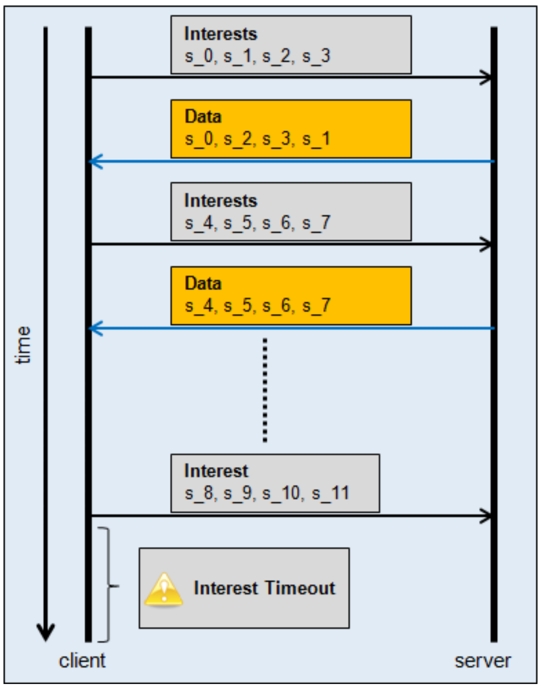


(a) Requests (b) Data processing

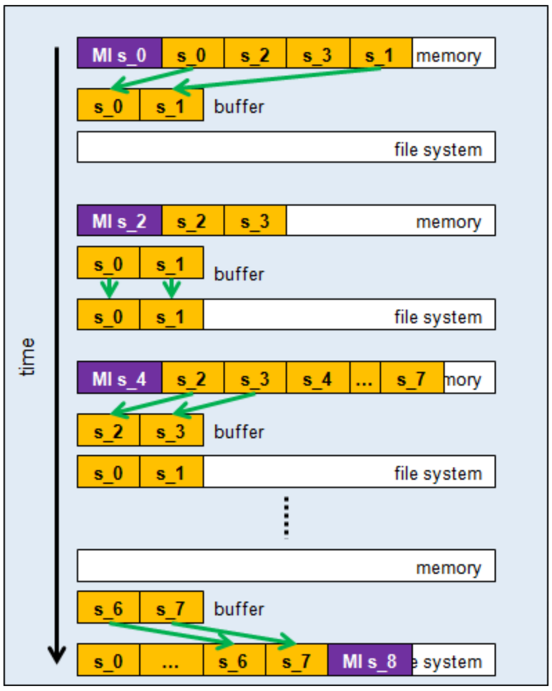
Figure 3.1: Sequential file transfer with resume capability

Figure 3.1 shows the extended behavior of the *ccnsimplecat* application introduced in Section 2.1.4. Figure 3.1(a) shows the request process in case of an Interest timeout. After n successfully received segments, the request for segment $n+1$ times out. Figure 3.2(b) illustrates the handling of the meta information. While the transfer makes progress the meta information is kept and updated in memory. In case of an Interest timeout the meta information is written to the **.meta** file.

3.3.2 ccncat_resume



(a) Requests



(b) Data processing

Figure 3.2: Pipelined file transfer with resume capability

Similar to *ccnsimplecat_resume*, Figure 3.2 shows the behavior of *ccncat* introduced in Section 2.1.4.2 with the resume capability feature. The request scenario considered in 3.2(a) is similar to Figure 3.1(a) but more than one segment is requested at the same time.
Figure 3.2(b) illustrates the data processing of the *ccncat_resume* application: Again meta information is stored in memory until an Interest timeout occurs. The difference between *ccnsimplecat_resume* and *ccncat_resume* is the moment of the meta information update. With *ccnsimplecat_resume* the next segment number to receive can be updated after every received segment, while with *ccncat_resume* one has to wait until an ordered block of continuous segments is received. So the meta information entry depends on the time of last block processing and not directly on the last received segment.

3.4 Data Structure for Meta Information

The meta information is a very important part to enable the resume capability. In this section we explain how we retrieve and update it in the file transfer applications.

Listing 3.1 shows the definition of the *metainfo* struct we have introduced to store the meta information within the transfer application. In order to ensure compatibility among both transfer applications, e.g. starting a transfer with pipelining using *ccncat_resume* and resuming it by *simplecat_resume* or vice versa, we use meta information which is equipped with all necessary parameters used for both application. Table 3.1 describes the function of the parameters in the *ccn_fetch* library.

```
struct metainfo {
        char ccn_name[MAX_URI_LENGTH];
        seg_t readSeg;
        intmax_t readStart;
        seg_t segsRead;
        long int partFileSize;
};
```

Listing 3.1: Meta information struct

field in stream struct	function in the fetch stream
readSeg	the segment for the current read position (next segment to receive)
segsRead	number of segments already read
readStart	the file read position at segment start

Table 3.1: Description of the stream information used in the ccn_fetch library

Since *ccnsimplecat_resume* uses less resume parameters we provide the missing information for *ccncat_resume* through a suitable mapping to enable interoperability between both applications. We will introduce this mapping in Section 3.5.1.

3.4.1 ContentName with Integrated Version Number

In CCNx each human readable ContentName is encoded to a binary sequence of components for internal use. Such a binary sequence of components enables an easier matching process between the ContentName in an expressed Interest and potential Data packet candidates. Because the human readable ContentName is a regular string of valid characters, it is used as filename for the partial data and meta information files. The different markers within the ContentName allow us to extract specific information from the string.

In this work, the version number is stored as part of the ContentName. We do not store it separately to avoid reassembling both fields again before every resume operation.

In the most common case a consumer requests data with a ContentName without any markers, so the CCNx framework adds the missing information about the version and the segment number automatically. In CCNx the order of the markers is always the same, so we can make assumptions about the appearance of the ContentName string. An example name string is shown in Figure 2.2 in Section 2.1.2 : The name prefix is followed by the version and finally the segment number is written. As a consequence we can take the segment marker as a stop indicator for the version information and extract the version string.

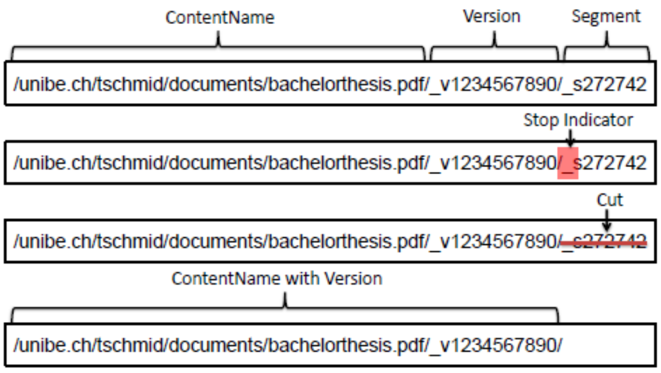

ContentName
Version
Segment
/unibe.ch/tschmid/documents/bachelorthesis.pdf/_v1234567890/_s272742
Stop Indicator
/unibe.ch/tschmid/documents/bachelorthesis.pdf/_v1234567890/_s272742
Cut
/unibe.ch/tschmid/documents/bachelorthesis.pdf/_v1234567890/~~_s272742~~
ContentName with Version
/unibe.ch/tschmid/documents/bachelorthesis.pdf/_v1234567890/

Figure 3.3: Example ContentName extraction process

Figure 3.3 illustrates with an example how a ContentName including version and segment information is processed to get only the necessary components of it.

3.4.2 Next Segment to Receive

The segment number following the version information in the naming string is already extracted by the existing file transmission applications in order to express subsequent Interests.

There are different ways of generating the number of the next segment to receive: In *ccnsimplecat_resume* we take the extracted segment number of the last received ContentObject and increment it. In *ccncat_resume* it is more complex due to the pipelining. The basic idea is to use the stream information which is generated to perform a pipelined transfer. The mechanism is explained in Section 3.5.2.

3.4.3 Position of the Partial Data File

The *ftell(...)* function of the C standard library is needed to get the file position of the partial file which is used to verify that the partial file has not been modified outside the transfer applications.

3.5 Implementation of the Resume Capability

There are two different implementations of the resume capability. In Section 3.5.1 we describe the implementation of *ccnsimplecat_resume* which implements the resume capability without pipelining. In Section 3.5.2 we introduce the implementation of the *ccncat_resume* applications which provides the resume capability with pipelining.

3.5.1 Resume Capability without Pipelining

3.5.1.1 ccnsimplecat

Figure 3.4 shows a flow chart of *ccnsimplecat*. First the application checks whether there is a syntactically correct ContentName. If the ContentName is valid, the version of the available ContentObjects is resolved and the first Interest is built. Otherwise the application terminates without transferring data.
When an Interest has been built, it can be expressed and processed. The part *"Process Interests for a given time"* describes very generally the main part of the transfer process. This part handles the content processing and consists of a processing function called *incoming_content(...)* and two data structures (*ccn_closure* and *mydata*) which we explain in the following paragraphs.

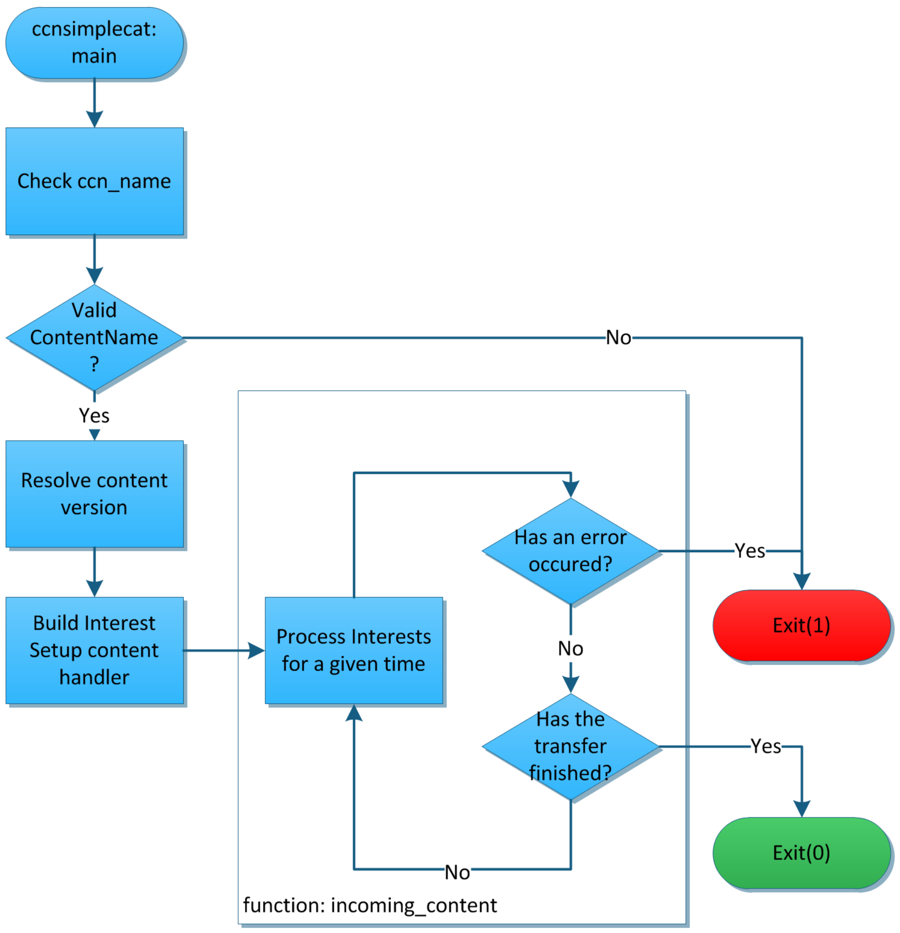


Figure 3.4: Application flow of ccnsimplecat

3.5.1.1.1 Processing of Incoming Content

The function *incoming_content(...)* will be called for every received segment.

