Lappeenranta University of Technology
Faculty of Technology Management
Degree Program of Information Technology

Master's Thesis

Mikko Gynther

REAL-TIME MUSICAL PAIR IMPROVISATION ON MOBILE DEVICES

Examiners of the Thesis: Professor Jari Porras
D.Sc. Kari Heikkinen
Instructor of the Thesis: M.Sc. Tommi Kallonen

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
Degree Program of Information Technology

Mikko Gynther

REAL-TIME MUSICAL PAIR IMPROVISATION ON MOBILE DEVICES

Master's Thesis
2011

100 pages, 27 figures, 28 tables, 3 appendices
Examiners: Professor Jari Porras
D.Sc. Kari Heikkinen
Keywords: real-time, wireless communication, musical improvisation, mobile devices

This thesis discusses the design and implementation of a real-time musical pair improvisation scenario for mobile devices. In the scenario transferring musical information over a network connection was required. The suitability of available wireless communication technologies was evaluated and communication was analyzed and designed on multiple layers of TCP/IP protocol stack. Also an application layer protocol was designed and implemented for the scenario.

The implementation was integrated into a mobile musical software for children using available software components and libraries although the used platform lead to hardware and software constraints. Software limitations were taken into account in design. The results show that real-time musical improvisation can be implemented with wireless communication and mobile technology. The results also show that link layer had the most significant effect on real-time communication in the scenario.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Mikko Gynther

REAALIAIKAINEN MUSIIKKI-IMPROVISAATIO PAREITTAIN MOBIILILAITTEILLA

Diplomityö
2011

100 sivua, 27 kuvaa, 28 taulukkoa, 3 liitettä
Tarkastajat: Professori Jari Porras
TkT Kari Heikkinen
Hakusanat: reaaliaikainen, langaton kommunikaatio, musiikki-improvisaatio, mobiililaitteet
Keywords: real-time, wireless communication, musical improvisation, mobile devices

Tämä työ käsittelee reaaliaikaisen musiikki-improvisaation suunnittelua ja toteutusta mobiililaitteille. Improvisaatio tapahtuu pareittain ja perustuu musiikillisen informaation siirtoon verkkoyhteyden yli. Käytettävissä olleiden langattomien teknologioiden soveltuvuutta arvioitiin ja kommunikointia analysoitiin ja suunniteltiin useilla TCP/IP -protokollapinon kerroksilla. Myös sovelluskerroksen protokolla suunniteltiin ja toteutettiin.

Toteutus integroitiin lasten mobiiliin musiikkisovellukseen käytettävissä olleita ohjelmistokomponentteja ja -kirjastoja hyödyntäen, vaikka ympäristö asetti laitteisto- ja ohjelmistorajoitteita. Ohjelmistokomponenttien puutteet huomioitiin suunnittelussa. Työn tulokset osoittavat, että musiikki-improvisaatio voidaan toteuttaa langattoman tiedonsiirron ja mobiiliteknologian avulla. Tulokset osoittavat myös, että linkkikerroksella oli suurin merkitys kommunikaation reaaliaikaisuuteen.

PREFACE

This theses was done in Communications Software Laboratory in Lappeenranta University of Technology as a part of the Seventh Framework Programme UMSIC project. The author would like to thank the Communications Software Laboratory and the UMSIC consortium for a challenging and motivating project.

