Abstract

Mobile ad hoc networks (MANETs) often suffer from disruptions and partitioning. Techniques to tackle these challenges such as caching and replication might result in data being widely distributed in the network. For enabling media streaming in such networks, the requirement for an improved signaling system has been raised. Therefore, we have in this thesis, designed and implemented a solution for signaling of media streaming in MANETs. This solution is able to create meta data that is kept and sent together with audio and video (AV) data, so specific data can be found and retrieved if disruption and partitioning occurs. This also provides the ability to gather meta data and create overviews of where AV data is distributed in the network. In addition it provides detailed control over the streaming process so users can choose whether missing data is important enough to be retrieved or not.

Since there are currently no existing media players that supports this functionality, we have also provided the design of an experimental media player. This media player is tailored to support the specifics of our signaling system, and makes it easier to illustrate benefits of the functionality. Through evaluation we have verified that the user is provided this fine-grained control of different streaming sessions, in addition to being informed with as much as possible details of the process.
Acknowledgements

I would like to thank my advisors Professor Thomas Plagemann, Professor Vera Goebel and Morten Lindeberg for invaluable feedback and guidance. Without their help, this work would never have been possible.

I would also like to thank all the other people at the DMMS research group, for providing a friendly working environment, making the process of writing this thesis enjoyable.

Lars Olav Dybsjord
University of Oslo
May, 2010
# Contents

1 Introduction 1  
1.1 Background .............................................. 1  
1.2 Motivation .............................................. 2  
1.3 Goals ..................................................... 3  
1.4 Outline ................................................... 3  

2 Background 5  
2.1 Media Streaming in MANETs ................................. 5  
2.1.1 Current Media Players .................................. 7  
2.2 Media Signaling ............................................ 8  
2.2.1 RTSP .................................................... 8  
2.2.2 SIP ..................................................... 8  
2.2.3 H.323 .................................................. 9  
2.3 DT-Stream Project ........................................... 9  
2.4 Definition of Node Roles ................................... 10  

3 Design 13  
3.1 Requirements ............................................... 13  
3.1.1 Media Player ............................................. 13  
3.1.2 Overlay Transport ....................................... 19  
3.1.3 Signaling ............................................... 21  
3.2 System Design .............................................. 25  
3.2.1 Architecture ............................................ 25  
3.2.2 Meta Data Manager ..................................... 27  
3.2.3 Stream Control Manager ................................ 29  
3.3 Summary .................................................... 39  

4 Implementation 43  
4.1 Implementation Overview .................................. 43  
4.1.1 Applications API (libdtss) ............................. 44  
4.1.2 Delay Tolerant Stream Signaling Daemon (dtssd) . . 44
4.1.3 Media Sessions ........................................ 45
4.2 Overlay API ........................................... 48
4.3 Applications API (libdtss) ............................. 51
4.4 Signaling Core (dtssd) ................................. 55
   4.4.1 Meta Data Manager ................................. 64

5 Evaluation ................................................. 67
   5.1 Goals .................................................. 67
   5.2 Experiments ............................................ 68
      5.2.1 Setup ........................................... 68
      5.2.2 Workload ........................................ 70
      5.2.3 Parameters ....................................... 71
      5.2.4 Scenarios ......................................... 72
   5.3 Results ................................................ 73
      5.3.1 Scenario 1: No Packet Loss ..................... 73
      5.3.2 Scenario 2: Fixed Packet Loss .................. 74

6 Conclusion and Future Work ................................ 79
   6.1 Conclusion ........................................... 79
   6.2 Future Work .......................................... 80
      6.2.1 Signaling Tuning ................................ 80
      6.2.2 Content Type Priority ............................ 81
      6.2.3 Overlay Layer Integration ....................... 81
      6.2.4 Evaluation ....................................... 81

Bibliography .................................................. 83

Appendices ..................................................... 85

A Source Code ................................................ 85
   A.1 Signaling Core (dtssd) ................................ 85
   A.2 Applications API (libdtss) ........................... 85
   A.3 Emulated Overlay Layer (libdtt) ..................... 86
   A.4 Emulated Media Player ................................. 86
List of Figures

1.1 Scenario: Emergency and Rescue ........................................... 3
2.1 Three network partitions .................................................. 6
2.2 Node Roles ................................................................. 11
3.1 Missing data ............................................................... 14
3.2 Progress bar ............................................................... 16
3.3 Window split ............................................................... 17
3.4 Data chunk index of a part of a stream ............................... 17
3.5 Data chunk index of a part of a stream with network overview 19
3.6 DT-Stream Media Player \cite{1} ............................................. 20
3.7 Moving node ............................................................... 21
3.8 Network stack ............................................................. 26
3.9 System components ...................................................... 26
3.10 Meta Data Storage ....................................................... 29
3.11 Push and pull based streaming ....................................... 30
3.12 Gaps of missing data ................................................... 31
3.13 AV data packet .......................................................... 31
3.14 Two sources with a file with the same URI ......................... 32
3.15 Stream control signaling .............................................. 37
3.16 METADATA message with sequence numbers ..................... 39
3.17 Generate overview to media player ................................ 40
4.1 DTSS ........................................................................... 44
4.2 Three node example ....................................................... 45
4.3 Sessions ....................................................................... 46
4.4 struct media_session_t .................................................. 49
4.5 Several streaming sessions ............................................. 49
4.6 Overlay API ............................................................... 49
4.7 Data types .................................................................. 51
4.8 libdtss ....................................................................... 52
LIST OF FIGURES

4.9 dtssd .................................................. 55
4.10 IPC message BNF ...................................... 56
4.11 IPC message handling ................................. 57
4.12 DTSS packet format ................................. 58
4.13 Network message BNF ............................... 58
4.14 DTSS signals .......................................... 60
4.15 DTSS session states .................................. 60
4.16 SIG\_METADATA with bitmap ..................... 62
4.17 seq_l .................................................. 62
4.18 SIG\_GET\_DATA with bitmap ..................... 63
4.19 Meta Data Storage .................................... 65
4.20 Meta Data Storage .................................... 66

5.1 Three node scenario .................................... 72
5.2 Scenario 1: The consumer has received AV data with sequence numbers 0 to 2499 from the stream 0 .................. 74
5.3 Scenario 2: The consumer is missing some AV data from the stream ......................................................... 74
5.4 Scenario 2: Show how AV data is distributed in the overlay and locally ......................................................... 75
5.5 Scenario 2: The result of gathering AV data with sequence numbers 800 to 1000 from Overlay Node 1 .................. 76
5.6 Scenario 2: Show where AV data packets with sequence numbers in the range 1472 to 2016 is located in the overlay and locally ......................................................... 76
5.7 Scenario 2: The consumer has received all the AV data it was interested in ......................................................... 77
List of Tables

3.1 Table of cached data .............................................. 29
3.2 Index of where packets belonging to one stream is located in
a network .......................................................... 39
4.1 Which variables are set in which type of node ............... 48
5.1 Experiments machine details ................................. 68
5.2 Scenario 2: Fixed packet loss during transmission ........ 73
5.3 Scenario 1: No packet loss .................................. 74
5.4 Scenario 2: Fixed packet loss .............................. 77
5.5 Scenario 2: Fixed packet Loss .............................. 77
Chapter 1

Introduction

This thesis is a part of the ongoing "DT-Stream" project at the distributed multimedia (DMMS) research group at University of Oslo. The DT-Stream project objective is:

"To develop new solutions that enable AV streaming services over heterogeneous, mobile and unstable networks which are found for instance in emergency and rescue operations." [2]

As a part of this project, this thesis aims to provide a signaling system for media streaming over delay and disruption tolerant networks (DTN [3]). As a main objective, we aim to provide users with a fine-grained control over the signaling of different media streams.

In this chapter we begin by introducing some background information (Section 1.1) followed by an discussion of the motivation for this thesis (Section 1.2). Finally, we states our goals in Section 1.3.

1.1 Background

The emerging technology of mobile devices currently provides improved possibilities for multimedia communication. In addition, many of these small devices supports interconnection with other devices through interfaces like Bluetooth, wireless local area network (WLAN), and infrared. Through especially WLAN (IEEE 802.11) and Bluetooth, higher bandwidth is supported, and enables relative high speed interaction between mobile devices in both ad-hoc\(^1\) and infrastructure mode. While the infrastructure mode is dependent on a access point, the ad-hoc mode supports a improvised setup of a

\(^1\)ad-hoc: for this purpose
network for a special purpose. Mobile ad-hoc networks (MANET) is a definition of such networks and is the domain we choose to study in this thesis.

The topology of such interconnected devices opens up for new possibilities, but also challenges the stability of the network and devices regarding energy and bandwidth consumption, when more data and processing is required. As a result, mobility, delay and disruption may occur, and affect the stability of the network. In such environments, transferring of multimedia data through the network, is therefore not a trivial task.

Currently, many devices provide AV services such as head mounted cameras, cellular phones with built in cameras, lightweight microphones and screen projection glasses. Unfortunately, many of these AV services cannot with todays technology be fully utilized in MANETs, due to communication limitations. These limitations are often the results of different nodes moving out of range from each other, making it difficult to stream continuous data, as these nodes lack connectivity. However, when this happens, it is still important that a destination waiting for data, receives as much as possible of the required data, e.g., when nodes reacquire connectivity.

1.2 Motivation

Scenario: Emergency and Rescue

Today, there are several scenarios where the technology of media streaming over MANETs can be useful. Amongst these, is emergency and rescue [4], which we study in this thesis. This is an area where we believe media streaming can provide great results, by improving communication between rescue and emergency operators. In such scenarios, infrastructure might neither be present, and ad-hoc solutions is the only possibility.

As an example, consider the event of a tunnel collapse where people might be trapped inside. When rescue personnel are working within the rescue site, efficient communication amongst each rescue personnel is an important factor. With video and/or audio conferencing present, the ease of communication efforts during such stressful operations can be expected. If the rescue personnel approaches the tunnel from both sides, there would be a lot easier to construct a picture of the scenario, using video and audio equipment to communicate, for all parties involved. In addition, there could be situations where new collapses could separate rescue personnel from each other. In this case, video and audio services can synchronize their information and procedures to continue on more effectively. Also, a command and control center
Introduction

with leaders that is managing the operation, may be located off site and requires communication to an from the rescue site. Figure 1.1 illustrates such a scenario.

1.3 Goals

With the scenario of emergency and rescue in mind, this thesis aims to provide a customized signaling system for optimized media streaming over MANETs, with focus on delay tolerance, visualization and presentation. When media data is streamed from a node to another, and mechanisms used to handle disruption (e.g., caching and replication) results in widely distributed data in the network, our signaling system should be able to retrieve this data. In addition, we strive to provide a fine-grained API for applications to gather information about the data in the network in addition to be able to retrieve the data itself.

1.4 Outline

In the following chapters, we begin by discussing background in Chapter 2. Our signaling system’s design is presented in detail in the subsequent Chapter 3, with its implementation described in Chapter 4. The evaluation of our system is performed in Chapter 5 followed by a conclusion and discussion of topics for future work in Chapter 6.
Chapter 2

Background

In this chapter we begin by introducing the concept of MANET video streaming in Section 2.1 and discuss the existing technology for signaling in Section 2.2. Then we give an overview of the DT-Stream project to see how we should design our signaling system, in Section 2.3.

2.1 Media Streaming in MANETs

In Mobile Ad-Hoc Networks (MANETs), heterogeneous devices interconnect to form a network without the need of any infrastructure like the Internet. Each device in the network is responsible for forwarding data to other devices, operating as individual routers. However, since these devices are mobile, routes between the different devices may change frequently depending on the movement. Therefore, to be able to have an adaptive routing, additional routing protocols is required. These routing protocols can be either reactive (on-demand), proactive (table-driven) or hybrid (both). Reactive protocols calculates the routes when it is needed, while proactive protocols calculates and updates the routing table in advance. Examples of reactive routing protocols are AODV [5] and DSR [6], while examples of proactive routing protocols are OLSR [7] and WRP [8].

Since the mobility in MANETs can result in sparse distribution of the devices in the network, routes between different devices might be disrupt for longer periods in time. As a result, this may lead to several partitions in the network where a group of devices is unable to communicate with another group (illustrated in Figure 2.1). These technical issues are addressed in the DTN domain. In this domain, the lack of continuously connected end-to-end paths makes it problematic to transport data with standard transport protocols such as UDP and TCP. As a result, the concept of store, carry...
Figure 2.1: Three network partitions

and forward has been introduced. This concept relies on each node's effort to store data that needs to be forwarded, until a route that brings the data closer to the destination arises. When this route arises, data is sent "towards" the destination. In this way, a device moving from one partition to another, can be able to store, carry and forward data from nodes in the first partition to nodes in the second partition. Since this technology provides the ability for two devices to communicate despite the lack of a continuously end-to-end path, this enables the study of media streaming over MANETs.

However, to be able to have a media streaming session in delay and disruption tolerant MANETs, the data transport and signaling of data transport must be determined. Thus, in the following, we give a short description of the two standard transport protocols TCP [9] and UDP [10], and the application layer protocol RTP [11]. Then we describe some of the current media players used for playing media streams, and how they perform in a DTN.

TCP

The Transport Control Protocol (TCP) is a connected-oriented transport layer protocol designed to provide reliable and in-order data transmission between end points in a network. It supports both flow control and congestion control to provide and maintain high transfer rate. As TCP is a connection oriented protocol, it might not behave sufficient in sparse MANETs [12]. As sparse MANETs can suffer from partitioning, TCP might fail due to timeout, or trigger congestion control when unexpected packet loss occurs.