In this function all processing, including writing data to the standard output, timeout handling and generating subsequent Interests is done. The function relies on data contained in two structs which we will introduce in the next paragraph.
Figure 3.5 illustrates the workflow of the *incoming_content(...)* function. The end states (which are illustrated as colored boxes on the right side) are return statements of the *incoming_content(..)* function except in the case of invalid data. An invalid data field results in an internal error and leads to immediate application termination. If the last expressed Interest times out it will be reexpressed in order to ensure that the requested data will arrive. Moreover the function checks whether the received segment is the final block. If the final block arrives the **done* flag in the *mydata* struct (see Listing 3.3) will be set to true and the data transfer is assumed to be finished. Otherwise there are still segments missing and the Interest for the next segment request will be built and expressed.

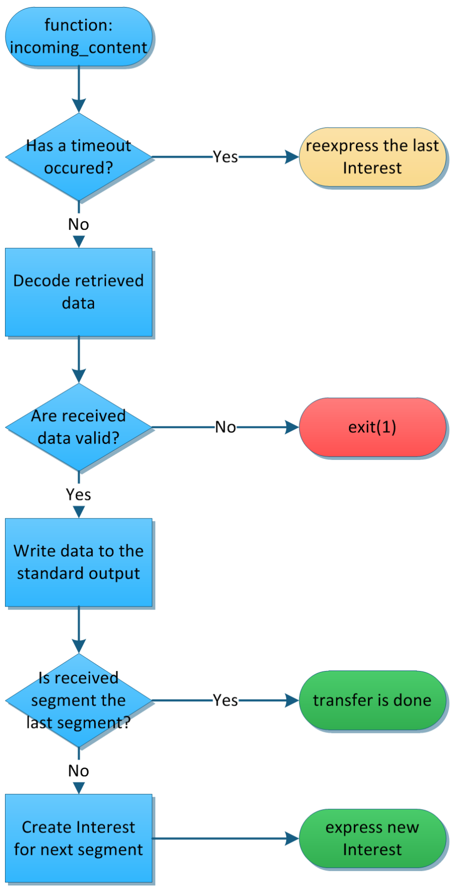

function: incoming_content
Has a timeout occured?
Yes
reexpress the last Interest
No
Decode retrieved data
Are received data valid?
No
exit(1)
Yes
Write data to the standard output
Is received segment the last segment?
Yes
transfer is done
No
Create Interest for next segment
express new Interest

Figure 3.5: Function flow of incoming_content(...)

3.5.1.1.2 Data Structures

The first structure is called *ccn_closure* and is shown in Listing 3.2. It contains data used by the *incoming_content(...)* function and is accessible from each call of *incoming_content(...)*.

The type *ccn_handler* hides a function pointer referencing the content processing function. In case of *ccnsimplecat* this is the *incoming_content(...)* function. Furthermore there is a void pointer called *data* which can be used to point to a data object on the heap. The application *ccnsimplecat_resume* introduces a structure for this case called *mydata* which is shown in Listing 3.3. Finally there is an integer field called *intdata* which is used to store the segment number of the last received segment.

```
struct ccn_closure {
    ccn_handler p;     /**< client-supplied handler */
    void *data;        /**< for client use */
    intptr_t intdata; /**< for client use */
    int refcount;      /**< client should not update this directly */
};
```

Listing 3.2: Struct: ccn closure

The struct *mydata* is used for storing states and flags which are accessible through the whole transfer process due to its allocation on the heap. The integer field **done* is used to store whether the final block has arrived or not. A value of 1 indicates that the application may terminate with success. Another integer field is called *allow_stale* and it contains a boolean value (0 or 1) indicating if stale data is allowed.

```
struct mydata {
    int *done;
    int allow_stale;
};
```

Listing 3.3: State information

3.5.1.2 Adding Resume Capability

Figure 3.6 shows the application work flow of the *ccnsimplecat_resume* application as an extended version of *ccnsimplecat*.

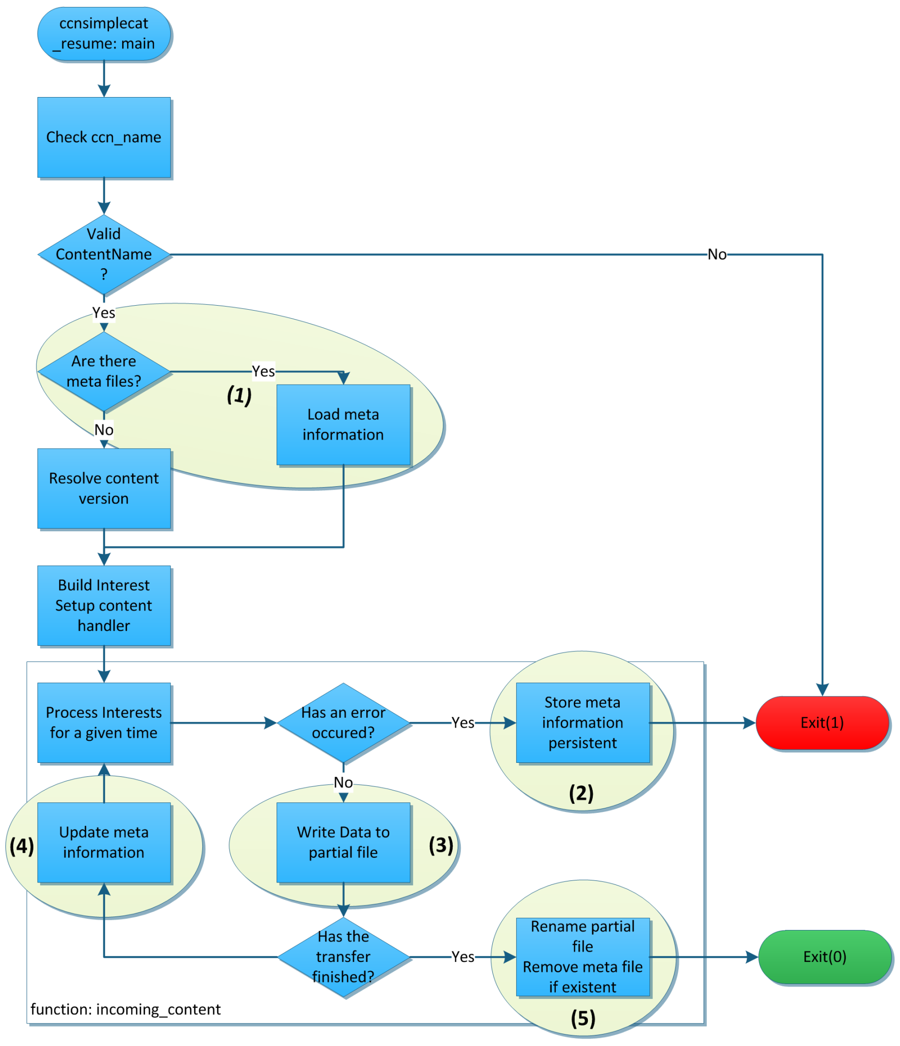
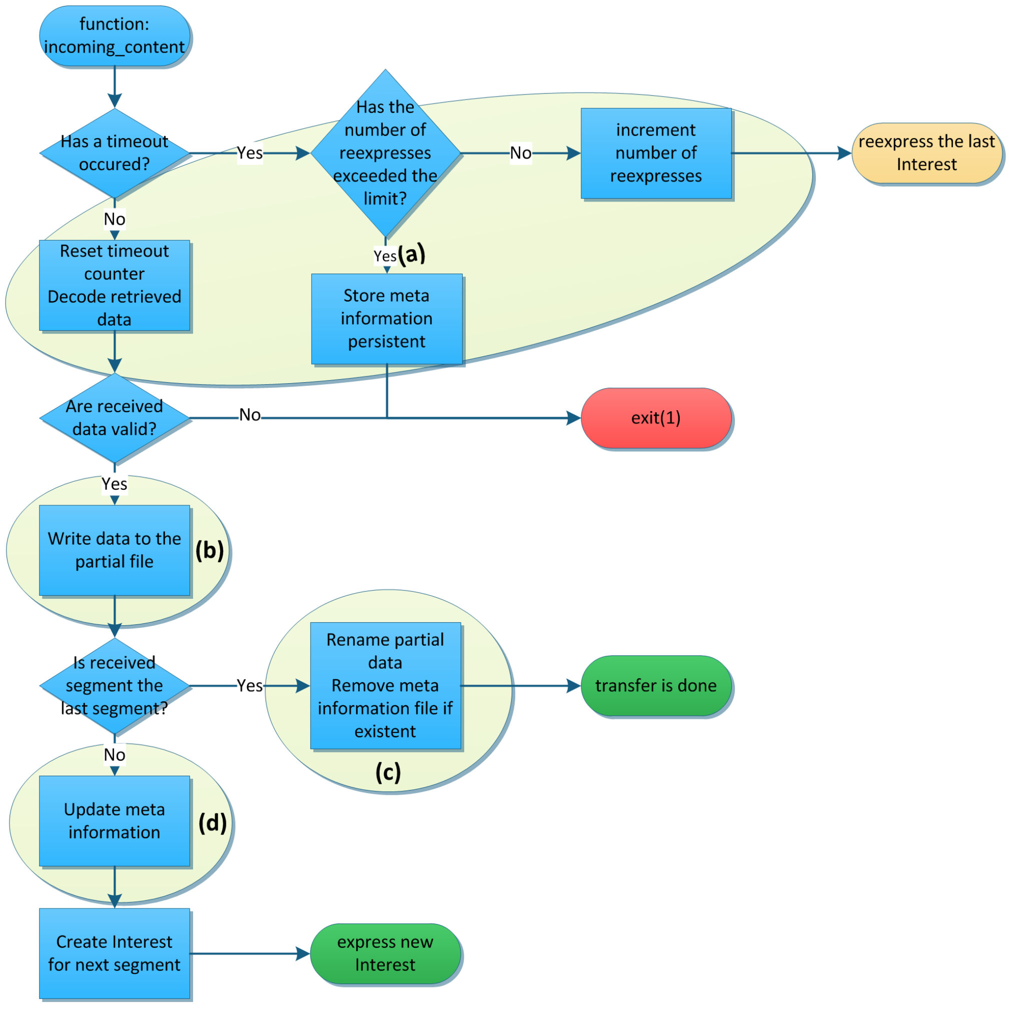

ccnsimplecat _resume: main
Check ccn_name
Valid ContentName ?
No
Yes
Are there meta files?
Yes
(1)
Load meta information
No
Resolve content version
Build Interest Setup content handler
Process Interests for a given time
Has an error occured?
Yes
Store meta information persistent
(2)
Exit(1)
No
Update meta information
(4)
Write Data to partial file
(3)
Has the transfer finished?
Yes
Rename partial file Remove meta file if existent
(5)
Exit(0)
function: incoming_content

Figure 3.6: Application flow of ccnsimplecat_resume

The first change in the work flow (see (1) in Figure 3.6) is the check whether the transfer could be resumed or has to be started from scratch. The application searches for available partial files and meta information filename**.part** and filename**.meta** introduced in Section 3.2.

If both of these files exist the initial Interest is built upon the loaded meta information. Otherwise the application has to resolve the version of the requested ContentObject using the *ccn_resolve_version(...)* function and the initial Interest starts with segment number zero which indicates the ContentObject is requested from the beginning. After this case distinction the Interest can be expressed.