TABLE OF CONTENTS

ABBREVIATIONS

ADHD	Attention-Deficit Hyperactivity Disorder
ALSA	Advanced Linux Sound Architecture
API	Application Programming Interface
BP	Beacon Period
CEM	Cognitive Engineering Module
CHUM	Child-centric Usability Module
CP	Cross-platform
DCCP	Datagram Congestion Control Protocol
DSSS	Direct Sequence Spread Spectrum
EDR	Enhanced Data Rate
FHSS	Frequency Hopping Spread Spectrum
GEMS	General Middleware Services
GPL	General Public License
GPS	Global Positioning System
GNU	GNU's Not Unix
GUI	Graphical User Interface
HSPA	High Speed Packet Access
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISM	Industrial, Scientific and Medical
JACK	JACK Audio Connection Kit
JamMo	Jamming Mobile
JSON	JavaScript Object Notation
LAN	Local Area Network
LLC	Logical Link Control
LOS	Line-of-sight
MAC	Media Access Control
Mac OSX	Macintosh Operating System X

MEAM	Musical Engineering and Authoring Module
MIDI	Musical Instrument Digital Interface
MIMO	Multiple-Input Multiple-Output
NTP	Network Time Protocol
OFDM	Orthogonal Frequency Division Multiplexing
OpenGL	Open Graphics Library
PeerHood	Peer-to-Peer Neighborhood
QoS	Quality of Service
RAM	Random Access Memory
RFC	Request for Comments
RTP	Real-time Transport Protocol
RTS / CTS	Request To Send / Clear To Send
RTT	Round Trip Time
SCTP	Stream Control Transmission Protocol
SIG	Special Interest Group
TCP	Transmission Control Protocol
TCP-RTM	Transmission Control Protocol Real-time Mode
TCP SACK	Transmission Control Protocol Selective Acknowledgement
TCPW	Transmission Control Protocol Westwood
UDP	User Datagram Protocol
UHF	Ultra High Frequency
UI	User Interface
UMSIC	Usability of Music in Social Inclusion of Children
USB	Universal Serial Bus
WAN	Wide Area Network
Wi-Fi	Wireless Fidelity
WLAN	Wireless Local Area Network
WVGA	Wide Video Graphics Array

1 INTRODUCTION

This thesis was done in Usability of Music in Social Inclusion of Children (UMSIC) project. The practical part of this thesis is a part of Jamming Mobile (JamMo) application which is developed in UMSIC. In this chapter UMSIC, real-time pair improvisation scenario, objectives, research questions and the structure of this thesis are presented.

1.1 Usability of Music in Social Inclusion of Children

UMSIC is a multidisciplinary and transnational science and technology project. It aims at measuring and increasing social inclusion of children by the means of musical collaboration and modern technology. The project started in 2008 and will end in 2011. [53]

UMSIC brings together expertise from developmental psychology, music education, music technology, music therapy, software engineering and human computer interaction for children. The project partners are University of Oulu, University of Central Lancashire, University of Jyväskylä, Lappeenranta University of Technology, University of Zürich, Systema Technologies, Institute of Education University of London and Nokia.

There are two target groups: children with Attention-Deficit Hyperactivity Disorder (ADHD) and immigrant children. Both groups may have difficulties with language and they are in a risk to be marginalized [53]. Music therapy studies have shown that musical activities improve self-regulation, self-esteem, self-expression, social functioning and interaction [53]. The use of technology improves digital literature and technological skills which is important in the modern world [53]. The social inclusion of children of ages 3-12 is measured in kindergartens and schools. Groups consisting of native and immigrant children are researched in regular schools. Children with ADHD are researched in specialized schools.

In UMSIC there are four different scenarios based on the level of collaborative activities. In Standalone scenario there is no collaboration between children and the musical activities take place on a single mobile device. In Ad hoc scenario two children collaborate with two mobile

devices. In Public scenario a group of children collaborate in a classroom environment with multiple mobile devices. Teacher is able to monitor and control the activities with a computer. In Networked scenario a group of children collaborate with mobile devices through a community server located in the Internet.

1.2 Real-time pair improvisation scenario

Musical improvisation is an activity in which a person creates new music by singing or playing an instrument during a performance. The performance is not throughly composed beforehand and a performer can reflect his or her mood to a great extent and spontaneously react to the stimulus of other musicians or an audience. In collective improvisation performers may imitate or complement each other. Turn-taking patterns, such as call-and-response (also known as question-and-answer) and sequential solos, are common. For instance jazz, blues, and various non-Western musical styles rely heavily on improvisation as opposed to classical Western music which is strictly formalized by a composer. Improvised music, however, typically has some compositional elements such as tempo, time signature, motif, theme, structure, chord progression or scale.

