University of Twente
Enschede - The Netherlands

Connecting distributed E-health applications by means of a generic control protocol

Thesis for a Master of Science degree in Telematics
from the University of Twente, Enschede, the Netherlands
Enschede, June 1, 2009

Wim van Ravesteijn

Report nr: BSS 09-13

GRADUATION COMMITTEE:
Dr. ir. B.J.F. van Beijnum (University of Twente)
Dr. ir. I.A. Widya (University of Twente)
Prof. dr. ir. H.J. Hermens (University of Twente)

Connecting distributed E-health applications by means of a generic control protocol

Thesis for a Master of Science degree in Telematics
from the University of Twente, Enschede, the Netherlands
Enschede, June 1, 2009

Wim van Ravesteijn

UNIVERSITY OF TWENTE,
Faculty of Electrical Engineering, Mathemathics and Computer Science,
Division of Biomedical Signals and Systems

Abstract

In this thesis a new protocol is designed to support the negotiating of a medical standard. There exist several vendors of medical equipment that use their own protocols and message formats to exchange and store medical data. This creates problems in inter-operability.

Using SIP we describe a control protocol that can negotiate the use of a medical standard. We created a new data format based on SDP to describe the medical standards (called 'MSDP') and medical data (called 'medical'). The last one is used to exchange single-valued measurements, to prevent a huge overhead of first negotiating a medical standard, and only afterwards sending this single value.

The thesis describes the data format of the newly designed protocol, explanations on the use and interfaces for the different components. To prove the concept, a prototype is implemented and evaluated.

Acknowledgments

I thank my primary supervisor at the University of Twente, Bert-Jan van Beijnum, for all the discussions, ideas and feedback he gave me during our meetings. The comments on improvements were of great value, and the discussions made me reflect my choices and rephrase certain parts of my thesis.

I also thank Ing Widya for his presence at several of the meetings, for his input during the discussions and the feedback on the content of the thesis.

I would like to thank my family for their support and trust in me during my studies. Also I would like to thank all my friends, both in Enschede as well as around Europe for giving me an enjoyable student life and many great experiences.

Wim van Ravesteijn

Enschede, the Netherlands
1 June 2009

Contents

List of Figures

List of Tables

List of Acronyms[1]

ANSI ..... American National Standards Institute A private non-profit organisation that oversees the development of voluntary consensus standards for products, services, processes, systems and personnel in the United States.

API ..... Application Programming Interface A set of routines, data structures, object classes and/or protocols provided by libraries and/or operating system services in order to support the building of applications.

ASCII ..... American Standard Code for Information Interchange A character encoding based on the English alphabet.

ASN.1 ..... Abstract Syntax Notation One A standard and flexible notation that describes data structures for representing, encoding, transmitting and decoding data.

CRLF ..... Carriage return - new line The character sequence carriage return followed by new line (ASCII 0x0d0a), used to mark the end of a line.

DNS ..... Domain Name System A hierarchical naming system for computers, services or any resource participating in the internet.

DSM-CC ..... Digital Storage Media - Command and Control Part 6 of MPEG-2, an ISO/IEC standard developed to provide the control functions and operations specific to managing MPEG-1 and MPEG-2 bit streams.

ECG ..... Electrocardiogram A recording of the electrical activity of the heart over time produced by an electrocardiograph, usually in a non-invasive recording via skin electrodes.

FTP ..... File Transfer Protocol A network protocol used to transfer data from one computer to another through a network such as the Internet.

GSM ..... Global System for Mobile communications The most popular standard for mobile phones in the world.

HTTP ..... Hyper Text Transfer Protocol A communications protocol used for retrieving documents from the Internet.

[1]Most definitions are taken from wikipedia.org

IANA ..... Internet Assigned Numbers Authority The entity that oversees global IP address allocation, root zone management for the DNS, media types and other Internet protocol assignments.

IETF ..... Internet Engineering Task Force Developer and promoter of Internet standards, in particular with standards of the TCP/IP and Internet protocol suite.

IO ..... Input/Output The communication between an information processing system (such as a computer), and the outside world - possibly a human, or another information processing system.

IP ..... Internet Protocol A protocol used for communicating data across a packet-switched internetwork.

IPv4 ..... Internet Protocol version 4 The fourth revision in the development of the Internet Protocol (IP) and the first version of the protocol to be widely deployed.

IPv6 ..... Internet Protocol version 6 The next-generation Internet Layer protocol for packet-switched internetworks and the Internet. Currently co-existing next to IPv4, but supposed to replace it totally in the future.

ISO ..... International Organization for Standardization An international-standard-setting body composed of representatives from various national standards organizations.

ITU ..... International Telecommunication Union International organisation to standardise and regulate international radio and telecommunications.

ITU-T ..... Telecommunication Standardisation Sector Coordinator of standards for telecommunications on behalf of the ITU.

Java ..... Java A programming language which can run on any Java virtual machine after being compiled to byte code, regardless of computer architecture.

MDCP ..... Medical Data Control Protocol A prototype developed in this thesis to prove the correctness of the proposed model.

MIME ..... Multi-purpose Internet Mail Extensions An internet standard that extends the format of e-mail to support text in character sets other than US-ASCII, non-text attachments, message bodies with multiple parts and header information in non-ASCII character sets.

MPEG ..... Moving Picture Experts Group A working group of ISO/IEC charged with the development of video and audio encoding standards.

MSDP ..... Medical Session Description Protocol A format for describing medical data stream initialisation parameters.

MSDP ..... Medical Session Description Protocol A format for describing medical data stream initialisation parameters.

MTU ..... Maximum Transmission Unit The size of the largest packet that a network protocol can transmit.

NAPTR ..... Name Authority Pointer DNS record type used to provide a general, flexible, extensible and standard mechanism to provide information about services provided for a certain domain.

NTP ..... Network Time Protocol A protocol for distributing time signals over computer networks to synchronise clocks.

OSI ..... Open Systems Interconnection An effort to standardise networking that was started by ISO along with the ITU-T.

PDA ..... Personal Digital Assistant A hand-held computer.

PER ..... Packed Encoding Rules A set of ASN.1 encoding rules for formatting data in binary.

QoS ..... Quality of Service The ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow.

RFC ..... Request for Comments A memorandum published by the IETF describing methods, behaviours, research, or innovations applicable to the working of the Internet and Internet-connected systems.

RSVP ..... Resource ReSerVation Protocol A transport layer protocol designed to reserve resources across a network to request specific levels of QoS for application data streams.

RTCP ..... Real-time Transport Control Protocol Out-of-band control information for an RTP flow, periodically transmitting control packets to participants in a streaming multimedia session.

RTP ..... Real-time Transport Protocol A standardised packet format for delivering audio and video over the Internet.

RTSP ..... Real Time Streaming Protocol A protocol for use in streaming media systems which allows a client to remotely control a streaming media server, issuing VCR-like commands such as "play" and "pause".

SASL ..... Simple Authentication and Security Layer A framework for authentication and data security in Internet protocols.

SDP ..... Session Description Protocol A format for describing streaming media initialisation parameters in an ASCII string.

SIP ..... Session Initiation Protocol A signalling protocol, widely used for setting up and tearing down multimedia communication sessions such as voice and video calls over the Internet.

SIPS ..... Session Initiation Protocol Secure The secure version of SIP.

SMTP ..... Simple Mail Transfer Protocol An Internet standard for electronic mail transmission across Internet.

SRM ..... Session and Resource Manager A network component in the DSM-CC specification managing sessions and resources.

SRV ..... Service Record DNS record type used to specify information on available services.

TCP ..... Transmission Control Protocol A core protocol of the Internet Protocol Suite, providing reliable, ordered delivery of a stream of bytes between two end systems.

TCP/IP ..... Transmission Control Protocol over Internet Protocol The set of communications protocols used for the Internet and other similar networks.

TLS ..... Transport Layer Security A cryptographic protocol that provides security and data integrity for communications over TCP/IP networks.

UA ..... User Agent An application running on an end system.

UAC ..... User Agent Client A User Agent that is behaving like a client in a client-server communication.

UAS ..... User Agent Server A User Agent that is behaving like a server in a client-server communication.

UDP ..... User Datagram Protocol A core protocol of the Internet Protocol Suite, providing unreliable transport of datagrams between two end systems. Datagrams may arrive out of order, appear duplicated, or go missing without notice.

UMTS ..... Universal Mobile Telecommunications System One of the third-generation (3G) cell phone technologies.

URI ..... Uniform Resource Identifier A compact string of characters used to identify or name a resource on the Internet.

XML ..... Extensible Mark-up Language A general-purpose specification for creating custom mark-up languages.

XMPP ..... Extensible Messaging and Presence Protocol An open, XML-inspired protocol for message oriented middleware.

Chapter 1

Introduction

1.1 Motivation

E-health applications are upcoming business, due to the fact that more and more people get older, and costs for healthcare increase accordingly. There are not enough people willing to work in healthcare and the costs get too high to continue the existing system. For this reason, one should investigate more in research on technical measurements to make the older people longer self-supporting.

In the emerging E-health market, there is a lot of organisations and vendors which developed their own standards for transmitting and storing the personal and vital sign data. As the patients will be at their own homes, or at least not in the healthcare centre, the system works in a distributed way. This results in application components being connected via a network, where different parties are involved, like the application components of the patient, the network provider and the application components of the healthcare centre. This makes merging the different systems complicated when they are developed by different vendors.

1.2 Objective

Design a generic control protocol that can connect distributed application components of different vendors of E-health applications. It should be able to negotiate different data formats and configure a communication channel based on common capabilities and such that optimal communication is achieved, and which accomodates communication between distributed applications with possible different health data formats.

1.3 Approach

First we'll investigate which standards are mainly in use nowadays, both for transmitting and storing vital sign data. We'll focus on ECG and blood pressure data, as these two cover both continues time signals and discrete time measurements. Next to that, we'll investigate about control protocols that can be used for negotiating the transfer of vital sign data. The negotiating will be related to the transfer protocol used and the format of the vital sign data.

Based on the good points of the existing standards, we'll try to propose a control protocol that is able to negotiate about different transfer protocols and data formats. This could lead

to inserting converters on sender and/or receiver side to come to a common data format used in the transfer that both sides understand. The protocol should be able to negotiate the transfer of vital sign data between devices and applications of different vendors. A prototype will be part of the final results.

1.4 Expected results

The result of the work should be a control protocol that is able to negotiate on a common transfer protocol and data format between different applications and devices. To support this, one or both entities will have to use converters to convert different formats to a common one (writing converters is not part of the assignment). The negotiation is based on least costs and requirements on the quality of the receiving entity. A prototype should prove the feasibility of the proposed solution.

1.5 Document structure

The remainder of this thesis is structured as follows. Chapter 2 gives some background information about existing control protocols and medical standards. Chapter 3 presents a general model we will use in the remainder of this thesis, as well as a list of requirements for the control protocol we will design. Chapter 4 presents the design. It starts with finding the most suitable existing control protocol, followed by making adjustments to meet all requirements. Chapter 5 presents the implementation of a prototype, starting with a refinement of the design followed by an overview of the implementation. Chapter 6 presents the evaluation and validation of the prototype. Chapter 7 presents the conclusions and further work.

Chapter 2

Background

This chapter presents techniques that are used further on in this thesis. It does not present new techniques, so anybody already familiar with these techniques can skip to the next chapter.

2.1 Control protocols

There exist already different control protocols. In this section we provide an overview of the most used ones.

2.1.1 SIP

The Session Initiation Protocol (SIP) is an application-layer control protocol for creating, modifying and terminating sessions with one or more participants. It is described in [23]. The protocol itself does not contain the media data itself, but the parameters needed to set up the connection. SIP can use different transport protocols, but must at least implement UDP and TCP. Proxy servers are used as intermediate hosts to route SIP packets. A registration service is used by users to upload their current location, which is used by proxy servers to route packets to the right location and offers users mobility in the network while still being reachable.

The syntax used by SIP is much identical to HTTP/1.1, as defined in [12]. SIP defines several methods: *REGISTER*, *OPTIONS*, *INVITE*, *ACK*, *CANCEL* and *BYE*. The response includes a status code. Also this status code is comparable to the ones defined in HTTP/1.1 (see [12]), with several additions. They are categorised in informational, successful, redirection, client error, server error and global failure responses. Message bodies are used to include information that cannot be included in the headers, like media descriptions, for which the Session Description Protocol (SDP, see [17]) is used.

As SIP is similar to HTTP, also extensions of HTTP can be used in SIP. For authentication, digest access authentication ([13]) can be used. This is a simple challenge-response mechanism. It can only be used for authentication, and not for confidentiality or integrity.

Locating SIP servers ([22]) can be done by using DNS. *NAPTR* and *SRV* records are used to inform clients and proxies where a certain contact can be found. This includes mechanisms for prioritising certain servers and load balancing.

In [20] an extension to SIP that specifies event notifications is described. To receive notifications, one first has to subscribe to these messages. Different categories of notifications can be used, and one can subscribe to one or more chosen categories.

More information about SIP and its extensions can be found in appendix A.

2.1.2 RTSP

[25] describes the Real Time Streaming Protocol (RTSP). It is an application level protocol, which controls the delivery of real-time data. It can be used for on-demand and live streaming of data, for example audio and/or video. Several data streams can be controlled at once that need to be time-synchronised. Normally the data streams are not delivered over RTSP, although this is possible with interleaving. It runs on top of transport protocols like UDP, multicast UDP and TCP and is similar in syntax and operation to HTTP/1.1, as defined in [12].

Sessions play an important role in RTSP. The *session-id* is generated by the server, and included in every following request by the client. RTSP defines several methods: *DESCRIBE*, *ANNOUNCE*, *GET_PARAMETER*, *OPTIONS*, *PAUSE*, *PLAY*, *RECORD*, *REDIRECT*, *SETUP*, *SET_PARAMETER* and *TEARDOWN*. The response includes a status code. Also this status code is comparable to the ones defined in HTTP/1.1 (see [12]), with several additions. They are categorised in informational, successful, redirection, client error and server error responses. Message bodies are used to include information that cannot be included in the headers.

More information about RTSP can be found in appendix B.

2.1.3 XMPP

The eXtensible Message and Presence Protocol (XMPP) is described in [24] and is an open protocol for message oriented middleware. The protocol uses a XML stream to exchange messages between two hosts, which can be either client-server, or server-server. Security is provided by TLS and SASL. The protocol is adapted from the Jabber[1] protocol. A client only connects to one server, the servers in the network take care of routing and forwarding messages to other clients.

All implementations must support *message*, *presence* and *info/query* messages. These stanzas provide the core functionality of XMPP. *Message* is used for exchanging messages between clients, *presence* is used to inform subscribed clients about the presence information of a certain user and *info/query* is used to retrieve and set values of other clients and servers.

Messages are exchanged near real-time, but provisions can be made in the server to store messages when they cannot be delivered due to an off-line client. The initial aim of the protocol was to support instant messaging, but other uses are possible as well within the specifications of the protocol. Unless many other instant messaging protocols, this is an open standard, and everybody can run his or her own server.

[1] http://www.jabber.org/

2.1.4 DSM-CC

[6] describes the use of Digital Storage Media - Command and Control (DSM-CC). This is part 6 of the ISO/IEC MPEG-2 standard, developed to provide the control functions and operations specific to managing MPEG-1 and MPEG-2 bit streams. DSM-CC does not require a certain transport layer, but can run on different ones, as long as it detects and discards corrupted messages and delivers entire messages (in case segmentation is performed, messages must be re-assembled before delivery). The network on which DSM-CC is used may be non-homogeneous. The DSM-CC specification consists of various protocol areas, which can be used standalone, or combined with other areas.

Sessions play an important role in DSM-CC and have a network-wide unique session ID. It is used to group together the resources needed for the instance of a service. Sessions are set up between a client and server when the client wants to access a service and taken down at the end of the service. Session messages are send between user and network (U-N messages), via the Session and Resource Manager (SRM). The data is send over a direct connection between client and server (U-U connections). Multiple of such connections may exist. Resources are dynamic during a session, and can be altered with the *AddResources* and *DeleteResources* messages send by the server. Within one session, connections with different servers can exist.

Sessions are always set up by the client, but a client can be notified to set up a connection via the *PassThruReceipt* message. Not answering such a message means either the user is not present, or does not wish to set up the session. Usage of the *PassThruReceipt* is for example for notifying the client of an incoming call.

DSM-CC specifies a lightweight and fast protocol for uploading data or software from a server to a client. The client can inform the server via generic user compatibility descriptors about its capabilities, so the server can send the correct image. Uploads can be done via inter-active flow-controlled upload, as well as via a broadcast upload. Every control message contains enough information to enable a client to receive the data out of sequence, for example to tune to an upload channel if the upload is already under way. The ability to upload software can simplify the maintenance procedures for clients.

2.1.5 H.323

In [16] a description of H.323 can be found. H.323 is an umbrella standard, putting several existing standards together to offer interpersonal communications between end systems that are attached via some kind of network. The networks on which H.323 operates are the ones with a non-guaranteed QoS. As audio codecs, for example G.711, G.723.1, G.728 and G.729 can be used, while as video codec H.261 or H.263 can be used. Before a call is set up, an agreed coding standard is negotiated.

The output streams are transferred over the network using the real-time transport protocol (RTP). As part of the real-time transport control protocol (RTCP), information is send to the receiving system to synchronise the audio and video streams, the packet rate and packet transmission delay. The receiver sends via the RTCP information back about percentage of packets lost/corrupted and the inter-arrival jitter. The information in the RTCP packets is used to optimise the data transfer.

H.323 provides a *Gatekeeper*. The *Gatekeeper* should give permission to every end system that wants to get involved in a multimedia session, or needs more resources in an existing session. In this way, the *Gatekeeper* can make sure that not too many end systems get involved in a multimedia session, which would go beyond the capabilities of the network.

The *H.323 Gateway* is used to interconnect different types of networks. The *Gateway* provides translations between the different messages on each network type, both control and media messages. It also provides address translation in case different addressing schemes are used on the different networks.

2.2 Medical data standards

There exists different medical data standards. In this section we present the most important ones.

2.2.1 DICOM

In [18] an introduction to DICOM is given. DICOM prescribes an uniform, well-understood set of rules for the interchange of digital images. DICOM can support many different types of images, transfer syntaxes and service roles. Communication stacks used are point-to-point, ISO and TCP/IP.

More details of DICOM are described in [29]. Service classes define actions that can be executed, like store, find, get and move, which can be applied to information objects. Information objects carry information, for example images, reports or patients. They are defined by an Information Object Definition (IOD), which is a list of mandatory, optional and conditional attributes to be stored in that object. Service classes and IODs are combined in the service-object pair classes (SOP classes).

2.2.2 ecgML

ecgML is a mark-up language using XML for supporting ECG data exchange and analysis. It is created because SCP-ECG, DICOM and HL7 are considered to focus on specific applications and domains ([26]; [28]). A major advantage of using XML is that the information is not merged and intertwined with its representation format, making it more flexible to process, and avoids redundancy.

ecgML is developed using the knowledge of other standards, and tried to remove the drawbacks of these standards. Many elements are taken from these other standards. These include annotations relating to the acquisition protocols, patient information and analysis results ([27]). A drawback of using XML is that it needs lot of storage space, as a lot of overhead is created by the tags. A measurement solving this is that the actual ECG data is not included in XML format, but only a reference to the binary data is included. In this way, the advantages of the use of XML stay for all the extra data added, but not too much overhead is created.

2.2.3 FDA

The U.S. Food and Drug Administration (FDA) is an agency part of the USA government that is responsible for the safety regulations in lot of fields, including biological medical products and medical devices ([5]). They developed a standard called FDA XML Data Format, which is meant for waveforms as well as relevant submission information ([8]). During drug studies, subjects get either the drug under study or a placebo, and are periodically monitored. The data collected during monitoring needs to be in a standard format for further processing and for this an own format was created.

2.2.4 HL7

HL7 is both used for the name of the organisation as well as the name of the standard this organisation has developed. An overview of HL7 can be found in [1]. HL7 creates international standards for clinical text document mark-up, decision support, inter-system and inter-organisation messaging and user interface integration. 'Level Seven' comes from the application layer of the OSI communications model, which is the seventh layer.