The circles (2), (3), (4) and (5) in Figure 3.6 emphasize changes in the *incoming_content(...)* function which we show in Figure 3.7.



function: incoming_content
Has a timeout occured?
Yes
Has the number of reexpresses exceeded the limit?
No
increment number of reexpresses
reexpress the last Interest
No
Reset timeout counter Decode retrieved data
Yes
(a)
Store meta information persistent
Are received data valid?
No
exit(1)
Yes
Write data to the partial file
(b)
Is received segment the last segment?
Yes
Rename partial data Remove meta information file if existent
(c)
transfer is done
No
Update meta information
(d)
Create Interest for next segment
express new Interest

Figure 3.7: Function flow of incoming_content(...) with resume capability

Already in the first step of the work flow changes have been made. The timeout handling has been completely revised (see (a) in Figure 3.7): Instead of reexpressing the Interests for an infinite number of times, we have introduced a reexpression limit. If the limit is reached, the application stores the meta information into the meta information file and terminates afterwards (see (a) in Figure 3.7). This limit is useful to distinguish short from long connection interrupts. Short connection interrupts are caused by fluctuating signal strength or packet loss, so after one to three reexpressions the connectivity should be rebuilt and the transfer can continue. On the other hand during a long connection interrupt an Interest reexpression does not help because the content source is not reachable.

The next change affects the data output (see (b) in Figure 3.7): Instead of piping the whole data stream to the standard output, we implement a native file storing function that allows to store data on a persistent storage device. We store the partial data and meta information file in the current working directory by default. So the user has to start the transfer application from the same working directory as before otherwise the application is not able to find the partial data and meta information.

The next change (see (c) in Figure 3.7) affects the part which checks whether the last received data chunk is already the last segment of the file indicated by a final block identifier or if there are more segments to be received. If the received segment is the last segment, clean up operations are performed. These clean up operations include the deletion of existing meta data files and renaming the data file by removing the **.part** extension. If it is not the last segment, the meta information in the memory needs to be updated (see (d) in Figure 3.7) and the Interest for the next segment to receive has to be built.

The retrieved meta information is stored in the struct *metainfo* as explained above (see Listing 3.1). Because the same struct is used in the main method and in the *incoming_content(...)* function it is allocated on the heap.
To use the struct within the *incoming_content(...)* function a pointer is needed to know where on the heap the struct is allocated. Because the struct *mydata* (see Listing 3.3) is everywhere accessible within the *incoming_content(...)* function, we extended it with a pointer to the *metainfo* struct.

The ContentName including the version of the transferred file does not change during the data transmission. Therefore the ContentName with the version will be extracted at the beginning. After that, the information will be loaded into the *metainfo* struct.

The meta information will be updated within the *incoming_content(...)* function. We can get the next segment to receive using the incremented *intdata* field of the *mydata* struct which stores the segment number of the last transferred segment.

As there are five fields to fill but just three values necessary for the *ccnsimplecat_resume* application there exists a mapping of these values (see Table 3.2).

field in metainfo struct	**data in ccnsimplecat_resume**
ccn_name	ContentName with version
readSeg	next segment to receive
segsRead	next segment to receive
readStart	partial file size
partialFileSize	partial file size

Table 3.2: Mapping of the meta information value in ccnsimplecat_resume

After updating the meta information the Interest for the next segment to receive will be built and expressed.

3.5.2 Resume Capability with Pipelining

3.5.2.1 ccncat

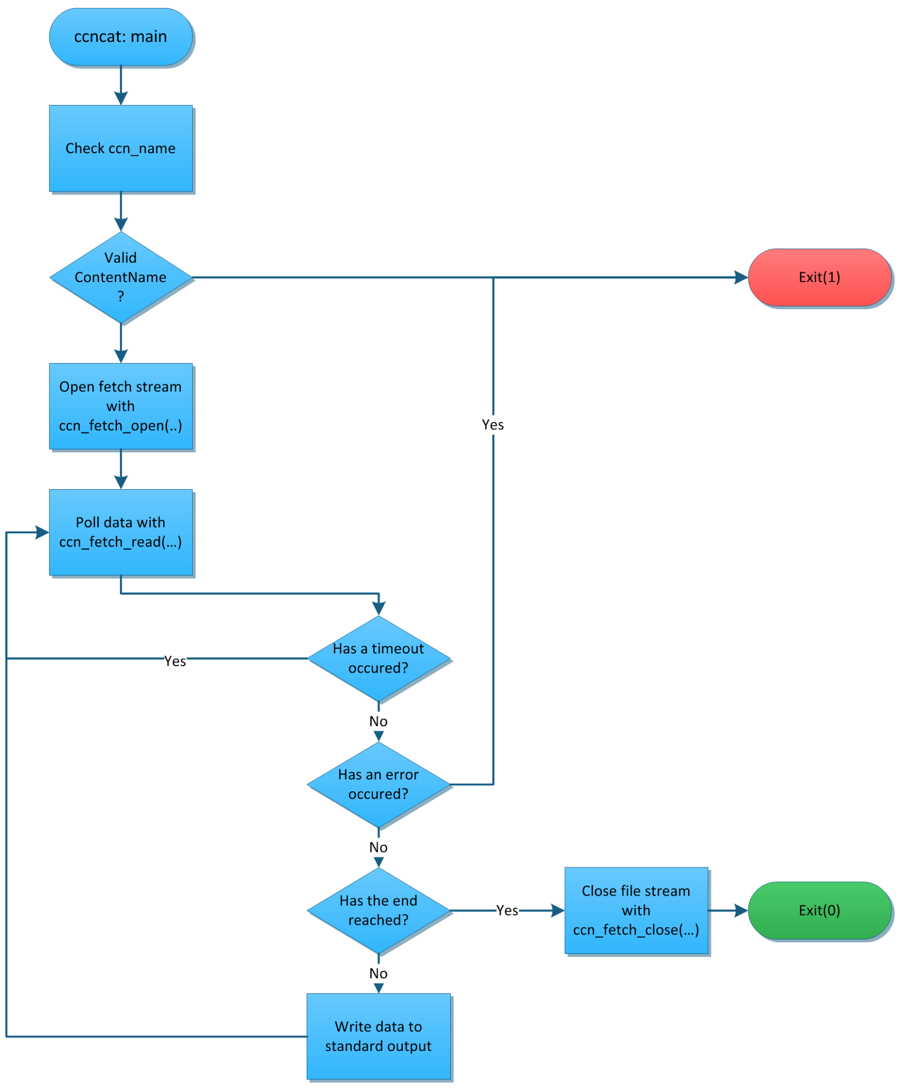
Comparing *ccncat* to *ccnsimplecat* shows us that the *ccncat* application is implemented on a higher abstraction level. The *ccncat* transfer application uses the *ccn_fetch* library which is providing streaming access for fetching segmented CCNx data including pipelined transfer, while *ccnsimplecat* implements the whole stream processing in a single function.

The *ccn_fetch* library provides three important functions, i.e. *ccn_fetch_open*, *ccn_fetch_read* and *ccn_fetch_close*. They are similar to commonly known file handling functions (*fopen(...)*, *fread(...)*, *fclose(...)*).

ccn_fetch_open
: The function *ccn_fetch_open(...)* creates a fetch stream struct for a given ContentName. More information about the fetch stream struct can be found in Appendix 6.1.

ccn_fetch_read
: The *ccn_fetch_read(...)* function fills a buffer with data received by the fetch stream and returns a positive integer value for the number of bytes fetched or a negative integer in case of an error. Because the buffer size is limited, only a small part of the data can be processed at the same time. Therefore, this operation is repeated periodically until the entire ContentObject is transferred.

ccn_fetch_close
: To close a fetch stream properly the function *ccn_fetch_close(...)* has to be called. Since the fetch stream contains pointers to memory allocated on the heap, clean up is necessary to avoid memory leaks.



ccncat: main
Check ccn_name
Valid ContentName ?
Exit(1)
Open fetch stream with ccn_fetch_open(..)
Yes
Poll data with ccn_fetch_read(...)
Yes
Has a timeout occured?
No
Has an error occured?
No
Has the end reached?
Yes
Close file stream with ccn_fetch_close(...)
Exit(0)
No
Write data to standard output

Figure 3.8: Application flow of ccncat

Figure 3.8 shows the work flow of the *main* routine of the *ccncat* application. The routine starts with a syntactical check of the ContentName parameter. If the ContentName is valid a new fetch stream will be opened with the *ccn_fetch_open(...)* function for the requested ContentName. If this stream has successfully been created, data could be read by polling *ccn_fetch_read(...)* within a while loop.

Unlike in the *ccnsimplecat* application, error handling is done in the main method specifically within the while loop of the polling process. Errors which could possibly occur are encoded with a specific number. So it is possible to handle different cases.

When data is received it is written to the standard output and the poll process can proceed.
Finally after leaving the while loop, the opened stream has to be closed properly with the *ccn_fetch_close(...)* function.

3.5.2.2 Adding Resume Capability

To set up a new fetch stream based on captured meta information, we have to adapt the underlying *ccn_fetch* library.

Because the function *ccn_fetch_open(..)* creates a fetch stream starting a transfer from the beginning, we have created a new function called *ccn_fetch_open_resume(...)* which allows to create a fetch stream based on meta information. The new function extends the existing *ccn_fetch_open* function with more parameters. To avoid a very long parameter list we hand over the whole *metainfo* struct (shown in listing 3.1) since almost all of the stored information is used for creating the fetch stream.

The changes of *ccn_fetch_open_resume(...)* compared to *ccn_fetch_open(...)* include the following points:

- The version resolving is no longer necessary since we can take the ContentName with version from the meta information
- Several stream information have to be set based on the meta information:

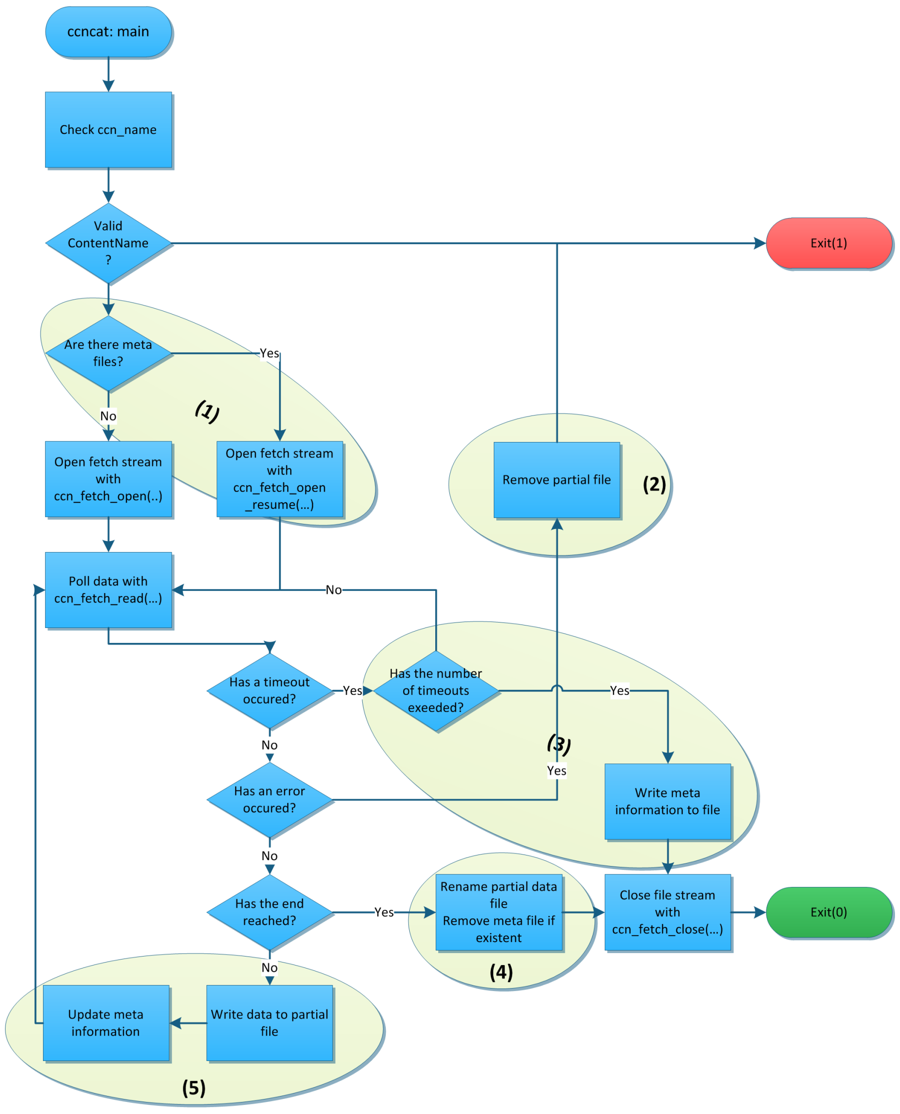
```
+ fs->readSeg = minfo.readSeg;
+ fs->readStart = minfo.readStart;
+ fs->readPosition = minfo.readStart;
+ fs->segsRead = minfo.segsRead;
```

- The stream has to be started from the segment saved in the readSeg variable instead of zero.

```
+ NeedSegment(fs, fs->readSeg);
- NeedSegment(fs, 0);
```

Figure 3.9 illustrates the adapted application work flow for the *ccncat_resume* transfer application.