UDP

The User Datagram Protocol (UDP) is another transport layer protocol. Its purpose is to provide high rate data transfer without any form for reliability
and ordering of segments\textsuperscript{1}. Since UDP does not provide error correction, UDP is sometimes preferred over TCP in cases such as real-time data transfer. In such cases, receivers may benefit from the segment being dropped, instead of waiting for error correction (which might stall all subsequent data sending).

**RTP**

Real-Time Transport Protocol (RTP) is an application layer protocol that defines a standardized packet format for transmitting AV data in a network. It can be transported over any transport protocol (e.g., TCP and UDP), but since audio and video often require timely delivery, UDP is often preferred.

### 2.1.1 Current Media Players

The actual presentation of the media to the end-user(s) is handled by a media application like e.g., a media player. The core functionality of a media player is to choose and present subsequent decoded video and/or audio frames onto the screen and speakers, based on the users preferences. The decoded frames are received from a decoder responsible for decoding encoded data. The preferences of the users determines which frames to be presented with commands such as play, fast-forward and fast-backward. Determined by the implementation of the media player, the decoder is not always a part of the implementation, but rather a library the media player uses. FFMPEG [13] is an example of such a library, which performs both encoding and decoding of data, and is an open source project.

As decoding local data often is a result of a file being played back, decoding remote data may be the result of a remote media streaming session. For this media streaming session to be possible, the media player requires some form of signaling to inform the remote node to send encoded data. The signaling is thus usually handled by implementations of application layer protocols such as HTTP [14], RTSP [15] and SIP [16]. Whether the signaling is a part of the media player is however depending on the design and implementation of the media player. Currently most media players implement the signaling as a part of the media player, as this simplifies its usage from an end user perspective.

\textsuperscript{1}Segment: Layer 4 PDU in the OSI model
Chapter 2

2.2 Media Signaling

The purpose of media signaling is to control the sending of one or more media streams in a network. However, since many of the current signaling protocols are unicast, these signaling protocols might not be sufficient for media streaming in sparse MANETs, where frequent disruptions and route changes may lead to wide spread distribution of media data.

Since these signaling protocols do not care about the transport of data, they leave this to the transport layer. If these signaling protocols require all data to be delivered, they use a reliable protocol such as TCP. If data delivery in a timely manner is more important, a protocol like UDP is more suitable. However, as previously discussed, TCP in MANETs are limited, thus protocols like UDP should be preferred for this thesis.

Since MANETs can experience a lot of packet loss, media streaming clients may miss much data. Therefore, a signaling system with increased functionality to handle packet loss is required, to provide better support for media streaming in MANETs. However, before we design such a signaling system, we describe some of the popular existing solutions for signaling media data in packet based networks.

2.2.1 RTSP

Real Time Streaming Protocol (RTSP) is a protocol for mediation of one or more media streaming sessions, and is often referred to as a ”network remote control”. The signaling messages is text-based (such as HTTP), and it uses RTP for transmission of media data. These messages can be transported over both TCP and UDP. RTSP offers no form of error correction, which makes the choice of a transport protocol for media data transmission important. If the RTSP implementation determines that all the media data must arrive, it uses a protocol such as TCP as transport protocol, while using a protocol like UDP for media data that must arrive within timely manner. RTSP supports unicast and multicast for one-to-many applications.

2.2.2 SIP

Session Initiation Protocol (SIP) is a similar signaling protocol for handling multimedia communication between two or more parties, such as video and voice calls in a network. It too, uses text for session communication which is not very optimized with respect to network utilization in MANETs. It uses RTP for media data transmission, and similar to RTSP does not provide
any error correction. As opposed to RTSP, SIP provides additional signaling functions to provide many of the same features as the public switched telephone system (PSTN). This includes features as:

- Causing a phone to ring
- Hear busy sound
- Hangup before and after someone has answered

### 2.2.3 H.323

H.323 is an International Telecommunication Union (ITU-T) recommendation for a two-way media communication over packet based networks. It includes other ITU-T protocols for further specifications of the H.323 system, and the two most important protocols that is relevant for this thesis is:

- H.225.0
- H.245

H.225.0 is responsible for the call signaling, while H.245 is responsible for handling media communication such as sending and receiving media data. This media data is encapsulated in RTP packets, similar to the other previously discussed signaling protocols. The signaling procedure in H.225.0 is based on the procedure for ISDN [17], and opposed to RTSP and SIP, does not send text-based signaling messages, resulting in better utilization of the network.

### 2.3 DT-Stream Project

DT-Stream [2] is an ongoing project at University of Oslo, aiming to study media streaming over DTNs. As a part of the DT-Stream project, a previous study of data transport named MOMENTUM [18] has proposed an overlay to enhance data transmission in sparse MANETs. Its primary goal is

"to deliver as much video as possible, with the best quality achievable in such an environment"

---

[^2]: [http://www.itu.int](http://www.itu.int)
MOMENTUM acquires this by taking advantage of the network layer, and extracting information from the routing protocol for later reuse in an overlay transport layer. It provides optimizations for multimedia distribution, reliable transport for high priority messages and utilize the concept of store, carry and forward.

2.4 Definition of Node Roles

Since we discuss detailed network topology in this thesis, we introduce the definitions of the different node roles in a network in the following list, and illustrated in Figure 2.2.

- **Source** - (1) A node that serves video-on-demand or live streams
- **Consumer** - (5) A node that wants data from another node in the network, to present to the user (for e.g., playback)
- **Overlay Node** - (1,2,3,5) Any node in an overlay
- **Caching Node** - (1,2,3,5) Any node in an overlay that performs caching
- **Intermediate Overlay Node** - (2,3) A node in an overlay positioned between two other nodes
- **Intermediate Caching Node** - (2,3) A node in an overlay positioned between two other nodes, that perform caching
Figure 2.2: Node Roles

Node definition:
1 - Network Node, Overlay Node, Source Node
2 - Network Node, Overlay Node, Intermediate Overlay Node
3 - Network Node, Overlay Node, Intermediate Overlay Node
4 - Network Node
5 - Network Node, Overlay Node, Consumer
Chapter 3
Design

In this chapter, we explain the design of a delay tolerant signaling system for multimedia streaming for the DT-Stream project. We start by introducing the requirements (Section 3.1), and then provide details on how these requirements are met with our design (Section 3.2).

3.1 Requirements

To be able to design a signaling system for multimedia streaming over a DTN, we need to identify the functional and non-functional requirements for the protocol. Since the signaling system is dependent on both a customized transport protocol and a customized media player for playing back the multimedia data, we need to introduce and define their functionality. Succeedingly, we present the requirements of the signaling system itself, based on the requirements of the media player and transport protocol.

3.1.1 Media Player

As we target DTN, playback of multimedia streams through the network is potentially highly unstable. Since today’s technology lacks proper support for full playback of incomplete streams, we have set some assumptions regarding a customized media player for the DTN domain. However, to be able to make these assumptions, we must review the current available media players to investigate what is missing for them to operate in a DTN environment.

Current Media Players

Most of the media players available are based on the assumption that received data should be shown from start to finish, by the use of streaming protocols
When playing back a media stream from the network, and the network presents packet loss, gaps of missing data will occur in the arriving stream. When the media player "discovers" this, it will either crash, halt, or in best case skip the missing data; leaving a solid colored screen or a scrambled view of the last frame it was able to present. Figure 3.1 a) shows a media player waiting for more data to play back, while displaying a solid color screen. Figure 3.1 b) however, shows a media player that has managed to skip data that is missing, and only displaying the newest received data. Depending on the implementation of these media players, these media players could have crashed due to the missing data.

Most media players also implement a **playout buffer**\(^1\), that keeps a logical infinite buffer with arriving media data that the media player consumes data from. As the network might provide jitter, this buffer makes the playback go smooth, as long as the buffer is not empty when the media player consumes data. Especially when playing back live streams, the delay of filling the buffer may be unacceptable as the result will be that the user will continuously watch non real-time data (always be one step behind). This is however not as much of a problem when dealing with video on demand, as this data is not real-time anyways.

**Delay Tolerant Media Player**

Since current media players do not provide the ability for a user to either skip the parts missing from a stream, or wait for it to arrive, we propose a media player that allows this type of control.

An example of this, is when a media player is presenting a stream to the user that has missing parts. The first thing we want from the media player is

---
\(^1\)Also called **jitter buffer**
that it should provide the user with information on which parts of the stream that are missing. Second, we want the user to be able to choose whether to wait for the missing part, or keep trying to play back the rest of the stream and skip the parts missing. As a result, if the user decides to wait for the parts of the stream that are missing, he should also get the opportunity to display these parts in another window. To give an overview of the streaming process to the user, the delay tolerant media player would need a progress bar.

**Progress bar**

The purpose of the progress bar is to inform the user of the status of the arriving multimedia stream. Therefore, the progress bar should present to the user what data is available, by e.g., showing a rectangular box colored at the parts that has arrived. It should be noted that if it is a live stream, the size of the bar (width) will be dynamic and change based on the data received over time (e.g., "growing"). While playing we should also have a playback line moving in real-time speed to tell the viewers where in a stream we are. However when a part of a stream is missing, the default action would be to continue playback of the stream until otherwise demanded. If this missing part arrives at later time, the progress bar will indicate this by changing colors. As a result, the user would be able to see which parts he can choose to play back in another window, while continuously watching the newest data (see next section). If the user wants to watch a part that is not marked as local in the progress bar in another window, the user should be aware that it may take time for this data to arrive. When the user chooses to watch this part, a content priority request would be set on this data.

Figure 3.2 illustrate an example of a progress bar.

**Window Split**

By allowing a user to split the media player window in two (or more), he could be able to watch several parts of a stream at the same time. For example, he can have one window play the parts of a live stream that is missing (playing it as it arrives) and at the same time have an additional window continue to play the newest data from the live stream (possibly skipping the parts not available). Figure 3.3 illustrates this. In 1) the user wants to see a previous part of a stream that has not arrived yet, and initiates a window split. At 2) the media player window is extended to show an additional window. The left window will thus continue to show the newest data from the live stream, while the right window will show the parts that has not arrived yet. Since
all the data the right window presents has not arrived yet, this presentation might be slow, as the media player must wait for the data to arrive, before it is presented.

It should also be noted that splitting the window to present two different parts of a stream concurrently, should be possible at all times. For example when a user watches a live stream, he might not remember what happened earlier in the stream and might want to go back and watch it again (at the same time as keeping an eye on the live stream).

**Data Chunk Index**

In addition to showing a progress bar of a media stream, it would also be useful to let the user know where the pieces of the stream are located in the network. If the underlying signaling system is able to gather the necessary information, the media player should support presenting it to the user. By letting the user see how the pieces are distributed in the network, it could be possible for the user to increase the probability of retrieving the data he is interested in, by e.g., changing physical location. An example for such a data chunk index is illustrated in Figure 3.4. The figure shows a list of nodes in the network with data belonging to the stream of interest. Each node’s progress bar indicates which chunks of the stream the node is currently in possession of. It should also be noted that the index illustrated in this figure is only made up of the parts of the stream that the user requested. This size should not be static, but rather support the user to customize it to its needs. This means it could show an index of any interval of the streams, and even an index of the complete stream.
1) Figure 3.3: Window split

2) Figure 3.4: Data chunk index of a part of a stream
Content Priority

We propose a content priority management for letting user input affect the signaling system for fetching data. When a part of a stream is missing, the user can tell the signaling layer that this missing data is important and has to be received. By doing this, the missing data will get a higher priority. When this priority is set, the signaling layer would start by gathering meta data from the network to know where to retrieve the data from. This meta data gathering is the same as for the data chunk index, and because of this, the data chunk index and the progress bar will be updated too. With awareness of where the data are in the network, the retrieval can begin. However, in addition to letting the signaling layer choose the best way to retrieve data, we let the user override this process. This override, can be done by looking at the video chunk index, which gives the users the possibility to choose exactly what data to get from which node.

Network Overview

We also want the users to be able to have an overview of the network topology with details on each node. By having this overview, the user can tune the data fetching by using the content priority feature together with the video chunk index. An example of such a feature is illustrated in Figure 3.5. The figure shows a overview of the network topology in the right window, and a data chunk index for each of the nodes in the left window.

Media Player Features

This section briefly summarises the requirements identified for the media player.

- Play back live streams, files (video on demand) and part of files
- Play back incomplete live streams and files
  - Continue the playback of the last (time wise) received data
  - Stop and wait for sequentially received data
  - Split the window while playing the latest data, to wait for sequential data from another part of the stream
  - Watch two parts of a stream at the same time (even though, there is no data missing)
- Skip to another place in the stream (FFWD, FBWD)
Sonia Anes Gonzalez, another master student in the DT-Stream project is currently working on designing a media player for the DTN domain. The purpose of this media player is mostly to provide the user with the control to prioritize complete streams from different sources, in addition to switch between different stream quickly. Compared to our design for a delay tolerant media player, her study is concerned with complete streams from different sources, while we discuss the priority for the different parts of one stream. The preliminary graphical design of here proposed media player is illustrated in Figure 3.6.

### 3.1.2 Overlay Transport

For our customized media streaming signaling system, we assume an underlying delay tolerant transport protocol. However, to be able to design
our signaling system, we need to set some requirements from this transport protocol. These requirements are listed here, with a detailed explanation following in the next sections.