In real-time pair improvisation scenario of UMSIC project two children aged from 7 to 12 create new music by playing virtual musical instruments along a backing track. Improvisation takes place in real-time which means that children can hear themselves and each other without a noticeable delay. Virtual instruments are software components that produce instrument sounds according to user input. They are shown on the screens of mobile devices and played by touching. A backing track is a musical recording which provides harmonic, rhythmic and or melodic content to improvise on. The backing track can be seen as a group of accompanying musicians whereas improvising children are soloists. Both devices are able to store the whole performance for later listening and editing. Real-time pair improvisation is a part of the Ad hoc scenario of UMSIC but it can be utilized in Public scenario as well. Ad hoc scenario and classroom environment in Public scenario indicate that peers are relatively close to each other as walls limit distances to a few meters.

Real-time pair improvisation scenario is presented in Figure 1. In the figure there are two

users with mobile devices. Both devices run a musical application and send a real-time data stream to each other. Sending data in real-time is required because users use headphones. Children have to use headphones because there may be many pairs improvising simultaneously in the same room. The use of speakers would create a lot of noise which would make creating music difficult. Because peers can not hear each other from speakers it is compulsory to transfer playing via a low latency network connection in order to avoid a noticeable delay. Playing can be transferred as audio or control data, such as Musical Instrument Digital Interface (MIDI). Improvisation may take place when the devices are in the same network without the use of a centralized server.

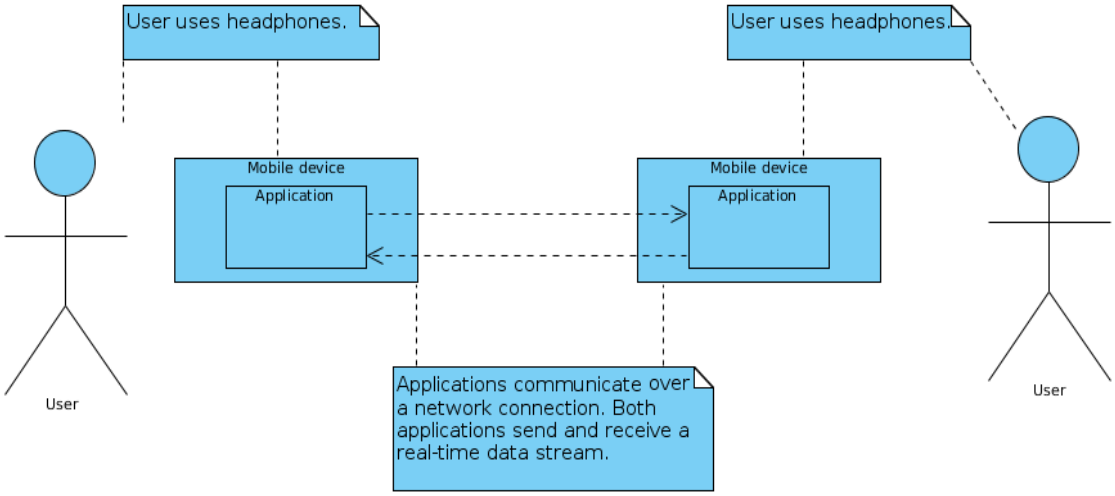

Figure 1: Real-time pair improvisation scenario

Similar scenarios has been studied in wired networks. Zimmermann et al. have studied the requirements for the scenario [57]. They created a system for transmitting MIDI and video data to provide authentic distributed musical collaboration. Williams & Chapman have created a system for transferring MIDI over the Internet and compared the performance of different networking protocols [56].