In *ccncat_resume*, different from *ccncat*, the application searches for meta files (see (1) in Figure 3.9). If there are no meta information files available the stream will be created with the *ccn_stream_open* function starting at segment number zero as in the original *ccncat* application. Otherwise the meta information contained in the file is loaded into the *metainfo* struct as introduced in listing 3.1. After that the fetch stream will be created with the *ccn_fetch_open_resume* function.



ccncat: main
Check ccn_name
Valid ContentName ?
Exit(1)
Are there meta files?
Yes
No
(1)
Open fetch stream with ccn_fetch_open(..)
Open fetch stream with ccn_fetch_open _resume(...)
Remove partial file
(2)
Poll data with ccn_fetch_read(...)
No
Has a timeout occured?
Yes
Has the number of timeouts exeeded?
Yes
(3)
Yes
No
Has an error occured?
Write meta information to file
No
Has the end reached?
Yes
Rename partial data file Remove meta file if existent
Close file stream with ccn_fetch_close(...)
Exit(0)
(4)
No
Update meta information
Write data to partial file
(5)

Figure 3.9: Application flow of ccncat_resume

The desired data can be retrieved by opening a fetch stream. As mentioned above we have to poll the *ccn_fetch_read(...)* within a while loop for new data fragments. The delivered data gets copied into a buffer and the function returns a value. If the return value is a positive integer the number indicates the number of bytes that are newly added to the empty buffer. As shown at point (5) in Figure 3.9, instead of printing it to the standard output we can save it to the partial data file and we have to update the *metainfo* struct with the current stream state. Therefore we copy the values stored in the stream struct (Table 3.1) to the *metainfo* struct (Listing 3.1).

Otherwise if the polling fails, we get a negative return value. The negative return value corresponds to an error code as shown below.
We compare the different actions between *ccncat* and *ccncat_resume*.

-2 = CCN_FETCH_READ_TIMEOUT

Reason: some of the Interests timed out
Action without resume: reexpress the timed out Interests and poll again
Action with resume: If timeout limit has not exceeded, reexpress the timed out Interests and poll again. Otherwise save the meta information persistently to a file, leave the while loop and terminate the application without a completed transfer (As shown at point (3) in Figure 3.9).

-4 = CCN_FETCH_READ_END

Reason: file has been successfully transferred
Action without resume: leave the polling loop
Action with resume: leave loop, rename the partial data file and remove the meta information file (As shown at point (4) in Figure 3.9).

other error codes < 0

Reason: fatal stream error
Action without resume: terminate application with exit code 1
Action with resume: we cannot guarantee that the partial data is in a consistent state so we remove the partial data file as well as the meta information file (if any) and terminate application with exit code 1 (As shown at point (2) in Figure 3.9).

After leaving the while loop we have to close the file stream properly, so that we can terminate the application successfully.

Chapter 4

Evaluation

In this chapter we want to measure the efficiency of the developed transfer applications.

In particular, we would like to answer the following questions:

- Are there any benefits of using the new file transfer applications with resume capability?
- What are the advantages of pipelined file transfers compared to sequential single segment file transfers?
- Does the block size influence the completion time of file transfers?
- When is it useful to resume an interrupted transfer and when should we drop the received data and start the transfer from scratch?

4.1 Application Scenario

The evaluation was performed based on an application scenario shown in Figure 4.1. There are two participants in this scenario: one of them is the content source, the other one consumes data from the content source. The consumer occasionally loses connectivity to the content source, i.e. corresponding to users losing connectivity because of mobility.

The following scenario is defined for wireless networks but can easily be adapted to wired networks by unplugging the network cables and temporarily leaving the local network. Based on this scenario we created a script to obtain test results and measured the transfer time for different file sizes. The test script runs on the consumer device and will be explained in section 4.3.

Below the resume scenario is described in four steps.

Step 1
The mobile devices have to be in connectivity range of each other (green circle) and the consumer requests data by sending Interests. The content provider serves the mobile device with data as long as the requested data exists in the repository.

Step 2
The mobile device leaves the serving range of the content provider but is continuously sending Interests for the not yet completed file transfer. The content provider does not get any of these Interests because the mobile device is too far away and consequently does not return any data objects.

Step 3
After a specific number of unsuccessful Interest requests without receiving data in return the mobile device is in a timeout state and the transfer will be interrupted.

Step 4
Later, the mobile device enters the connectivity range of the content provider again and requests the data again. In this step the transfer will be finally completed.

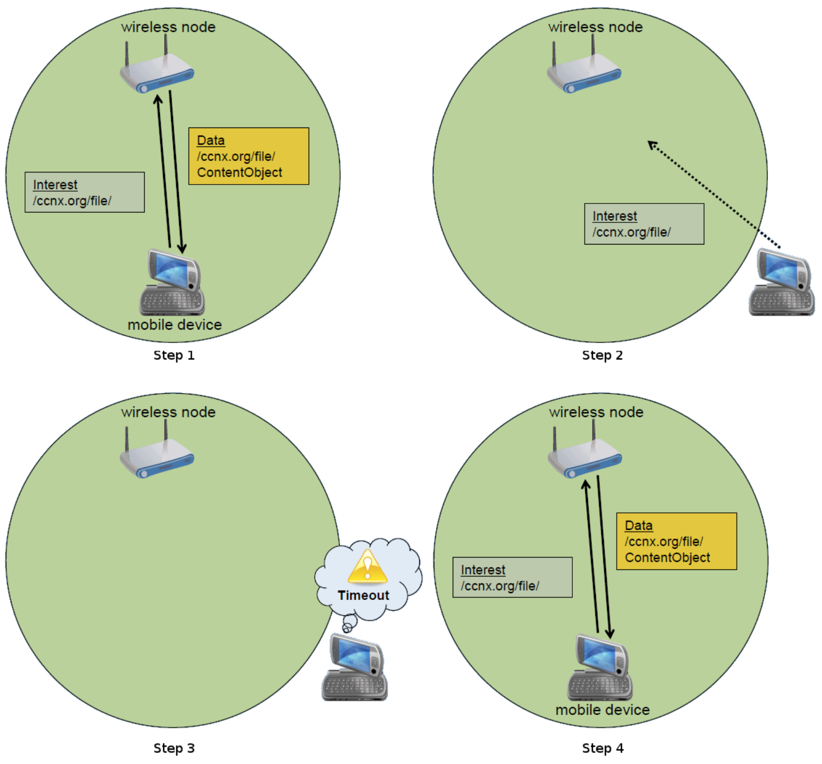


Figure 4.1: Application scenario explained in four steps

4.2 Hardware / Technology

The implementation was tested on PCEngines ALIX boards [5] using the ADAM operating system described below with the CCNx extension specified in the Technical Appendix 7.1.2. In the following we refer to these ALIX boards as mesh nodes.

Such a mesh node contains a mainboard with an AMD Geode CPU, two wireless LAN cards and a flash card used as storage device. Detailed hardware information is provided in the Technical Appendix 7.1.1.

With this hardware we are able to test different transfer scenarios using wireless and wired connections.

4.2.1 ADAM Operating System

ADAM[6] is the acronym for Administration and Deployment of Adhoc Mesh networks. The ADAM operating system provides a build environment to configure and deploy certain mesh networks. It is based on the Linux kernel and is designed to run on different embedded architectures. Due to this goal the number of features is limited, nevertheless the operating system provides all of the necessary tools used in mesh networks. This approach allows to keep the image of the operating system small so that it can be directly run out of the memory or does not need a lot of disk space. The build environment is package based and supports an easily integration of additional features and tools.

4.3 Testing Setup

In this section we describe the testing setup. There is one mesh node which shares content through a CCNx repository service and another node which consumes the data. Because it is difficult to control disconnections due to mobility on embedded systems, we implemented an evaluation mode to test the transfer applications. The evaluation mode requires a command line parameter for the breakpoint and is enabled on the requester. The breakpoint value is indicating after which number of transmitted segments a connection timeout occurs. Additionally, we compare the wireless transmission results with wired transmissions on the same nodes.

Figure 4.2 gives an overview of the configuration of the different network interfaces. For the main tests a wireless adhoc network with the IEEE802.11a standard has been used. Additionally a wired connection over a Fast-Ethernet (100MBit) device was configured. The wired connection was particularly used to start the test script over an SSH connection. Therefore, the wireless interface was only used for CCNx traffic while control commands from the control station were transmitted over the Ethernet cable. Furthermore, the control station is used to start the test script.

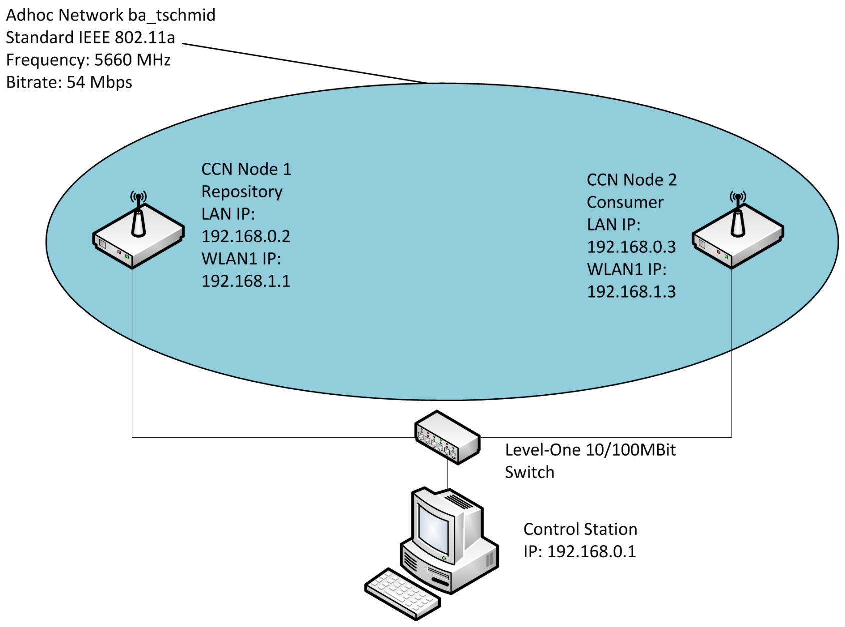


Figure 4.2: Applied testing architecture with mesh nodes

The content source was running the CCNx repository *ccnr* which was prepared with data files containing randomly generated data. For example a 1MB file can be generated with the following command:

```
dd if=/dev/urandom of=/path/to/random/1MB bs=1024 count=1024
```

Listing 4.1: Command to generate random files

After creation, the file has been written to the CCNx repository with different block sizes using the *ccnseqwriter* command. For example, for a block size of 1024 bytes we run the following command:

```
ccnseqwriter -b 1024 -r ccnx://ccnx.org/filename
```

Listing 4.2: Insert files into CCNx repository

The test script was running on the consumer node simulating the scenario introduced above.
The test script does the following:

1. Start the transfer with one of the developed transfer applications and request the complete file. Simulate a connection loss with an evaluation parameter after a configurable number of received segments. Measure the transfer time of the first transfer until the breakpoint segment arrives and write it into a variable. On the contrary to the real world after the simulated connection loss no more additional Interests will be expressed therefore the requester does not receive any additional segments.