- **Reliable Transport**
- **AV Data Cache**

### Reliable Transport

As our signaling system needs to send some high priority control messages (such as PLAY and STOP), it is very important that the destination receives these. To provide such guarantees we assume that a packet that is sent through the overlay is received some time in the future, possibly with extensive delay, but is never dropped. As an example, assume that a node moves out of range and is no longer connected to the network overlay (Figure 3.7 a and b). If the node that is no longer connected to the overlay needs to send a high priority message to a destination that is part of the overlay, it sends the message to its transport layer. Since each node in the overlay knows about all the other connected nodes in the overlay, the transport layer knows if it is possible to send the message to the destination. Therefore, when the overlay transport layer receives the message, it will look up the destination node to see if it is presently located in the overlay. If this is not the case, the message should be buffered until the destination appears in the overlay again. When the node that is sending the messages moves in range of the overlay (Figure 3.7 c), the message should be sent to the destination. However, if the destination node is not a direct overlay neighbour of the sender,
it will be routed through an intermediate overlay node. If this intermediate overlay node has no longer any knowledge of the destination by the time the message arrives (due to e.g., another network partition), the same procedure should be taken here; put the message in a buffer, wait for the destination node to appear in the overlay, and send the packet to the destination.

It should be noted that the route this last node sends the message, may be different from the initial route, since it is up to each intermediate overlay node to calculate the route it chooses to send data to the destination on. This should however not inflict other problems than some delay, and the guarantee that the packet should arrive at the destination is still held.

**AV Data Cache**

To allow a consumer to fetch missing AV data from the network, different nodes needs to perform caching. In this thesis, we assume that the overlay transport handles this. However, for the signaling layer to be able to retrieve this data for further processing (such as sending the data to a consumer requesting it), the AV Data Cache need to provide an interface to the signaling layer. The decision of which node that should perform caching, is however out of the scope of this thesis, but for simplicity’s sake we assume that all nodes in the overlay performs caching for all sent and received data. This also means, that when a source sends data to a consumer, the overlay transport should cache data locally before it is sent to the consumer. As a result, the consumer can retransmit data from the source, if data is lost in the network.

### 3.1.3 Signaling

In this thesis we aim to provide a signaling system for a media player that can be highly customized and give verbose information to the user regarding the availability of a stream. As we have already discussed the requirements for the media player and the transport layer, we must now identify the requirements for the signaling system, to see how it can support the media
player requirements.

To identify these requirements, we introduce an example to illustrate some of the problems we are aiming to solve. The example considers the case where a tunnel has collapsed. Two emergency and rescue operators, Bob and Alice, are approaching the tunnel from each side. When they rendezvous at the center of the tunnel (where the collapse has taken place), Bob and Alice sets up a video conference, as Alice needs help from Bob to resolve an issue. Alice is using a head mounted camera to send live video to Bob, which Bob is displaying with a ordinary media player. As Bob needs to move to another location due to some external event, he moves out of Alice’s wireless range. At this time, Alice is streaming out data, but no one is receiving. Two minutes later, Bob moves to another location, and is back in Alice’s wireless range. Unfortunately, a lot of the data Alice has transmitted, is lost, and Bob continues to receive the latest data that Alice is sending. However, since Bob is mostly interested in the lost data, some questions arises:

- How does Bob know what part of the stream he has lost?
- How does he tell Alice that he wants the parts that has been lost?
- Has Alice cached the data that Bob wants retransmitted?

Now, it seems that Charlie, a third operator that was located close by Alice, received a lot of the data destined for Bob. At the time Alice and Bob got reunited (within wireless range), Charlie, is also available. Charlie is however a bit closer to Bob than Alice, and can apply transmission of data faster to Bob. Since Charlie performed caching, he has a lot of the data that Bob had lost. As a result, some new unresolved questions arises:

- How can Bob know if Charlie has any data he wants?
- Could Bob ask Charlie for the data that had been lost?
- If both Alice and Charlie are available and can retransmit data, which one of them should Bob receive data from? Alice, Charlie or both?

As current media players do not give any information about what parts of the stream are lost, it is difficult for Bob to ask both Alice and Charlie to retransmit the lost data. Even if he had known what data that is lost, it would still be very difficult for Bob to actually ask for the retransmission and have the media player display it. If we take a step back and ignore the fact that we are working with a DTN, the process of asking Alice for a retransmission would be the following:
1. Figure out a way to extract which pieces of data that are actually missing from the media player.

2. Set up a new streaming session (using e.g. RTSP) to Alice, with the absence of the media player. Hoping that she has cached all the data she has been streaming out into the network, so that Bob can ask for it.

3. Ask Alice for the specific data that Bob is missing.

4. Open a new media player session to play back the newly received data as playing it with the old media player session is currently impossible.

It should also be mentioned that the playback of the retransmitted data in a new media player session is neither a trivial task. If Bob moved out of range exactly when a partitioned video frame was sent, he could only get the first part. If he in the second retransmitted session received the last part, he would need to locate the already received first part of the frame, so the new media player session would be able to present the whole frame. If this is not done, the result would be that some frames would not be presented to the user. In addition, since we are working with a DTN and disruptions can occur frequently, this whole process, including the part of retransmission, could be very difficult and time demanding.

As a result, we conclude that to provide the user with different ways of expressing data of interest, we need the signaling system to allow the media player to perform fine grained instructions. This means that the media player should be able to control as much as possible of the signaling system, thus the signaling system and the functionality of the media player will rely heavily on each other. As a result, we have chosen to approach our solution of the signaling of data from the consumers perspective, meaning that we provide most features to the consumer. From this perspective we will give the user possibility to get an overview of the distributed data in the network, and making the user choose which parts of the stream it wants to have retransmitted. By choosing this approach, we omit the process of distributing data from the senders side and considers this out of the scope of this thesis.

Since we have argued that more information and customizability should be given to the users through the media player, we must include these requirements into the signaling system. Therefore, we divide the signaling system into the following components:
- **Stream Control Manager** - Controls the sending of the AV data

- **Meta Data Manager** - Manages information on AV data located in the network

In the following, we thoroughly explain these components.

**Stream Control Manager**

To be able to control the playback of a media stream, we would need a Stream Control Manager. This Stream Control Manager needs to provide a way to control the sending of AV data in the network. This includes both the sending of data from the source as well as from the intermediate caching nodes. In addition, it should support retransmission of data from both the source and the intermediate caching nodes.

To support the requirements of the media player, this management would need the following features:

- **Start** the sending of a stream
- **Stop** the sending of a stream
- **Fetch** only a part of a stream
  - From the source
  - From the intermediate caching nodes
- **Skip** parts of a stream for
  - Fast forward
  - Fast backward

**Meta Data Manager**

To be able to locate and identify lost data, some meta data has to be maintained. Where this meta data should be maintained is however another question. Therefore we analyze two different meta data management models, namely:

- **central**, or
- **distributed**
If we choose to have a central node in the network maintaining all of the meta data for each streaming session, we can save a lot of network usage. As a result, each node must query this central node to receive all the information they need, to get an overview of the data distributed in the network. However, when working with a DTN, it is highly likely that a connection to this node cannot be established at all times, because of potential partitioning. As a result, we can see that this model does not fit the task.

The distributed model, might in our case, be more suitable. In this way, consumers does not need to rely on a single point of failure such as a central meta data management. Instead meta data are scattered out in the network. How this data is scattered is however out of the scope for this thesis.

Therefore, we propose a simple approach to let each node maintain meta data for its locally cached AV data. Unfortunately, this requires a lot more network usage, as each node that wants an overview of the distributed AV data would need to send meta data requests to all the nodes. This can either be done by flooding, or just by sending to a subset of the nodes in the overlay.

3.2 System Design

The purpose of the system is to handle the communication and signaling of media streams in DTNs. It relies on the underlaying overlay layer for delivering data to the respective destination. The signaling layer is the core element in this thesis and we use this section to discuss the sending of control messages, meta data messages and data messages.

3.2.1 Architecture

In Figure 3.8, we show the placement of the signaling layer in the network stack. The figure shows that the signaling layer in our architecture is located just below the application layer and just above the overlay layer. In the TCP/IP model however, the overlay and signaling layer belongs to the application layer.

System Components

In this section we introduce the different abstract components of our system. Figure 3.9 show an overview of the system with the following components in the signaling layer:
Figure 3.8: Network stack

Figure 3.9: System components
• **Application Programming Interface** - Interface to allow applications to control the signaling system

• **Stream Control Manager**
  
  - **Dispatcher** - Generating, sending and flushing AV data
  
  - **Signaler**
    
    * **Stream Control Signaler** - Signal the control of data transmission
    
    * **Meta Data Signaler** - Signal the sending and receiving of meta data. The meta data is gathered and stored in the Meta Data Manager.

• **Meta Data Manager** - Maintains meta data for local and remote data (received from Meta Data Signaler). The local meta data contains reference to the AV Data Cache in the overlay layer.
  
  - **Meta Data Storage**

  The components in the transport layer are not a part of our signaling system, but since we assume that the caching is done in the overlay layer, we include them in the figure.

### 3.2.2 Meta Data Manager

The purpose of the Meta Data Manager is to maintain an overview of where the AV data is located. On the source node and the intermediate caching nodes, this Meta Data Manager will only maintain a overview of the local AV data. On the consumer node, the Meta Data Manager can however maintain an overview of AV data distributed in the entire network including the AV data located locally. To maintain an overview of AV data in the entire network, signaling of meta data is required (discussed in detail in Section 3.2.3). The purpose of the Meta Data Signaler is thus to gather meta data from other nodes in the network and give the gathered meta data to the Meta Data Manager. The Meta Data Manager must thus support the following operations:

• Instruct the Meta Data Signaler to request meta data from other nodes

• Send and receive meta data to and from the Meta Data Signaler
Since our signaling system should be highly customizable to the user, the Meta Data Manager would also need to provide some control to the application layer. It should though be noted, that the application layer should not communicate directly with the Meta Data Manager, but rather communicate through the application programming interface (illustrated in Figure 3.9). To support the requirements set in the requirements section, the Meta Data Manager must provide the following functions to the application layer:

- **Get meta data** - Instruct the Meta Data Signaler to gather meta data, and return the meta data to the user

- **Get highest received sequence number** - So the application can know the size of the progress bar

- **Get latest received sequence number** - To instruct the application of which AV packets that has been received (in receiving order)

**Data Model**

In this thesis, we assume that AV data is cached in the overlay layer. Therefore, when the Dispatcher component at the source (discussed in detail in Section 3.2.3) sends and AV data packet (including meta data) to the network, the overlay layer will cache the outgoing packet if the node is performing caching. Also, when the overlay layer in another node receives a packet from the network, it would cache the packet in the AV Data Cache if this node too, is performing caching.

Since the overlay layer handles caching of AV data packets, we assume that the overlay layer provides us with references to them. These references should be stored together with the meta data for each AV data packet located in the overlay layer. The meta data and the references, should be stored in the Meta Data Storage in the Meta Data Manager, so the other components can query it for meta data requests. Table 3.1 shows an example of the meta data stored together with AV data references in the Meta Data Storage. Here, the data pointers reference AV data packets cached in the AV Data Cache located in the overlay layer. The meta data (stream id and sequence number) is used to identify the packets that are cached in the overlay layer and which stream the packets belong to. A more detailed description of the meta data generation is discussed in Section 3.2.3.

Figure 3.10 illustrates an example of the content of the Meta Data Storage on a consumer after it has requested meta data from the network. The Node
attribute illustrates which overlay node that has which AV data (Node 0 is locally cached AV data).

### 3.2.3 Stream Control Manager

As discussed in the requirements section, our signaling system should support the streaming of data from a source to a consumer. In addition to this, it should allow a consumer to fetch missing data from both a source and intermediate caching nodes, if e.g., a network disruption has occurred.

Therefore we have divided the signaling of data transmission into two groups:

- Sender Oriented (Push based) - A source continuously send packets of data to a consumer
• Receiver Oriented (Pull based) - A consumer fetches specific packets of data from a source or intermediate caching nodes

In this thesis, we have chosen to take an initial sending oriented approach. By this we mean, that after a consumer has asked a source to start streaming, the source sends packets of data (push based) to the consumer continuously. Figure 3.11 a) shows how the pushing of data starts after the consumer has asked the source to send a stream. By doing this, it would be less overhead in the network, since the consumer do not have to ask the source for each packet it has not yet received (pull based). Figure 3.11 b) shows a pull based data transfer.

However, since we are working with DTNs and do not have the privileges of connection oriented transport protocols, the source will not be informed when a packet is lost in the network due to e.g. a network disruption. As a result, the source will not retransmit lost packets, and the consumer will experience missing data from the stream (Figure 3.12). Therefore, we introduce a receiver oriented data transfer model, to let the consumer require missing data (pull based) from any node it wants. This receiver oriented data transfer model will thus be able to request data from all the nodes in the overlay. As a result the user can choose to use the split screen functionality discussed in Section 3.1.1, having a media player window playing back the data that is fetching.
Dispatched

For the Meta Data Manager to be able to maintain an overview of the cached data, the Meta Data Manager needs to distinguish the different packets cached in the AV Data Cache. Therefore, AV data sent out to the network needs some meta data so the Meta Data Manager can know which and where in a stream a data packet belongs. When a consumer requests a stream from a source, the source must generate this meta data for each packet of encoded data (gathered from the application layer) it is sending to the consumer. As a result, we define each AV data packet to include both AV data and meta data. The format of an AV data packet is illustrated in Figure 3.13).