[29] describes version 2.5, which was the latest fully approved version by ANSI. Messages are used to transfer data between systems. A three-character code is used in every message to identify its type. The HL7 standard is organised in different chapters that all consist of several segments. Messages consists of segments, which have a name and segment ID which identifies the segment type. Segments can be mandatory or optional, and may occur only once or can be repeated in a message.

Every message starts with a message header segment, containing information about the message, processing and version IDs, sending and receiving applications and facilities, date and time, security and message type. After the header segment, the patient identification segment follows, which includes personal data of the patient. This is followed by the observation request segment, which includes information about a specific observation to be made. After the observation is made, observations are added.

All data in HL7 should be represented in ASCII text, but as this requires more bytes, in practice encapsulation and binary encoding is applied.

2.2.5 SCP-ECG

The European Committee for Standardization (CEN) developed the Standard Communication Protocol for Computer-Assisted Electrocardiography (SCP-ECG) ([10]). The standard is supported by the OpenECG portal[2]. The scope of the standard is data acquisition, encoding, transmission and storage.

No transport protocol is defined in SCP-ECG, which means implementations can freely choose a protocol to exchange the data, as long as the format defined in SCP-ECG is kept. The data structure defines different sections, both mandatory and optional. The pointers and header information, including patient and ECG acquisition data are mandatory, other sections are optional.

[2] http://www.openecg.net/

Chapter 3

Requirements

In this chapter we'll first describe a general model that we'll use during the remainder of this thesis. In the second part we'll describe the requirements for the control protocol we'll build.

3.1 General model

As we do not discuss different models in this thesis, we will assume a certain model. In this section we will describe the model that will be used in the remainder of this thesis.

We define a model with separation of the control plane and the data plane. The control plane will negotiate about the way data will be transferred in the data plane. In the data plane, there is a possibility to insert filters and/or converters in the data stream. This is negotiated by the control plane.

As there do not exist global[1] implemented mechanisms to negotiate service parameters on networks, negotiation will take place directly between the sender and receiver devices. One device can both be sender and receiver. For example, for the communication between a patient and hospital the hospital is the receiver, but when a doctor queries the data, the hospital is the sender. It is possible that these devices are aware of their own network connections, and take this information into consideration when negotiating the data transfer. For example, when the sending device is aware that the UMTS connection is not available, but only a GSM connection, this will influence the maximum data flow. With the information that only GSM is available, it can decide that certain transfer methods are not going to work, and will thus not negotiate these.

The data layer should be capable of transferring both continuous time signals as well as discrete time signals. In this thesis we'll not go into detail how data connections are working, as we'll focus on the control plane, which will negotiate about the parameters used in the data plane.

Discrete time signals often are a single value only. Negotiating the parameters of a data channel and setting up the data channel might need more bytes to be transferred than the actual data would need. In such cases, it would be worth to see if the control plane could

[1] There do exist protocols for negotiating service parameters, like RSVP [7], but they are only implemented in some local networks, and not available on a global scale.

include methods to transmit such data to limit bandwidth use. This will not be a requirement, but if possible, it would be a nice feature.

After negotiating the parameters of a data connection, the control plane will pass these parameters to configure the data plane. The data plane is responsible for setting up the connection, including inserting filters and/or converters in the channel. One control connection should be able to negotiate several data channels, so that for every source to destination connection there is only one control connection set up, while there can exist zero to several data connections between the source and destination device, depending on the needs.

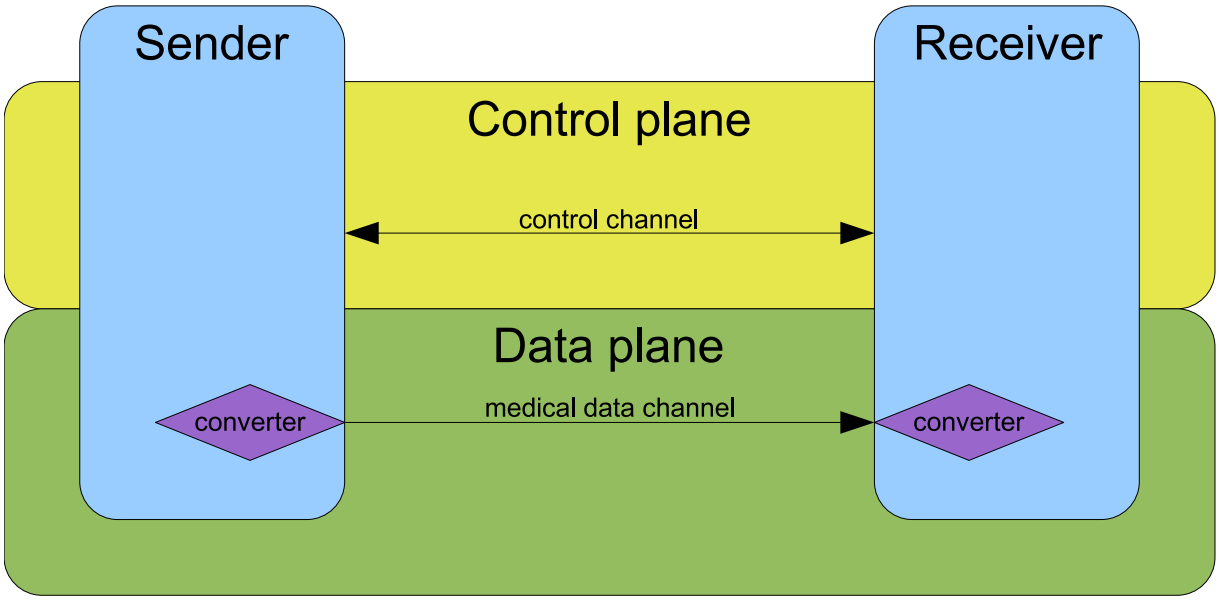

Figure 3.1: General model

Figure 3.1 shows the model. There is a sender and receiver having several communication channels. In the control plane, there is one control channel that takes care of negotiating all the data channels in the best way. In the data plane, converters and a data channel are visible. Converters are optional. There can exist several of such data channels with converters. The definition of 'in the best way' depends on the system settings. This may be based on the following parameters:

- Network traffic: this can both be related to the limits of the network, or to cost-efficient use. Certain channels might be more expensive, and their use should be limited if possible.
- Local resource usage: both the sender and receiver hardware might have limited resources to process the conversion of signals. At both sides this can be related to limited computing power of the device used, or to shared resource usage for all connected other parties.
- Signal quality: for certain usage, a minimum or maximum signal quality might be required.

Both the sender and receiver have static capabilities used for the negotiation, based on the device in use. One could think about the kind of network connection they have (GSM will limit the data rate that is possible for example) or their processing power (certain converters will ask for too many resources for example). Both the sender and receiver also have preferences used for the negotiation. They are normally set by the user of the device. One could think about minimum or maximum sampling frequency or bit rate of the transmitted signals.

Based on the static capabilities and preferences of both sender and receiver, the parametrised negotiation will take place. Negotiation will only take place when a connection is set up, on a running connection no reconfiguration will be taking place.

In the remainder of this thesis a negotiation protocol will be designed, prototyped and tested.

3.2 Requirements

Before we can design a control protocol that serves the objectives, we need to define clear requirements. The requirements are based upon the general model described above. With these requirements we can search for an existing control protocol that covers most of our needs and that can be extended easily with the missing requirements. We define the following criteria:

3.2.1 Not linked to existing medical standards

The control protocol should be able to connect different medical standards for data transfer. To prevent the favouring of a specific one, the control protocol should not depend on an existing medical standard. This makes it more likely that all vendors will implement it later on. This requirement is highly appreciated.

3.2.2 Peer to peer

The model described in section 3.1 does not define a client-server model, but a peer to peer. The control protocol that will be developed in this thesis should be peer to peer. This requirement is mandatory.

3.2.3 Open standard

The protocol designed should become an open standard, so everybody can implement it, and use it without any costs. This will also put restrictions on the standards used as base for the new protocol, as they should allow reuse. This requirement is mandatory.

3.2.4 Low overhead

Many devices used in the medical world have only a limited amount of processing power and a limited bandwidth connecting to other devices. This makes it necessary for the control protocol to use as little as possible processing power and bandwidth when negotiating about the actual data transfer. The processing power is hard to measure, as it also depends on which program components are used. In [11] a comparison between storing ECG data in the XML and ASN.1 format is made. It can be concluded that ASN.1 is much more efficient in

space usage than XML. Important to keep in mind is that using extensive compression to limit the bandwidth will cause extra processing power to be used, which might result in more overhead in other fields. This requirement is highly appreciated.

3.2.5 Low latency

The setting up of a connection should be done as quick as possible. There exists the possibility that a request for data is inter-active, so connection set-up should not take too long. Also, the medical data might be part of monitored critical values, and any delay in transmitting them could cause health risks for the patient. This requirement is highly appreciated.

3.2.6 Both sender and receiver should give their preferences

When negotiating about a data transfer, both the sender and the receiver should be able to give their preferences and capabilities. As the roles of sender and receiver are flexible (one time the sender can be the one initiating the connection, while the other time it can be the receiver initiating the connection), both should be able to give their preferences for the connection. This is needed because both the sending and receiving side might have limitations or minimal requirements for the data. In these preferences, hosts can include for example network costs, network capabilities, local resource usage, local resource availability and required signal quality. This requirement is mandatory.

3.2.7 Both sides should know the agreement

To be able to set up a connection, both the sender as well as the receiver should know the actual agreement, to be able to prepare themselves for the data transfer. It should not matter who makes the final decision, but both sides should know the final agreement afterwards. This requirement is mandatory.

3.2.8 Both sides can conclude the agreement (but only one at a time)

The side making the final decision of the agreement should be independent of who initiates the control connection. For example, if the host setting up the control session thinks it has more processing power than the peer, it could request to make the final decision, but when it thinks the peer has more processing power, it can propose to let the peer decide. Negotiating about who makes the decision should not cause extra messages to be exchanged. This requirement is not mandatory.

3.2.9 Different protocol options of the same standard should be supported

Some of the medical standards only define a storage format, but do not specify how data should be transferred over the network. Due to the lack of this specification, different transfer methods may exist. The control protocol should be aware of this, and make sure that the final agreement is unambiguous for both sides. This requirement is mandatory.

3.2.10 The control protocol should be extensible with new data protocols

At this moment, several standards for storing and transferring medical data exist. It can be expected that over time, more standards will emerge. There is also the possibility that new

implementations of existing standards emerge that are not fully compatible with existing ones. For this reason it should be possible to extend the control protocol with the new standards and implementations that emerge. This requirement is highly appreciated.

3.2.11 Support to transfer the actual medical data for small data sets

In some cases, the amount of data that needs to be transferred might be less than what it would take to set up a connection, for example when transferring a single blood pressure. In such cases, it would save time and data transfer if the control protocol would transfer the data itself instead of negotiating a data channel to transfer the small piece of data. This requirement is not mandatory, but would be nice to have support for.

3.2.12 Secure

As sensitive medical data is transferred, the protocol should be secure, in the sense of authentication and integrity. Although the control protocol does not include the actual data, it should make sure it does not tell a host to set up a malicious connection with an untrusted host. When actual medical data is transferred, according to the requirement 3.2.11, confidentiality is an important issue as well. This requirement is mandatory.

3.2.13 Possibility to reconfigure

Although not included in the first draft of the protocol, it should be possible in the future to extend the protocol to provide the possibility of reconfiguring an existing connection. This would mainly influence the life-time of a session that is set up for the negotiation, but without considering this point already now, it might be very complicated to add this functionality later on. This requirement is mandatory.

Chapter 4

Design

In this chapter the design of the control protocol is described. We start with choosing an existing control protocol that suits already best our needs, and that will be extended in the second part with the missing functionality.

4.1 Choosing an existing control protocol

There exist a lot of control protocols already. Several of them are described in section 2.1. By using one of them, and extend it to meet the requirements specified in section 3.2, we save a lot of time, and prevent a lot of mistakes in the design, as these protocols have been extensively tested, and proved themselves in practise.

First, we will evaluate the existing control protocols to see how much they support of the requirements. Everything that is not supported yet, should be possible to be build into the existing protocols.

4.1.1 Not linked to existing medical standards

The protocol should not be based on an existing medical standard.

SIP

SIP is designed for multimedia sessions, and not for medical use (though this doesn't prevent it from being used for medical applications).

RTSP

RTSP is designed for transferring real-time data, and not for medical use (though this doesn't prevent it from being used for medical applications).

XMPP

The origin of XMPP is instant messaging, not related to medical use.

DSM-CC

DSM-CC is designed for handling video streams, and not for medical use (though it doesn't forbid being used for medical applications).

H.323

H.323 is designed for handling video and audio streams, and not for medical use (though it doesn't forbid being used for medical applications).

4.1.2 Peer to peer

The protocol should support peer to peer.

SIP

SIP has been designed for peer to peer. There is full support.

RTSP

RTSP is made for controlling multimedia streams from a server to a client. But, also clients can ask a server to join a session for recording, so it should be possible to make RTSP peer to peer.

XMPP

Messages in XMPP can be exchange to any other client as far as the server configuration allows. Also messages to a server itself are allowed.

DSM-CC

DSM-CC is made for transferring and controlling multimedia streams from a server to a client. It is not possible to make it peer to peer.

H.323

H.323 is made for transferring and controlling multimedia streams from a server to a client. It is not possible to make it peer to peer.

4.1.3 Open standard

SIP

SIP is an open standard. It is defined by a RFC, meaning it can be freely reused.

RTSP

RTSP is an open standard. It is defined by a RFC, meaning it can be freely reused.

XMPP

XMPP is an open standard. It is defined by a RFC, meaning it can be freely reused.

DSM-CC

DSM-CC is an ISO standard. ISO has the copyright of these standards, and charges for receiving copies.

H.323

H.323 is an ITU-T standard. The standard as such is freely available, but patents might exist for certain parts of the standard.

4.1.4 Low overhead

The protocol should limit the bandwidth use and processing power.

SIP

SIP sends messages in plain text. This is less efficient than binary encoding. Commonly used keywords in SIP do have a compact form, mainly to limit the size of messages to prevent exceeding the MTU of the used transport protocol.

RTSP

RTSP sends messages in plain text. This is less efficient than binary encoding. There do not exist compact forms of keywords.

XMPP

XMPP uses XML to submit messages. This creates a lot of overhead in the messages, while on the other hand they are easy to process by computers or read by humans.

DSM-CC

DSM-CC uses the MPEG transport stream, which is a binary format.

H.323

H.323 uses binary encoding, namely ASN.1 with Packed Encoding Rules (PER)

4.1.5 Low latency

The connection set up should be done as quick as possible, meaning in as little as possible request-response sequences.

SIP

For setting up a connection, normally 3 messages are enough, provided that both sides support a common format. These messages are *INVITE*, *200 OK* and *ACK*.

RTSP

At least three request-response sequences are needed, one for retrieving a media description, one for setting up a connection and one for starting the stream.

XMPP

There is the need for a continues connection between the client and a server. If messages are send to a user that is connected to the same server, no connections have to be set up to send a message. If a message is send to a user on another server, a connection might have to be set up first, but it might exist already. Depending on the implementation of the negotiation, only one request-response could be enough for negotiation, without the need to set up any connection.

4.1.6 Both sender and receiver should give their preferences

Both the sender and receiver should give their own preferences and capabilities as input to the negotiating process.

SIP

When setting up a session, a SDP body can be included to describe the capabilities of the initiating side. The answering side builds a final agreement from the received SDP body together with its own capabilities. If the initiating side does not include a SDP body, the receiving side includes a SDP body with its capabilities, after which the initiating side builds a final agreement based on the received SPD body together with its own capabilities. Only one side is aware of the capabilities of both, the other side only of the common capabilities.

RTSP

RTSP is intended to provide the interaction between media servers and clients interested in the contents on the media server. To set up a media connection, the client requests a description from the server. As this description is not defined, it should be possible to include a description which also specifies preferences. It is possible to inform a server via *ANNOUNCE* about an existing session. Afterwards it can be asked to *RECORD* it if the initiating side wants to deliver data to the receiving side.

XMPP

XMPP is not made directly for session negotiation, but, it can transfer messages to other clients. Depending on how the negotiation process is implemented, it is possible that both sender and receiver give their preferences.

4.1.7 Both sides should know the agreement

After the agreement has been concluded, both sender and receiver should be aware of this agreement.

SIP

When the initiating side includes a SDP body, the receiving side generates the agreement, and includes it in the response to the initiating side. If the initiating side did not include a SDP body, the receiving side will send its own one, after which the initiating side generates the agreement. This agreement is included in the *ACK* request.

RTSP

RTSP is intended to provide the interaction between media servers and clients interested in the contents on the media server. Because of this, a client retrieves a description of the available media presentation, and afterwards requests to receive it. In a *DESCRIBE* response, or *ANNOUNCE* request, several options can be included. By afterwards looking at the *SETUP* request of the initiating side, the receiving side knows the final choice.

XMPP

XMPP is not made directly for session negotiation, but, it can transfer messages to other clients. Depending on how the negotiation process is implemented, it is possible that both sender and receiver know the final agreement.

4.1.8 Both sides can conclude the agreement (but only one at a time)

The side concluding the agreement should be independent of which side requests the connection set up.

SIP

The initiating side decides who will conclude the agreement. If the initiating side includes a SDP body in the initial request, the receiving side will conclude the agreement. If the SDP body is not in the initial request, it will be included in the response to this request, and the initiating side concludes the agreement.

RTSP

With RTSP, it is always the client (initiating side) that sets up the connection, and decides about the final parameters to use if there are several possibilities. The server has no control, unless only providing one option.

XMPP

XMPP is not made directly for session negotiation, but, it can transfer messages to other clients. Depending on how the negotiation process is implemented, it is possible that either sender or receiver concludes the agreement.

4.1.9 Different protocol options of the same standard should be supported

If a protocol has different options, they should be distinguishable in the negotiation.

SIP

SIP lacks the mechanism to negotiate a medical data protocol. It does provide a mechanism to negotiate media sessions via SDP. Depending on how SDP will be extended or replaced for negotiating medical data protocols, implementing this is possible.

RTSP

The description of the media sessions is not defined. It is possible to include any kind of description, if this description allows the support of different options of the same protocol, this is possible.

XMPP

XMPP is not made directly for session negotiation, but, it can transfer messages to other clients. Depending on how the negotiation process is implemented, it is possible to support different protocol options of the same standard.

4.1.10 The control protocol should be extensible with new data protocols

The data protocol should not be hard-coded in the control protocol, but be adjustable at run-time.

SIP

This is much related with section 4.1.9 When designing the negotiating mechanism this should be taken care of.

RTSP

If the message body is defined in such a way that it can be extended, this is possible.

XMPP

XMPP is not made directly for session negotiation, but, it can transfer messages to other clients. Depending on how the negotiation process is implemented, it is possible that new data protocols are added afterwards.

4.1.11 Support to transfer the actual medical data for small data sets

The control protocol should be able to contain medical data if this is a small data set (optional).

SIP

SIP supports including different message bodies via the MIME specification. When a message body that can include medical data is defined, this can be included in the SIP messages.

RTSP

RTSP allows the inclusion of message bodies in several of its methods. With properly defined message bodies, medical data can be included.

XMPP

XMPP is made to transfer messages to other clients. As different kind of messages can be defined, it is possible to include actual medical data.

4.1.12 Secure

Authentication and integrity should be provided by the control protocol, as well as confidentiality when medical data is transferred via the control protocol.

SIP

SIP is derived from HTTP/1.1, for which several security mechanisms are defined, including TLS. SIP proxies can normally freely route SIP packets, and security with TLS is only hop-by-hop, meaning all intermediate proxy servers can read the contents of the messages. As forwarding might include also untrusted proxies, and because our model only defines direct connections in section 3.1, the proxies should not be used. Authentication can be provided by Digest Access Authentication (see [13]). This is not the most strong authentication, but combined with TLS it should be sufficient, especially since only negotiation takes place, and not the transfer of actual medical data. Only when actual (small amounts of) medical data are transferred, additional encryption of the data attachments should be considered.