2. After a certain delay (see CCND_DEFAULT_TIME_TO_STALE=1 below) start the transfer again and run the application until the file transfer has been completed. Measure the transfer time of the second transfer and save it to a variable.

3. Calculate the total transfer time using both the variables and write the result as well as the current evaluation parameters into a result file.

4. Repeat step 1 to 3 for a configurable number of times

5. Adapt the breakpoint by a configurable value and start again with step 1.

The configuration of the CCNx instances on the mesh nodes contains the following settings:

CCND_DEFAULT_TIME_TO_STALE=1
In order to ensure that the consumer always consumes Data packets out of the CCNx repository and not a cache, the default stale time is set to 1 second. This means that received data has a life time of one second in the cache. Additionally, a delay of one second has been inserted between both transfers of the test script to guarantee that already received ContentObjects are stale before the second transfer starts.

CCND_DEBUG=0
The debug output has been disabled to avoid performance waste on the resource constrained ALIX devices.

Unicast face allocation
The current version of the CCNx framework implements the data transfer on the application layer above the IP stack. Because of that the CCNx daemon has to know on which IP address and port number he has to listen and respond to incoming Interests. For this case we have to manually configure the IP address and port for each mesh node in the test scenario.

4.4 Results

In this section we compare the results of the evaluation scenario described in section 4.1-4.3. In subsection 4.4.1 we analyze the impact of different block sizes on the transfer time. In subsection 4.4.2 we have a look at the influences of different pipeline sizes on the transfer time. After that in subsection 4.4.3 we analyze the differences of the transfer time using different network connections, more precisely wired and wireless connections. Finally in subsection 4.4.4 we analyze the overhead of processing and holding the meta information to resume a file transfer.

4.4.1 Influence of the Block Size

In this section the transfer applications *ccnsimplecat* and *ccnsimplecat_resume* which are providing sequential file transfer with pipeline size of 1 will be examined. In particular, the influence of different block sizes is compared to transfer times of file transmissions. Table 4.1 shows the test setup used to evaluate the results shown in this subsection.

parameter	**unit**	**value**
connection type		wireless unicast
block size	Bytes	1024, 2048, 4096
file size	MB	5
number of breakpoints [1]		10
runs per breakpoint [2]		10

Table 4.1: Test cases ccnsimplecat vs. ccnsimplecat_resume

Figure 4.3 shows the comparison of *ccnsimplecat* and *ccnsimplecat_resume* when transferring a 5MB file using different block sizes. The y-axis shows the average total transfer time of the file over a wireless IEEE 802.11a link. The total transfer time consists of the sum of the first incomplete transfer time until the breakpoint segment arrived and the second transfer time until the transfer completion as explained in section 4.3. The x-axis shows the breakpoints converted to kilobytes which indicates the points where the application simulates the connection timeout during the first transfer try. The vertical bars drawn over the data points are showing the standard deviation of the test results which have been small for all of the test cases.

[1]The breakpoint value denotes how many segments have been transmitted until a connection loss is simulated. The value in the table indicates the number of different breakpoints tested in a test case.

[2]This value indicates how many transfer times have been measured per breakpoint.

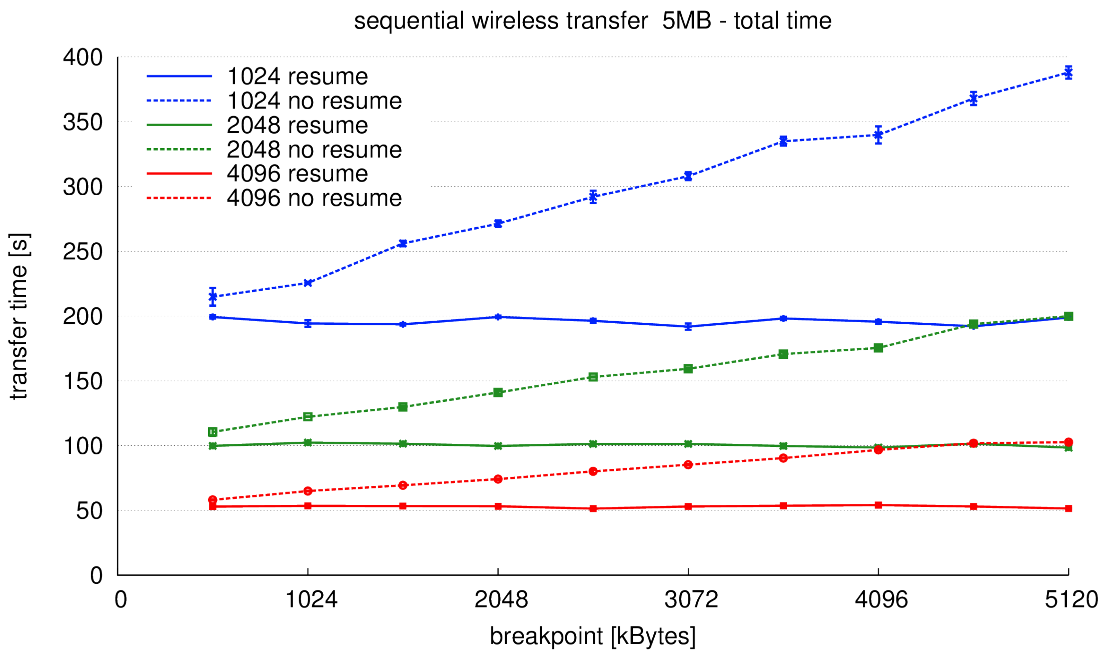


Figure 4.3: ccnsimplecat vs. ccnsimplecat_resume

The following properties can be observed:

1. Independent of the breakpoint, the total transfer time of the *ccnsimplecat_resume* application is constant. Because of the stored partial data we do not have to request already received data chunks again. In contrast, the time cost of *ccnsimplecat* increases the later the download stops. In the worst case of the given test scenario, the download gets interrupted immediately before the transfer has finished. This causes that almost twice the amount of data has been transmitted and therefore it requires twice the amount of time.

2. The block size of the transferred segments has a significant impact on the transmission speed. When using small block sizes, the overhead of CCNx, e.g., CCN header, signatures, etc, for data transmissions is greater. Although larger block sizes result in fragmentations at the IP layer, larger block sizes still result in shorter transfer times. Moreover the number of transmitted packets is larger when using small block sizes which results in processing overhead due to the verification of more signatures.

The current version of the CCNx framework uses a block size of 1024 bytes as default value. Moreover the maximal block size is 4096 bytes with CCNx version 0.6.

4.4.2 Influence of the Pipeline Size

In this subsection the influence of the pipeline size on the file transfer time will be analyzed. For this case the *ccnsimplecat* and *ccncat* applications have been tested according to test scenario 4.1 with and without resume capability.
Table 4.2 summarizes the parameters used to analyze the influence of the pipeline size on the transfer time of a 5MB file.

parameter	**unit**	**value**
connection type		wireless unicast
pipelines		2,4,8,16
block size	Bytes	4096
file size	MB	5
number of breakpoints		10
runs per breakpoint		10

Table 4.2: Test cases ccncat vs. ccncat_resume

Figure 4.4 compares *ccncat* and *ccnsimplecat* with and without resume capability over wireless IEEE802.11a links. The *ccncat* application is evaluated with different pipeline sizes while *ccnsimplecat* uses sequential file transfers corresponding to a pipeline size of one.

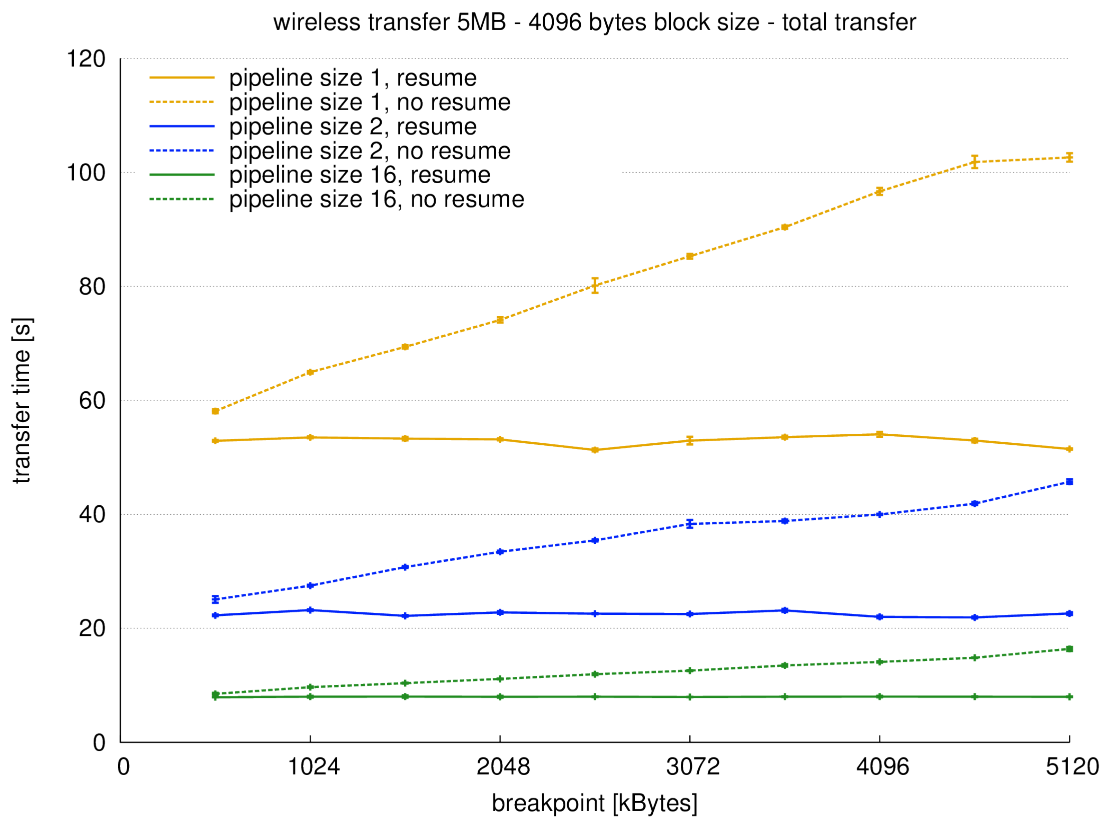


Figure 4.4: ccnsimplecat vs. ccncat with / without resume capability (pipeline sizes 1, 2, 16)

The y-axis denotes the total transfer time of a 5MB file according to the scenario described in section 4.1. The x-axis denotes again the breakpoints in kilobytes. In order to have a better overview in Figure 4.4 only the results of the pipeline sizes 1, 2 and 16 using a block size of 4096 bytes are shown.

Figure 4.5 shows the transfer times of the applications with resume capability using 1, 2, 4, 8 and 16 pipelines. The axis labeling is equal to the labeling of Figure 4.4 above.