Since we are working with DTNs, packet loss is highly likely and we require that some nodes caches data for e.g. later retransmission. Since several streams can be present in the network at the same time, some nodes may be caching data from more than one stream at the time. For a consumer to be able to request retransmission of data from a node that is caching more than one stream, it is necessary for the caching node to keep track of the different streams that exists in the network. Therefore, it is important that AV data is sent together with meta data to distinguish itself from other AV data. The meta data that is needed to provide ambiguity is thus a stream identification and a sequence number.

The stream identification is a hash constructed of the source’s address and the URI\(^2\) of the stream the consumer is requesting. We include the

\(^2\)Unified Resource Identifier
source’s address in the hash to decrease the possibility of a hash collision. If we had not included them, a consumer requesting two streams with the same URI from two different sources at the same time (Figure 3.14), we would get a hash collision. By using a hash (we propose a 32 bit hash), the overhead per AV data packet would only be 4 bytes for the stream identification. If we had not used a hash, and included the sources address in addition to the URI as meta data in the AV data packet, the overhead per AV packet would be much higher. If we consider the example where an URI of a stream is ”teststream.avi”, and we assume the source address is IPv4 (4 bytes), the overhead would be 4 + strlen("teststream.avi") = 18 bytes. As we can see, we can reduce much overhead per AV data packet, by generating a hash instead.

The sequence numbers, should however be generated at a per packet basis, and thus constructed each time the source gathers data from the application layer. Therefore, the sequence numbers would identify where the AV data packet is located in a stream at the signaling level. The data encapsulated in the AV data packet, might be encapsulated in another container with its own protocol for fragmentation at the application layer (such as RTP). However, the implementation of the signaling system should keep this in mind, and support dynamic AV data packet sizes, so the sequence numbers can be relative to other protocols sequence numbering. The reason we have not used pre-existing packet formats (such as RTP) at the signaling level, is that these protocols has higher packet overhead.

Now, when the AV data packet is including the meta data, the packet is sent to the consumer through the overlay layer. If the overlay layer has enabled caching (e.g. it is performing caching), the AV data packet would be cached in the AV Data Cache at the overlay layer. A reference to the location where the packet is cached is then returned to the signaling layer, so the Meta Data Manager can keep track of the data cached in the overlay
Design

When a consumer requests some parts of a stream (receiver oriented), the data sending is much more simpler. In this case, the node receiving the request (any node with cached data), queries the Meta Data Manager and sends the AV data to the consumer if present.

We can summarize the data transmission with the following algorithm:

1. If the source is pushing data to the consumer (sender oriented)
   (a) Retrieve encoded AV data from the application layer
   (b) Make AV data packet
      i. Include meta data
         A. Stream identification
         B. Sequence number
   (c) Send the AV data packet to the consumer through the overlay
      i. Cache the AV data packet in the AV Data Cache in the overlay layer and return a reference to location of the cached data.

2. If a node is responding to specific data requests (receiver oriented)
   (a) Locate the cached data by looking up the sequence number in the Meta Data Manager
   (b) Send the cached AV data packet to the consumer through the overlay

Signaling

For the individual nodes to be able to communicate with each other at the signaling layer, they would need a set of common messages. As stated in the requirements section (3.1), these would include messages to signal meta data and AV data. Therefore, we have chosen to divide the signaling into two parts:

- Stream Control Signaler - Signaling AV data transmission
- Meta Data Signaler - Signaling meta data transmission
These are both illustrated in Figure 3.9 in the “Signaling” component.

Stream Control Signaler

The Stream Control Signaler is one of the most important parts of our system. Its purpose is to control the sending of AV data from both the source and the intermediate caching nodes. The following list gives an overview of the Stream Control Signaler messages (optional arguments is included in brackets):

- **PLAY <stream_id> <URI>** (sender oriented) - Instruct the source to push AV data to the consumer
- **PAUSE <stream_id>** - Instruct the source to pause the pushing AV data from a stream to the consumer
- **FINISH <stream_id> [sequence_number]** - Instruct either source or consumer that a streaming session is finished
- **ERR <stream_id> <error_number>** - Instruct the consumer that an error has occurred
- **GET_DATA <stream_id> <start_sequence_number> <end_sequence_number>** (receiver oriented) - Request specific AV data from a stream

PLAY <stream_id> <URI>

The purpose of the PLAY message is to start the sending of a file or a live feed from a source. For the source to send a stream to the consumer, the source would need to know which file or live feed to send. The file or live feed must therefore be identified by an URI which must be sent together with the PLAY message (how the consumer gets this URI is however beyond the scope of this thesis). In addition to the <URI>, a hash of the source, consumer, and the <URI> must be computed and included in the PLAY message. This hash is the <stream_id> and when the source responds to any messages from the consumer, this hash should be included in the message. Since the system relies on non-connection oriented communication, we use this hash to identify communication belonging to one streaming session. As a result, when the source starts to send data to the consumer, the data being sent
Design

should include the computed hash, so we know which session the data belongs to. The format of the <URI> and the hash algorithm for the <stream_id> is also beyond the scope of this thesis.

Since we are working with MANETs and want to minimize the network usage and setup handshake latency, we have chosen to not send a setup message to initiate a streaming session (as done in RTSP and SIP). Instead, we let the PLAY message initiate the stream setup as our system presently does not need any advanced setup options.

When the source receives the PLAY message, it answers by starting to send AV data packets to the consumer. To decrease the number of control messages in the network, the source will not reply with a specific message to indicate that the initiation of the playback was successful (as RTSP and SIP do). Instead, when the consumer receives the first AV packet of a stream it knows that the setup was successful. However, if the PLAY message did not result in a successful setup at the source (e.g. the stream with the URI was not located at the source), the source issues a ERR message to the consumer with a describing error message. If an ERR message is not sent back to the consumer, the consumer would never know what went wrong, and is unable correct the error (e.g. send the correct URI with another PLAY message). Since we assume reliable transport of signaling messages, the PLAY message will arrive at the source, and the consumer will receive an ERR message if something went wrong.

**PAUSE <stream_id>**

The PAUSE message is needed to pause (or resume) the sending of a stream with the stream identification <stream_id>. It should be noted that only pausing the media player and not the entire streaming session might be a better solution. By choosing this solution, the consumer would receive and buffer data that is ready to be played back when the media player is done pausing. However, if a pause is expected to last for a while, this message could be used.

**FINISH <stream_id> [sequence_number]**

If a source is sending a file instead of a live feed to the consumer, end of file might be reached at one point. When this happens the source must send a FINISH message to the consumer to indicate that there is no more data coming. This FINISH message will also include the sequence number of the last packet sent, so the consumer knows if it should be waiting for more AV data that may be delayed or lost in the network.
In the case where the consumer is streaming either a live feed or a file from the source and is not interested in continuing the streaming, the consumer sends a FINISH message to the source. The important difference between the FINISH message and the PAUSE message however, is that the FINISH messages should free up the resources at the source, and does not support resuming the same streaming session later. When a FINISH message is sent from the consumer, it is not necessary for the consumer to include any sequence numbers as the consumer is not interested in any more data anyways.

ERR <stream_id> <error_number>

The ERR message should be sent to a node in response to a message that resulted in an error. By having a table with error messages each node can look up the <error_number> to know what went wrong. By including a number instead of a describing error message together with the ERR message, some overhead might be lowered, as long messages in ASCII text takes more space than a number (we propose 1 byte = 256 different error messages). We leave it to the application to present the error message to the end user.

GET_DATA <stream_id> <start_sequence_number> <end_sequence_number>

The GET_DATA message is used to send a request to any node in the overlay (running our signaling system) for a specific range of AV data packets that a consumer is interested in. However, since the consumer might not know which node that has the AV data it is interested in, its function might not always be useful. This is why we need the Meta Data Signaler. By having the Meta Data Signaler query the network for meta data, the consumer can construct an index telling where the AV data is located in the network. Now, when the consumer needs some AV data, it can know where it is located, and use the GET_DATA message to request the specific AV data from that location.

In Figure 3.15 we show three examples of a stream management signaling between two peers. In a) The consumer initiates a PLAY request to the source, which the source answers by starting to push AV data packets. The consumer has computed a hash (731) as a stream id which the source includes in its response. In the first case, a file (test.avi) is being streamed. When the file is fully read and sent, the source sends a FINISH message including the stream id (731) and the last sequence number sent (123), to the consumer. Almost the same happens in b) but this time, a live feed is being played back (with stream id 871). The consumer indicates at a later time that it has received what it needs, and sends a FINISH message to the source which terminates
Figure 3.15: Stream control signaling
the session and frees up resources. In c) however, the file "notavailable.avi" was not available at the source, so an ERR message was returned with an error number to identify what caused the error. All these scenarios would be the same even if intermediate nodes were present, as these would just forward the messages to the correct node.

Meta Data Signaler

As we discussed in the GET_DATA message section, the Meta Data Signaler is also very important. For the consumer to be able to fetch AV data (receiver oriented) from the network without overflowing the network with GET_DATA messages, we let the consumer ask the nodes in the network to send us a list describing which data they are caching. Even if a user is not interested in retrieving data from the other nodes, requesting these lists from the other nodes will provide an overview of where the AV data is located in the network. By doing this, the requirement to provide a network overview in the media player (Section 3.1.1) can be fulfilled. The Meta Data Signaler will therefore need the following meta data signaling messages:

- **GET_METADATA** <stream_id> <sequence_number_from> <sequence_number_to>
  - Request a list of meta data belonging to a stream with sequence numbers <sequence_number_from> to <sequence_number_to>

- **METADATA** <stream_id> <sequence_number_1> ... <sequence_number_n>
  - A list of meta data belonging to a stream with sequence numbers <sequence_number_1> to <sequence_number_n> a node has cached

GET_METADATA <stream_id> <sequence_number_from> <sequence_number_to>

The GET_METADATA message is used to request a list of which AV data packets in the range <sequence_number_from> to <sequence_number_to> belonging to a stream (<stream_id>) that a node is caching. When a node receives this message and has AV data packets in the requested range and belonging to the requested stream, the node must send a METADATA message in return. The GET_METADATA message is sent when either the application layer wants an overview of the network, or the signaling system itself wants to perform data fetching (e.g. the signaling system is set to retrieve a complete stream without any data gaps).

METADATA <stream_id> <sequence_number_1> ... <sequence_number_n>
Design

Figure 3.16: METADATA message with sequence numbers

<table>
<thead>
<tr>
<th>Caching Node</th>
<th>Sequence Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.0.2</td>
<td>1,2,3,4,5,15,16,17,18,23,...</td>
</tr>
<tr>
<td>10.0.0.3</td>
<td>1,2,3,4,5,6,7,8,9,10,11,15,17,18,...</td>
</tr>
<tr>
<td>10.0.0.5</td>
<td>19,20,21,22,23,24,...</td>
</tr>
<tr>
<td>10.0.0.8</td>
<td>6,7,8,9,15,16,17,18,22,23,24,25,...</td>
</tr>
</tbody>
</table>

Table 3.2: Index of where packets belonging to one stream is located in a network

This message is used as a response to a received GET_METADATA message. When a GET_METADATA message is received, the Meta Data Signaler will query the Meta Data Manager for a list of the locally cached AV data packets in the requested range, belonging to a stream. The Meta Data Signaler will then generate a METADATA message containing the sequence numbers of these packets. Figure 3.16 illustrates an example of a METADATA message. This message is then returned to the node that sent the GET_METADATA message. When a node receives a METADATA message containing sequence numbers, it forwards the message to the Meta Data Manager. The Meta Data Manager will then store the list of all the sequence numbers belonging to a stream in the Meta Data Storage. Table 3.2 illustrates an example of a resulting meta data index of where data is located in the network. This index is also available for the application to query through through the API.

In Figure 3.17 we show an example of meta data gathering in the network. If the signaling system gets a request from either the application layer (e.g. a user wants an network overview) or the signaling layer itself does automatic meta data gathering, the Meta Data Signaler generates a GET_METADATA message. This message is thus flooded to the network (shown by the black arrows in the figure). The dotted arrows show how each node having sequence numbers belonging to the requested stream replies with a METADATA message. These messages are then merged in the signaling layer.

3.3 Summary

The signaling system must provide an API to the application for customizing the streaming process. This API is communicating with both the Meta Data Manager and the Stream Control Manager. The Stream Control Manager
Figure 3.17: Generate overview to media player
is responsible for handling data transfer and signaling. The signaling has been divided into a Stream Control Signaler and a Meta Data Signaler component. These components handle all signaling between the different nodes in a network. The Stream Control Signaler is responsible for handling data transmissions, while the Meta Data Signaler is handling meta data exchange. The Stream Control Manager sends and queries meta data to and from the Meta Data Manager as the Meta Data Manager is responsible for storing and controlling both local and remote meta data. Since the overlay layer handles AV data caching, the Meta Data Manager only contains meta data with references to the data cached in the overlay layer. When a data request for cached AV data arrives at the Stream Control Signaler, the Stream Control Signaler instructs the Dispatcher to start sending the data requested. The Dispatcher then queries the Meta Data Manager which returns a reference to the overlay layer. When the Dispatcher receives the returned reference, it instructs the overlay layer to send the data with the respective reference to the node requesting the data (the consumer). However, before a consumer requests data from a caching node, it should know if the caching node has this data first. This can be accomplished by querying the network for the meta data regarding a stream using the Meta Data Signaler. Therefore, when a node receives meta data from other nodes in a network after a query, it merges all the meta data in the Meta Data Manager. When the node wants some data regarding a stream, it queries the Meta Data Manager requesting a list of which caching nodes in the network that has the certain data. Based on this list, it can determine from which node it wants to request data. For the Meta Data Manager to distinguish different different AV data, it relies on the Dispatcher to generate AV data packets with AV data and meta data. The meta data included to support ambiguity is thus a stream id and sequence numbers to indicate where a AV data packet is located in a stream.