RTSP

RTSP is derived from HTTP/1.1, for which several security mechanisms are defined. Attachments can be encrypted to provide confidentiality.

XMPP

XMPP implements TLS and SASL. This provides authentication, integrity and confidentiality.

4.1.13 Possibility to reconfigure

It should be possible to reconfigure an ongoing data transfer.

SIP

SIP provides the possibility to reconfigure an existing session.

RTSP

In RTSP both the sending as well as the receiving side can request a reconfiguration. When the server announces a reconfiguration, the client cannot reject and has to follow the instructions in the announcement.

XMPP

XMPP is not made directly for session negotiation, but, it can transfer messages to other clients. Depending on how the negotiation process is implemented, it is possible that reconfiguration takes place.

4.1.14 Summary and conclusion

In table 4.1, an overview of the evaluation of all requirements for the different protocols is given. '+' means it is available, '+ -' means it is not available by default, but it is possible to make it available as extension, and '-' means it is not available by default, and also not possible to make it available. The requirements marked with an asterisk (*) are mandatory according to the requirements defined in section 3.2. When looking at the requirement 'peer

Requirement	SIP	RTSP	XMPP	DSM-CC	H.323
Not linked to existing medical standards	+	+	+	+	+
Peer to peer*	+	+ -	+	-	-
Open standard*	+	+	+	+ -	+ -
Low overhead	+ -	+ -	-	+	+
Low latency	+	+ -	+		
Both should give their preferences*	+	+	+ -		
Both sides should know the agreement*	+	+	+ -		
Both sides can conclude the agreement	+	-	+ -		
Different protocol options supported*	+ -	+ -	+ -		
Extensible with new data protocols	+ -	+ -	+ -		
Transfer the actual medical data	+ -	+ -	+ -		
Secure*	+ -	+ -	+		
Possibility to reconfigure*	+	+ -	+ -		

Table 4.1: Summary of the requirements conformance of currently available control protocols

to peer' we can see that this is not possible for both *DSM-CC* and *H.323*, meaning we cannot use these protocols to continue our work, as this requirement is mandatory.

When comparing *SIP*, *RTSP* and *XMPP*, we see that *SIP* has the best overall score, followed by *XMPP*, followed by *RTSP*. Based on this result, we will continue with the development of the control protocol using *SIP*.

As can be seen, SIP already provides most of our requirements. The following items need to still be designed:

- SDP-like document type for negotiating medical data protocols.
- A document type for sending actual medical data, possibly including extra security measurements.

4.2 Document for negotiating medical data protocols

As can be seen in the previous section, SIP provides already a lot of functionality that we need according to our requirements. Some functionality is missing, and needs to be added. In this section, we'll describe the document for negotiating medical data protocols, based on our findings in the previous section.

4.2.1 Negotiating parameters

With the choice of SIP, negotiation requires only two data transmissions. In the first transmission, A sends its preferences to B. B will compare this with its own preferences and conclude the best option. In the second transmission, B sends back the conclusion to A. A and B are not fixed, and independent of who initiates a connection and who answers a connection request.

There is always one host that decides the final agreement, using the preferences received from the other host. For making the proper decision, meaning the most optimal connection parameters, a host should be able to describe its preferences in full detail. This includes for example bitrate, delay, jitter and resolution, which can be bound to a certain acceptable range. Also, a host might only have a limited amount of understood standards, maybe even a limited number of different transport protocols of the same standard. The document describing the preferences and requirements should be able to describe this in full detail, so that a proper decision can be made.

The description should be fully standardised, so that all hosts have the same understanding of the requirements and preferences of each other. To make a proper decision, not only the requirements should be communicated, but also the parameters of the supported formats, as they might not be totally fixed for a certain standard. It might happen that certain parameters are unknown, in which case they cannot be taken into account. This happens for example with the delay when the speed to convert a signal depends on the available processor power, but the available processor power is unknown. When a parameter is unknown for a standard, it should still be possible to accept that standard if the host sending the offer is fine with that. It should be possible to specify this in the offer.

4.2.2 Media type for negotiating medical data protocols

The SDP document (see appendix A.6 and [17]) does provide functionality to negotiate multimedia sessions, but not for negotiating medical data sessions. The ideas of SDP are very useful though. We'll build the new format based on SDP. This new media type will need a name. Medical Session Description Protocol, abbreviated as MSDP, seems to be a suitable name for the new standard. The media type registration at IANA could be "application/msdp".

4.2.3 Medical Session Description Protocol (MSDP) definition

In section 5 of [17] the SDP description is split into *session description*, *time description* and *media description*. Both the *session description* and *time description* can be partly reused, while the *media description* has to be replaced with a description suitable for our purpose.

The *session description* of SDP can be partly reused. The *v* type letter (protocol version) must be *0*. The *o* type letter (origin) and *s* type letter (session name) are compulsory, while the *i* type letter (session information) is optional, like in SDP. The *u* (Uri), *e* (e-mail) and *p* (phone) type letter must not be used. The *c* type letter (connection data) is again compulsory. The *b* type letter (bandwidth information) may only be used in case streaming is used, as in case of non-streaming this does not make sense. The *k* type letter (encryption key) must be omitted, as this method of key exchange is not secure enough, because the MSDP messages might be exchanged in plain text.

The *time description* must be used in case streaming is used. In case non-streaming transfer is used, the *t* type letter (time) denotes the time the measurement was taken. A range denotes longer-lasting measurements, and a same start and end time denote that the measurements were taken at one point in time. The *r* type letter (repeat times) must be omitted, as for sending new measurements, new sessions need to be set up. The *z* type letter (time zones) may be used.

The *media description* of SDP should be totally omitted, and replaced with a *medical description*. One control session should be able to control several data sessions. This means that several medical channel descriptions are allowed in one MSDP message, like SDP can include zero or more media descriptions. The type letter *m* can be reused to define a medical channel. The syntax for *m* is:

```
m EQUAL <transfer> SP <content>
<transfer> = "stream" / "packet"
<content> = "bloodpressure" / "ecg" / "mixed" / token  ; 'token' should be pre-defined
```

The parameter `<transfer>` specifies if the channel is a streaming one, or non-streaming (denoted by 'packet'). Non-streaming can be used for transferring recorded measurements as one file, while streaming can be used to display real-time recordings. They are represented by the keywords *stream* and *packet*. The parameter `<content>` defines the type of data that is being send over this medical channel. Pre-defined values have to be used to make sure that there is a common understanding. For the moment we will define only *bloodpressure*, *ecg* and *mixed*, but more values should be defined in future. The value *mixed* can be used when several kind of measurements are combined into one file or stream. A medical channel means one file or stream in the remainder of this text.

By defining the type of the medical channel and how it is transferred, one can easily parse the MSDP message, and recognise the parts that are useful to process. Because it is possible to define more than one medical channel type, it is possible to negotiate the parameters for different medical channels at once. This can be used when different kind of measurements belong together, but are not stored into one format that can handle all at once. In case a format is used that is able to handle different medical channels at once, the *mixed* type should be used.

After the definition of the medical channel the requirements or available parameters for the received data are described. This is done by the type letter *n*. When the MSDP offer is generated by the sender, these lines define the parameters of the original signal that is received from the source (for example a sensor). This is used by the receiver to see if loss or padding will happen when using certain standards. When the MSDP offer is generated by

the destination, these lines define the requirements of the data to be received. The syntax for n is:

```
n EQUAL <key> HCOLON <range> [SP "optional"]
<key> = "bitrate" / "delay" / "jitter" / "resolution" / token  ; 'token' should be pre-defined
<range> = <minimum> "-" <maximum>[SLASH <step>] /
          "-" <maximum> /
          <minimum> "-" [SLASH <step>] /
          <value>
<minimum> = 1*DIGIT
<maximum> = 1*DIGIT
<step> = 1*DIGIT
<value> = 1*DIGIT
```

The parameter `<key>` defines the parameter described, directly followed by a colon and then the acceptable range, where the minimum and maximum are separated by a dash. Only positive integers can be used for the range, and the minimum and maximum value are included in the range. Either the minimum or maximum might be omitted in case that side is unbound. If the value is totally unbound it must not be mentioned as parameter. If only specific values in the specified range are allowed (only available in case a lower bound is given) the step value is provided immediately after the range, separated by a slash. To require a specific value, only that value must be mentioned and the dash must be omitted. The parameter `optional` means the acceptable range can be ignored in case it is unknown for a certain standard.

This type letter can be repeated several times to describe different parameters, but can also be totally omitted in case no specific requirements apply. Parameter names and their units of measurement must be standardised. For the moment we define *bitrate* (in kbits/s), *delay* (in ms), *jitter* (in ms) and *resolution* (in bits), but more parameters can be defined in future. *Bitrate* and *resolution* are measured at the output, while *delay* and *jitter* are values caused by using a certain converter.

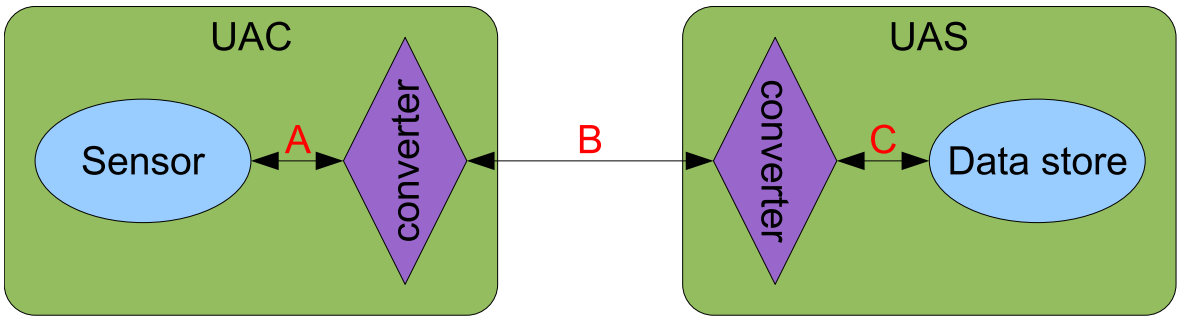

Figure 4.1: Specification from which point n is seen.

Figure 4.1 shows a typical view on the data channel in use. In the UAC a sensor measures data and makes this available. It goes through a converter, and is then transferred over the network to the UAS. In the UAS it goes first through a converter, and then for example to a data store.

When the UAC generates a MSDP offer, the n type lines immediately after the m type line define the parameters of the data that comes from the sensor, in figure 4.1 denoted by 'A'.

The n type lines following the q type lines (defined below) define the parameters of the data that is transferred to the UAS (denoted by 'B') in case the standard defined in the q type line is used. If the values for the same parameter differ between the ones defined after the n type lines and the ones defined after the q type lines, it means in some cases that either padding needs to be done, or loss of data results (for bitrate and resolution). When the UAS generates a MSDP offer, the n type lines immediately after the m type line define the parameters of the data it wants to receive (denoted by 'C'). For this, also the data generated by the sensor in the UAC (denoted by 'A') needs at least this value, as well as the data after the converter (denoted by 'B').

After the requirements zero or more different options for sending the data are defined with the type letter q. The syntax for q is:

```
q EQUAL <preference> SP <standard> [SP <transport> [SP <parameter>]...]
<preference> = 1*DIGIT  ; range 0-100
<standard> = "DICOM" / "ecgML" / "FDA" / "HL7" / "SCP-ECG" / token  ; 'token' should be pre-defined
<transport> = token  ; 'token' depending on <standard>
<parameter> = token  ; 'token' depending on the <standard> <transport> combination
```

The parameter `<preference>` defines the preference for this option, where the preference is a value between 0 and 100, with 0 meaning not accepted (not very useful to specify) and 100 most preferred. This is followed by the name of the standard (`<standard>`), and optionally followed by a specific transport protocol (`<transport>`), for example when the transport protocol is not defined. Names of the standard are pre-defined. This may be followed by specific parameters needed for this standard, where their order should be standardised, and depends on the `<standard>` and `<transport>` combination. For the moment we define for `<standard>` *DICOM*, *ecgML*, *FDA*, *HL7* and *SCP-ECG*. Transport and possible other parameters needed for negotiation should be defined before these standards can be used.

Take as example standard *SCP-ECG*, which does not specify a transport protocol. We will specify now that the transport can be done over HTTP, FTP and SMTP (but more options might exist). For both HTTP and FTP we leave the choice of port number undefined, while for SMTP we define the fixed port number 25 (in this example). In this case, the following needs to be defined:

- SCP-ECG HTTP `<port>`
- SCP-ECG FTP `<port>`
- SCP-ECG SMTP

When using these values, `<port>` needs to be replaced by a real port number. Because SMTP has a fixed port number, there is no need to define an extra parameter here, so it should be omitted. Using this specifications, a resulting q type line could be:

```
q=75 SCP-ECG HTTP 80
```

This means that with a preference of 75 standard SCP-ECG using the HTTP implementation on port 80 is an option.

There is a need to define the parameters in this fixed way, as otherwise different hosts can have different understandings of the parameters. In case the registered transport appears

later to need more parameters, a new name, with different parameters, needs to be registered. Consider for example that there exists an implementation of the *SCP-ECG* standard using a HTTP protocol that can specify the trunk size, next to the port number. Reusing the name *HTTP* would cause misunderstandings between hosts, so this new registration will have to find a new name, for example *HTTP-ts*, resulting in a registration "SCP-ECG HTTP-ts `<port> <trunksize>`".

The `<preference>` is set by the host generating the MSDP message, and gives the preference of that host for this option. There is no pre-defined formula to calculate this value. A host is free to define the formula to calculate this value, and define the parameters used for the calculation. One host might for example give preference to standards requiring low processing power, while another one might prefer low amount of data to be transferred. To allow negotiations in an easy way, the `<preference>` value is not defined further. The preferences of both hosts are compared without looking at how this value was calculated. In this way, no priority is given to the preferences of a specific host, and both have equal chances to have their preference used. If both hosts have the same interpretation of the calculation of the preference value (for example, both prefer low processing power), the final agreement will be the optimum for both hosts. If, on the other hand, both hosts have a different interpretation of the calculation of the preference value, the conclusion might be sub-optimal for one host, but the overall conclusion will be the best one possible. This is because a host is supposed to share all its possibilities with the other, which makes it possible to calculate the best possible overall conclusion.

Every option should be followed by a description of the parameters of this option. This is done with the type letter n as described above. Ranges are allowed if the standard supports different options, but unbound values are not allowed. Unbound values are not allowed because it is unrealistic that any value for a parameter would be possible in a certain standard. When the chosen standard defines a fixed value for a parameter, it is not useful to specify it, and the value should be hard-coded, so it can be used for concluding the negotiation. For example, when standard X defines a resolution of 16 bits (and not 8, 12 or 16 bits), there is no possibility to negotiate or change this value, and specifying it in the MSDP message would waste bandwidth. For this reason, when a host implements standard X, it must be aware that the resolution is 16 bits. The second parameter (keyword *optional*) must not be used. If *optional* is received, it must be ignored.

4.2.4 Concluding the agreement

Setting up a SIP connection requires a connection set-up request from the initiator to the intended receipt. In this request a session description (in our case the MSDP) can be included, but it is also possible to request the intended receipt for a proposal (by not including a session description at all). This is comparable to the two cases described in figure A.1, which explains the SIP session set-up using SDP messages. The host receiving the proposal can compare it with its own preferences and capabilities, and return a final agreement to the other host.

It might happen that the host sending the MSDP offer puts higher requirements than are possible by the receiver of the offer. In this case no agreement is possible. This is made clear by sending an empty MSDP answer (no m, n and q type lines). When receiving this answer, the host knows that the requirements were too high. It can now either decide it doesn't want

a connection, or it can lower its requirements. When sending a new proposal, there is again the risk the requirements are too high. For this reason, it can opt to ask the other host to send an offer instead, so it can decide itself if the available parameters of the other host are sufficient for communication.

Because a PDA is low in resources, the calculation of the final agreement can best be done by a server, but it is not known in advance which of the two hosts involved in a connection set-up is the server, and first negotiating this would also consume resources and cause delay. Connection set-up is normally done between either a PDA and server, or between two servers. If the initiating host is a server, the answering host can be a PDA or a server. It is the most save if the initiating host would do the calculation of the final agreement, as in that case for sure not a PDA will do so. If the initiating host is a PDA, the answering host is for sure a server. In this case, the answering host should do the calculation, because it normally has more resources.

The initiating host is the one deciding who will calculate the final agreement. If it includes a proposal in the connection request, it will be the answering host deciding, if it does not include a proposal in the connection request, the answering host will include the proposal, and the initiating host will decide. In case the initiating host knows it is a PDA, the answering host is a server, meaning that it has more resources. In this case, the initiating host should send a proposal, so the server (answering host) can make the final decision. If, on the other hand, the initiating host knows it is a server, it should not include a proposal, as in this case the answering host is forced to return a proposal, after which the initiating host can make the final decision. If the initiating host is not able to detect if it is a PDA or server, it is not a big issue, but in this case the calculation might take a bit more time and consume more resources, like the battery of the PDA.

The proposal that is send is based on the capabilities of the host and user preferences. The capabilities define which converters are available, which formats are understood and which formats don't exceed the maximum transfer rates of the connected network. The user preferences define the requirements of the user of the system. In case of a storage server, these user preferences can also be based on the needs of the users requesting the data afterwards. The user preferences can request a minimum quality of the data to be able to make a proper evaluation, or maximum quality to prevent the need for too much storage space. Merged together the capabilities and user preferences form the proposal that is used for the negotiation. This merged proposal is described by the MSDP. The `<preference>` in the q line can be used to define the relative preference for a certain standard compared to the other standards mentioned.

4.2.5 Protocol messages

SIP is normally used for setting up media sessions. In this kind of sessions, loss of data is acceptable, especially when happening at the beginning or ending of a data stream. With medical data this is a big issue though, meaning that data should only be send when the receiving side is ready to receive the data. SIP does not provide the possibility to detect when the receiving side is ready to receive the data, due to its optimisation in amount of message exchanges needed to set up a session. As soon as the session parameters are concluded, or the conclusion is received, a host should prepare for the data transmission. This includes inserting

filters in the data channel, opening ports and preparing protocol stacks. No signalling to the other host is done via SIP when a host is ready to receive data.

To make sure data is only send after the receipt is ready to receive the data, an extra message will have to be included to notify the sender it can start sending data. In appendix A.7 and [9] an extension is described that allows to exchange messages over SIP. As soon as the receipt of the data is ready to receive the data, it should notify the control plane, which will send a *MESSAGE* with a body of the type *text/plain* and the contents "ready to receive" to the control plane of the sender. The control plane of the sender will confirm the message with a *200 OK*, and inform the data plane to start transmitting data. This means that compared to normal SIP operation, an extra message is exchanged before data is send, instead of just starting to send data and don't bother about the first part being lost.

As we don't use SIP in the way it was initially meant for, we should make sure that we don't communicate with 'normal' SIP servers, as this doesn't make sense. This can be achieved by including a *Required* header: X-Medical. This value is also put in the *Supported* header. We do not follow the rule of SIP that in case a requested *Required* header is not supported by the other peer that we should fall back to basic *SIP* operation, as this would neglect the purpose for which we introduced it.

When supporting X-Medical, the implementation should be able to accept the following content types: application/msdp, application/medical and text/plain. The application/medical might be omitted in case no support for direct transferring of medical data is included. The accepted content types should be mentioned in the *Accept* header.

Another extension to SIP is that in the *From* header, the parameter *isSender* must be included in an *INVITE* request. It can have the value 'true' or 'false'. The value 'true' means that the initiator of the session is the sender of the data, while the value 'false' means the initiator of the session is the receiver of the data.

Figure 4.2 shows the transactions in the case the UAC sends the MSDP offer. In the control plane first an *INVITE* request is send, including an MSDP offer. The UAS first replies with *180 Ringing* (which can be omitted in case the next reply (*200 OK*) is send immediately), followed by a *200 OK* reply, including a MSDP answer. The UAS also sends a configure message to the data plane, including the MSDP answer, so the data plane can configure itself. After the UAC receives the *200 OK* reply, it sends a *ACK* request to confirm the receipt of the *200 OK*. It also sends the MSDP answer to the data plane, so it can configure itself for the data transfer. Now both data planes know the configuration parameters, and will configure themselves.