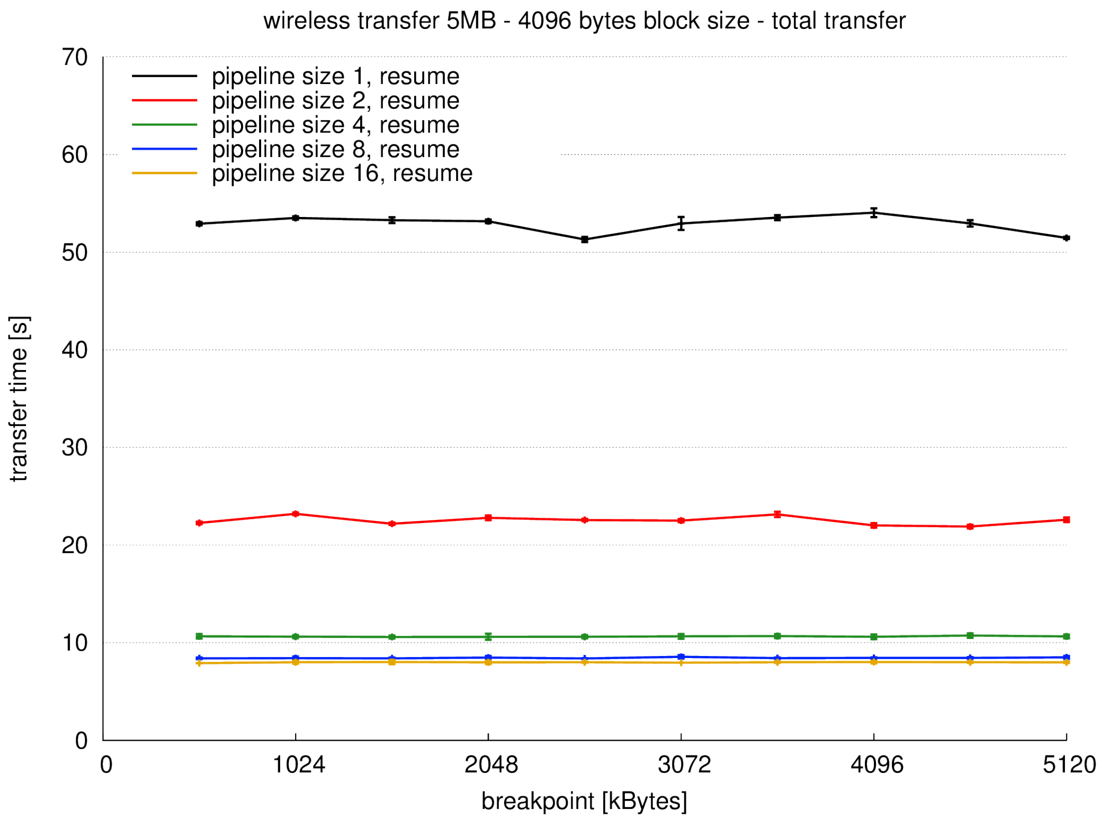


Figure 4.5: ccnsimplecat_resume vs. ccncat_resume

From Figure 4.4 and Figure 4.5 we conclude the following:

1. As already seen in Figure 4.3 the transfer time of the application with resume capability is constant.

2. In general, increasing the pipeline sizes results in shorter transfer times. Doubling the pipeline from 1 to 2 and 2 to 4 halves the transfer time. Adapting the pipeline size from 4 to 8 or 8 to 16 leads to a smaller performance gain as from 2 to 4. A reason for that is the processing overhead for the signature verification on the mesh nodes. In CCNx each Data packet is signed so that receivers can verify that the packet has been sent from the desired publisher. Therefore, every receiver has to validate the signature of each received packet which causes some processing overhead. On resource constrained devices like on ALIX boards processing power is limited. No matter how fast the network connection is, there exists a point where the CPU is not able to process all of the received ContentObjects and send new Interest for subsequent segments at the same time. Another reason is the oc-

cupancy of the wireless medium. Increased medium occupation results in longer waiting time and more collision.

Pipeline sizes larger than 16 were not tested because this is a hard coded upper limit in the CCNx source code.

Table 4.3 shows the relative performance gain of pipelined file transfer. To calculate this relative values the mean of the transfer times for each breakpoint of the *ccncat_resume* application has been taken. The performance is always based on the sequential file transfer which is denoted with pipeline size 1. The performance gain value shows the enhancement factor of the data rate per time. The time gain indicates how much time has been saved compared to a pipeline size of 1 and can be calculated with the following formula:

$$\text{time gain} = 100\% - \frac{\text{transfer time with pipeline size x}}{\text{transfer time with pipeline size 1}} \cdot 100\%$$

Therefore a time gain of 100% denotes an instant transfer with a transfer time of zero seconds.

pipeline size	**avg transfer time**	**performance gain**	**time gain**
1	52.91	100.00%	0.00 %
2	22.52	234.97%	57.44 %
4	10.64	497.26%	79.89 %
8	8.44	626.65%	84.05 %
16	7.99	662.34%	84.90 %

Table 4.3: Performance gain of the resume capability using multiple pipelines

Viewing the values above again illustrates the fact that pipeline sizes bigger than 8 do not yield a similar performance gain as from 1 to 2 and 2 to 4.

4.4.3 Influence of the Network Connection

In this subsection the influence of different network connections on the transfer time will be analyzed. The tests are performed over wireless IEEE802.11a and wired Fast-Ethernet (100MBit) links.
Table 4.4 shows the parameters used to analyze the influence of network connection on the transfer time of a 5MB file.

parameter	**unit**	**value**
connection type		wireless / wired unicast
pipelines		16
block size	Bytes	4096
file size	MB	5
number of breakpoints		10
runs per breakpoint		10

Table 4.4: Test cases wired and wireless

Figure 4.6 shows the results when using a block size of 4096 Bytes and a pipeline size of 16 for wired and wireless transmissions.

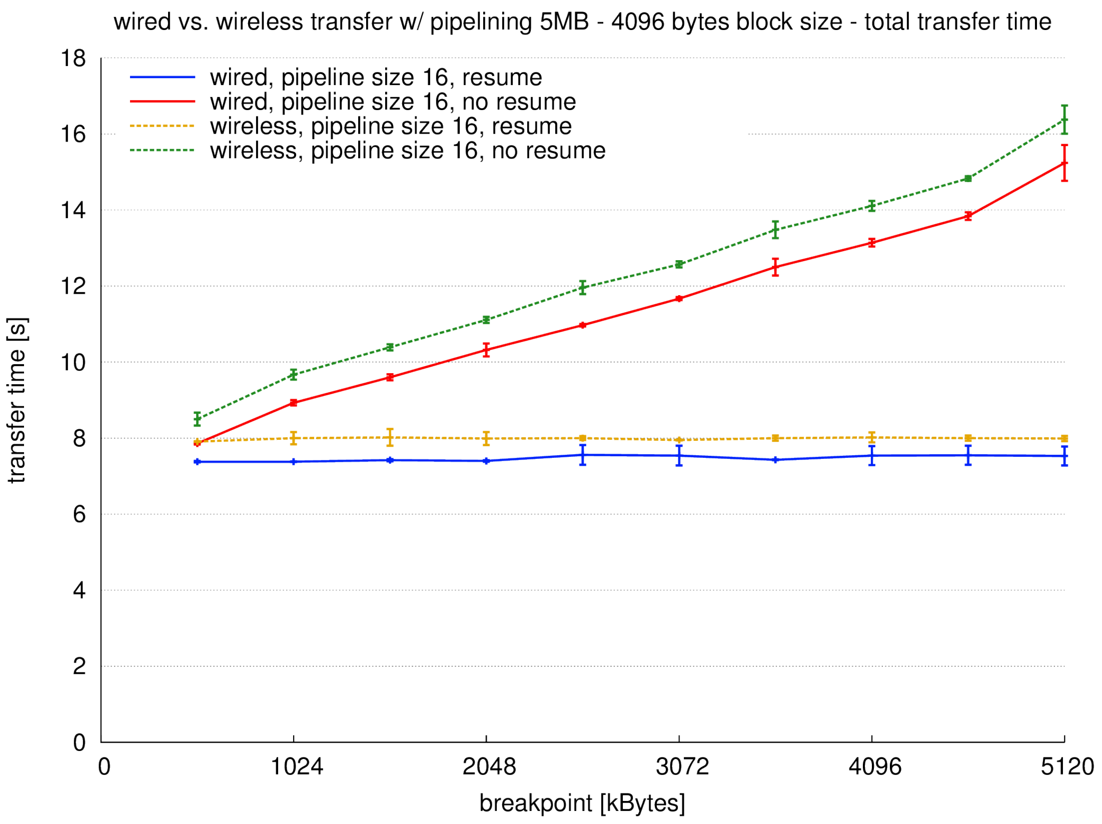


Figure 4.6: wireless vs. ethernet with pipelining

The x-axis denotes the breakpoints and the y-axis shows the total transfer time of a 5MB file in seconds. As already seen above, the transfer time of the application with resume capability is constant while the application without resume has linear growing of transfer times with increasing breakpoints.

The evaluation showed that the difference in transfer times between wired and wireless unicast transmission is small. Although the maximal bandwidth of a fast-ethernet connection (100MBit) is almost double the maximum bandwidth of an IEEE802.11a wireless connection (54MBit), the performance gain in using ethernet over wireless connection is only minimal. As explained in section 4.4.2 the signature verification mechanism is the bottleneck which limits the data throughput on resource constrained devices.

4.4.4 Processing Overhead

In this subsection the processing overhead of applications with resume capability will be determined. Most of the additional actions happen at application start, therefore the actions which may produce overhead are only executed once per transfer. The list of the additional actions is shown below:

Application start

- Check for partial data and meta information file
- Allocate meta information struct on the heap
- String parsing to obtain the (partial) filename
- Open or create partial data file

During the transfer

- Write received data chunks to the partial file
- Update meta information in memory after receiving data chunks

After the transfer

- Remove the meta information file if there is any
- Rename partial file

To analyze the impact of the additional actions the transfer time for transferring a whole file at once without disruption has been measured with all four existing file transfer applications. This step has been repeated 100 times to calculate the mean transfer time and the standard deviation.

Table 4.5 shows the different test cases with the associated results.

application	file size	block size	pipelines	avg. transfer time	std. dev.
ccncat	5 MB	4096	16	8.08s	0.11
ccncat_resume	5 MB	4096	16	8.09s	0.12
ccncat	10 MB	4096	16	15.78s	0.49
ccncat_resume	10 MB	4096	16	15.77s	0.50
ccnsimplecat	5 MB	4096	1	56.14s	0.26
ccnsimplecat_resume	5 MB	4096	1	56.69s	0.22
ccnsimplecat	10 MB	4096	1	112.47s	0.36
ccnsimplecat_resume	10 MB	4096	1	112.56s	0.68

Table 4.5: Transfer time for transferring a file without disruptions

The values in table 4.5 indicate that the applications with resume capability do not require significant processing overhead. The transfer times and the standard deviation of the *ccncat* applications differ only in the second decimal digit while the *ccnsimplecat* applications differ in the first decimal digit. A reason for this behavior is the fact that meta information will only be written to a file when a connection timeout occurs. Holding and updating meta information in memory does not take much more time. Another reason is the size of the *metainfo* struct. The meta information fields in this struct are limited by its type definitions in C. Because of this type definitions the meta information can not be greater than 2032 bytes on a regular *x86* machine.

4.4.5 Discussion

After analyzing the results we are able to answer the questions formulated at the beginning of the chapter.

Are there any benefits of using the new file transfer applications with resume capability?