The following gives a summary of the signaling messages needed in the signaling system:

- **PLAY <stream_id> <URI>** - Instruct the source to push AV data to the consumer
- **PAUSE <stream_id>** - Instruct the source to pause the pushing of AV data from a stream to the consumer
- **FINISH <stream_id> [sequence_number]** - Instruct either source or consumer that a streaming session is finished
• ERR <stream_id> <error_number> - Instruct the consumer that an error has occurred

• GET_DATA <stream_id> <start_sequence_number> <end_sequence_number>
  - Request specific AV data from a stream

• GET_METADATA <stream_id> <sequence_number_from> <sequence_number_to>
  - Request a list of AV data packets belonging to a stream

• METADATA <stream_id> <sequence_number_1> ... <sequence_number_n>
  - A list of AV data packets belonging to a stream that a node has cached
Chapter 4
Implementation

In this chapter, we describe the implementation of our system, which is presented in the design chapter. We start by giving a short overview of the different components, and then provide more details on their functionality.

4.1 Implementation Overview

Figure 4.1 shows an abstract overview of the implementation, with our system indicated in the DTSS (Delay Tolerant Stream Signaling) component. As the figure shows, DTSS is divided into two parts:

- dtssd - Delay Tolerant Stream Signaling Daemon, and
- libdtss - Library for Delay Tolerant Stream Signaling.

The dtssd is the signaling core, responsible for handling meta data and stream control management, while libdtss is a library that provides an API to the application layer. As current media streaming applications (e.g., media players) are implemented together with different signaling protocols, problem arises when the application suffers from failure unrelated to the signaling protocol. An example of such failure can be a result of problems in the decoding process, and if this failure cause the application to crash or hang, the signaling protocol will also be affected. Therefore, instead of implementing our signaling system as a part of another application, our signaling system will operate on its own, and can be viewed as an own layer. This layer is (as shown in the figure) located just above the overlay layer, and just below the application layer. This means, that when an application wants to start a streaming session, it has to communicate with the signaling layer that handles the signaling of the streaming process. By doing this, the signaling
Figure 4.1: DTSS

layer would not be affected if the application e.g., crashes and the application is able to resume the previous actions when restarted.

### 4.1.1 Applications API (libdtss)

For applications (such as media players) to be able to communicate with the signaling core (dtssd), we have developed a library. This library is called libdtss (library for delay tolerant stream signaling) and is developed to provide the application with fine-grained control of the signaling system. For an application to be able to use our signaling system, this library must be included in the application. The library makes use of IPC to talk to a already running dtss daemon, using unix sockets for instructions and status information, and shared memory for data transmissions.

### 4.1.2 Delay Tolerant Stream Signaling Daemon (dtssd)

dtssd is the component that handles the signaling core. For the entities in the network to be able to signal the actions of a stream, all nodes involved must have this component running in the background (daemon) listening for instructions. These instructions can be received from the overlay layer e.g., signaling messages, and from the application layer, e.g., a media player initiating the playback from a source. As the overlay layer and signaling layer is a part of a virtual network stack located in the application layer (Figure 3.8), our system needs to be implemented in user space. As a result, dtssd must be started on demand (or together with the operating system booting process), if a device should participate in the signaling system. To illustrate how the daemons on the different nodes communicate, consider a scenario with three nodes (Figure 4.2). Assume that the consumer wants to play back
Implementation

Figure 4.2: Three node example

a stream from the source using a media player. The media player connects to the signaling system and instructs it to start a streaming session from the source. The signaling system receives the instruction from the media player and sends a signaling message to the source through the overlay layer, instructing it to start streaming. Since the route in the figure includes one intermediate caching node, the signaling message is routed through this node. When the signaling message arrives at this node, the overlay layer caches the message, and forwards it if the destination is known. The signaling message is also sent to the signaling system, for processing.

When the overlay layer at the source receives the signaling message from the Intermediate Overlay Node, it sends the message to the signaling system. When the signaling system receives this message, it starts pulling data from the application layer (e.g., encoder) and pushing it to the consumer through the overlay layer.

4.1.3 Media Sessions

When an application wants to stream data from a source, it connects to dtssd. dtssd then initiates a media session by creating a `media_session_t` structure to store information and state about the media session. An identifier to this media session is then returned to the application, which the application uses for further handling. When this media session is created, dtssd can issue signaling messages to the source. However, to simplify the handling of node addresses, we assume that the overlay provides us with an identifier for each node in the overlay. As a result, dtssd must acquire an overlay node identifier from the overlay layer to be able to send signaling messages to the source. Thus, when dtssd creates a media session between a source and a consumer, it queries the overlay layer using the source’s address, for a this identifier. Since the transport layer should handle route changes, this identifier will always remain the same. When dtssd receives this overlay node identifier from the overlay layer, it stores it in the media session structure, and signaling messages can be sent to the source. In addition, when
dtssd receives data from the overlay, it is received together with an overlay node identifier to indicate where the data came from. Thus, when the source receives a play¹ message for a stream that is not already initiated, it creates a media session structure with the overlay node identifier the signaling messages was received from. This identifier is then used when the source sends data to the consumer.

Figure 4.3 shows a media session between a source and a consumer, with overlay node identifiers for each overlay node (dt_node_t). As we can see, the media_session_t is the communication channel between the Media Player and dtssd, but can also be viewed as a virtual session from the source to the consumer (grey horizontal line). As the overlay layer handles route changes, the consumer requires only one overlay node identifier (black horizontal line), for all communication to and from the source, even if network paths change.

The media session structure (media_session_t) is listed in Figure 4.4 with an explanation of the different variables listed in the following:

- **const char *uri** - The URI of the file to be streamed from the source

¹Note that play is used as setup and play
Implementation

- `char *shm` - A pointer to a memory area that the dtssd running at the consumer use to transfer data to the connected application (IPC)

- `dt_node_t dt_sid` - Overlay node identifier for either the source or the consumer

- `int state` - The state of the media session. Possible values is listed in Figure 4.15

- `int source` - Set to one if the current node is a source

- `uint32_t stream_id` - A unique stream identification generated from the source’s address and the URI

- `seq_t seq` - Latest sent sequence number at the source or the latest received sequence number at consumer and intermediate overlay nodes

- `seq_t final_seq` - Received together with a SIG_FINISH message. The value is the sequence number of the final sent AV data packet from the source

- `int fd` - A file descriptor used to read data from the application layer at the source

- `meta_data_t *first` - Pointer to the first meta data record

- `meta_data_t *last` - Pointer to the last meta data record

- `meta_data_t *current` - Pointer to the last data record that the application layer has queried

- `int bitmap[NODE_MAX][BITMAP_SIZE]` - A bitmap of where AV data packets is located in the network. `bitmap[0]` is always locally cached AV data

It should be noted that this structure will not be fully populated in any node. As an example, consider the `int fd` variable. This is only needed at the source, since the source’s purpose is to gather AV data from the application layer. Since the consumer will never fetch data from the application to send to another node, this is not needed on this end. Table 4.1 shows a complete overview of which variables is set in which type of node.

The variables in the media session structure is discussed in more detail in the following sections.
### Table 4.1: Which variables are set in which type of node

<table>
<thead>
<tr>
<th>Variable</th>
<th>Source</th>
<th>Intermediate Overlay Node</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>const char *uri</td>
<td>set</td>
<td>unset</td>
<td>unset</td>
</tr>
<tr>
<td>char *shm</td>
<td>unset</td>
<td>unset</td>
<td>set</td>
</tr>
<tr>
<td>dt_node_t dt_sid</td>
<td>set</td>
<td>unset</td>
<td>set</td>
</tr>
<tr>
<td>int state</td>
<td>set</td>
<td>Not Implemented</td>
<td>set</td>
</tr>
<tr>
<td>int source</td>
<td>set</td>
<td>unset</td>
<td>set</td>
</tr>
<tr>
<td>uint32_t stream_id</td>
<td>set</td>
<td>set</td>
<td>set</td>
</tr>
<tr>
<td>seq_t seq</td>
<td>set</td>
<td>set</td>
<td>set</td>
</tr>
<tr>
<td>int fd</td>
<td>set</td>
<td>unset</td>
<td>unset</td>
</tr>
<tr>
<td>meta_data_t *first</td>
<td>set</td>
<td>set</td>
<td>set</td>
</tr>
<tr>
<td>meta_data_t *current</td>
<td>unset</td>
<td>unset</td>
<td>set</td>
</tr>
<tr>
<td>meta_data_t *last</td>
<td>set</td>
<td>set</td>
<td>set</td>
</tr>
<tr>
<td>int bitmap[1][1]</td>
<td>set</td>
<td>set</td>
<td>set</td>
</tr>
</tbody>
</table>

If different streams should be played back at the same time, several media sessions must be initiated from the application to the dtssd, as shown in Figure 4.5. In this figure, three different media players are communicating with dtssd, with each media player having its own media session.

### 4.2 Overlay API

Presently, there exists no overlay implementations that fits our requirements. As DT-Stream is an ongoing project, we expect such an overlay to be available in the near future. Until then, we assume that the overlay implementation will have an API with the blocking functions\(^2\) illustrated in Figure 4.6. The syntax of the diagram in the figure is:

\[
\text{<function-name> (<arg1_name>:<arg1_type>,...): <return type>}
\]

In the following, we discuss the purpose of these functions, which are prefixed by "dt_" to indicate Delay Tolerant functions.

Since we assume that the overlay layer handles reliable transmission, these function will only return erroneous if fatal errors occurs. For instance, the data cache is full or the overlay layer has crashed.

```c
dt_node_t
dt_init (const struct in_addr dst_addr, const uint16_t dst_port);
```

\(^2\)Blocking: Waiting for the completion of an event
typedef struct media_session
{
    const char *uri;
    char *shm;
    dt_node_t dt_sid;
    int state;
    int source;
    uint32_t stream_id;
    seq_t seq;
    seq_t final_seq;
    int fd;
    meta_data_t *first;
    meta_data_t *current;
    meta_data_t *last;
    int bitmap[NODE_MAX][BITMAP_SIZE];
    pthread_mutex_t bitmap_lock;
    pthread_mutex_t queue_lock;
} media_session_t;

Figure 4.4: struct media_session_t

Figure 4.5: Several streaming sessions

Figure 4.6: Overlay API
When the signaling layer has control messages or data messages to send to another node in the overlay, it must be sent through the overlay layer. However, as previously discussed, to simplify the handling of the destination addresses, we assume that the overlay layer has identifiers for all nodes in the overlay. Therefore, when the signaling layer has data to be sent to a destination it must use a overlay node identifier, so the overlay layer knows which overlay node to send to. Since the signaling layer initially do not know which overlay node identifier to use when sending data to a destination, it must use this function to acquire it. This function thus takes the destination node’s address and port (\texttt{dst\_addr} and \texttt{dst\_port}) as arguments, and returns the overlay node identifier for that destination. The name of the overlay node identifier type is \texttt{dt\_node\_t}.

\begin{verbatim}
void *
dt_send (dt_node_t dt_sid, uint8_t type, const char *buf,
         size_t length);
\end{verbatim}

When the signaling layer has acquired an overlay node identifier (by calling \texttt{dt\_init}), the signaling layer can send data through the overlay layer by calling \texttt{dt\_send}. This function requires an overlay node identifier (\texttt{dt\_sid}), what type of data that is to be sent (\texttt{type}), the data itself (\texttt{buf}), and the length of the data (\texttt{length}). The different types of data (\texttt{type}) can either be \texttt{DT\_CONTROL} or \texttt{DT\_DATA} (Figure 4.7), where these indicates the reliability of the data in the overlay layer. \texttt{DT\_CONTROL} indicates that the data sent is guaranteed to be delivered (as previously discussed), while \texttt{DT\_DATA} is sent without any guarantee. \texttt{DT\_CONTROL} is therefore useful when sending signaling messages that require reliability, while \texttt{DT\_DATA} is useful to send AV data packets that don’t require reliability.

Since the overlay layer is responsible for caching data that is sent to the network, this function returns a pointer to the address where the data is cached in the overlay layer. If this cached data is to be retransmitted at another time (e.g., a consumer requests missing data), \texttt{dt\_flush} should be used to send this data.

If an error occurs during the sending of data, this function should return \texttt{NULL}.

\begin{verbatim}
void *
dt_receive (dt_node_t *dt_sid, uint32_t *intermediate,
             size_t *length);
\end{verbatim}

This function is used by the signaling layer to receive data that from the overlay layer that has been received from the network. The data that has been
received from the network, could either be destined for this node (consumer or source), or just forwarded to another destination (intermediate overlay node). If the data is not destined for this node, the value of the address `intermediate` is pointing to will be set to 1, indicating to the signaling layer that it is only an intermediate overlay node. If the data is destined for this node, the value of the address `intermediate` is pointing to will be set to 0, indicating to the signaling layer that it is either a source or a consumer, and that it has arrived at the destination. `dt_receive` also takes a pointer to a overlay node identifier (`dt_sid`) as a parameter to store which overlay node the data has been received from.