Both the data planes inform their control plane that they are configured. The control plane of the receiving host generates a *MESSAGE* with a body of the type *text/plain* and the contents "ready to receive" and sends this to the UAC. The UAC confirms this message with the *200 OK* reply, and informs the data plane to start sending the data.

In this case it is the UAC that transfers the data to the UAS, so after it finishes transferring the data (including confirmation from the UAS, which should be arranged in the data protocol) it will inform the control plane it finished. Now the UAC control plane will send a *BYE*

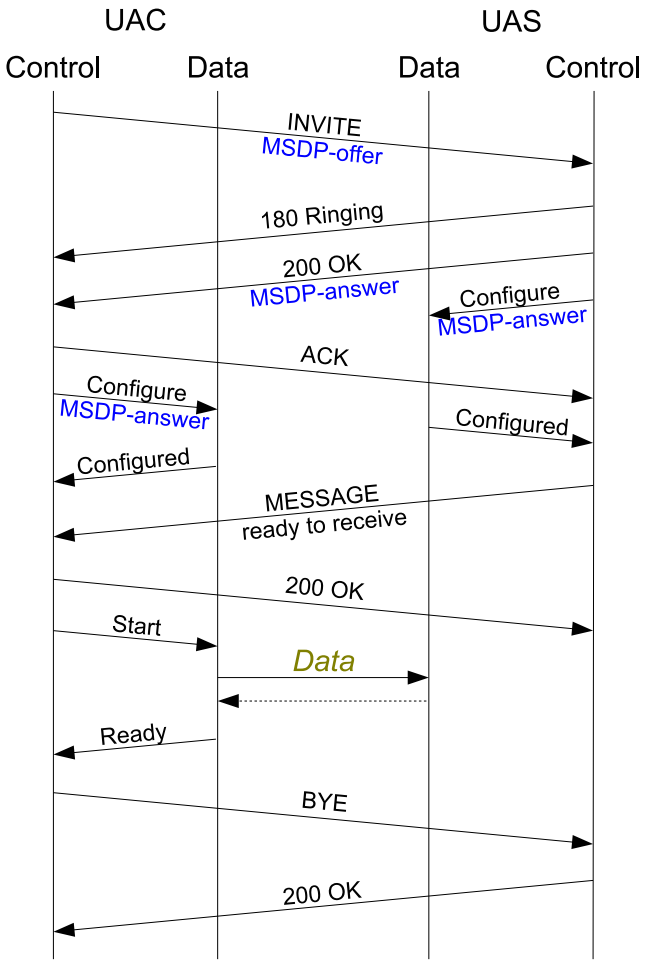

Figure 4.2: Session transactions for the case where the UAC sends the MSDP offer.

request to the UAS to close the control session as well. The UAS replies with a *200 OK*, and the control session is closed as well.

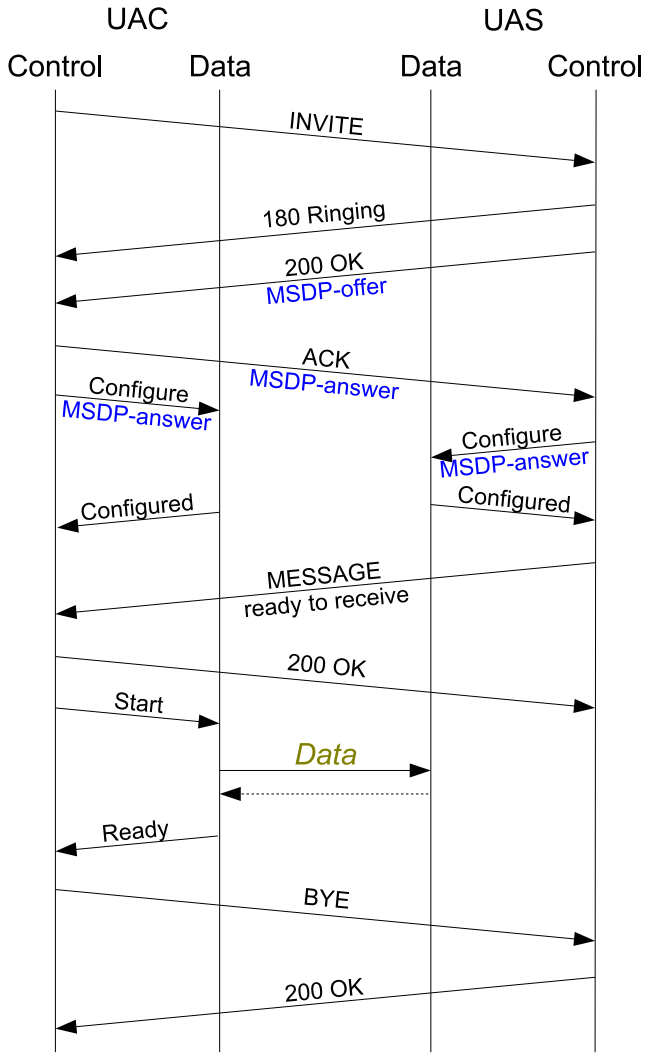

Figure 4.3: Session transactions for the case where the UAS sends the MSDP offer.

Figure 4.3 shows the transaction in the case the UAC does not include a MSDP offer in the *INVITE* request, and as a result the UAS will have to include one in the *200 OK* reply. The final agreement is made by the UAC, which sends it back to the UAS in the *ACK* message. It also informs the data plane to configure itself with these parameters.

When the UAS receives the *ACK*, including the MSDP body, it will inform the data plane to configure itself. When the data plane is configured, it will inform the control plane. The UAS, which is the receiver of the data, will generate a *MESSAGE* with a body of the type *text/plain* and the contents "ready to receive" and sends this to the UAC. The UAC will confirm this message with a *200 OK* reply, and inform the data plane it can start transmitting the data.

As it is also here the UAC that transfers the data to the UAS, it is the UAC closing the control session.

Both in figure 4.2 and figure 4.3 it is the UAC that transfers data to the UAS. For this reason it is the UAC that initiates the closing of the control session. Only in case the receiver of the data (in this case the UAS) suddenly decides it does not want to receive the data, it can close the control sessions by sending a *BYE* request. In case the data transfer would be from the UAS to the UAC (the client requests data, instead of offering to send it), the figures would be the same until the *Configured* messages from the data plane to the control plane. The *MESSAGE* and the related *200 OK* will go in the other direction, and the *Start* will be generated in the UAS. The *Ready* message from the data plane to the control plane would be send in the UAS instead, and also the *BYE* request would be send by the UAS.

4.2.6 Security considerations

Although no medical data will be transferred by SIP, it is still advised to use TLS for setting up connections, including authentication. As for now we only define a direct connection between the UAC and UAS, without the use of proxies. Because of this, the use of TLS provides us with an end-to-end encryption, because no intermediate hops exist.

4.2.7 Example

```
1  m=packet ecg
2  n=bitrate:10-20 optional
3  n=resolution:8-16/4
4  q=100 SCP-ECG http 80
5  n=resolution:12
6  n=delay:60
7  q=70 ecgML http 8080
8  n=bitrate:18
9  q=10 HL7 ftp 21
10 n=bitrate:10
11 n=resolution:12-14
12 n=jitter:50
```

Figure 4.4: Example of MSDP offer (medical description only)

An example of a medical channel in a MSDP packet that is send by the destination host of the medical data is shown in figure 4.4. Line 1 of figure 4.4 defines that a file (*packet*) with ECG data will be delivered. Line 2 and 3 of figure 4.4 define that a bitrate between 10 and 20 kbits/s and a resolution of 8, 12 or 16 bits is requested. In case bitrate is not known it is allowed to still select it (because of the keyword *optional*), but the resolution must be provided and within the range specified. Line 4, 7 and 9 of figure 4.4 define the different protocols possible, including their options. With preference 100 the SCP-ECG standard over HTTP (on port 80) is requested, with preference 70 the ecgML standard over HTTP (on port 8080) and as last option with preference 10 the HL7 standard over FTP (port 21). Line 5 and 6 of figure 4.4 define the parameters for SCP-ECG (line 4 of figure 4.4), line 8 of figure 4.4 defines the parameter for ecgML (line 7 of figure 4.4) and line 10, 11 and 12 of figure 4.4 define the parameters for HL7 (line 9 of figure 4.4). Most of the parameters have a single value, but line 11 of figure 4.4 defines that the resolution can be varied between 12 and 14 bits by HL7.

When a host receives a proposal from another host in a session set-up, it will calculate a final agreement, based on the received MSDP and its own capabilities and user preferences. The result is returned back in the resulting MSDP. This MSDP reply will only contain one option for the communication (meaning one q-type per m-type).

```
1 m=packet ecg
2 q=80 ecgML http 8080
3 n=bitrate:18
4 n=resolution:16
```

Figure 4.5: Example of MSDP answer (medical description only)

When the example of figure 4.4 is received, and the host supports both ecgML and HL7 with the same preference, but not SCP-ECG, the resulting reply (medical description only, the session and time description are like they are in normal SDP messages) is shown in figure 4.5. Line 1 of figure 4.5 is the same as in the initial proposal. Line 2 of figure 4.5 defines the chosen option, including the parameters. The first parameter, the preference value, does not really matter in the reply as only one option is provided, but should give the 'final score' for this choice, so the other side can see what the average of the `<preference>` values for the choice is. The final score is the average of the `<preference>` values of both the sender and receiver, or any value that is between the two `<preference>` values and related to the way of calculating the best option. If no final score is calculated, a value of 0 must be used. The higher the value for final score (still within the range 0-100), the closer to the overall optimum the final agreement is. When looking at the two MSDP examples, the host that calculated the conclusion had a `<preference>` value for ecgML of 90 ((70 + 90) / 2 = 80). Providing the parameters as requirements (directly after line 1 of figure 4.5 with type letter n) does not make sense, as only one option is available. Still, the chosen option should be followed by the final parameters using fixed values, and not ranges (line 3 and 4 of figure 4.5), so both hosts have the same parameters for the chosen standard.

4.3 Document type for sending actual medical data

The goal of the document type for sending medical data is to limit the amount of data transfer. When one only wants to send a single measurement (for example blood pressure), it is far from efficient to first negotiate a standard in which the data will be send, and afterwards transferring the data via this standard. For this reason, we want to be able to send simple measurements directly over the control protocol in short messages.

4.3.1 Media type for sending medical data

SIP allows for including MIME attachments, which we can define to be able to contain medical data. Standard SIP is not designed to exchange messages in the way we need them. There exists an extension to SIP that allows the exchange of messages, as described in appendix A.7 and [9]. While the goal of this extension is Instant Messaging, it can also be used for our needs. Messages are exchanged between two SIP endpoints, without the need to set up a session first. According to [9] the implementation must support the "text/plain" media type, but for our work we'll have to introduce a new media type, and not use "text/plain". The media type registration at IANA could be "application/medical".

4.3.2 Medical document definition

When comparing to MSDP, there is several similarities in the needs, with the difference that in the medical document we transmit actual values instead of negotiating for a certain standard. MSDP and *medical* will be closely linked, as they are both transferred over SIP by the same hosts. SDP provides a compact form to transfer data from one host to another. For these reasons, we will base the medical document on both SDP and MSDP.

The goal of the *medical* document is to limit the overhead caused by negotiating a transmission standard. The overhead is large in case a small data set needs to be transferred. For this reason, only small data sets will be covered by the *medical* document. For the moment, we'll only handle blood pressure, but other measurements should be easily included later. For example ECG data does not have a small data set, for this reason, the *medical* document will not support the transmission of ECG.

The *session description* of SDP can be partly reused. The *v* type letter (protocol version) must be *0*. The *o* type letter (origin) and *s* type letter (session name) are compulsory, while the *i* type letter (session information) is optional, like in SDP. The *u* (Uri), *e* (e-mail) and *p* (phone) type letter must not be used. The *c* type letter (connection information) must be omitted, as no connection will be set up. The *b* type letter (bandwidth information) must be omitted, as in case of a single measurement, bandwidth is not relevant. The *k* type letter (encryption key) must be omitted, as the data is already in the *medical* document itself, including encryption keys in the actual document does not make sense.

The *time description* of SDP can be partly reused. The *t* type letter (time) should denote the time of the measurement. Normally the start and end time should be the same time, but in case it took a certain time to measure the value (for example, when measuring blood pressure, it takes about 20 seconds to measure it) the time frame in which the measurement was made may be given as well. The *r* type letter (repeat times) must be omitted, as for sending new measurements, new messages need to be exchanged. The *z* type letter (time zones) may be used.

The *media description* of SDP should be totally omitted. Instead, we'll include the measurements. The type letter *m* will be used to define a medical measurement. It is allowed to include several different measurements made at the same time, for example a blood pressure and temperature measurement. The receiving host either confirms the receipt with *200 (OK)*, or replies with a suitable *4xx* or *5xx* response. In case several measurements are combined into one message, only one response is possible. Only in case all measurements are accepted, a *200 (OK)* response is send, in all other cases, even if all but one measurement are accepted, a suitable *4xx* or *5xx* response has to be send. It is impossible to separately accept/reject measurements if they were combined into one single message, as the response is kept as short as possible and does not allow for separately confirming measurements. The syntax for *m* is:

```
m EQUAL <content> HCOLON <value>
<content> = "bloodpressure" / token  ; 'token' should be pre-defined
<value> = token  ; 'token' depending on <content>
```

The parameter `<content>` defines the kind of medical data that has been measured. Depending on `<content>` , `<value>` is defined. For blood pressure, both systolic and diastolic

pressures are measured in millimetres of mercury (mm Hg). The values are denoted separated by a dash, for example 120-75, meaning 120 mm Hg systolic and 75 mm Hg diastolic.

4.3.3 Protocol messages

The set of protocol messages is very limited, because there is no negotiation or session set-up. Every medical messages needs to be confirmed by the receipt, so the sender knows that the transmission of the message succeeded. The confirmation can be done with standard SIP messages. For a successful transmission, including acceptance of the medical data, a *200 OK* should be returned. In case of an error, the most suitable response from the *4xx* or *5xx* class should be used.

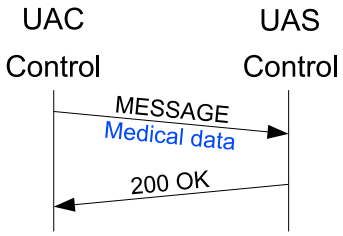

Figure 4.6: Message transactions for the exchange of medical data

Figure 4.6 shows the message transactions for the exchange of medical data. No session needs to be set-up first, and only the control plane is involved. First the UAC sends a *MESSAGE*, including a body of the type "application/medical", which contains the medical measurements, including timing information. After acceptance of the *MESSAGE*, the UAS returns a *200 OK* reply.

4.3.4 Security considerations

Because medical data will be transferred by SIP when using this extension, TLS for setting up connections and authentication must be used. As we define a direct connection between the UAC and the UAS, without the use of proxies, this provides us with end-to-end security.

4.3.5 Example

```
1  m=bloodpressure:120-75
```

Figure 4.7: Example of medical document (medical description only)

Figure 4.7 shows the medical description of a medical document. The *session description* and *time description* have been omitted. Only one measurement is transferred, in this case a blood pressure, with 120 mm Hg systolic and 75 mm Hg diastolic.

Chapter 5

Implementation

In this chapter we describe a design refinement. The control plane of this prototype will be called Medical Data Control Protocol (MDCP). We specify the finite state machine of MDCP and the interfaces provided by MDCP as well as the interfaces used by MDCP.

5.1 MDCP design

In chapter 4 we have described a new protocol for negotiating medical data exchange. In this section we'll further detail the design, for example by defining interfaces and state diagrams.

5.1.1 SIP implementation

Our design is build on top of the SIP protocol. As there exist reference implementations of this protocol, we will base our implementation on it. We also prefer to use Java for implementation, as the language is wide spread. JAIN-SIP is such an implementation. More information about this implementation as well as a download is available in [2] and [3].

5.1.2 SDP implementation

The messages defined in the previous chapter are based on SDP. This means that the easiest solution is adjusting an existing implementation of SDP. JAIN-SIP does also implement SDP, though at first sight it didn't look like the easiest one to use. For this reason, we have adjusted the code as provided by jSDP ([4]). The parts we don't need are removed, and other parts are added to support MSDP and the medical document type. Some existing classes and methods of jSDP are adjusted to meet our needs.

5.1.3 Components

The prototype that we'll build for proving our concept will consist of various components. First of all, there will be a user interface to operate the computer logic. The user interface communicates both with the MDCP and data plane. These two also communicate between each other. The MDCP will be build on top of SIP, which itself will run on top of the TCP transport layer. The data plane will communicate via different application layer protocols, or directly with a transport layer protocol, depending on the medical standard used for exchanging the data or on the choice made during negotiating whenever the transport protocol

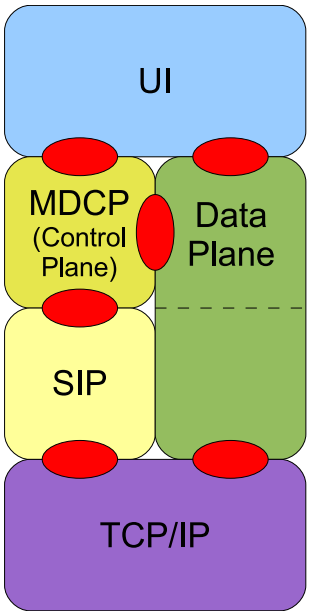

Figure 5.1: The different protocol elements that are important for us. The red ellipses denote interaction points between the different components.

is left open by the medical standard. Figure 5.1 shows these different protocol elements as protocol stack and their interaction points.

For us, especially the MDCP and its interaction with neighbouring components is important. This means we have to define interfaces for three interaction points:

- User interface
- SIP layer
- Data plane

All three interaction points are bi-directional.

In our prototype the user interface is operated by a human being. In reality, this is often not the case. A patient will often carry a device that automatically sends measurements due to a trigger, for example a timer, or a notified change in measurements received from the connected sensors. A central storage server will automatically store received measurements from patients, and send it to a medical person on request. The medical person would either get measurements on his screen automatically, or requests the data of a certain patient. Only in the last case a direct interaction with a human being results in the exchange of data, in other cases, an automated system takes care of it.

For the data plane, there is also a data plane factory to create a data plane instance for every connection. The data plane factory can also be queried for its capabilities. Communication with the data plane factory is only initiated by the MDCP.

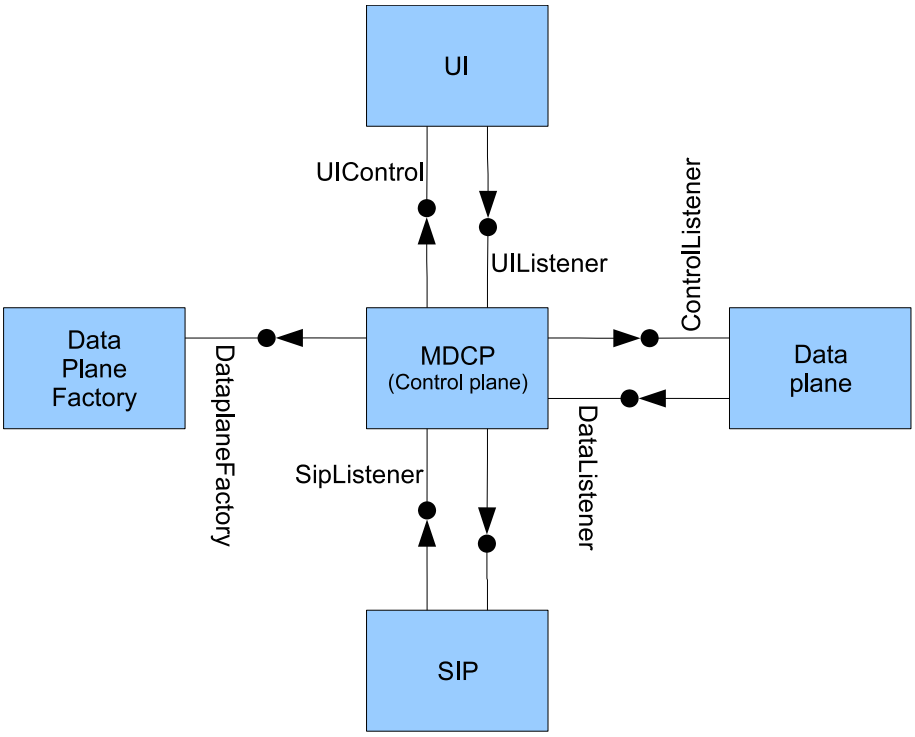

Figure 5.2: The UML diagram including interfaces.