The effective transfer time[3] using the resume capability for transferring a file with disruptions requires approximately the same time as the transfer of the entire file without disruptions. Compared to the applications without resume capability, the transfer time of the new applications with resume capability is independent of the breakpoint. Without the resume capability the transfer time increases with a later breakpoint position.

What are the advantages of pipelined file transfers compared to sequential single segment file transfers?

Using pipeline size four or higher leads to a huge performance increase compared to sequential single segment transmissions. Therefore, it is beneficial to use the *ccncat* or *ccncat_resume* application to perform fast file transfers. Though the performance gain with a pipeline size of 8 to 16 is not as much as up to a pipeline size of 4.

Does the block size influence the completion time of file transfers?

The choice of the block size affects the transfer time considerably. The behavior is inversely proportional: doubling the block size approximately halves the transfer time. Certainly this effect is constrained, the CCNx framework allows a maximal block size of 4096 bytes. Because of that behavior the default block size of the CCNx frame work should be raised from 1024 bytes to 4096 bytes. However, this may have disadvantages in networks with many collisions, since large block sizes may require fragmentation. If a fragment collides the entire packet needs to be retransmitted including correctly received fragments. Therefore, the loss of individual packets is more critical than with smaller block sizes.

When is it useful to resume an interrupted transfer and when should we drop the received data and start the transfer from scratch?

Since through the resume capability there is no performance degradation, it is always useful to apply the resume capability for file transfers even for small files. But there are use cases where it makes no sense to use the resume capability. For example: For video streaming it is not necessary to receive each segment. If a single Interest times out the frame should be skipped. We do not have to reexpress the Interest because the missing data will not be used to display the video later.

[3]effective transfer time: time to transfer a whole file without the time spent during interruptions

Chapter 5

Conclusion

5.1 Summary

In this bachelor thesis we extended the implementation of two different file transfer applications with resume capability and compared their transmission performance with the original implementation. With this resume capability new application possibilities for the CCNx framework have been opened. Mobile devices are capable of receiving complete data files even if a transfer is intermittent. Devices with short contact times are able to share data without need to transfer everything at once.
During the evaluation we measured parameters in order to maximize transfer speed and presented performance limitations on resource constrained devices. Based on this parameters we analyzed the benefits of the applications with the resume capability. Regarding the results one can infer that the resume capability enables a constant effective transfer time independent of interruptions.

5.2 Conclusion

The evaluation of these two new file transfer applications led to new insights. The resume process causes no significant processing overhead but enables a constant effective transfer time independent of the number of interrupts. The influence of different block sizes has been determined. A block size of 4096 Bytes reduced the transfer times significantly compared to smaller block sizes. Furthermore the benefits of pipelined file transfer have been evaluated. A pipeline size of 2 approximately halves the transfer times compared to single sequential file transfer and a pipeline size of 16 reduces it by 84.9 percent. To reach the best transfer performance the parameters have to be adapted to the underlying hardware. On resource constrained devices the CPU limits the maximal bandwidth even if the network connection allows a higher one.

5.3 Future Work

The implementation of the file transfer applications with resume capability demonstrates the potential of the resume approach. Nevertheless there are many open questions, problems and potential optimizations to address.

The CCNx framework provides several transmission parameters which may be considered in future implementations. For example the *freshnessSeconds* parameter indicates how long the content remains valid. After the time has elapsed, content will be marked as stale and can be removed from the ContentStore. Based on this information it may be determined whether partial data should be stored or not. For example, if a content file is only valid for a few seconds, it may not be required to store partial information. Furthermore, the *freshnessSeconds* can also be used to automatically remove incomplete files that can no longer be completed.

Another issue affects security and consistency of ContentObjects. The implemented resume capability does not store the PublisherPublicKeyDigest which allows to identify the publisher that signed the ContentObject. Without this information it is possible when resuming a transfer, that another publisher shares a file with the same ContentName but different content. Therefore, we are not able to distinguish the original from the new publisher. If the new publisher does not have any evil intentions we only get inconsistent data, but the publisher may also be able to insert malicious data. Future implementations may therefore compare the identity of the PublisherPublicKeyDigest with the identity in the signature of the received ContentObject.

Additionally, the storage location for the partial data files has been a design choice and was set to the working directory of the user. In future implementations it might be useful to create a shared data folder to remove the restriction that one can only resume from the same directory of the stored partial files.

Chapter 6

Appendix

6.1 Code Listings

To avoid a bloated document the source code is only delivered in an electronic format (attached DVD). In the following section the created and edited source code files will be declared.

6.1.1 Source Code of the Transfer Applications

6.1.1.1 Edited Source Code

The following files have been edited for this thesis:

`/src/ccnx/csrc/cmd/ccnsimplecat.c`
The evaluation mode parameter has been added in this file.

`/src/ccnx/csrc/cmd/ccncat.c`
The evaluation mode parameter has been added in this file.

`/src/ccnx/csrc/lib/ccn_fetch.c`
The function *ccn_fetch_open_resume(...)* has been implemented in this file.

`/src/ccnx/csrc/include/ccn/fetch.h`
The declaration of the *metainfo* struct used in the *ccn_fetch(...)* library has been added in this file.

6.1.1.2 Newly Created Source Code

The following files have been newly created for this thesis:

`/src/ccnx/csrc/cmd/ccnsimplecat_resume.c`
The transfer application ccnsimplecat_resume has been implement in this file.

`/src/ccnx/csrc/cmd/ccncat_resume.c`
The transfer application ccncat_resume has been implement in this file.

6.1.2 ccn_fetch

```c
struct ccn_fetch_stream {
        struct ccn_fetch *parent;
        struct localClosure *requests; // segment requests in process
        int reqBusy;            // the number of requests busy
        int maxBufs;            // max number of buffers allowed
        int nBufs;              // the number of buffers allocated
        struct ccn_fetch_buffer *bufList;       // the buffer list
        char *id;
        struct ccn_charbuf *name; // interest name (without seq#)
        struct ccn_charbuf *interest; // interest template
        int segSize;    // the segment size (-1 if variable, 0 if
           unknown)
        int segsAhead;
        intmax_t fileSize;      // the file size (< 0 if unassigned)
        intmax_t readPosition; // the read position (always assigned)
        intmax_t readStart;     // the read position at segment start
        seg_t readSeg;          // the segment for the readPosition
        seg_t timeoutSeg;       // the lowest timeout segment seen
        seg_t zeroLenSeg;       // the lowest zero len segment seen
        seg_t finalSeg;         // final segment number (< 0 if not
           known yet)
        int finalSegLen;        // final segment length
        intmax_t timeoutUSecs;  // microseconds for interest timeout
        intmax_t timeoutsSeen;
        seg_t segsRead;
        seg_t segsRequested;
};
```

Listing 6.1: Definition of the fetch stream

Chapter 7

Technical Appendix

7.1 Configuration of Wireless Mesh Nodes with ADAM

7.1.1 Hardware

Since the main benefits of the content centric networking approach can be realized on mobile wireless devices, the goal of the evaluation was to analyze the impact of the resume capability on resource constrained mobile devices. For this case the following embedded system hardware has been used:

- ALIX.3D2 system board (LX800 / 256 MB / 1 LAN / 2 miniPCI / USB)
- 2 x Wistron DNMA92 Atheros 802.11a/b/g/n miniPCI wireless card
- 4 x Antenna reverse SMA dual band

7.1.2 Software

As operation system the ADAM operating system developed at the University of Berne has been used[6]. Because the CCNx framework was no embedded in the existing build scripts the existing scripts had to be extended with a new package. In the following part of this document the necessary changes for running the CCNx framework on ALIX devices is explained.

7.1.2.1 Setting Up the Build Environment

In order to generate an image of the ADAM operating system the build environment created by Daniel Balsiger[3] has been used. As a basis the trunk of the adam image builder hosted on the CDS subversion repository was used. The repository is accessible via https://subversion.cnds.unibe.ch/svn/adam/trunk/. The additonal buildscripts and the configuration for building ADAM with CCNx support is hosted on a separate branch https://subversion.cnds.unibe.ch/svn/adam/branches/ccnx

7.1.2.2 Setting Up the Cross-compiling Environment

As mentioned in Section 7.1.1 these embedded systems are resource constrained devices, so compiling a whole operating system would take a lot of time. To avoid this it is useful to apply a cross compiling environment on a performant workstation.

The cross-compiling environment can be set up as shown in listing 7.1

```
# setup the environment
root@hostname:~/image-builder# ./build-tool setup alix
```

Listing 7.1: Setting up the cross-compiling environment

After setting up the the profile for the cross compiling environment, the toolchain itself has to be compiled. Listing 7.2 shows how to start the compiling process for the toolchain.

```
# setup the toolchain for cross compiling
root@hostname:~/image-builder# ./build-tool toolchain alix
```

Listing 7.2: Build the tool chain for the cross-compiling environment

7.1.2.3 Additional Configuration for Compiling the CCNx Framework

The CCNx framework has some dependencies which are not in the default ALIX buildprofile. Because of that the *buildconfig* which lies in the folder *image-builder/buildconfig/alix* has to be updated. The packages *expat* and *ccnx* (which will be introduced later) have been added to the default build configuration.

```
## Update the following line in ~/image-builder/buildconfig/alix/
   buildprofile

BOARDPACKAGES="zlib openssl curl dropbear openntpd nostromo busybox
   linux hotplug2 libnl iw  module-init-tools iproute2 iputils
   ipv6calc iptables flex radvd sudo db cfengine olsrd sqlite2
   tunslip codeprop marwis libpcap ptpd libgpg-error libgcrypt crda
   strace expat ccnx"
```

Listing 7.3: Default board packages

Further the *rc2* encryption in the *openssl* package has to be activated because it is used by the ccn keystore.

```
## Update the following lines in ~/image-builder/buildscripts/
   packages/openssl.sh

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 7.4: Build openssl with rc2 support

While running CCNx applications on an embedded system it is not possible to use graphical user interfaces. Because of that only the C applications are compiled in the created package build script.

The buildscript for the CCNx framework is shown in listing 7.5 and is stored in the *image-builder/buildscripts/packages/ccnx.sh* folder.