`dt_receive` returns a pointer to where the data is cached in the overlay, and the length of the cached data in the third parameter (`length`).

```c
int dt_flush (dt_node_t dt_sid, uint8_t type, const char *buf, size_t length);
```

When the signaling layer receives a request to send data that is cached in the overlay layer to the consumer, this function must be called. As the Meta Data Manager in the signaling layer is responsible for maintaining references to the data cached in the overlay, the signaling layer is able to provide `dt_flush` with the reference it needs to send the data to the consumer. This reference is a pointer that is previously returned to the signaling layer when either, `dt_receive` or `dt_send` has been called. Therefore, when the signaling layer calls `dt_flush` with this data reference (`buf`), `dt_flush` sends this data to the overlay node referenced by `dt_sid`.

If an error occurs, `dt_flush` returns 0.

### 4.3 Applications API (libdtss)

For an application to be able to use the signaling system, it needs a way to communicate with it. For this reason, we have made a library that an
application can include to have access to the signaling system’s interface. This interface is therefore constructed based on what an application needs to control a stream through DTNs. In addition, this library must also support the requirements set for the delay tolerant media player described in Section 3.1.

This library is constructed of the files libdtss.c and libdtss.h found in Appendix A.2. Figure 4.8 shows the most important functions of the libdtss component.

```c
int dtss_setup_stream (const char *dst_addr, const uint16_t dst_port, const char *uri);
```

This function sets up a local session between an application and the signaling system. It returns a media session handle (int) for further handling of the signaling process. `dst_addr` and `dst_port` is the address and port of the source, and the stream is identified by `uri`.

```c
int dtss_start_stream (const int session);
```

This function starts the stream with the session handle `session` initiated in a previous call to `dtss_setup_stream`.

```c
int dtss_pause_stream (const int session);
```

This function pauses a stream with the session handle `session` started by a previous call to `dtss_start_stream`.

<table>
<thead>
<tr>
<th>libdtss</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtss_setup_stream(dst_addr:const char *,dst_port:const uint16_t,filename:const char *): int</td>
</tr>
<tr>
<td>dtss_start_stream(session:const int): int</td>
</tr>
<tr>
<td>dtss_pause_stream(session:const int): int</td>
</tr>
<tr>
<td>dtss_finish_stream(session:const int): int</td>
</tr>
<tr>
<td>dtss_get_highest_seq(session:const int): seq_t</td>
</tr>
<tr>
<td>dtss_get_latest_seq(session:const int): seq_t</td>
</tr>
<tr>
<td>dtss_get_data(session:const int,seq:const seq_t,buffer:char *,length:size_t): ssize_t</td>
</tr>
<tr>
<td>dtss_gather_data(session:const int,from_seq:const seq_t,to_seq:const seq_t,dt_node:int): int</td>
</tr>
<tr>
<td>dtss_get_metadata(session:const int,from_seq:const seq_t,to_seq:const seq_t,signal:int,dt_node:int,buf:char *,length:size_t): int</td>
</tr>
</tbody>
</table>

Figure 4.8: libdtss
int
dtss_finish_stream (const int session);

This function finishes a stream with the session handle session started by a previous call to dtss_start_stream.

seq_t
dtss_get_highest_seq (const int session);

This function returns the highest sequence number belonging to a stream with the session handle session. By retrieving this sequence number, an application can calculate the width of a progress bar discussed in Section 3.1.1.

seq_t
dtss_get_latest_seq (const int session);

This function returns the latest sequence number arrived, from the stream with the session handle session. For instance, if the signaling layer has received AV data packets with sequence numbers in the following order: 1, 3, 2, 4, each call to dtss_get_latest_seq will return the latest sequence number from this order. Therefore, the first call will return 1, second call will return 3, third call will return 2 and the fourth call will return 4. By doing this, an application can populate a progress bar by subsequent calls to this function, having the latest arrived packet marked in the progress bar first.

ssize_t
dtss_get_data (const int session, const seq_t seq,
               char *buf, size_t length);

This function retrieves an AV data packet with sequence number seq from the stream with the session handle session. By asking specific which AV data packet to retrieve, an application can itself choose which part of a stream it wants to present to the users, as described in the design chapter (Section 3.1.1).

int
dtss_gather_data (const int session, const seq_t from_seq,
                  const seq_t to_seq, int dt_node);
This function is used to gather AV data in the range \texttt{from\_seq} to \texttt{to\_seq} belonging to the stream with session handle \texttt{session}, from the overlay node referenced by \texttt{dt\_node}. The signaling layer will only fetch AV data that is not already cached in the overlay layer. If both \texttt{from\_seq} and \texttt{to\_seq} is 0, the signaling layer should fetch the complete stream. If \texttt{dt\_node} is zero, the signaling layer would flood \texttt{GET\_METADATA} messages to all the nodes in the overlay. Thus, when \texttt{METADATA} messages is received as a response, the signaling layer would send \texttt{GET\_DATA} messages to the nodes that have cached AV data packets in the range \texttt{from\_seq} to \texttt{to\_seq}.

\begin{verbatim}
int dtss_get_metadata (const int session, const seq_t from_seq,
                     const seq_t to_seq, int signal,
                     const int dt_node, char *buf,
                     size_t length);
\end{verbatim}

This function gathers and returns meta data for AV data packets in the range \texttt{from\_seq} to \texttt{to\_seq} that belongs to the stream with session handle \texttt{session} from the overlay node \texttt{dt\_node}. If \texttt{signal} is set to 1, this function instructs the signaling core to flood \texttt{GET\_METADATA} messages to all other nodes in the overlay, unless \texttt{dt\_node} is non-null. If \texttt{dt\_node} is more than 0, this indicates that a specific node in the overlay should be queried for meta data.

This function will return 1 on success with a bitmap of the sequence numbers a node is caching in \texttt{buf}, and the length of the buffer in \texttt{length}. If this function is called with \texttt{signal} set to 1, this function will probably return a bitmap that is not fully updated, as the process of retrieving meta data from the overlay may take some time. Thus subsequent calls to this function should be called periodically with \texttt{signal} set to 0, to check if a bitmap relative to the overlay node referenced by \texttt{dt\_node}, has been updated. As previously mentioned, if this function is called with \texttt{dt\_node} set to 0, and \texttt{signal} set to 1, the signaling core floods \texttt{GET\_METADATA} messages, but as this function can only return meta data for one node at the time to the application, only the local meta data will be returned in \texttt{buf}.

This function thus supports the requirements to provide a Data Chunk Index discussed in Section 3.1.1.

A better solution for this process is to let applications register callback functions that are called whenever meta data arrives at the signal core and the meta data storage is updated. This is however left to future work.
4.4 Signaling Core (dtssd)

The signaling core is implemented in the two files; dtssd.c and dtssd.h (Appendix A.1), and is responsible for handling all meta data and stream control management. The signaling core receives instructions from the application layer through IPC from the libdtss. Figure 4.9 shows the most important functions of the dtssd component, which are explained in detail in the following.

```c
void *
handle_ipc_thread (void);
```

This function is running as an own pthread\(^3\) and is handling the IPC between dtssd and the libdtss library. Since we should support several applications using the signaling core at the same time, this function implements the use of unix sockets for each of the applications that want to connect to it through the libdtss library. These unix sockets will only handle instructions and status information to and from libdtss, while AV data will be transferred through shared memory. The address for the shared memory will however be transferred through the socket at the initial setup (dtss_setup_stream), for later use when data needs to be transferred.

Since the communication between dtssd and libdtss is implemented by using sockets, we also require an IPC message protocol. This protocol is very simple as this is not our primary interest in this thesis. The message format is described in Figure 4.4 in Backus-Neur Form (BNF), while the message handling is illustrated in Figure 4.11.

\(^3\)pthread: POSIX Threads
\[
\begin{align*}
\langle \text{ipc message} \rangle & \rightarrow \langle \text{setup} \rangle | \langle \text{command} \rangle | \langle \text{return} \rangle \\
\langle \text{setup} \rangle & \rightarrow \text{DTSS\_SETUP} \ (\text{dst. addr}) \ (\text{dst. port}) \ (\text{URI}) \\
\langle \text{dst. addr} \rangle & \rightarrow \text{char} \ * \\
\langle \text{dst. port} \rangle & \rightarrow \text{uint16} \_t \\
\langle \text{URI} \rangle & \rightarrow \text{char} \ * \\
\langle \text{command} \rangle & \rightarrow \langle \text{commands} \rangle \ (\text{dtss session id}) \ (\text{opt. seq. number}) \\
\langle \text{dtss session id} \rangle & \rightarrow \text{uint8} \_t \\
\langle \text{commands} \rangle & \rightarrow \text{DTSS\_START\_STREAM} | \text{DTSS\_PAUSE\_STREAM} | \text{DTSS\_FINISH\_STREAM} | \text{DTSS\_GET\_DATA} | \text{DTSS\_GATHER\_DATA} | \text{DTSS\_GET\_METADATA} | \text{DTSS\_HIGHEST\_SEQ} | \text{DTSS\_LATEST\_SEQ} \\
\langle \text{opt. seq. number} \rangle & \rightarrow \text{uint32} \_t | \text{void} \\
\langle \text{return} \rangle & \rightarrow \langle \text{status} \rangle | \langle \text{dtss session id} \rangle \\
\langle \text{status} \rangle & \rightarrow \text{int} \\
\end{align*}
\]

Figure 4.10: IPC message BNF
void *
handle_transport_thread (void);

The purpose of this function is to handle instructions and data from the
overlay, as opposed to handle_ipc_thread that handles instructions from
the application layer. handle_transport_thread is also implemented as a
pthread, and calls the blocking function dt_receive in the overlay API to
retrieve data from the overlay. The format of the DTSS network packets
(received from dt_receive) is illustrated in Figure 4.12, and listed as BNF
in Figure 4.4. Figure 4.12 a) show the complete DTSS packet (dtss_pkt_t),
while Figure 4.12 b) and c) shows how AV data packets (avdata_t) and
DTSS control packets (control_t) are encapsulated in the DTSS packets.
These packet formats are slightly different from what we stated in the design,
to reduce overhead. For instance, stream id is a field in the DTSS packet
header and not in the encapsulated AV data packet and DTSS control packet.

void *
dispatch_thread (void);

This function is also running as an own pthread and has the responsibility
to send and flush AV data packets to the consumers. It will loop through
Chapter 4

Figure 4.12: DTSS packet format

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtss header</td>
<td>type</td>
</tr>
<tr>
<td>dtss payload</td>
<td>stream id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>av data</th>
<th>sequence number</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>SIG_PLAY (URI)</td>
<td>SIG_PAUSE</td>
</tr>
<tr>
<td></td>
<td>SIG_ERROR (error number)</td>
<td>SIG_FINISH (sequence number)</td>
</tr>
<tr>
<td></td>
<td>SIG_GET_METADATA (from seq) (to seq)</td>
<td>SIG_METADATA (from seq) (to seq) (metadata)</td>
</tr>
<tr>
<td></td>
<td>SIG_GET_DATA (from seq) (to seq) (metadata)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>metadata</th>
<th>sequence number</th>
<th>metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>from seq</td>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>to seq</td>
<td>sequence number</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>type</th>
<th>AVDATA</th>
<th>CONTROL</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved</td>
<td>void</td>
<td></td>
</tr>
<tr>
<td>length</td>
<td>uint16_t</td>
<td></td>
</tr>
<tr>
<td>stream id</td>
<td>uint32_t</td>
<td></td>
</tr>
<tr>
<td>sequence number</td>
<td>uint32_t</td>
<td></td>
</tr>
<tr>
<td>error number</td>
<td>uint8_t</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>uint8_t (data)</td>
<td>uint8_t</td>
</tr>
</tbody>
</table>

Figure 4.13: Network message BNF
all the media sessions, sending the data scheduled to be delivered for each session. The data to be sent is either retrieved from the application layer or flushed from the overlay layer. When data is retrieved from the application layer, this function must generate an AV data packet and use dt_send to send data to the consumer. The generation process includes the meta data generation responsible for creating an unique sequence number for each AV data packet, and attaching the stream_id set in the media session structure. If the data should be flushed from the overlay layer (cached data), it uses dt_flush.

If this function is retrieving data from the application layer and the application layer has no more send (e.g., the live stream has ceased), this function will call dtssd_signal_stream_finished to signal the consumer that the stream is finished.

```c
int dtssd_setup_session (char *shm, const char *dst_addr,
                         const uint16_t port, const char *uri);
```

This function is called from handle_ipc_thread and is responsible for setting up a media session between the source and consumer. It calls the function dt_init in the overlay API to acquire a overlay node identifier to the source (dst_addr and dst_port), and initiates a media_session_t structure. This identifier is stored in the dt_sid variable in the media session structure.

The dtssd_setup_session returns a handle (int) for the media session initiated, that the application can use to further control the media session. Therefore, all subsequent calls to other signaling core functions, require this handle.

```c
int dtssd_signal_stream_start (media_session_t *m);
```

When a media session has been initiated by a previous call to dtssd_setup_session, this function may be called to signal the start of a stream with the media session identified by m. This function thus sends a SIG_PLAY message to the source. This message is the same as discussed in the summary of the design chapter. The prefix SIG_ is only noted to indicate that it is a signaling message (Figure 4.14 show a complete list of all the signals).