Figure 5.2 shows the UML diagram, including the interfaces. Our prototype will focus on MDCP, but of course also the other parts need dummy implementations to show the proper functioning. The interfaces take care of the communication between the MDCP and the other components. They are defined in section 5.1.5.

5.1.4 States

Our design is based on SIP. The state diagrams of SIP are shown in figure 5-8 of [23]. SIP defines different transaction diagrams for *INVITE* and *non-INVITE* requests. In our implementation we'll have to communicate with SIP following these transaction diagrams.

Figure 5.3 shows the state diagram of the MDCP. The MDCP communicates with the user, the data plane as well as with the SIP layer. All the communication with the user is in black, the communication with SIP is in blue and the communication with the data plane is in dark yellow. In the first line of a state transition (in italic) an event is shown that triggers zero or more actions, which are shown in the following lines. The top-right path, passing the *Ringing* state and that starts with receiving *INVITE* has twice *configure data plane* as action on transition to *Connected* state, but only one is executed. This depends on whether the MSDP offer is included in the *INVITE* request, or in the *200* answer. In case the MSDP offer is included in the *INVITE*, the first *configure data plane* is executed, otherwise the second. In the *connected* state, the data receiver sends a *MESSAGE* when the data plane finishes configuration, while the sender waits for *MESSAGE* before it starts sending data. Which host is the data receiver is independent from where the transaction started.

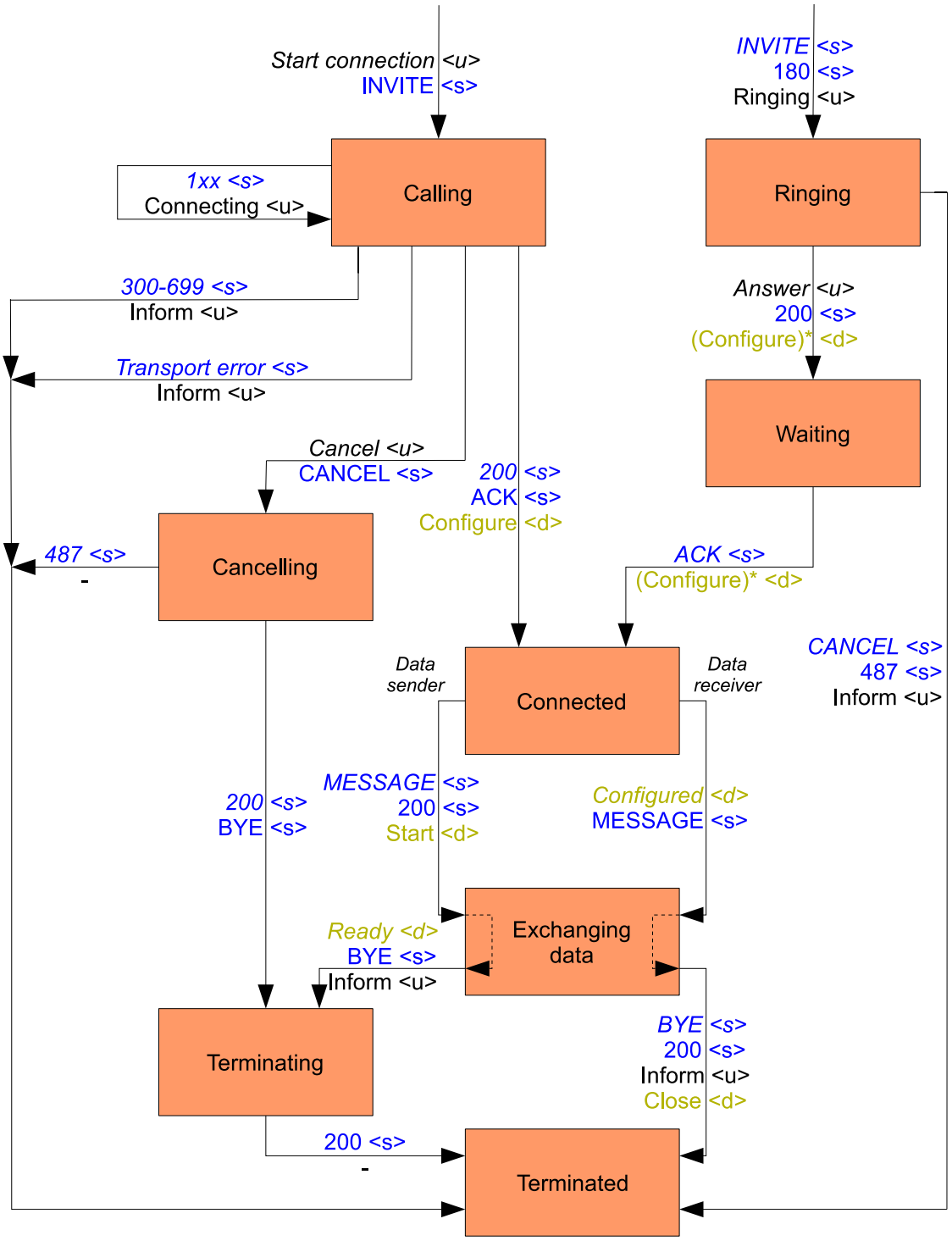

Figure 5.3: The simplified MDCP state diagram. In blue and marked with "<s>" the communication with the SIP layer, in dark yellow and marked with "<d>" communication with the data plane and in black and marked with "<u>" communication with the user. The first line (in italic) shows an event, the other lines actions. The action marked with an asterisk is only executed once, either before or after the *Waiting* state.

The diagram doesn't show all transactions that are possible. At any moment, it is possible that a host withdraws from the session. In the *Ringing* state this is done by responding with a *488 (Not acceptable here)* answer, in the other states this is done by sending a *BYE* request, after which a transition to the *Terminating* state is made. Whenever a host receives a *BYE*, it should respond with a *200 (OK)* and go directly to the *Terminated* state, while stopping all communications. It should inform the data plane to cancel the transmission, and unload everything related to the session.

5.1.5 Interfaces

Seen from MDCP, there are four interaction points. All these interaction points, except the one with the data plane factory, are bi-directional. This means we have to define in total seven interfaces.

User interface to MDCP

The user interface takes care of the communication between the user and the MDCP. The user can control the behaviour of the MDCP via this interface, and the MDCP can show information to the user. When looking at figure 5.3 one can see only three methods that are invoked by a user: starting a connection, answering an incoming connection and cancelling a connection. The interface is shown in figure 5.4.

```
1  public interface UIListener {
2     CallIdHeader startConnection(String destination, Requirement[] parameters,
3           boolean isSender, boolean sendMsdpOffer);
4     boolean answerCall(CallIdHeader callID, Requirement[] parameters);
5     boolean cancelCall(CallIdHeader callID);
6     boolean sendMeasurement(String destination, String content, String value);
7     boolean sendMeasurement(String destination, String content, String value,
8           Date start);
9     boolean sendMeasurement(String destination, String content, String value,
10          Date start, Date end);
11    void close();
12 }
```

Figure 5.4: The UIListener interface

To start a connection (lines 2-3 of figure 5.4) the following parameters should be provided: the destination, requirements for the connection, if the initiator is sender or receiver of the data, and if a MSDP offer should be send in the initial message (*INVITE*). The destination is a SIP URI. The requirements are given in an array, where the array might be empty in case no requirements apply. The controller must know if the initiator is the data sender or receiver, as this defines if from the *Connected* state the left or right path will be taken (see figure 5.3). The last parameter is used to control which host will make the decision: the initiator or the answering host. As this is decided by the initiating host only, the idea is that if the host is low in resources, it should include an MSDP offer, so the other host makes the final decision. If the host has enough resources it should not include the MSDP offer, so in the *200 (OK)* response, it will receive the offer, and make the decision. With this information, the MDCP can set up the connection. The session identification number (*Call-ID*) is returned in case

the session is being set up, or a *null* value in case a session cannot be set up. The type of the return value is CallIDHeader.

To answer a call (line 4 of figure 5.4) the session identification number (*Call-ID*) has to be provided. This number is reported to the user when he is informed about an incoming call. Also the requirements are given in an array, where the array might be empty in case no requirements apply. The return value is a boolean that reports true in case the incoming call was answered, and false otherwise.

A call can be cancelled (or rejected; line 5 of figure 5.4) at any moment. To cancel a call the session identification number (*Call-ID*) has to be provided. The return value is a boolean that reports true in case the call will be cancelled, and false otherwise.

Not mentioned in figure 5.3, but also part of the protocol is the exchange of measurements. To exchange measurements (lines 6-10 of figure 5.4), a destination, content description, value, and optionally a starting and ending time of the measurement are needed. The destination is a SIP URI. The content description is a pre-defined token, which also defines how the value should be understood. Optionally, the starting and ending time stamp of the measurement can be given. If both are omitted, the current time is used for both, if only start is given, this value is used both for start and end.

The *close* method (line 11 of figure 5.4) is used to make MDCP shut down properly. This will close the SIP sockets, and cleanly close all other resources, including open calls.

MDCP to user interface

The MDCP has to report certain events to the user. This is done via the user interface. There are several messages that need to be reported: incoming connection, finished connected, incoming measurements, debug messages, informative messages and error messages. The interface is shown in figure 5.5.

```
1 public interface UIControl {
2     void notifyIncomingCall(CallIdHeader callID, String caller);
3     void notifyFinishedCall(CallIdHeader callID);
4     boolean notifyMeasurement(String caller, String content, String value);
5     void notifyDebug(CallIdHeader callID, String msg);
6     void notifyInfo(CallIdHeader callID, String msg);
7     void notifyError(CallIdHeader callID, String msg);
8 }
```

Figure 5.5: The UIControl interface

An incoming connection (line 2 of figure 5.5) is communicated to the user interface with the session identification (*Call-ID*) and the caller identification (*From*). This provides information to the user who is trying to contact him, and also an identification, which is used to answer or cancel/reject that call, and makes it possible to handle several calls in parallel. There is no return value.

When a call finishes (line 3 of figure 5.5), either because it ended normally, or because it was

cancelled, the user interface is informed. As parameter the session identification (*Call-ID*) is provided, while it has no return value.

Measurements can also be received (line 4 of figure 5.5), and are reported to the user interface. Although several measurements can be combined into one SIP message, they will be reported to the user interface one by one. The caller identification (*From*), the content description and the value are provided. When the measurement is accepted, the true value is returned, if measurement sending is not supported, or the measurement is not accepted, false must be returned, which causes the MDCP to send a negative acknowledgement. When several measurements are combined into one message, and one is rejected, all measurements will be marked as rejected in the response. This is because the reply is kept as short as possible, and only tells if all measurements are accepted, or all rejected, and not per measurement. We use a reliable transport protocol (TCP, preferably over TLS), meaning there is no need to repeat the received message. This does prevent though that only part of the received measurements is accepted.

The debug (line 5 of figure 5.5), informative (line 6 of figure 5.5) as well as the error (line 7 of figure 5.5) message carry two parameters, namely the session identification (*Call-ID*) and a text message. The *Call-ID* must be *null* in case the message is not related to a certain call. There is no return value.

SIP layer to MDCP

The interface between the SIP layer and the MDCP has already been defined by the JAIN-SIP API (see [2]). We'll describe the interface we have to implement in the MDCP here. There are in total six methods in the *SipListener* interface. The interface is shown in figure 5.6.

```
1 public interface SipListener extends java.util.EventListener {
2     void processRequest(RequestEvent evt);
3     void processResponse(ResponseEvent evt);
4     void processTimeout(TimeoutEvent evt);
5     void processIOException(IOExceptionEvent evt);
6     void processTransactionTerminated(TransactionTerminatedEvent evt);
7     void processDialogTerminated(DialogTerminatedEvent evt);
8 }
```

Figure 5.6: The SipListener interface

The method processRequest (line 2 of figure 5.6) is called when a new request arrives, with the event as parameter. The method processResponse (line 3 of figure 5.6) is called when a response arrives, with the event as parameter. When a time-out is noticed by the SIP layer, the method processTimeout (line 4 of figure 5.6) is called, with the event as parameter. In case of an IO exception the method processIOException (line 5 of figure 5.6) is called, with the event that resulted in this error as parameter. When a transaction is terminated, the processTransactionTerminated (line 6 of figure 5.6) method is called, with the event as parameter. In case a dialog is terminated, this is notified by calling the processDialogTerminated (line 7 of figure 5.6) method, which also takes the event causing it as parameter. None of these methods have a return value. The difference between dialogs and transactions is explained in appendix A.2.

MDCP to SIP layer

The JAIN-SIP API defines the methods that are available. These are too many to describe here. For a detailed overview you can read the API itself (see [2]).

MDCP to data plane factory

The MDCP needs to know about the capabilities of the data plane. For this reason we introduce a data plane factory, which is responsible for informing the MDCP about its capabilities, as well as for creating new data planes. The interface is shown in figure 5.7.

```
1 public interface DataplaneFactory {
2    public MsdpSessionDescription getMsdpCapabilities();
3    public ControlListener createControlListener(DataListener dl, CallIdHeader callId,
4          MsdpSessionDescription msdp, boolean isSender);
5 }
```

Figure 5.7: The DataplaneFactory interface

To know about the capabilities of the data plane (line 2 of figure 5.7), the factory returns a MSDP session description. This includes all requirements and preferences of the factory, as well as all different standards it supports, with its parameters.

The factory is also responsible for creating new data planes (which implement the *ControlListener* interface; lines 3-4 of figure 5.7). To create a new data plane, the responsible controller, the session identification (*Call-ID*), a MSDP description containing the configuration parameters, as well as if the data plane will send or receive data have to be given. With these parameters, the factory should decide which data plane implementation to initiate. The new data plane created is returned.

Data plane to MDCP

The communication from the data plane to the MDCP is very limited, and consists only of three messages: configuration succeeded, configuration error and transfer finished. For every session a new data plane will be created, which has a 1-1 relationship to a sub controller in the MDCP (a MDCP sub controller is created for every call, and is explained further in section 5.2.3). Because of this, there is no need for an identification, like the session identification, to be included. The interface is shown in figure 5.8.

```
1 public interface DataListener {
2    boolean configurationSucceeded();
3    void configurationError(String msg);
4    void transferFinished();
5 }
```

Figure 5.8: The DataListener interface

The configuration succeeded message (line 2 of figure 5.8) is used by the data plane to notify the MDCP it is configured, and ready to exchange data. The receiver communicates this

via the MDCP to the sender, so the sender knows transmission can start. The sender will, once both the local data plane as well as the remote data plane have confirmed they are configured, inform the data plane the transmission can start. The return value is true in case this notification is expected to arrive (MDCP in the *waiting* or *connected* state), and false in all other case. In case false is returned, the data plane should free the reserved resources and not expect incoming data.

The configuration error message (line 3 of figure 5.8) is used in case it is impossible to configure the data plane, or in case the data transmission fails. As parameter a textual explanation of the error is given. After this message, the data plane should free the resources involved in this transfer. There is no return value.

The sending data plane should inform the MDCP when the transmission is finished (line 4 of figure 5.8). That means that the data has been transferred, and is confirmed in the data plane by the receiver. After this message, the data plane should free the resources involved in this transfer. There is no return value.

MDCP to data plane

The MDCP can send three different messages to the data plane: configure, start sending and close connection. The interface is shown in figure 5.9.

```
1 public interface ControlListener {
2     void configure();
3     boolean startTransfer();
4     boolean closeTransfer();
5 }
```

Figure 5.9: The ControlListener interface

The configure message (line 2 of figure 5.9) is used to inform the data plane it should prepare for a data transmission. This means opening ports, and inserting filters. The configuration parameters are already included in the constructor. The MDCP must be informed when the configuration is done. The data plane should return control immediately, and not wait until configuration succeeded. A separate message is used for this.

In case the host is the sender, it should wait for the start message (line 3 of figure 5.9) from the MDCP before starting the actual data transfer. The return value is true in case it can start sending data, or false otherwise. In case transfer fails, this should be communicated to the MDCP by a separate message.

When the transmission is cancelled, or when the transfer is finished, a message to close the connection (line 4 of figure 5.9) is send. When this message is received, a possible transmission should be stopped, and the resources related to the session should be freed. The return value is a boolean which is true in case the session can be closed, or false otherwise.

5.2 Packages

Our prototype of the protocol contains several Java classes. These classes are grouped in packages. In table 5.1 an overview of the packages and a short description is given. In the remainder of this section, they will be explained in more detail.

Package	Description
core	The interfaces implemented in other packages
core.sdp	Basic SDP implementation, with the used parts of the session and time description
core.sdp.medical	The medical description, which extends basic SDP
core.sdp.msdp	The MSDP description, which extends basic SDP
core.sdp.util	Supporting classes used by SDP
dataplane	The data plane implementation
gui	A graphical user interface implementation
mdcp	The MDCP implementation
ui	The user interface implementation

Table 5.1: Package structure

5.2.1 MSDP messages (package 'core.sdp.msdp')

In section 4.2 we have defined a new message type, namely application/msdp. Several classes are needed to represent the extension to SDP. This package also extends some of the classes defined in 'core.sdp' to make them able to work with the MSDP message type. The factory is also able to merge MSDP messages with requirements, as well as decide the optimal standard to exchange the data.

When merging a MSDP message with the requirements, first the general requirements section of the MSDP will be merged with the supplied requirements. Afterwards all options (different standards) will be checked if they still are possible with the new requirements set. In this way, only the options that match the requirements have to be transferred. It is possible that no single option matches the requirements. In this case the whole medical description is removed. A method using the merge method should check if there is still a medical description left after merging, if not, it should stop the negotiating process, as no standard is available with the given requirements.

The decision method will first merge the general requirements of the sender MSDP with the MSDP of the receiver, and afterwards do the same the other way around. It might happen that no option at either the sender or receiver matches all requirements, after which the deciding method will return a MSDP without medical description. After the previous steps, a list of common options (standards) is made, and the preference values are compared. The common option with the highest sum of the square root of the preference value of the sender and the square root of the preference value of the receiver will be taken as conclusion. The concluded preference value will be the square of this highest sum:

$$concluded\_preference = \left(\frac{\sqrt{preference\_sender} + \sqrt{preference\_receiver}}{2} \right)^2$$

The reason of using the square root for the calculation is to prefer preferences with a value closer together above preferences where the values are further apart, while having the same average. Consider for example a matching standard, where the preference for the sender is 10, and the preference of the receiver is 90. The average preference will be 50. This is also the case when both sender and receiver have preference 50. In general, using a very low preference value is not good, as it means the standard is far from preferred, and should only be used as a last resort. Using the formula above makes sure that the standard where both sender and receiver have preference 50 is chosen:

$$\left(\frac{\sqrt{50}+\sqrt{50}}{2}\right)^2 = 50$$

$$\left(\frac{\sqrt{10}+\sqrt{90}}{2}\right)^2 = 40$$

As can be seen, when both have preference 50, the concluded preference is higher than when one has 10 and the other 90.

5.2.2 Data plane (package 'dataplane')

The goal of the prototype we implement is to prove the concept we described in chapter 4. We will not exchange actual medical data, but only pretend we do so. This will give us an easy way to test if the ideas we presented work, without having to find implementations of the different medical protocols.

First of all, the data plane factory presents some random capabilities, and for different hosts, different capabilities should be presented, to simulate reality. Also the data plane factory should be able to create a dummy data plane which imitates a real data plane. This data plane does not have the capability to exchange data, but simulates this by using random timers. First of all, when it is requested to configure, it starts a timer with a value between 2 and 8 seconds. When the timer fires, it informs the MDCP configuring succeeded. This randomness matches reality: under different system loads, configuration can take longer or shorter. The values are not realistic, but chosen to slow down the whole process, and to allow human interaction during the process (like cancelling an ongoing data transfer).