```
#!/bin/bash
##################################################
. ${BUILDDIR}/buildscripts/functions
VERSION="0.6.1"
SHA1SUM="467316966851227cc2f731707a14e18b0c6cb1fd"
URL="https://www.ccnx.org/releases/"
FALLBACK="http://www.iam.unibe.ch/~rvs/research/adam/fallback/"
BUILD_DEPS="toolchain expat openssl"
##################################################

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}
```

Listing 7.5: CCNx buildscript

With the additional package script it is possible to compile an operating system image which runs the CCNx framework on ALIX nodes. For the evaluation some hackings of the wifi frequency regulation had to be done. The kernel has been patched to have no regulatory domains. These hacks have to be inserted in the buildscript of the linux kernel. The changes are shown in listing 7.6

```
#!/bin/bash

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

VERSION="3.2"
SHA1SUM="3460afa971049aa79b8f914e1bfd619eedd19f55"
URL="http://www.kernel.org/pub/linux/kernel/v3.0"
FALLBACK="http://www.iam.unibe.ch/~rvs/research/adam/fallback"
BUILD_DEPS="toolchain"
##################################################

set -e
set -u

download_bz2 linux

cd ${BUILDDIR}
rm -rf linux*

tar -xjvf ${SRCDIR}/linux-${VERSION}.tar.bz2
mv linux-${VERSION} linux-${VERSION}-${BOARDNAME}
ln -s linux-${VERSION}-${BOARDNAME} linux
cd linux-${VERSION}-${BOARDNAME}



cat > regd.diff << EOF
57a58,63
>
> /* Everything allowed */
> #define ATH9K_2GHZ_UNLIMITED  REG_RULE(2412-10, 2484+10, 40, 6, 20,
    0)
> #define ATH9K_5GHZ_UNLIMITED  REG_RULE(5150-10, 5850+10, 40, 6, 30,
    0)
>
>
62c68
<       .n_reg_rules = 5,
---
>       .n_reg_rules = 2,
64c70
<       .reg_rules = {
```

```
---
>       /*.reg_rules = {
67c73,77
<       }
---
>       }*/
>       .reg_rules = {
>                 ATH9K_2GHZ_UNLIMITED,
>                 ATH9K_5GHZ_UNLIMITED,
>         }
72c82
<       .n_reg_rules = 4,
---
>       .n_reg_rules = 2,
74c84
<       .reg_rules = {
---
>       /*.reg_rules = {
78c88,92
<       }
---
>       }*/
>       .reg_rules = {
>                 ATH9K_2GHZ_UNLIMITED,
>                 ATH9K_5GHZ_UNLIMITED,
>         }
83c97
<       .n_reg_rules = 3,
---
>       .n_reg_rules = 2,
85c99
<       .reg_rules = {
---
>       /*.reg_rules = {
88c102,106
<       }
---
>       }*/
>       .reg_rules = {
>                 ATH9K_2GHZ_UNLIMITED,
>                 ATH9K_5GHZ_UNLIMITED,
>         }
93c111
<       .n_reg_rules = 3,
---
>       .n_reg_rules = 2,
95c113
<       .reg_rules = {
---
```

```
>       /*.reg_rules = {
98c116,120
<       }
---
>       }*/
>       .reg_rules = {
>                ATH9K_2GHZ_UNLIMITED,
>                ATH9K_5GHZ_UNLIMITED,
>         }
103c125
<       .n_reg_rules = 4,
---
>       .n_reg_rules = 2,
105c127
<       .reg_rules = {
---
>       /*.reg_rules = {
109c131,135
<       }
---
>       }*/
>       .reg_rules = {
>                ATH9K_2GHZ_UNLIMITED,
>                ATH9K_5GHZ_UNLIMITED,
>         }
301c327
<       struct ieee80211_supported_band *sband;
---
>       /*struct ieee80211_supported_band *sband;
313c339
<                       continue;
---
>                       continue;*/
324c350
<               if (!(ch->flags & IEEE80211_CHAN_DISABLED))
---
>               /*if (!(ch->flags & IEEE80211_CHAN_DISABLED))
328c354
<       }
---
>       }*/
EOF
patch --ignore-whitespace ${BUILDDIR}/linux-${VERSION}-${BOARDNAME}/
   drivers/net/wireless/ath/regd.c regd.diff

cat > regd.diff << EOF
111c111
<       .n_reg_rules = 5,
---
```

```
>       .n_reg_rules = 2,
113c113
<       .reg_rules = {
---
>       //.reg_rules = {
115c115
<               REG_RULE(2412-10, 2462+10, 40, 6, 20, 0),
---
>               //REG_RULE(2412-10, 2462+10, 40, 6, 20, 0),
118,120c118,120
<               REG_RULE(2467-10, 2472+10, 20, 6, 20,
<                       NL80211_RRF_PASSIVE_SCAN |
<                       NL80211_RRF_NO_IBSS),
---
>               //REG_RULE(2467-10, 2472+10, 20, 6, 20,
>               //      NL80211_RRF_PASSIVE_SCAN |
>               //      NL80211_RRF_NO_IBSS),
123,126c123,126
<               REG_RULE(2484-10, 2484+10, 20, 6, 20,
<                       NL80211_RRF_PASSIVE_SCAN |
<                       NL80211_RRF_NO_IBSS |
<                       NL80211_RRF_NO_OFDM),
---
>               //REG_RULE(2484-10, 2484+10, 20, 6, 20,
>               //      NL80211_RRF_PASSIVE_SCAN |
>               //      NL80211_RRF_NO_IBSS |
>               //      NL80211_RRF_NO_OFDM),
128,130c128,130
<               REG_RULE(5180-10, 5240+10, 40, 6, 20,
<                         NL80211_RRF_PASSIVE_SCAN |
<                         NL80211_RRF_NO_IBSS),
---
>               //REG_RULE(5180-10, 5240+10, 40, 6, 20,
>                   //      NL80211_RRF_PASSIVE_SCAN |
>                     //    NL80211_RRF_NO_IBSS),
135,138c135,144
<               REG_RULE(5745-10, 5825+10, 40, 6, 20,
<                       NL80211_RRF_PASSIVE_SCAN |
<                       NL80211_RRF_NO_IBSS),
<       }
---
>               //REG_RULE(5745-10, 5825+10, 40, 6, 20,
>               //      NL80211_RRF_PASSIVE_SCAN |
>               //      NL80211_RRF_NO_IBSS),
>       //}
>
>       .reg_rules = {
>                 REG_RULE(2412-10, 2484+10, 40, 6, 20, 0),
>                 REG_RULE(5180-10, 5825+10, 40, 6, 20,0),
>           }
>
EOF
patch --ignore-whitespace ${BUILDDIR}/linux-${VERSION}-${BOARDNAME}/
    net/wireless/reg.c regd.diff

patch_src linux
echo Building for $BOARDARCH

make distclean
make ARCH=${BOARDARCH} CROSS_COMPILE=${CROSS_TARGET}- mrproper

rm -f .config*
cp ${BUILDCONFDIR}/config-linux .config >${BUILDDIR}/initramfs.cpio
rm -rf ${INSTALLDIR}/lib/modules -rf

make ARCH=${BOARDARCH} CROSS_COMPILE=${CROSS_TARGET}- oldconfig
make ARCH=${BOARDARCH} CROSS_COMPILE=${CROSS_TARGET}- vmlinux
make ARCH=${BOARDARCH} CROSS_COMPILE=${CROSS_TARGET}- modules
make ARCH=${BOARDARCH} CROSS_COMPILE=${CROSS_TARGET}-
    INSTALL_MOD_PATH=${INSTALLDIR} modules_install

KVERS=$( grep -E "^(VERSION|PATCHLEVEL|SUBLEVEL)" Makefile | cut -d =
     -f 2 | xargs echo | sed -e 's/ /./g' )

if [ -d ${INSTALLDIR}/lib/modules/${VERSION}-${BOARDNAME} ]; then
        ${INSTALLDIR}/cross-tools/bin/depmod.pl -F System.map -b ${
            INSTALLDIR}/lib/modules/${VERSION}-${BOARDNAME}
elif [ -d ${INSTALLDIR}/lib/modules/${KVERS}-${BOARDNAME} ] ; then
        ${INSTALLDIR}/cross-tools/bin/depmod.pl -F System.map -b ${
            INSTALLDIR}/lib/modules/${KVERS}-${BOARDNAME}
elif [ -d ${INSTALLDIR}/lib/modules/${KVERS} ] ; then
        ${INSTALLDIR}/cross-tools/bin/depmod.pl -F System.map -b ${
            INSTALLDIR}/lib/modules/${KVERS}
else
        ${INSTALLDIR}/cross-tools/bin/depmod.pl -F System.map -b ${
            INSTALLDIR}/lib/modules/${VERSION}
fi
```

Listing 7.6: Hacking the regulatory domains of the wifi devices

7.1.2.4 Compiling the Packages

Finally the needed image can be built. The process can be done straightfoward as explained in the README file. In the first step every package has to be compiled. Listing 7.7 shows the command to execute for that step.

```
root@hostname:~/image-builder# ./build-tool packages alix
This will install the following packages in order from left to right:

zlib openssl curl dropbear openntpd nostromo busybox linux hotplug2
   libnl iw  module-init-tools iproute2 iputils ipv6calc iptables
   flex radvd sudo db cfengine olsrd sqlite2 tunslip codeprop marwis
   libpcap ptpd libgpg-error libgcrypt crda strace expat ccnx

Packages will be built in:     /home/alix-builder
Package sources are in:        /home/alix-builder/sources
Extra configuration is in:     /home/alix-builder/buildconfig
Packages will be installed to: /home/alix-builder/target
Packages get compiled by:      /home/alix-builder/buildscripts/
   packages/<package>.sh
Buildlog is in:                /home/alix-builder/buildlogs/<package
   >.buildlog
Cross-Compiler used:           i586-linux-uclibc-gcc
Target system triplet is:      i586-linux-uclibc

Are these values reasonable ? Begin installation [y/n]
```

Listing 7.7: Compiling the packages

7.1.2.5 Creating the Image Files

After compiling the packages the image can be built. The file `README.build` in the `doc` folder contains a step by step manual how to build the necessary images. The main steps of this document are explained in the following lines.

To run the ADAM operating system it is necessary to build three different image files:

- The standalone image which is perfectly adjusted to test node hardware without bricking it.

```
./image-tool gen_standalone <boardname> <version>
```

Listing 7.8: Generating the standalone image

  Executing the command shown in listing 7.8 generates an image file called *<boardname>-image-standalone-<version>.bin.gz*

- The real image which needs a configuration image for working in a mesh network.

```
./image-tool gen_image <boardname> <version>
```

Listing 7.9: Generating the real image

  Executing the command shown in listing 7.9 generates the image file called *<boardname>-image-<version>.bin.gz*

- The configuration image which contains ssh public keys, the hostname, the network configuration etc.

```
./image-tool gen_config <boardname> <hostname>
```

Listing 7.10: Generating the configuration image

  The configuration image is called *config-<hostname>.default* and can be created running the command shown in listing 7.10.

To setup the configuration follow step 10 of the `README.build` manual.

The configuration files used for running the ALIX Mesh Nodes mentioned in the Evaluation chapter can be found at the path `/experiments/scripts/setup/nodes/` of the attached DVD.

7.1.2.6 Installing the Images on the Device

After generating the necessary image files for running the ADAM operating system the flash disk have to be formatted and the images have to be installed.

There exists a step by step manual how to create a bootable flash drive with the ADAM operation system. The manual is called `README.alix` and can be found in the `doc` folder of the ALIX image builder.

When all of this steps have been fulfilled, the flash drive can be plugged into the flash drive slot of the ALIX mainboard. The operating system is ready to run.

7.1.2.7 Initialize Connectivity to the Mesh Node

There are several ways of connecting and controlling the mesh node:

RS-232 Serial Link

The unix tool *minicom* is necessary to open a console for sending commands over a serial link.

First the serial cable has to be plugged into the RS-232 of the mesh node and the usb serial adapter on the host computer. Thereafter the console can be opened with the following command:

```
minicom -D /dev/ttyUSB0 -b 9600
```

Listing 7.11: Open serial communication console using minicom

Then a common unix login prompt should appear.

SSH Connection using one of the configured network devices

When the network devices are configured correctly, it is possible to get access to the mesh node using a ssh console. The only thing one has to know is the IP address of the node's network device. To open the console the following command has to be used:

```
ssh root@192.168.0.1
```

Listing 7.12: Open ssh console

A SSH connection can also be used to transfer files from or to the node. There exists a command line tool called `scp` with this functionality. For example for transferring a file to the node the following command has to be used:

```
scp /path/to/file root@192.168.0.2:/path/on/mesh/node/
```

Listing 7.13: Send file to node using scp

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] "Project CCNx," https://ccnx.org, 2012.

[3] D. Balsiger, "Administration and deployment of wireless mesh networks," Master's thesis, University of Berne, April 2009. [Online]. Available: http://cds.unibe.ch/research/pub_files/Ba09.pdf

[4] "CCN_REPO(1) manual page," October 2012. [Online]. Available: http://www.ccnx.org/releases/latest/doc/manpages/ccn_repo.1.html

[5] "Pcengines alix," http://pcengines.ch/alix.htm, 2012.

[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.