In addition to signal the source to start streaming, this function will also set a state variable in the media_session_t that the media session is currently in a playing state. This state variable is set to _PLAYING listed in Figure 4.15.
# Signals

```c
enum {
    SIG_PLAY,
    SIG_PAUSE,
    SIG_FINISH,
    SIG_ERROR,
    SIG_GET_METADATA,
    SIG_METADATA,
    SIG_GET_DATA,
};
```

Figure 4.14: DTSS signals

# States

```c
enum {
    _PLAYING,
    _WAITING,
    _PAUSING,
    _FINISHED,
    _ERROR,
};
```

Figure 4.15: DTSS session states

When the source receives a SIG_PLAY message it starts pushing data (sender oriented) to the consumer and sets the state variable in media_session_t to _PLAYING.

```c
int
dtssd_signal_stream_pause (media_session_t *m);
```

This function is called when the application wants to pause or resume the sending of a stream (m) from a source. It sends a SIG_PAUSE message to the source. When a source receives this message, it sets the state variable in the media_session_t structure to _PAUSING (if the state is currently set to _PLAYING). dispatch_thread will discover this, and cease the sending of AV data to the consumer. If the state is currently set to _PAUSING, it is changed back to _PLAYING, and dispatch_thread will continue to send AV data to the consumer.

```c
int
dtssd_signal_stream_finished (media_session_t *m);
```
This function is called either at the source when a stream is finished (the application layer has no more data to send to a consumer), or when a consumer is no longer interested in the receiving AV data. `dtssd_signal_stream_finished` will thus notify the other end that the streaming should cease with a `SIG_FINISHED` message.

If the application layer at the source has no more data to send to the consumer, the signaling layer generates a `SIG_FINISHED` message with the last sent sequence number, and send it to the consumer. This way, the consumer (when receiving this `SIG_FINISHED` message) can know if AV data packets are lost in the network. An example is when a `SIG_FINISHED` message arrives at a consumer\(^4\). The consumer notice that the sequence number included in the `SIG_FINISHED` message is 10 numbers higher than the sequence number of the last received AV data packet. Now the consumer knows that 10 AV data packets has not yet arrived and might be missing. If the `SIG_FINISHED` message had not included this sequence number, the consumer would not be able to know if all data has arrived.

```c
int dtssd_signal_get_metadata (dt_node_t dt_sid,
                               uint32_t stream_id,
                               seq_t from_seq, seq_t to_seq);
```

`dtssd_signal_get_metadata` sends a `SIG_GET_METADATA` message for the media session \(m\) to the overlay node identified by `dt_sid`. To minimize network overhead, the signaling system only ask the other nodes in the network for a list of the sequence numbers in the range `from_seq` to `to_seq`. The value of `from_seq` and `to_seq` is determined by the user of the application which is used as arguments to `dtss_signal_get_metadata` in `libdtss`.

```c
int dtssd_signal_send_metadata (dt_node_t dt_sid,
                                uint32_t stream_id,
                                seq_t from_seq, seq_t to_seq);
```

When a node receives a `SIG_GET_METADATA` message from the network (handled in `handle_transport_thread`), it calls `dtssd_signal_send_metadata` to send a list of sequence numbers of the locally cached AV data packets back to the consumer (the overlay node referenced by `dt_sid`). The list of sequence numbers that is to be sent back to the consumer, is gathered from the Meta Data Manager, and belongs to the stream with the identification

\(^4\)Keep in mind that control messages are guaranteed to be delivered
typedef uint32_t seq_t;

stream_id (the hash of the source and URI). When this list is created it is sent to the consumer in a SIG_METADATA message.

As the size of a seq_t variable is 4 bytes (Figure 4.17), sending a list of e.g., 10 000 sequence numbers (seq_t) would take 40 000 bytes. If such list is to be sent from several nodes in the network at the same time, the overhead for network data would become high. Therefore, to decrease the network overhead cost, we send a bitmap instead. By including the from_seq and the to_seq in the SIG_METADATA message, it would be easy for the consumer to know which sequence number the sender of the SIG_METADATA are caching, by looking at the offset from from_seq. An example of a SIG_METADATA message is illustrated in Figure 4.16. This SIG_METADATA message instructs the consumer that he is caching AV data packets; 11, 13, 15, and 16. And not; 10, 12, 14 and 17, in the range 10 to 18. By using a bitmap instead of sequence numbers, a SIG_METADATA message with 10 000 sequence numbers would only use 1250 bytes (10 000/8).

To reduce the calculation cost to construct such a message from the AV Data Cache, we require the Meta Data Manager to update a local bitmap every time a AV data packet arrives. As a result, dtssd_signal_send_metadata can just copy the range of the bitmap that the consumer requested.

```c
void dtssd_md_update_index (dt_node_t dt_sid, media_session_t *m, 
    seq_t from_seq, seq_t to_seq, 
    char *metadata, uint16_t length);
```

dtssd_md_update_index is called when a consumer receives a SIG_METADATA message in response to a SIG_GET_METADATA message. The sequence numbers of the AV data packets that the remote node has cached for the media stream m, is thus located in metadata. These sequence numbers are in the range from_seq to to_seq, and the overlay node the SIG_METADATA message is
received from is referenced by `dt_sid`. As a result, `dtssd_md_update_index` updates the local index table of where AV data are located in the network (relative to the overlay node identifier), stored in the Meta Data Storage (previously illustrated in Figure 3.10).

As discussed for the previous function, sequence numbers can take a lot of space. Therefore, performance decreases by having the Meta Data Manager store the sequence numbers. Thus, as already mentioned, when a consumer receives `SIG_METADATA` messages from the network, it will store the sequence numbers as a bitmap. This also result in lower overhead when the application request an overview of the network, as the index can be returned to the application as a bitmap.

```c
int dtssd_signal_get_data (dt_node_t dt_sid, media_session_t *m
  seq_t from_seq, seq_t to_seq);
```

`dtssd_signal_get_data` does the actual fetching of AV data from the other nodes in the overlay. It issues a `SIG_GET_DATA` message to the overlay node `dt_sid` requesting AV data packets in the range `from_seq` to `to_seq`, belonging to the media session `m`.

As the consumer might already have some of the AV data packets in the range `from_seq` to `to_seq`, it becomes unnecessary to request all the AV data packets in this range. Therefore, to reduce the network cost, we include a bitmap of locally received AV data packets in the `SIG_GET_DATA` message. As a result, an exact request for AV data packets is sent. Figure 4.18 shows an example of an `SIG_GET_DATA` message with a attached bitmap. The value 1 indicates that the consumer is already caching the AV data packet with sequence number relative to the `from_seq`, while 0 indicates that it is not caching the AV data packet. From the figure, we can see that the consumer is caching AV data packets; 11, 13, 15, and 16, while 10, 12, 14, and 17 is not cached. The node that receives the `SIG_GET_DATA` message then knows that the consumer only requests AV data packets with sequence number; 10, 12, 14, and 17.

```c
uint32_t
hash (struct in_addr source, uint16_t source_port, char *uri);
```
This function generates a unique unsigned 32 bit hash based on the source’s address (source and source_port), and the URI of the media session stream (uri). The returned value is stored in the stream_id variable in the media session structure (Figure 4.4).

### 4.4.1 Meta Data Manager

Since the Meta Data Storage is implemented as a very simple data structure, the Meta Data Manager is not implemented as a component. This is because each function can easily manage the Meta Data Storage itself. The Meta Data Storage is constructed of a linked list of meta data structures (meta_data_t) that contains pointers to where data is cached in the overlay layer, in addition to a bitmap that keeps an overview of where AV data packets is located. This linked list and bitmap is a part of the media session structure (described in Section 4.1.3). The meta data structure and the meta data storage part of the media session structure is listed in Figure 4.19.

One instance of the meta data structure is a record of an AV data packet cached in the overlay, and the purpose of the different variables in the structure are the following:

- **seq_t seq** - The sequence number of the cached AV data packet
- **uint16_t payload_length** - The length of the AV data packet
- **void *cache** - A pointer to where the AV data packet is cached in the overlay layer
- **meta_data_t *next** - A pointer to the next meta data structure (linked list)

Whenever a AV data packet is sent or received with `dt_send` or `dt_receive`, a pointer to where the AV data packet has been cached in the overlay layer is returned. This pointer is thus stored in a newly created `meta_data_t` structure which is added to the linked list in the media session structure (relative to the stream id the AV data packet belongs to). In addition, the bitmap is updated to keep an overview of where AV data packets is located. All sent and received AV data packets is marked in `bitmap[0]`, to illustrate that the AV data packet is cached locally. The other indices of the `bitmap[]` array is updated when a meta data gathering has occurred. When meta data has arrived from other nodes in the network, `bitmap[x]` is updated, where x is the overlay node identifier to the overlay node the meta data was received from. When the application layer invokes the previously
typedef struct meta_data meta_data_t;

struct meta_data
{
    seq_t seq;
    uint16_t payload_length;
    void *cache;

    meta_data_t *next;
};

typedef struct media_session
{
    ...

    /* Meta Data Storage Linked List */
    meta_data_t *first;
    meta_data_t *current;
    meta_data_t *last;
    int bitmap[NOE_MAX][BITMAP_SIZE];

    ...
}
media_session_t;

Figure 4.19: Meta Data Storage
explained `dtss_get_metadata`, `dtss_get_metadata` extracts the requested portions of this bitmap and returns it to the application layer.

An example of the Meta Data Storage is illustrated in Figure 4.20. As we can see, each media session has a linked list to meta data structures of locally cached AV data packets. The `bitmap[0]` in the figure also shows that AV data packet 0, 2 and 5 are locally cached. The other values of the indices in the bitmap are currently set to 0, as meta data requests has not been issued to the other overlay nodes.
Chapter 5
Evaluation

In this chapter we evaluate the implementation of our signaling system for handling media streams over DTNs. We begin by expressing the goals of our evaluation (Section 5.1), and then describe the details of the experiments (Section 5.2). Finally, we compare the results of the experiments to the requirements set in the design chapter (Section 5.3).

5.1 Goals

Our objective in this evaluation is to verify that our signaling system fulfills the requirements set in the design for 1) traditional media streaming through wired networks and 2) for our targeted network: sparse MANETs with frequent path disruptions. For this we need to evaluate the Stream Control Manager and the Meta Data Manager as well as the interaction between the signaling layer and the application layer. Therefore, we verify the following to support the functional requirements:

- **Stream Control Manager**
  - **Sender Oriented**
    * Start the streaming of a file or a live stream from a source
    * Pause the streaming of a file or live stream from a source
    * Finish the streaming of a file or a live stream from a source
  - **Receiver Oriented**
    * Fetch only a part of a file or a live feed from:
      - a source
      - intermediate caching nodes
Table 5.1: Experiments machine details

<table>
<thead>
<tr>
<th></th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel (R) Pentium(R) 4 CPU 2.80GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>1 GiB [19]</td>
</tr>
<tr>
<td>OS</td>
<td>GNU/Linux version 2.6.30-1-686 (Debian 2.6.30-6)</td>
</tr>
</tbody>
</table>

5.2 Experiments

In this section we show through experiments how our system fulfills the requirements set in the design. We start by explaining the experiments setup and essential components. The source code for these components are found in Appendix A.

5.2.1 Setup

Since the overlay transport and the media player in the DT-Stream project are not yet finished, we are unable to fully test our system in a real environment. Instead, we have chosen to emulate both a media player and a overlay transport to test the requirements set in Section 3.1. Both these components will be explained in detail in the following sections.

All experiments are performed on one single machine. Therefore, every instance of the signaling system (dtssd) running, represents one virtual node and uses the localhost to handle network transmissions. Details of the machine the experiments are performed on is listed in Table 5.1.

Emulated Media Player

The purpose of the emulated media player is to communicate with the signaling system (through IPC) to verify that our signaling system is working. Due to limitations in time, the emulated media player will not present video
and audio to the user. Thus it only has a text user interface, and the presentation of the signaling process (such as the progress bar and data chunk index) is limited.

The emulated media player is started by issuing the following command from a unix shell:

```
$ ./mp_dummy <URI> <source address> <source port>
```

It takes the URI of the stream to be played back as the first argument, and the IP address and port where the sources signaling daemon (dtssd) is running as second and third argument. When the emulated media player is started, it connects to the dtssd (if it is running) through IPC. The emulated media player gets a session id from dtssd, which is bound to the URI and the source arguments.

The emulated media player is implemented with a simple user interface to provide commands regarding the started session to the user. These are:

- `s` - Start the streaming process
- `p` - Pause/resume the streaming process
- `q` - Quit and finish the streaming process
- `g` - Get AV data packets from the overlay. The user is prompted to input a range of sequence numbers of the interested data in addition to which node it want the data retrieved from.
- `m` - Show progress bars of where AV data packets are located in the overlay. The user is prompted to input a range of sequence numbers of the interested data in addition to which node it want to show metadata from.

After the streaming process has started, the emulated media player continuously polls the dtssd for both the highest received sequence number and the latest received sequence number to populate a simple progress bar. Arrived data is indicated by ‘=’, while missing data is indicated by ‘.’ in the progress bar. When the emulated media player receives the sequence numbers of the received packets, it can request the data that it want to present to the viewer. However, since this thesis is about signaling, the decoding and the presentation of the data is not implemented in this emulated media player.
Emulated Overlay Layer (libdtt)

As there is currently no overlay transport implementation that supports our functional requirements, we have chosen to implement a simple emulated overlay transport. This emulated overlay layer simply uses TCP between each two nodes in the overlay and is therefore capable of guaranteeing that signaling messages will be delivered to the receiver. However, to achieve the same functionality over a real MANET, we do not expect that TCP is used, as TCP does not fit this task. Therefore, in a real MANET we assume a custom protocol for control packets with delivery guarantees as discussed in Section 3.1.2.