Also the request to start the data transfer results in the starting of a timer with a value between 2 and 8 seconds. When the timer fires, the MDCP is informed the transfer finished. Also this randomness matches reality: due to network load, or different amounts of data to be exchanged, the transfer can take longer or shorter. Monitoring a patient for longer time (streaming) is possible. In such cases a connection exists for longer time. Unlimited duration could create problems though: due to a host disappearing from the network, a connection might not be properly closed, meaning a connection is kept open forever. To prevent this, a limited time should be set (could be in the range of hours). To support connections existing longer than the time limit, keep-alive messages must be introduced so a connection is not closed due to exceeding the time limit. For now, such functionality is not included, and we rely on SIP for this (SIP does allow for longer-existing connections).

The simulation of the behaviour of configuration and data exchange results in a very simple implementation. On the other hand, it does react as a real implementation, which means it can be used for testing purpose.

5.2.3 MDCP (package 'mdcp')

The MDCP consists of 2 classes: the main controller and the sub controllers. The main controller is responsible for creating the SIP sockets when it is started. It also creates a data plane factory, which is responsible for communication with the data plane. The main controller is the access point for the user interface, as well for all incoming SIP messages. When a connection request is received from the user interface, or an *INVITE* request via SIP, a new sub controller is created to handle this session. Subsequent requests related to this session are forwarded to the responsible sub controller. For every session a new sub controller is created. The exchange of single measurements is also the responsibility of the main controller. Only one main controller is created during execution for an application (listening on one port for incoming connections).

The sub controller is responsible for a single session. Commands from the user interface and incoming requests and responses from SIP are forwarded to the sub controller from the main controller. Handling them is the responsibility of the sub controller. The sub controller also maintains the state of the MDCP dialogue, as defined in figure 5.3. Whenever a sub controller is in the *Terminated* state, the garbage collector of the main controller may delete it. At the latest when the sub controller is in the *Connected* state, a 1-to-1 relationship with the data plane is established. For every session, the data plane factory is requested to create a new instance of a data plane. The sub controller does communicate directly to the user interface, while answers come back via the main controller, as the user interface only has an interface towards the main controller, and not to the sub controllers.

5.2.4 User interface (package 'ui')

To control the behaviour of MDCP, we need a user interface. In reality this could be a system that automatically answers or initiates requests. To prove the concept, we will manually control the MDCP. To keep things simple, we will use a simple text interface. It might look less clear than a graphical interface, but speeds up development, and can be run in a non-graphical environment, like command line.

The user interface supports the following commands:

- `exit`: close the program
- `help`: display a short message with all available commands
- `connect <destination> <send|recv> [<msdp>]`: start a new call with `<destination>` (a SIP URI) as either sender (`send`) or receiver (`recv`) and optional some requirements for the connection (via the pre-defined requirements in the user interface as shown with the `msdp` command).
- `answer <call> [<msdp>]`: answer an incoming call, where `<call>` is replaced by the call number as internally used in the user interface, and optional some requirements for

the connection (via the pre-defined requirements in the user interface as shown with the `msdp` command).

- `cancel <call>`: cancel an active call, where `<call>` is replaced by the call number as internally used in the user interface. This command is also used to reject an incoming call.
- `send <destination> <content> <value>`: send a single measurement to `<destination>` (a SIP URI). `<content>` defines the content type (for example blood pressure), and `<value>` the value. The content type defines the format of the value.
- `calls`: show a list of active calls, including the internal call number used in the user interface.
- `msdp`: show available MSDP messages, which can be used in the `connect` and `answer` commands.
- `msdp full`: show available MSDP messages including their full body.

5.2.5 Other packages

Interfaces (package 'core')

This package defines the interfaces as implemented by the data plane, MDCP and user interface. The are defined in section 5.1.5.

Basic SDP (package 'core.sdp')

This package is mainly a copy of a part of the jSDP (see [4]) implementation of SDP. Only the parts we use are copied, and some parts have been adjusted to better match our needs and some unused parts have been removed. The adjustments are related to the removing of the variables we don't use in our implementation and adjusting the constructors and parsers that make use of these variables.

Medical messages (package 'core.sdp.medical')

In section 4.3 we have defined a new message type, namely application/medical. This message type needs to be represented in code. This package defines all classes that are not available in basic SDP, and also extends some of the classes to make them able to work with the medical message type.

SDP support classes (package 'core.sdp.util')

This package defines some data structures used by the basic SDP (implementation taken from jSDP; see [4]).

Graphical user interface (package 'gui')

This package provides a graphical user interface which can replace the user interface (package 'ui'). Functionality is the same as in 'ui', but the interface is graphical instead of command line based.

Chapter 6

Evaluation

In this chapter we describe the evaluation of our prototype. First we evaluate the prototype itself, and in the second part we validate the requirements set in section 3.2.

6.1 Evaluation of the prototype

In this section we describe several tests of the prototype. First we test the functioning of exchanging SIP messages for the overall control of the data exchange. After that, we look into detail in the exchanged MSDP message, to see if indeed the optimal solution is found. We finish with checking the exchange of measurements.

6.1.1 Exchanging measurements

The designed protocol allows for exchanging medical measurements. Our prototype implements this part as well. Figure 6.1 shows the exchanged SIP messages. One can see that

```
1  > MESSAGE sip:wim@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
2  > Call-ID: 8310222386edfa394db72d0a79cff3b3@192.168.48.128
3  > CSeq: 1 MESSAGE
4  > From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
5  > To: "wim" <sip:wim@willempie.kuipnet.void:3031>
6  > Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bK7d169d12e2f8054465de770ef3cc0d7d;received=192.168.48.128;rport=3030
7  > Max-Forwards: 0
8  > Content-Type: application/medical
9  > Content-Length: 148
10 >
11 > v=0
12 > o=wim 3452845863 3452845863 IN IP4 192.168.48.128
13 > s=Medical data transmission
14 > i=Medical data
15 > t=3452845862 3452845862
16 > m=bloodpressure:125-80
17
18 < SIP/2.0 200 OK
19 < Call-ID: 8310222386edfa394db72d0a79cff3b3@192.168.48.128
20 < CSeq: 1 MESSAGE
21 < From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
22 < To: "wim" <sip:wim@willempie.kuipnet.void:3031>;tag=Medical_v1.0
23 < Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bK7d169d12e2f8054465de770ef3cc0d7d;received=192.168.48.128;rport=3030
24 < Content-Length: 0
```

Figure 6.1: Sending a measurement

the measurement is put in the medical body of the SIP request. The measurement is correctly shown at the receiving host, and confirmed with *200 OK*. Exchanging a measurement in the other direction shows exactly the same result, thus we can conclude that exchanging measurements works.

6.1.2 Finding optimal data protocol

To show the correct functioning of the decision mechanism, we will do some connection set-ups, using different requirements. This should result in different optimal solutions. First, we see the default offers a host would send without any requirements. Figure 6.2 shows the default MSDP offer of host 1, while figure 6.3 shows the default MSDP offer of host 2.

```
1  v=0
2  o=msdp-test 2890844526 2890842807 IN IP4 192.168.48.1
3  s=MSDP Sample message
4  i=MSDP Sample message in the DataplaneFactory
5  c=IN IP4 192.168.48.128/127
6  t=2873397496 2873404696
7  m=packet ecg
8  n=bitrate:10-20
9  n=resolution:8-16/4
10 q=100 SCP-ECG http 80
11 n=resolution:12
12 n=delay:60
13 q=70 ecgML http 8080
14 n=bitrate:18
15 q=10 HL7 ftp 21
16 n=bitrate:10
17 n=resolution:12-14
18 n=jitter:50
```

Figure 6.2: Default MSDP offer host 1

```
1  v=0
2  o=msdp-test 2890844526 2890842807 IN IP4 192.168.48.1
3  s=MSDP Sample message
4  i=MSDP Sample message in the DataplaneFactory
5  c=IN IP4 192.168.48.128/127
6  t=2873397496 2873404696
7  m=packet ecg
8  n=bitrate:5-20 optional
9  n=resolution:4-16/4 optional
10 q=80 SCP-ECG http 2080
11 n=resolution:10
12 n=delay:20
13 q=100 ecgML http 3080
14 n=bitrate:14
15 n=resolution:4-14
16 n=jitter:10
17 q=50 HL7 ftp 3021
18 n=bitrate:10
```

Figure 6.3: Default MSDP offer host 2

When connecting without any additional parameters, it means the optimum of figure 6.2 and figure 6.3 has to be found. To check this, we connect without any additional requirements, and observe the resulting MSDP response. This can be seen in figure 6.4. As result, the SCP-ECG standard is chosen with optimum value 89 (line 10 of figure 6.4).

Now we will send another offer from host 1, but apply as requirement 'n=resolution:14 optional'. Due to this, the SCP-ECG capabilities will vanish from the offer. This can be seen in figure 6.5. Host 2 composes the MSDP response, without applying additional requirements. As a result, ecgML is concluded, with optimum value 84 (line 10 of figure 6.6).

As third test, we send an offer from host 2, and apply as requirements 'n=bitrate:10' and 'n=delay:10 optional'. Due to this, only HL7 is left in the offer (line 11 of figure 6.7). Host 1 composes the MSDP response, without applying additional requirements. As a result, HL7 is concluded, with optimum value 26 (line 11 of figure 6.8).

```
1  v=0
2  o=msdp-test 2890844526 2890842807 IN IP4 192.168.48.1
3  s=MSDP Sample message
4  i=MSDP Sample message in the DataplaneFactory
5  c=IN IP4 192.168.48.128/127
6  t=2873397496 2873404696
7  m=packet ecg
8  n=bitrate:10-20 optional
9  n=resolution:8-16/4 optional
10 q=89 SCP-ECG http 2080
11 n=resolution:10
12 n=delay:20
```

Figure 6.4: Response when connecting without any additional requirements

```
1  v=0
2  o=msdp-test 2890844526 2890842807 IN IP4 192.168.48.1
3  s=MSDP Sample message
4  i=MSDP Sample message in the DataplaneFactory
5  c=IN IP4 192.168.48.128/127
6  t=2873397496 2873404696
7  m=packet ecg
8  n=bitrate:10-20
9  n=resolution:14 optional
10 q=70 ecgML http 8080
11 n=bitrate:18
12 q=10 HL7 ftp 21
13 n=bitrate:10
14 n=resolution:12-14
15 n=jitter:50
```

Figure 6.5: Offer from host 1 with requirement 'n=resolution:14 optional'

When looking at the three given test scenarios, one can see that the response always is as expected. Based on this, we can conclude that for the given test scenarios, the prototype behaves as expected.

6.1.3 Controlling data transfer

For the controlling of the data transfer we use SIP. There is also MSDP bodies included, but they are only used to exchange parameters of the data plane. In this section we'll discuss the four possible scenarios, and test them in the prototype. The four scenarios are composed by the initiator that can be either the sender or receiver, and the initiator can or can not include the MSDP offer in the invite.

```
1  v=0
2  o=msdp-test 2890844526 2890842807 IN IP4 192.168.48.1
3  s=MSDP Sample message
4  i=MSDP Sample message in the DataplaneFactory
5  c=IN IP4 192.168.48.128/127
6  t=2873397496 2873404696
7  m=packet ecg
8  n=bitrate:10-20 optional
9  n=resolution:14 optional
10 q=84 ecgML http 3080
11 n=bitrate:14
12 n=resolution:4-14
13 n=jitter:10
```

Figure 6.6: Response from host 2 on offer in figure 6.5

```
1  v=0
2  o=msdp-test 2890844526 2890842807 IN IP4 192.168.48.1
3  s=MSDP Sample message
4  i=MSDP Sample message in the DataplaneFactory
5  c=IN IP4 192.168.48.128/127
6  t=2873397496 2873404696
7  m=packet ecg
8  n=bitrate:10 optional
9  n=resolution:4-16/4 optional
10 n=delay:10 optional
11 q=50 HL7 ftp 3021
12 n=bitrate:10
```

Figure 6.7: Offer from host 2 with requirements 'n=bitrate:10' and 'n=delay:10 optional'

```
1  v=0
2  o=msdp-test 2890844526 2890842807 IN IP4 192.168.48.1
3  s=MSDP Sample message
4  i=MSDP Sample message in the DataplaneFactory
5  c=IN IP4 192.168.48.128/127
6  t=2873397496 2873404696
7  m=packet ecg
8  n=bitrate:10 optional
9  n=resolution:8-16/4 optional
10 n=delay:10 optional
11 q=26 HL7 ftp 3021
12 n=bitrate:10
```

Figure 6.8: Response from host 1 on offer in figure 6.7

Initiator sends data and sends offer

In this scenario the initiator includes a MSDP offer in the *INVITE*, and later on sends the data. The test output (SIP messages) is shown in figure 6.9, where the direction of the messages is shown by the arrows in front of it. We describe the scenario seen from the left host. In line 1 - 15 of figure 6.9 the *INVITE* is send, including the MSDP offer. First a *180 Ringing* is received (line 17 - 23 of figure 6.9), later a *200 OK* (line 25 - 38 of figure 6.9), including the MSDP response. Based on this the data plane is configured. The *200 OK* response is answered by sending *ACK* (line 40 - 47 of figure 6.9). The left host sends the data to the right host. Line 49 - 61 of figure 6.9 shows the *MESSAGE* we receive to inform us that the other host is ready to receive the data. We answer this with *200 OK* (line 63 - 69 of figure 6.9). When we finish sending the data, we notify the right host by sending *BYE* (line 71 - 78 of figure 6.9). This is answered with *200 OK* (line 80-86 of figure 6.9).

Initiator receives data and sends offer

In this scenario the initiator includes a MSDP offer in the *INVITE*, and later on receives the data. The test output (SIP messages) is shown in figure 6.10, where the direction of the messages is shown by the arrows in front of it. We describe the scenario seen from the left host. The first four messages (line 1 - 47 of figure 6.10) are exactly the same as in the case where the initiator sends data and sends the offer. As the direction of the data is opposite, the last four messages (line 49 - 86 of figure 6.10) are the same as in the case where the initiator sends data and sends the offer, but are send in the other direction.

Initiator sends data and receives offer

In this scenario the initiator doesn't include a MSDP offer in the *INVITE*, and later on sends the data. The test output (SIP messages) is shown in figure 6.11, where the direction of the

messages is shown by the arrows in front of it. We describe the scenario seen from the left host. In line 1 - 12 of figure 6.11 the *INVITE* is send. First a *180 Ringing* is received (line 14 - 20 of figure 6.11), later a *200 OK* (line 22 - 35 of figure 6.11), including the MSDP offer. The *200 OK* response is answered by sending *ACK* (line 37 - 47 of figure 6.11), including the MSDP response. Based on this the data plane is configured. The last four messages (line 49 - 86 of figure 6.11) are again the same as in the scenario where the initiator sends data and sends the offer.

Initiator receives data and receives offer

In this scenario the initiator doesn't include a MSDP offer in the *INVITE*, and later on receives the data. The test output (SIP messages) is shown in figure 6.12, where the direction of the messages is shown by the arrows in front of it. We describe the scenario seen from the left host. The first four messages (line 1 - 47 of figure 6.12) are exactly the same as in the case where the initiator sends data and receives the offer. As the direction of the data is opposite, the last four messages (line 49 - 86 of figure 6.12) are the same as in the case where the initiator sends data and receives the offer, but are send in the other direction.

Conclusion

The four tested scenarios all show the exchange of the messages that would be exchanged in such scenario. Next to that, the user interface tells the transfer is finished after sending or receiving of *BYE*. This means that in case of a successful data transfer, the initial state is restored. The controlling of the data transfer works as expected.

6.2 Validation of requirements

In section 3.2 we listed several requirements for the control protocol. Table 4.1 lists the summary of the comparison. Many requirements were already fulfilled by SIP, some where not. In this section we will discuss the requirements that were not fully fulfilled yet.

6.2.1 Low overhead

Because we choose to work with SIP, we have to exchange messages in plain text, creating a certain overhead. Also the message bodies containing the negotiating parameters are send in plain text, namely a format closely linked to SDP, which is also a plain text protocol. The choice for using SIP resulted in a certain overhead that can not be solved. The amount of data exchanged is still reasonable. The whole control session requires seven messages, although eight messages might be exchanged, where the additional message is the *180 (Ringing)* response. It is not required in a connection set-up, but automatically send because the user interface needs too long to answer a connection request (SIP requires a response to an *INVITE* within 200 ms). In case there is no involvement of a human on accepting an incoming connection request, and a system automatically accepts an incoming connection within 200 ms, the *180 (Ringing)* response may be omitted.

6.2.2 Different protocol options of the same standard should be supported

As SDP did not include support for negotiating medical data protocols, this requirement was not fulfilled by SIP. The design of the MSDP messages does allow for this. Next to the standard, optionally the transport can be given, as well as an unlimited amount of parameters (to be defined per standard). This definition allows for supporting different protocol options of the same standard.

6.2.3 The control protocol should be extensible with new data protocols

The MSDP implementation does not limit the used standards, although some are included in the code. This means that the controller can support any kind of standard. The data plane should be able to understand the data protocol. This means that the data plane needs to be extended to support new protocols. As there are defined interfaces for the data plane, and a factory model is used, extending the data plane to support new data protocols can be easily achieved.

6.2.4 Support to transfer the actual medical data for small data sets

The newly defined application/medical message type allows for exchanging medical data. The SIP extension of the *MESSAGE* method is required for this, but as it is an existing extension, it can easily be used.

6.2.5 Secure

Our prototype does not implement any security mechanisms. To make a prototype that shows the correctness of the design, this was not required. Still, SIP allows the use of authentication, integrity and confidentiality, which should easily be implemented into the prototype. For confidentiality and integrity TLS can be used. As this only allows for hop-to-hop security, the use of (forwarding) proxies must be forbidden. This is achieved by setting the *MaxForwards* header to 0.

6.2.6 Summary

The low overhead requirement is not totally met, as we use plain text and not binary encoding. The amount of messages required by the control protocol are only 7 or 8. MSDP allows for supporting different protocol options of the same standard. In MSDP the data protocols are not fixed, and due to the use of a factory adding extra data protocols is possible. Actual medical data can be transferred using the control protocol. Extensions for SIP exist that can provide the required security.