Musical data has also been transferred over wireless technologies. Bartolomeu et al. have researched and optimized Bluetooth protocol stack for transferring MIDI data [3]. Wireless Local Area Network (WLAN) has been used in MIDI transfer by Maekawa et al. [35]. They achieved acceptable network latency. Also audio has been streamed over WLAN [13].

1.3 Objectives and research questions

The focus of this thesis were the communications between the devices and the implementation of the real-time pair improvisation scenario. The objective was to design and implement the pair improvisation scenario in JamMo by using existing musical and Graphical User Interface (GUI) components.

In the scenario there were many crucial aspects which were not part of this thesis. The musical engineering aspects on a single device such as the implementation of virtual instruments and a multi-track sequencer, were not considered. Moreover, the GUI components for playing virtual instruments were not created. Furthermore, group management and device discovery were not considered. Components with these functionalities were used and slightly expanded or modified when required to suit the scenario.

Educational aspects of musical improvisation were not part of this thesis. Turn-taking patterns or scales were not designed to be forced. Instead, it was decided that during an improvisation session children should be free to play their own virtual instruments when and how they want to. Teachers can instruct children to play in a particular style, e.g. pentatonic blues with the black keys of a keyboard, just like with physical instruments. Different musical styles can be utilized with different backing tracks.

It was not specified how the devices should communicate in real-time. This lead to Research Question 1:

Research Question 1: How should the devices communicate in real-time pair improvisation scenario?

JamMo is a large piece of software consisting of several modules. Real-time pair improvisation required the use, extension and creation of multiple components in different modules which lead to Research Question 2:

Research Question 2: How to implement and integrate the real-time pair improvisation scenario in JamMo?

1.4 Structure

The structure of this thesis is presented in Figure 2. Chapters form three parts. Chapters 1 and 2 describe the background of this thesis. Chapters 3 to 6 consider the practical part. Chapters 7 discusses the done work. The chapters consider the following topics:

- Chapter 2 **Jamming Mobile** introduces the software developed in UMSIC and in this thesis.
- Chapter 3 **Real-time pair improvisation** considers the scenario of the thesis from a technical viewpoint. Concepts, technical requirements and constraints are considered.
- Chapter 4 **Communications design** discusses the design aspects of needed communications on various networking layers.
- Chapter 5 **Communication protocol** defines the protocol used in communications.
- Chapter 6 **Implementation** considers the implementation and the issues during implementation.
- Chapter 7 **Conclusion** discusses the done work, its results and future work.