The emulated overlay layer also provides non-reliable transport for media data, and an AV data cache for sent and received AV data packets. The emulated overlay layer follows the specification described in Section 4.2

5.2.2 Workload

To conduct the experiments, we require a workload to test the system with. As the purpose of our signaling system is to send and receive AV data and meta data to and from the different nodes in the network, these two data types will be our workload.

AV data

The AV data is gathered from the application layer at the source. When this AV data is gathered, an AV data packet is created containing both AV data and meta data. For our experiments, we have chosen to create AV data packets of 4108 bytes in size (4096 bytes AV data and 12 bytes meta data) to be sent to the overlay layer for network transmission.

Since we are mostly interested in how the AV data packets are distributed in the network, the content of the AV data packets is not interesting.

To be able to run the test, the quantity of the data must be of some magnitude. As a result, the consumer will request the file test.ts from the source. This file is about 10 MB in size, and the application layer at the source will fragment this file into 2500 packets that should be sent to the consumer. The sequence number in each packet is thus a number between 0 and 2499.

Meta Data

For the consumer to be able to construct an overview of where AV data is located in the network, exchange of meta data must be done. The meta data
Evaluation

exchanged will be located in the payload of the signaling message METADATA, and the size of the payload will be dynamic depending on the requested metadata.

5.2.3 Parameters

The parameters needed to verify our signaling system are listed in the following:

- Caching Nodes
- Packet Loss

Caching Nodes

Since one of our most important goals is to show how a user can have as much as possible control of a streaming session, we need to define how the number of nodes in the network affects our signaling system. Since our signaling system is based on an optimistic caching model to retrieve missing data, the more nodes in the network, the more nodes are caching. As this thesis is not concerned about finding optimal ways of choosing caching nodes, every node in the network is set to perform caching (source and consumer included). However, since we are mostly concerned with verifying that our system actually provides a more fine grained control of a streaming session for the user, determining how the system scales to the number of nodes in the network, is out of the scope for this thesis. Since the DT-Stream project currently lacks a suitable implementation of an overlay layer, we leave this evaluation to future work. As a result, our experiments will only be conducted by a limited set of nodes, to verify that a consumer can:

- Receive data from a source (2 nodes; source and consumer)
- Receive data from one intermediate caching node (3 nodes; source, consumer and intermediate caching node)
- Get an index over where AV data is located in the network (3 nodes; source, consumer and intermediate caching node)

Packet Loss

A consumer will not fetch data it already has in local storage, if instructed by the user. Therefore, to verify that the receiver oriented fetching works, the consumer must experience missing packets. As a result, we edit the source
code for the emulated overlay layer to provide fixed packet loss, to evoke worst-case scenarios. This packet loss will thus emulate the behaviour of a sparse MANET with frequent packet loss.

5.2.4 Scenarios

To verify our signaling system, we have set up some scenarios with different parameters for the experiments. All these scenarios have in common that each consumer are streaming a file fragmented into 2500 AV data packets ordered by sequence numbers, from the source.

Scenario 1: No Packet Loss

The first scenario is set to verify our stream control management and meta data management for streaming in traditional wired networks. The network is constructed of 3 nodes as illustrated in Figure 5.1, and we verify the following functionality:

- **Start** a streaming session from a source (sender oriented)
- **Pause** the streaming session back to the source
- **Finish** the streaming session
- **Gather** meta data from the intermediate overlay node and the source

Scenario 2: Fixed Packet Loss

In the second scenario we verify that a consumer can retrieve missing data from both an intermediate caching node and the source itself. The topology of the scenario is the same as for Scenario 1, but this time we emulate packet loss. This packet loss is emulated to occur at fixed positions between the source and the intermediate caching node, and between the consumer and the intermediate caching node. Table 5.2 shows the range of AV data packets that is dropped to generate packet loss. Each row shows where the packet loss takes place. The first row in the figure shows how the source node (S)
Table 5.2: Scenario 2: Fixed packet loss during transmission

<table>
<thead>
<tr>
<th>Between Node</th>
<th>Missing AV data packets (seq.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow IC1$</td>
<td>500 - 1000</td>
</tr>
<tr>
<td>$IC1 \rightarrow C$</td>
<td>1500 - 2000</td>
</tr>
</tbody>
</table>

emulates packet loss and packets with sequence numbers 500 to 1000 are not sent to intermediate caching node ($IC1$). In the second row, $IC1$ emulates packet loss and packets with sequence numbers 1500 to 2000 are not sent to $C$.

In this scenario we thus verify that the following is working:

- Sender oriented data streaming
- Receiver oriented data fetching, from;
  - the source ($S$)
  - the intermediate caching node ($IC1$)
- Gather meta data from the intermediate overlay nodes and the source

5.3 Results

In this section we verify that our system behaves as expected. Since the system relies on the users experience and the users actions, we have chosen to show the results using screen capture of the streaming sessions.

5.3.1 Scenario 1: No Packet Loss

Table 5.3 shows the results of Scenario 1. Figure 5.2 shows how the emulated media player is able to display an index of the distributed AV data packets in the overlay.

The first line shows status information about the file being streamed, while the first progress bar shows which data that has arrived locally. In this scenario, we see that the complete file has arrived with no packet loss, and the state variable in the status line is set to \texttt{FINISHED}. The other progress bars illustrates where AV data packets is located in the overlay (and locally) for a user prompted range of sequence numbers. This range is 0 to 2528.
5.3.2 Scenario 2: Fixed Packet Loss

Figure 5.3 illustrates the progress bar of the streaming session when all data has been pushed (sender oriented) from the source to the consumer. We can see that the fixed packet loss has occurred, and the consumer is missing gaps of data.

Figure 5.4 illustrates how the user has issued meta data requests to the nodes in the overlay and got an overview of the distributed AV data packets based on their sequence numbers. In this listing, the user has requested the meta data for all AV data packets in the overlay with the sequence numbers in the range 480 to 1024. As we can see, Overlay Node 1 (S) is the only node...
Figure 5.4: Scenario 2: Show how AV data is distributed in the overlay and locally

that has replied that it has these AV data packets. Based on this information, the user can issue a data request to this node to get data of interest.

In this scenario, we assume that the user is only interested in AV data packets with sequence numbers ranging from 800 to 1000. This is to illustrate that the data gathering is operating individually and is not relying on the previous meta data gathering that gathered meta data from the range 480 to 1024. Figure 5.5 thus illustrates the resulting AV data distribution in the overlay, after the user has requested this data. In this listing, we show that the consumer has received this data from Overlay Node 1. However, since data from Overlay Node 1 is routed through Overlay Node 2 (IC1), Overlay Node 2 caches a copy of these packets (also shown in the listing).

In addition to verify that data can be retrieved from the source, we verify that data can be retrieved from intermediate nodes. Figure 5.6 illustrates a data chunk index of AV data packets with sequence numbers 1472 to 2016 and their location in the overlay. As we can see, both Overlay Node 1 and 2 has this data. However, since we should verify that data can be retrieved from an intermediate node, we show how the user requests data from Overlay Node 2. In this scenario, the user is interested in all AV data packets with sequence number 1472 to 2016, and issues data requests to Overlay Node 2. The resulting Data Chunk Index is illustrated in Figure 5.7.

Table 5.4 and 5.5 shows a summary of the verified results.
Figure 5.5: Scenario 2: The result of gathering AV data with sequence numbers 800 to 1000 from Overlay Node 1

Figure 5.6: Scenario 2: Show where AV data packets with sequence numbers in the range 1472 to 2016 is located in the overlay and locally
Figure 5.7: Scenario 2: The consumer has received all the AV data it was interested in

Table 5.4: Scenario 2: Fixed packet loss

<table>
<thead>
<tr>
<th>Sender Oriented Streaming</th>
<th>Receiver Oriented Fetching</th>
<th>Meta Data Gathering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.5: Scenario 2: Fixed packet Loss

<table>
<thead>
<tr>
<th>Receiver Oriented Fetching</th>
</tr>
</thead>
<tbody>
<tr>
<td>From S</td>
</tr>
<tr>
<td>From IC1</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusion and Future Work

6.1 Conclusion

The primary goal of this thesis, is to construct a signaling system for handling optimized streaming of media data over DTNs with emphasize on mechanisms that enables visualization and presentation. Thus, two of the important subjects of our thesis are the design of:

- a delay tolerant media player to determine the requirements for
- a signaling system

The delay tolerant media player’s purpose is to provide a more fine-grained control of the signaling system in addition to give the user of the media player more information about the streaming process. The signaling systems purpose is however to provide both a sender and receiver oriented data transmission model, to let the user (through the delay tolerant media player) customize the signaling process to its needs.

Since this thesis’ primary domain is sparse MANETs, data can be widely distributed. Therefore, with our solution, a user is able to instruct the signaling system to gather data using receiver oriented data transfer, for data that is cached in intermediate overlay nodes. This means that whenever a node is present in the overlay and performs caching, a user can fetch the cached data from it. Another important feature that our system provides, is the ability to let the user gather an overview of where data is located in the overlay.

In this thesis we have designed a delay tolerant media player to provide requirements for a signaling system that increases customizability of media

79
streaming sessions in DTNs. The signaling system has been design based on the requirements from this delay tolerant media player. In addition, we have implemented both an emulated overlay layer and an emulated media player to evaluate the signaling system. In the evaluation of the signaling system, we have verified that the signaling system fulfils the requirements set by the designed delay tolerant media player, using the emulated overlay layer and media player. The signaling system is able to support both sender and receiver oriented data transmissions, and is able to provide an overview of where AV data is distributed in the network with the use of meta data gathering. We have also implemented an API for the signaling system to provide fine-grained control of the streaming sessions.

### 6.2 Future Work

Due to time limitations, some improvements can yet be made to our system. Thus, we list in this section some of the improvements and future research that can be made to make signaling in DTNs even better.

#### 6.2.1 Signaling Tuning

One primary purpose of our signaling system is to provide fine-grained control of receiver oriented data and meta data gathering. For this reason, the signaling core have functionality to retrieve data and meta data from specific nodes in the overlay. However, the case when a user wants to gather some specific data from the overlay, and does not care where the data is retrieved from, remains future work. Therefore, we propose to research the design of algorithms for optimal gathering of data already distributed in the network. We are especially interested in algorithms similar to the following:

- Investigate where data is distributed using a minimal amount of control messages

- If several nodes has the same data a user requests
  - Use algorithms such as Dijkstra’s shortest path algorithm, and/or
  - Fetch different parts of the requested data from different nodes simultaneously

Such approaches share commonalities with P2P systems, which might be interesting to study and apply to our signaling system.


6.2.2 Content Type Priority

In our signaling system, we have not considered the content of the data arriving from the application layer at the source. As different types of content might need different priority, we propose a future study of adding additional meta data to the AV data packets to provide the consumer with more information regarding the content, for further optimization of data gathering. The MOMENTUM project has already considered this by e.g., adding additional priority to the different types of frames from a stream, which could be integrated in our signaling system.

6.2.3 Overlay Layer Integration

When the implementation of an overlay layer developed for the DT-Stream project is finished and supports our requirements, we propose an integration between the overlay layer and our signaling system. This integration will provide the opportunity for further evaluation of our signaling system, as discussed in the next section.

6.2.4 Evaluation

Since our thesis is bound by the lack of a fully implemented overlay layer, we were unable to conduct some experiments to give a complete evaluation. Therefore, when the implementation of an overlay layer is is working properly and is able to cooperate with our signaling system, additional experiments must be made. Experiments that this thesis lack and is needed to improve the evaluation, is listed in the following:

- Determine how the system scales relative to the number of nodes in a overlay
- How does future developed algorithms for improved data fetching compares to each other?
- Hold a survey to figure out how users experience the signaling systems optimizations and customizability
Bibliography


Appendix A

Source Code

Attached is a CD-ROM containing source code for DTSS and the emulated overlay layer and media player. The source code can also be found at the following URL:

http://larsod.at.ifi.uio.no/dtss

A.1 Signaling Core (dtssd)

The signaling core is constructed of the following files:

- dtss-0.0.1/src/dtssd.c - Stream Control Manager, Meta Data Manager and Meta Data Storage
- dtss-0.0.1/src/dtssd.h - Header for dtssd.c
- dtss-0.0.1/src/common.c - Helping functions
- dtss-0.0.1/src/common.h - Header for common.h
- dtss-0.0.1/src/encoder.c - Emulated application layer at the source node to provide AV data to dtssd.c
- dtss-0.0.1/src/encoder.h - Header for encoder.h

A.2 Applications API (libdtss)

libdtss is constructed of the following files:

- dtss-0.0.1/src/libdtss.c
- dtss-0.0.1/src/libdtss.h
A.3  Emulated Overlay Layer (libdtt)

The emulated overlay layer is constructed of the following files:

- dtss-0.0.1/src/libdtt.c
- dtss-0.0.1/src/libdtt.h

A.4  Emulated Media Player

The emulated media player is constructed of the following files:

- dtss-0.0.1/src/mp_dummy.c
- dtss-0.0.1/src/mp_dummy.h