As can be seen, all requirements that were not fulfilled by basic SIP, are fulfilled after adding our extensions. This means that our design is a solution for the described problem.

```
> INVITE sip:wim@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
> Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
> CSeq: 1 INVITE
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0;isSender=true
> To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>
> Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bKf1daffc2343da7e98f728b7cb8e38a7d;received=192.168.48.128;rport=3030
> Max-Forwards: 0
> Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>
> Require: X-Medical
> Supported: X-Medical
> Accept: application/msdp,application/medical,text/plain
> Content-Type: application/msdp
> Content-Length: 307
>
> [MSDP offer not shown]

< SIP/2.0 180 Ringing
< Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
< CSeq: 1 INVITE
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0;isSender=true
< To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bKf1daffc2343da7e98f728b7cb8e38a7d;received=192.168.48.128;rport=3030
< Content-Length: 0

< SIP/2.0 200 OK
< Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
< CSeq: 1 INVITE
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0;isSender=true
< To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bKf1daffc2343da7e98f728b7cb8e38a7d;received=192.168.48.128;rport=3030
< Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>
< Require: X-Medical
< Supported: X-Medical
< Accept: application/msdp,application/medical,text/plain
< Content-Type: application/msdp
< Content-Length: 348
<
< [MSDP response not shown]

> ACK sip:sip-test@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
> Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
> CSeq: 1 ACK
> Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bKb2f9e2b7dff714a067fd40d449b906b1;received=192.168.48.128;rport=3030
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> Max-Forwards: 0
> Content-Length: 0

< MESSAGE sip:sip-test@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
< Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bK41249b0990e38dd2f0572de9aa4190c4;rport=3031
< CSeq: 1 MESSAGE
< Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
< From: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< Supported: X-Medical
< Accept: application/msdp,application/medical,text/plain
< Max-Forwards: 0
< Content-Type: text/plain
< Content-Length: 16
<
< ready to receive

> SIP/2.0 200 OK
> Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bK41249b0990e38dd2f0572de9aa4190c4;rport=3031
> CSeq: 1 MESSAGE
> Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
> From: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> Content-Length: 0

> BYE sip:sip-test@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
> Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bK0859d7a2dc6cf6a2ca89a0804ba90c7e;rport=3030
> CSeq: 2 BYE
> Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> Max-Forwards: 0
> Content-Length: 0

< SIP/2.0 200 OK
< Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bK0859d7a2dc6cf6a2ca89a0804ba90c7e;rport=3030
< CSeq: 2 BYE
< Call-ID: e9a1f8bcc2de4c9dc9ad2804c1e01fde@192.168.48.128
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< Content-Length: 0
```

Figure 6.9: Initiator sends data and sends offer

```
> INVITE sip:wim@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
> Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
> CSeq: 1 INVITE
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0;isSender=false
> To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>
> Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bK3f81fffc38e9ca2fbee2c7a9b0f53ec3;received=192.168.48.128;rport=3030
> Max-Forwards: 0
> Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>
> Require: X-Medical
> Supported: X-Medical
> Accept: application/msdp,application/medical,text/plain
> Content-Type: application/msdp
> Content-Length: 307
>
> [MSDP offer not shown]

< SIP/2.0 180 Ringing
< Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
< CSeq: 1 INVITE
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0;isSender=false
< To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bK3f81fffc38e9ca2fbee2c7a9b0f53ec3;received=192.168.48.128;rport=3030
< Content-Length: 0

< SIP/2.0 200 OK
< Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
< CSeq: 1 INVITE
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0;isSender=false
< To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bK3f81fffc38e9ca2fbee2c7a9b0f53ec3;received=192.168.48.128;rport=3030
< Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>
< Require: X-Medical
< Supported: X-Medical
< Accept: application/msdp,application/medical,text/plain
< Content-Type: application/msdp
< Content-Length: 316
<
< [MSDP response not shown]

> ACK sip:sip-test@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
> Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
> CSeq: 1 ACK
> Via: SIP/2.0/TCP willempie.kuipnet.void:3030;branch=z9hG4bK7ec20440b8d9ebd367e87cc58bb0cc4b;received=192.168.48.128;rport=3030
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> Max-Forwards: 0
> Content-Length: 0

> MESSAGE sip:sip-test@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
> Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bKc46f28395a8fc776ea822c820dc06ecb;rport=3030
> CSeq: 2 MESSAGE
> Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> Supported: X-Medical
> Accept: application/msdp,application/medical,text/plain
> Max-Forwards: 0
> Content-Type: text/plain
> Content-Length: 16
>
> ready to receive

< SIP/2.0 200 OK
< Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bKc46f28395a8fc776ea822c820dc06ecb;rport=3030
< CSeq: 2 MESSAGE
< Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< To: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< Content-Length: 0

< BYE sip:sip-test@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
< Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bK3642c12692f4edf3bcd5a153598a8c59;rport=3031
< CSeq: 1 BYE
< Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
< From: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< Max-Forwards: 0
< Content-Length: 0

> SIP/2.0 200 OK
> Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bK3642c12692f4edf3bcd5a153598a8c59;rport=3031
> CSeq: 1 BYE
> Call-ID: 38355d2597d96a02daab6eec97ef748f@192.168.48.128
> From: "wim" <sip:wim@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> Content-Length: 0
```

Figure 6.10: Initiator receives data and sends offer

```
1  > INVITE sip:wim@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
2  > Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
3  > CSeq: 1 INVITE
4  > From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0;isSender=true
5  > To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>
6  > Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK0f3553a2c7b31075acb05c6a9e26d0ff;received=192.168.48.128;rport=3031
7  > Max-Forwards: 0
8  > Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>
9  > Require: X-Medical
10 > Supported: X-Medical
11 > Accept: application/msdp,application/medical,text/plain
12 > Content-Length: 0
13
14 < SIP/2.0 180 Ringing
15 < Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
16 < CSeq: 1 INVITE
17 < From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0;isSender=true
18 < To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
19 < Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK0f3553a2c7b31075acb05c6a9e26d0ff;received=192.168.48.128;rport=3031
20 < Content-Length: 0
21
22 < SIP/2.0 200 OK
23 < Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
24 < CSeq: 1 INVITE
25 < From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0;isSender=true
26 < To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
27 < Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK0f3553a2c7b31075acb05c6a9e26d0ff;received=192.168.48.128;rport=3031
28 < Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>
29 < Require: X-Medical
30 < Supported: X-Medical
31 < Accept: application/msdp,application/medical,text/plain
32 < Content-Type: application/msdp
33 < Content-Length: 307
34 <
35 < [MSDP offer not shown]
36
37 > ACK sip:sip-test@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
38 > Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
39 > CSeq: 1 ACK
40 > Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK69a04567aa51630cbf7737206e919241;received=192.168.48.128;rport=3031
41 > From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
42 > To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
43 > Max-Forwards: 0
44 > Content-Type: application/msdp
45 > Content-Length: 316
46 >
47 > [MSDP response not shown]
48
49 < MESSAGE sip:sip-test@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
50 < Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bK782b4b8ef247ca85ddcd3b12a1a6e369;rport=3030
51 < CSeq: 1 MESSAGE
52 < Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
53 < From: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
54 < To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
55 < Supported: X-Medical
56 < Accept: application/msdp,application/medical,text/plain
57 < Max-Forwards: 0
58 < Content-Type: text/plain
59 < Content-Length: 16
60 <
61 < ready to receive
62
63 > SIP/2.0 200 OK
64 > Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bK782b4b8ef247ca85ddcd3b12a1a6e369;rport=3030
65 > CSeq: 1 MESSAGE
66 > Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
67 > From: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
68 > To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
69 > Content-Length: 0
70
71 > BYE sip:sip-test@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
72 > Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bKc3f383b20226a4a6393d67bfe54b1ad0;rport=3031
73 > CSeq: 2 BYE
74 > Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
75 > From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
76 > To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
77 > Max-Forwards: 0
78 > Content-Length: 0
79
80 < SIP/2.0 200 OK
81 < Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bKc3f383b20226a4a6393d67bfe54b1ad0;rport=3031
82 < CSeq: 2 BYE
83 < Call-ID: da9e143e7b6a369f9ce77b2d0bfd6049@192.168.48.128
84 < From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
85 < To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
86 < Content-Length: 0
```

Figure 6.11: Initiator sends data and receives offer

```
> INVITE sip:wim@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
> Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
> CSeq: 1 INVITE
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0;isSender=false
> To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>
> Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK22ba8e07f3ded348bccc79ac982ff653;received=192.168.48.128;rport=3031
> Max-Forwards: 0
> Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>
> Require: X-Medical
> Supported: X-Medical
> Accept: application/msdp,application/medical,text/plain
> Content-Length: 0

< SIP/2.0 180 Ringing
< Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
< CSeq: 1 INVITE
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0;isSender=false
< To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK22ba8e07f3ded348bccc79ac982ff653;received=192.168.48.128;rport=3031
< Content-Length: 0

< SIP/2.0 200 OK
< Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
< CSeq: 1 INVITE
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0;isSender=false
< To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK22ba8e07f3ded348bccc79ac982ff653;received=192.168.48.128;rport=3031
< Contact: "sip-test" <sip:sip-test@willempie.kuipnet.void:3030;transport=tcp>
< Require: X-Medical
< Supported: X-Medical
< Accept: application/msdp,application/medical,text/plain
< Content-Type: application/msdp
< Content-Length: 307
<
< [MSDP offer not shown]

> ACK sip:sip-test@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
> Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
> CSeq: 1 ACK
> Via: SIP/2.0/TCP willempie.kuipnet.void:3031;branch=z9hG4bK433eb15d73754de59dfa26a7e81b84d5;received=192.168.48.128;rport=3031
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> Max-Forwards: 0
> Content-Type: application/msdp
> Content-Length: 348
>
> [MSDP response not shown]

> MESSAGE sip:sip-test@willempie.kuipnet.void:3030;transport=tcp SIP/2.0
> Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bKf6238ab739ab4335849829c5e5d3dcc0;rport=3031
> CSeq: 2 MESSAGE
> Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
> From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> Supported: X-Medical
> Accept: application/msdp,application/medical,text/plain
> Max-Forwards: 0
> Content-Type: text/plain
> Content-Length: 16
>
> ready to receive

< SIP/2.0 200 OK
< Via: SIP/2.0/TCP 192.168.48.128:3031;branch=z9hG4bKf6238ab739ab4335849829c5e5d3dcc0;rport=3031
< CSeq: 2 MESSAGE
< Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
< From: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< To: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< Content-Length: 0

< BYE sip:sip-test@willempie.kuipnet.void:3031;transport=tcp SIP/2.0
< Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bK3c0d0479f7281b69c8dd94ebe7f73dd6;rport=3030
< CSeq: 1 BYE
< Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
< From: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
< To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
< Max-Forwards: 0
< Content-Length: 0

> SIP/2.0 200 OK
> Via: SIP/2.0/TCP 192.168.48.128:3030;branch=z9hG4bK3c0d0479f7281b69c8dd94ebe7f73dd6;rport=3030
> CSeq: 1 BYE
> Call-ID: a2370232ac3b75ac26da0533c2168747@192.168.48.128
> From: "wim" <sip:wim@willempie.kuipnet.void:3030;transport=tcp>;tag=Medical_v1.0
> To: "sip-test" <sip:sip-test@willempie.kuipnet.void:3031;transport=tcp>;tag=Medical_v1.0
> Content-Length: 0
```

Figure 6.12: Initiator receives data and receives offer

Chapter 7

Conclusions and future work

This chapter presents the conclusions of our work, as well as suggestions for future work.

7.1 Conclusions

The objective of this thesis is to design a generic control protocol that can connect distributed application components of different vendors of E-health applications. It should be able to negotiate different data formats and configure a communication channel based on common capabilities and such that optimal communication is achieved, and which accommodates communication between distributed applications with possible different health data formats (as described in section 1.2).

We first created a list with requirements in section 3.2 which had to be fulfilled. Using this list of requirements, several control protocols were compared to see which one did match best, and needed smallest amount of adjustments to fit our needs. As a result of this comparison, SIP appeared to fit best our needs.

The parts not covered by SIP were identified, and we continued looking for solutions of the missing parts. As a result, two new media types were defined: MSDP and medical. Both are closely related to SDP, which is already used in combination with SIP.

After presenting the new message types and the extended rules for the protocol messages of SIP, we continued with designing a prototype. Interfaces were defined, and state diagrams drawn. Some dummy implementations were made to support testing, while the core part, the MDCP, was fully implemented.

The prototype has been tested to confirm the proper functioning of the designed protocol. Different scenarios were applied, and in all cases the behaviour was as expected. Also the list of requirements that were not fulfilled by basic SIP was compared again with the designed protocol.

When looking at the prototype and the comparison with the requirements, it can be concluded that the presented design of the control protocol is a solution to the objective of this thesis. Using MSDP and medical it is possible to connect distributed application components of different vendors of E-health applications.

7.2 Future work

During the work on this thesis, several issues appeared that were considered outside of the scope of this thesis. These issues can be a starting point for future research.

The control protocol solves the issue of incompatible protocols. But, before negotiating about a common protocol can be done, it should first be decided which data needs to be exchanged. When a measuring device wants to upload recorded data, this is not an issue, but when certain data is needed for evaluation by a medical person, there is no way to define this. For example, a nurse might want to see the ECG and blood pressure of a certain person for a certain date and time period. To solve this issue, search functionality should be added to the control protocol, so certain data can be queried, also when using devices of different manufacturers. Implementing this in the control protocol makes sure there is a querying method independent of medical standards.

When streaming a measurement for a longer time, it might happen that conditions change, due to a mobile user moving between different types of networks. Changed conditions might mean that the chosen standard is not longer the optimal solution. It can also happen that due to changed conditions the available bandwidth shrinks below the required value, making it impossible to continue the streaming in the current configuration. To solve this issue, it should be possible to reconfigure an ongoing transfer.

To make the design easier, we have removed the use of proxies from SIP. Proxies provide greater flexibility, but they result in more complicated security models. They allow for distributed storage of the data, load balancing between servers and locating of mobile users (this would allow a medical person to connect to a patient to receive real-time data when the patient does not continuously stream this to a central storage server). Research can be done on how proxies can be included again, while not breaking the functionality provided in the current model.

This thesis does present a model that can solve the problems mentioned. A prototype is implemented, but only a dummy data plane is implemented. Further research should be done on the different medical standards, and an implementation in the data plane should be developed. Also converters should be implemented to make sure the data plane can support different standards.

Appendices

Appendix A

SIP

The Session Initiation Protocol (SIP) is a control protocol used for negotiating voice and video connections over the internet. The protocol does not contain the media data itself, but the parameters needed to set up the connection. Several extensions are possible. A detailed overview can be found in this section.

A.1 Session Initiation Protocol (SIP)

[23] describes the Session Initiation Protocol (SIP). SIP is an application-layer control protocol for creating, modifying and terminating sessions with one or more participants. It can use different transport protocols, but must implement at least UDP and TCP. SIP does not handle the actual data, it only negotiates the parameters for the actual data transfer. Proxy servers are used as intermediate hosts to route SIP packets. A registration service is used by users to upload their current location, which is used by proxy servers to route packets to the right location and offers users mobility in the network while still being reachable.

The syntax used by SIP is much identical to HTTP/1.1, as defined in [12]. A message starts with a request line (request) or status line (answer), followed by the headers, an empty line, and optionally by a message body. The request line consists of a method (*REGISTER* for registering contact information, *OPTIONS* for querying servers about their capabilities, *INVITE*, *ACK* and *CANCEL* for setting up sessions and *BYE* for terminating sessions), followed by a single space character, the request URI (which indicates the called party), a single space character and the SIP version. The status line consists of the SIP version, followed by a single space character, a status code, a single space character and a textual description of the status. The first digit of the status codes defines the class of the response. The following classes are defined in SIP/2.0:

- *1xx* A provisional answer, the request is still being processed and a final answer will follow.
- *2xx* The action was successfully received, understood and accepted.
- *3xx* Redirection to another location, further action should be taken to complete the request.
- *4xx* The client made an error. The request contained a bad syntax or the server cannot fulfil the request.

- *5xx* The server made an error. The request is probably valid, but the server couldn't fulfil the request.
- *6xx* There was a global failure. There is no server that can fulfil this request.

Within every class, different status codes have pre-defined meanings, as defined in section 21 of [23]. They provide a more detailed description of what happened.

The request line or status line is followed by several header lines, all separated by a CRLF. Header lines might be split over several lines, as long as each extra lines starts with at least one space or horizontal tab. The line break and the white space character at the beginning of the next line are treated as a single space (SP) character. The order of the header field rows with the same field-name relative to each other is important, while the order of different field-names is not important.

If a message body is present, the Internet media type must be given by the *Content-Type* header field. Bodies of the MIME type *multipart*, as defined in [14], may be used. The size of the body must be provided in the *Content-Length* header. The *chunked* transfer encoding of HTTP/1.1 as defined in [12] is not allowed for SIP.

A user agent (UA) represents an end system, and contains a user agent client (UAC) for sending requests and a user agent server (UAS) for responding to requests. Valid SIP requests send by a UAC must at a minimum contain the following header fields: *To*, *From*, *CSeq*, *Call-ID*, *Max-Forwards* and *Via*. Valid SIP responses send by a UAC must contain the same *From*, *Call-ID*, *CSeq* and *Via* header in the same order as in the request. The *To* should also be the same, but if no *tag* is provided, this must be added.

The main SIP headers are described below:

- *To* This header specifies the desired recipient of the request, or the address-of-record of the user or resource that is the target of this request. The SIP URI scheme must be supported by all SIP implementations, while other URI schemes are allowed as well. This header is only valid in a dialogue.
- *From* This header specifies the logical identity of the initiator of the request. It contains a URI and optionally a display name. A *tag* parameter must be added.
- *Call-ID* Every dialogue should have a global unique call-ID. Messages can be connected to a dialogue via this header. Both client and server use the same call-ID in a dialogue.
- *CSeq* This header specifies the order of messages. Every subsequent message in a dialogue must have a higher *CSeq* number than the previous. It also includes the method, which should be the same as in the request line.
- *Max-Forwards* This is a counter that is decremented at every hop, to prevent unlimited forwarding of a message in for example a loop. When the value reaches 0 before the request reaches its destination, it will be rejected with a 483 (too many hops) error response.
- *Via* This header records the hosts a request travelled via. It is used to route responses. The *branch* parameter is compulsory and must be unique across space and time for all requests send by a particular UA.

- *Supported* This header specifies which extensions to SIP the UAC supports.
- *Require* This header specifies which extensions must be supported by the UAS to process the request.
- *Proxy-Require* This header specifies which extensions must be supported by a proxy to process the request.

A request that did not receive a final response yet, but did receive a provisional response (this applies to *INVITE*, which can take longer to process) can be cancelled by sending a *CANCEL* request. This request must include identical header fields, including *tags*, for *Request-URI*, *Call-ID*, *To*, *CSeq* (except the method which should be changed into *CANCEL*) and *From* as the request it tries to cancel. If present in the initial request, also the *Route* header should be identical. *Require* and *Proxy-Require* header fields are not allowed in *CANCEL* requests.

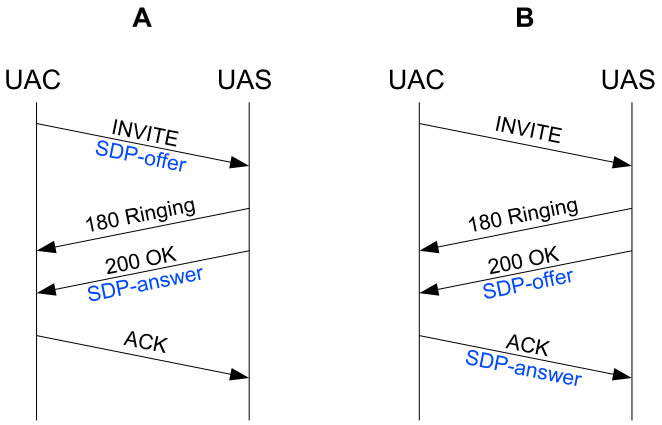

Figure A.1: SIP session set-up transaction. A shows the case where the UAC sends the SDP offer, B shows the case where the UAS sends the SDP offer.

Figure A.1 shows the messages exchanged in a session set-up without the use of a proxy server (using *INVITE*). The UAC sends an *INVITE* request to the UAS. The UAS replies with a *180 Ringing* response, to inform the UAC the request was received, and the UAS is waiting for user input. When the user picks up the phone (in case where the UAS is a phone) the UAS sends a *200 OK* reply. After receiving this, the UAC sends a *ACK* request to the UAS, which has the same *CSeq* number as the initial *INVITE* request (while the request method is changed from *INVITE* into *ACK*). From this moment on, the UAC and UAS have a session, which can be cancelled by either side by sending a *BYE* request, which should be replied with a *200 OK* message, after which the session does not exist any more. A and B differ in the way SDP messages are exchanged. In A the initial offer is generated by the UAC in the *INVITE* request, and answered (concluded) by the UAS with the *200 OK* reply. In B the UAC does not send an offer with the *INVITE* request. Now the UAS must include an offer in the *200 OK* reply. The UAC concludes the session parameters, and sends them to the UAS in the *ACK* request.

SIP allows clients to register with a server. This helps proxies in routing requests to mobile hosts. The following header fields are compulsory in a *REGISTER* request, except for *Contact*:

- *Request-URI* The domain of the location service for which the registration is meant.
- *To* The address for which the registration is to be created, queried or modified.
- *From* The address of the person responsible for registration, same as *To*, or different in case of registration by a third party.
- *Call-ID* Should be the same for all registrations done by a UAC.
- *CSeq* A counter that is incremented by 1 for each subsequent *REGISTER* request.
- *Contact* A particular endpoint of the user mentioned in *To*. This would normally contain the full host name of the host where the client of the user runs. The *expires* parameter can be used to indicate how long the registration should be valid, the *q* parameter is used to indicate the relative preference for a certain *Contact* address.

A 2xx response to a *REGISTER* contains all bindings for a particular address (mentioned in *To*) in the *Contact* header. A *REGISTER* request without a *Contact* header can be used to query the active bindings. Setting *expires* in the *Contact* header to "0" removes a binding. Using "*" for the *Contact* header with *expires* set to "0" removes all bindings for the registration in the *To* header.

OPTIONS requests can be used to query the capabilities of the server. An *Accept* header can be included to tell the UAS which formats can be used when generating the response. A response should contain the *Allow*, *Accept*, *Accept-Encoding*, *Accept-Language* and *Supported* header.

Existing sessions can be updated by sending another *REGISTER* message with new parameters. If accepted, the session will continue with the newly agreed parameters, otherwise the old ones will be used. To end a session, a *BYE* message must be send.

SIP proxies are used to route requests to UAS and responses to UAC. Every proxy makes its own routing decisions, and modifies the request before it is being forwarded. Responses travel back the same way in reverse order as a request went. A SIP message can pass several proxies on its way to the destination.

A.2 Relation between dialogs, transactions, requests and responses

Figure A.2 shows the relations between dialogs, transactions, requests and responses in SIP. Requests and responses are the messages exchanged over the network. A request includes a method, while a response is a reply to a request, including a status code. Due to their header values, messages are related to each other. The combination of a request and response is done in a transaction. This transaction has a client transaction (the request) and a server transaction (the response). A response is always related to a request, while a request might not have a related response (currently, only an *ACK* request does not have a related response).

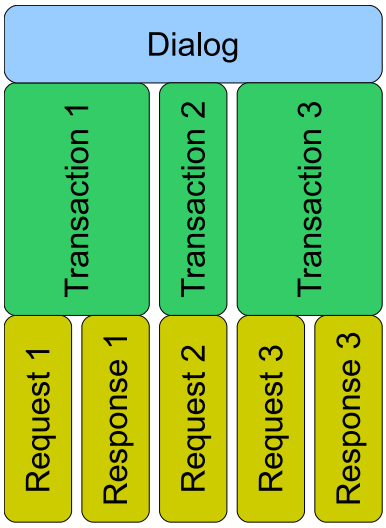

Figure A.2: Relations between dialogs, transactions, requests and responses in SIP.

When setting up a session, where several messages are related, a dialog is created. Requests and responses are part of a dialog because of common header values. A dialog is set-up by an *INVITE* request, and ended by a *BYE* request. All messages within that dialog are related to each other. Between two host, different dialogs can exist concurrently, and it is also possible to exchange messages outside of a dialog when a dialog exists. Dialogs are not always compulsory to be used, only in case a session is needed (this happens in an end host when sending or receiving an *INVITE*). For example an *OPTIONS* or *MESSAGE* request can exist outside of a dialog.

An example of messages exchanged in figure A.2 could be:

- Request 1: *INVITE*
- Response 1: *200 OK*
- Request 2: *ACK*
- Request 3: *BYE*
- Response 3: *200 OK*

A.3 Digest access authentication

[13] describes Basic and Digest Access Authentication. As Basic Access Authentication is far from secure, SIP only allows the usage of Digest Access Authentication. This does not guarantee a 100% secure system, but solves several issues that exist in Basic Access Authentication.

A server should send a 401 (unauthorised) response message to inform the client it needs to send credentials to use the service. An intermediate proxy uses the 407 (proxy authentication

required) response to inform the client it needs to send credentials to use the proxy. In the headers of such a reply, the authentication method is put, as well as the parameters needed for this. Whenever a server or proxy has once send a 401 or 407 response, the client may include the credentials in every subsequent request, to prevent an extra request/response sequence. In the Digest scheme a simple challenge-response mechanism is used, supported by a nonce generated by the server. The authentication information is communicated towards a server by sending a checksum over the user name, password, given nonce, HTTP method and the requested URI. The nonce changes regularly and should be unique, for example by using a hash of a time-stamp, HTTP ETag and private key. The nonce is set by the server. If a server would only accept the same nonce or digest once, a total protection against replay attacks is provided. As a counter-measurement against man in the middle attacks, a client can insert a cnonce, a nonce created by the client, so the man in the middle cannot make it easier to calculate the password from the response by using a pre-known nonce. This is because also the cnonce will be used in calculating the hash.

A.4 Locating SIP servers

DNS provides a way to publish records related to SIP, as described in [22]. First, NAPTR [19] records can be used to inform other parties about the services available for a certain domain, as well as their preference for a certain service. Examples of NAPTR records are:

```
;                    order pref flags  service    regexp replacement
utwente.nl. IN NAPTR   50   50   "s"  "SIPS+D2T"  "a"   _sips._tcp.utwente.nl.
utwente.nl. IN NAPTR   90   50   "s"  "SIP+D2T"   "a"   _sip._tcp.utwente.nl.
utwente.nl. IN NAPTR  100   50   "s"  "SIP+D2U"   "a"   _sip._udp.utwente.nl.
```

The *order* defines the order preferred by the receiving domain (utwente.nl in this example). A client must use the entry with the lowest *order* it supports. In case there is several entries with the same *order*, it should use the one with the lowest *pref*, but may for certain reasons choose one with higher *pref*. Ignoring the *order* is not allowed, the first one supported must be taken.

After finding the protocol to be used, a second DNS query is needed to find the actual server handling this protocol. The records that describe this are SRV records, defined in [15]. Examples of SRV records are:

```
;                            prio weight port target
_sip._tcp.utwente.nl. IN SRV    1    10    5060 pbx1.utwente.nl
_sip._tcp.utwente.nl. IN SRV    1    40    5060 pbx2.utwente.nl
_sip._tcp.utwente.nl. IN SRV    5    10    5060 pbx-backup.surfnet.nl
```

Prio defines the order in which the entries are to be tried. The lower numbers must be tried first. When several entries with the same *prio* exist, the *weight* defines which server is contacted. In the above example, both pbx1.utwente.nl and pbx2.utwente.nl share the same *prio*, so the *weight* defines which one to take. The total sum of *weight* is 50 for *prio* 1, meaning that pbx1.utwente.nl should be contacted with chance 10/50 and pbx2.utwente.nl should be contacted with chance 40/50. This can be used for load-balancing the servers. If a connection set-up fails to a server, all the other servers with the same *prio* should be tried first before

moving to a higher *prio*. In the above example, this would mean 20% of the requests arrives at pbx1.utwente.nl and 80% of the requests arrives at pbx2.utwente.nl. Only if both these servers don't respond, pbx-backup.surfnet.nl will be tried. In SRV records, both port and host name can be defined.

A.5 Event notifications

[20] defines an extension to SIP that specifies event notifications. When SIP is already used, and one has the need to send notifications of certain events, this extension is useful. To receive notifications, one has to subscribe first to a certain group of notifications. All *SUBSCRIBE* and *NOTIFY* messages are confirmed by the receiver. The *Event* header is used to inform the notifier for which kind of events notifications should be send. Only one *Event* header is allowed per message. A subscription has a limited duration, after which a new subscription must be made to keep receiving notifications. Immediately after confirming a *SUBSCRIBE* message, a first *NOTIFY* should be send. Subscribers should be prepared to receive the first *NOTIFY* message before the *SUBSCRIBE* transaction is completed, this due to out-of-order messages. The duration of a subscription is defined with the *Expires* header, where the notifier defines this value, while considering the requested subscription time of the subscriber. With an *Expires* header of 0, unsubscription is requested.

A notifier must immediately respond to a *SUBSCRIBE* transaction, and is not allowed to wait for user input. If the event package is understood, the subscriber is authorized to subscribe and no other barriers for creating the subscription exist, a "200 OK" response is send, immediately followed by a notify message containing the current resource state. If a subscription request cannot immediately be answered, for example when user input is required for authorisation, a "202 Accept" response is send, also immediately followed by a *NOTIFY* message. Later on, if the request was rejected, a notification with *Subscription-State* header with value "terminated" can be send to inform the subscriber. Every ending of a subscription, be it because a resource is not available, a time-out or a subscriber request, should generate a *NOTIFY* with *Subscription-State* header with value "terminated", as in this case the corresponding dialogue can be terminated. A single *SUBSCRIBE* request may trigger several *NOTIFY* requests.

Every *NOTIFY* message must have a *Subscription-State* header, with a value of "active", "pending" or "terminated". In case of "active" or "pending" state, an *expires* parameter with the remaining time of subscription should be included. In case of "terminated" state, a *reason* parameter should be included.

The *Allow-Events* header indicates support for *SUBSCRIBE* and *NOTIFY* messages. They may be returned in the response to an *OPTIONS* request, as well as in all methods which initiate dialogues and their responses.

A.6 Session Description Protocol (SDP)

[17] specifies the Session Description Protocol (SDP). SDP is a standard representation of information used to announce and set up multimedia sessions. SDP itself does not contain

multimedia data, neither is it a stand-alone protocol, it always needs a transport protocol, such as SIP, RTSP, SMTP and HTTP. It is denoted by the media type "application/sdp".

The options in the session description have a fixed order, this to allow simple parsers and easy error detection. Every field in the description consists of one case-significant character, followed by the "=" sign and then directly the value, where the line is ended by a CRLF. The description starts with a description of the whole session. After this session description, descriptions of zero or more media sessions follows, where the options set in the session description are used as default values, while the media description can override the defaults.

Now the compulsory fields will be described.

- *Version number* (v=). Current version number is 0.
- *Origin* (o=). This announces the originator of the session, including user name, unique session identification, session version (increased when the session data is modified), network type (currently only "IN" is defined meaning "Internet"), address type ("IP4" or "IP6" standing for IPv4 and IPv6) and unicast address of the session creating host.
- *Session name* (s=). A textual description of the session.
- *Connection data* (c=). It consists of network type, address type and connection address (see also origin). *Connection data* should either be present in the session description, or in every media description.
- *Timing* (t=). It defines the start and stop time, with NTP timestamps. A time stamp of 0 for stop time defines the session is unbounded, but does not start before start time. If also start time is 0, the session is permanent, although it is discouraged to use a value of 0.
- *Attributes* (a=). This is primary used for extending SDP. It can be used both in the session description as well as in the media description. It can contain both binary attributes and value attributes, where the attribute and value are separated by a semi-colon. It is often used to define the RTP payload type numbers, or the direction of the session ("recvonly" for receive only, "sendrecv" for bi-directional communication, "sendonly" for sending only and "inactive" for making the session inactive, for example when a user is put on hold).
- *Media descriptions* (m=). It consists of a media type (currently defined are "audio", "video", "text", "application" and "message"), port where the media stream is send, the transport protocol to be used (currently defined are "udp" denoting an unspecified protocol running over UDP, "RTP/AVP" denoting RTP used under the RTP profile for audio and video conferences with minimal control running over UDP and "RTP/SAVP" denoting secure RTP running over UDP) and the media format description, which might be repeated several times to announce several accepted formats. The order gives the preference of the format to be used. The RTP payload type numbers used here should be assigned with the *attribute* (for example: a=rtpmap:98 L16/11025).

Media descriptions can be added and removed in subsequent offers, as described in [21]. Adding a new media description is done by adding an extra media description line at the

end of the SDP message, or by replacing an old one. To remove a media description, the specific description line should be replaced with a new description containing a port number of 0. Changing a description is done by replacing the line describing the media that should be altered. The order must not be changed in any case, as the order is used to distinguish different streams by the receiver.

A.7 Instant Messaging

[9] specifies an extension to SIP for transferring Instant Messages. It allows for transferring MIME bodies inside and outside sessions. The method that is added to SIP for this is *MESSAGE*. Message requests are confirmed, but this might be a provisional *202 Accepted* response from a intermediate host. A *200 OK* response only confirms the receipt of a message by the final UAS, not that it has been processed or seen by the user of the UAS. A UAS that supports the *MESSAGE* method must understand bodies of the type "text/plain" and may understand bodies of the type "message/cpim".

Because SIP does not include congestion control, a new message should only be send after the receipt of the confirmation of the previous message. This to prevent the control network being flooded with media data, for which it was initially not intended. Messages must not exceed 1300 bytes, this because upstream proxies might chose to send a message request over UDP.

SIP does normally not include the actual media data. For this reason, security plays a more important role when using the *MESSAGE* method, which includes the actual media data. End-to-end authentication, body integrity and body confidentiality mechanisms must be implemented when using the *MESSAGE* method. Using the SIPS URI mechanism provides hop-by-hop protection, but all proxies must be trusted by the user, as they can see the message body in plain text.

Appendix B

RTSP

[25] describes the Real Time Streaming Protocol (RTSP). It is an application level protocol, which controls the delivery of real-time data. It can be used for on-demand and live streaming of data, for example audio and/or video. Several data streams can be controlled at once that need to be time-synchronised. Normally the data streams are not delivered over RTSP, although this is possible with interleaving. It runs on top of transport protocols like UDP, multicast UDP and TCP and is similar in syntax and operation to HTTP/1.1, as defined in [12].

Sessions play an important role in RTSP. They are identified by a *session-id*, which is normally assigned by the server in a random way. This to prevent that one can spoof the *session-id* of somebody else and control his or her media sessions. When a request that generates a session (*SETUP*) arrives, the server generates a *session-id* in the response. The client has to include that *session-id* in every following request.

Several methods are defined for RTSP:

- *DESCRIBE* This method is used to get a complete description of a media object as identified by the request URL.
- *ANNOUNCE* This method is used by the client to inform the server about the description of a presentation, and the server uses it to update a session description.
- *GET_PARAMETER* This method is used to retrieve the value of a parameter of the stream specified in the URI. If this method is send without a body, it can be used to check if the server is alive. The format of the body is not specified.
- *OPTIONS* This method is used to query a server for the support for certain functionality.
- *PAUSE* This method is used to interrupt an ongoing data stream. It is possible to request to stop in future. When continuing the data stream afterwards (with *PLAY* or *RECORD*), it will continue at the point where it was paused. After a certain time of being paused, the server may free the resources associated with the session.
- *PLAY* This method is used, after all outstanding *SETUP* requests have been acknowledged, to request the server to start sending data for the sessions defined in the *SETUP* method. It is possible to request the playing of parts of a media stream, and also to

request the playing of several parts at once without waiting for the previous parts to finish playing.

- *RECORD* This method can be used to ask a server to record a session to which it has been invited before. The server may store the stream at a different location than requested in the URI, but should send a *201* (created) response, including a *Location* header with the new storage location.
- *REDIRECT* This method is used by the server to inform the client that it should connect to another host if it wants to continue to receive the stream. The client should issue a *TEARDOWN* to the current server, and a *SETUP* to the new location, as pointed out in the *Location* header of the *REDIRECT* request.
- *SETUP* This method is used to set up a session. It can also be used for updating an existing session.
- *SET_PARAMETER* This method is used to set the value of a parameter of the stream specified in the URI. One should only request the change of one parameter at a time, to allow the server to either accept or reject it. If several parameters are updated, the server should either update all, or reject all. The format of the body is not specified.
- *TEARDOWN* This method stops the stream delivery and frees the resources of the session.

The response starts with a status line, that includes a status code. The status codes are the same as in HTTP/1.1 (see [12]), with several additions. The first digit of the status code categorises the status:

- *1xx* is an informational response, the request is received and is being processed.
- *2xx* is a response informing that the action was successfully received, understood and accepted.
- *3xx* is a redirection response, that tells the client that a new request will have to be send to a different location.
- *4xx* is a response informing that the client used bad syntax in the request, or that the request cannot be fulfilled.
- *5xx* is a response informing that the server failed to fulfil a valid request.

Both requests as well as responses can in several cases contain a message body. This message body is defined by certain headers. The message body is used to include information that cannot be included in the headers. The *Content-Length* must be used to define the length of the body.

Bibliography

[1] Introduction to hl7. Available from: http://www.hl7.com.au/FAQ.htm.

[2] Jain sip api specification. Available from: http://jcp.org/en/jsr/detail?id=32.

[3] Java api for sip signalling. Available from: http://jain-sip.dev.java.net/.

[4] jsdp: a java implementation of sdp protocol. Available from: http://jsdp.sourceforge.net/.

[5] U.s. food and drug administration. Available from: http://en.wikipedia.org/wiki/Fda.

[6] V. Balabanian, L. Casey, N. Greene, and C. Adams. An introduction to digital storage media-command and control. *Communications Magazine, IEEE*, 34(11):122–127, November 1996. Available from: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=544202.

[7] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp). Rfc, The Internet Society, September 1997. Available from: http://tools.ietf.org/html/rfc2205.

[8] B. Brown, M. Kohls, and N. Stockbridge. Fda xml data format design specification, draft. 2002. Available from: http://xml.coverpages.org/FDA-EGC-XMLDataFormat-C.pdf.

[9] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle. Session initiation protocol (sip) extension for instant messaging. Rfc, The Internet Society, December 2002. Available from: http://tools.ietf.org/html/rfc3428.

[10] CEN. Standard communication protocol for computer-assisted electrocardiography. 2000. Available from: http://www.tc251wgiv.nhs.uk/pages/pdf/censcp019.pdf.

[11] B. Erfianto. Design of a vital sign protocol format using xml and asn.1. Master's thesis, Faculty of Computer Science, University of Twente, The Netherlands, 2004.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Rfc, The Internet Society, June 1999. Available from: http://tools.ietf.org/html/rfc2616.

[13] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. Rfc, The Internet Society, June 1999. Available from: `http://tools.ietf.org/html/rfc2617`.

[14] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part two: Media types. Rfc, The Internet Society, November 1996. Available from: `http://tools.ietf.org/html/rfc2046`.

[15] A. Gulbrandsen, P. Vixie, and L. Esibov. A dns rr for specifying the location of services (dns srv). Rfc, The Internet Society, February 2000. Available from: `http://tools.ietf.org/html/rfc2782`.

[16] F. Halsall. *Multimedia communications: applications, networks, protocols, standards*. Addison Wesley, 2001.

[17] M. Handley, V. Jacobson, and C. Perkins. Sdp: Session description protocol. Rfc, The Internet Society, July 2006. Available from: `http://tools.ietf.org/html/rfc4566`.

[18] S. Horii. Primer on computers and information technology. part four: A nontechnical introduction to dicom. *Radiographics*, 17(5):1297–1309, 1997. Available from: `http://radiographics.rsnajnls.org/cgi/reprint/17/5/1297.pdf`.

[19] M. Mealling. Dynamic delegation discovery system (ddds) - part three: The domain name system (dns) database. Rfc, The Internet Society, October 2002. Available from: `http://tools.ietf.org/html/rfc3403`.

[20] A. B. Roach. Session initiation protocol (sip)-specific event notification. Rfc, The Internet Society, June 2002. Available from: `http://tools.ietf.org/html/rfc3265`.

[21] J. Rosenberg and H. Schulzrinne. An offer/answer model with the session description protocol (sdp). Rfc, The Internet Society, June 2002. Available from: `http://tools.ietf.org/html/rfc3264`.

[22] J. Rosenberg and H. Schulzrinne. Session initiation protocol (sip): Locating sip servers. Rfc, The Internet Society, June 2002. Available from: `http://tools.ietf.org/html/rfc3263`.

[23] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol. Rfc, The Internet Society, June 2002. Available from: `http://tools.ietf.org/html/rfc3261`.

[24] P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. Rfc, The Internet Society, October 2004. Available from: `http://tools.ietf.org/html/rfc3920`.

[25] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (rtsp). Rfc, The Internet Society, April 1998. Available from: `http://tools.ietf.org/html/rfc2326`.

[26] H. Wang, F. Azuaje, G. Clifford, B. Jung, and N. Black. Methods and tools for generating and managing ecgml-based information. 2004. Available from: `http://mimic.mit.edu/Archive/Publications/Wang04.pdf`.

[27] H. Wang, F. Azuaje, B. Jung, and N. Black. A markup language for electrocardiogram data acquisition and analysis (ecgml). *BMC Medical Informatics and Decision Making*, 3(1):4, 2003. Available from: `http://www.biomedcentral.com/1472-6947/3/4`.

[28] H. Wang, B. Jung, F. Azuaje, and N. Black. ecgml: Tools and technologies for multimedia ecg presentation. May 2003. Available from: `http://www.idealliance.org/papers/dx_xmle03/papers/04-05-02/04-05-02.html`.

[29] J. Wijsman. An information model for coherent ecg data sets. Bachelor's thesis, Faculty of Science and Technology, University of Twente, The Netherlands, August 2007.