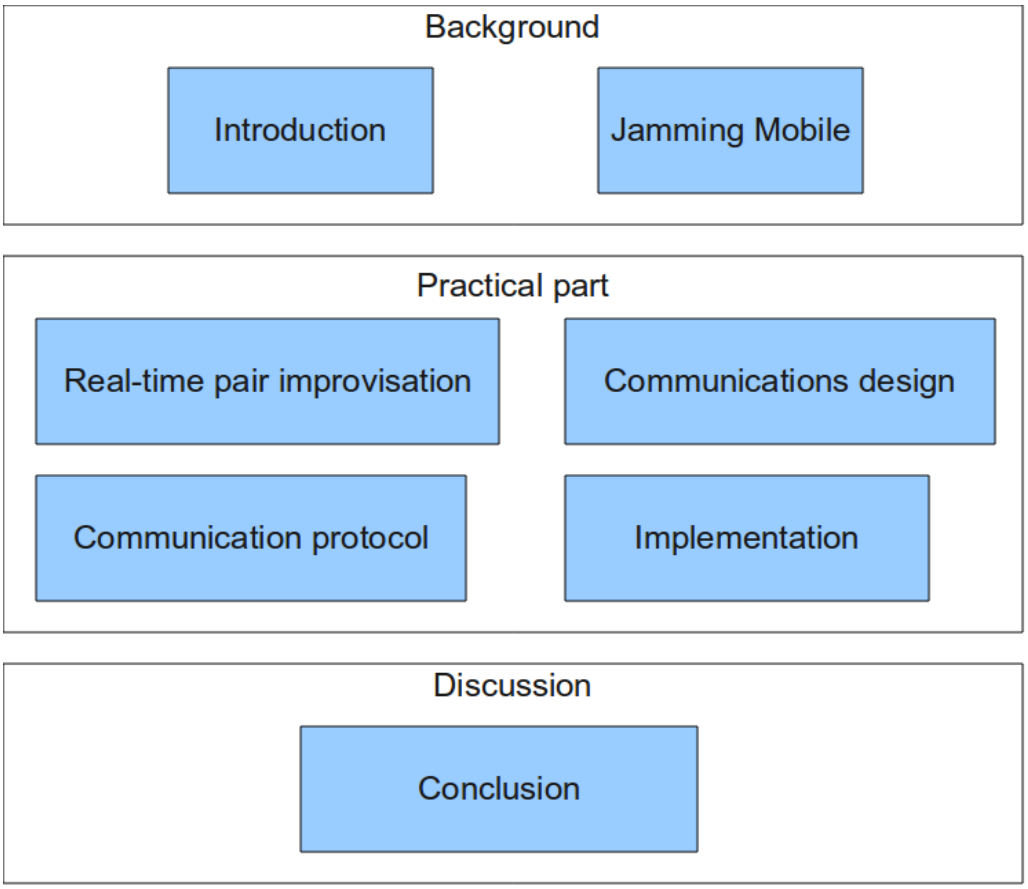

Figure 2: Structure of the thesis

2 JAMMING MOBILE

In this chapter Jamming Mobile (JamMo), its target device and the communication technologies of the target device are presented.

JamMo is a music collaboration and learning tool. It is intended for children aged from 3 to 12. Based on age the features are different. Children aged from 3 to 6 can play a singing game and a simple loop composition game in Standalone and Ad hoc scenarios. In loop composition game only prerecorded audio loops can be used in compositions. Children aged from 7 to 12 can also play virtual instruments and use recorded singing or playing with instruments as parts of compositions in Public and Networked scenarios.

JamMo is targeted at Nokia N900 and Maemo operating system but it can be built and run on other Linux distributions, such as Ubuntu Linux, as well. JamMo is written in C programming language. Many subcomponents are GObject [17] classes. GObject is the object oriented programming model of GLib [16] cross-platform software utility library.

In addition to JamMo application Teacher Server and Community Server are developed. The servers are important in Public and Networked scenarios as they provide possibilities to monitor the activities of children and to share created music. JamMo and the servers are presented in Figure 3.

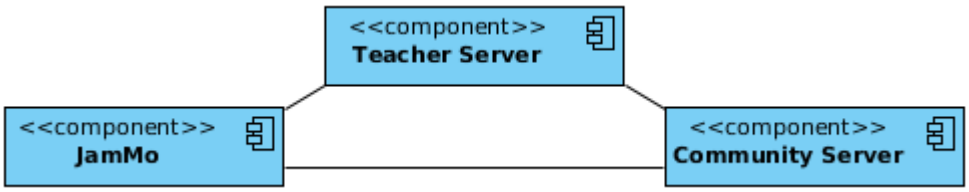

Figure 3: Components in UMSIC project

Installation instructions and binary releases are available on the JamMo website [47]. The latest source code of JamMo can be downloaded from Gitorious project hosting service [15]. In total JamMo consists of more than 70 000 lines of code (including unit test programs).

JamMo is released under General Public License (GPL) version 2.

JamMo uses musical and graphical materials produced especially for JamMo. The musical materials were made professionally and contain several backing tracks and dozens of audio loops and virtual instrument samples. The graphical materials have been designed to look appealing to children.

2.1 JamMo components

JamMo consists of four components: Child-centric Usability Module (CHUM), Musical Engineering and Authoring Module (MEAM), General Middleware Services (GEMS) and Cognitive Engineering Module (CEM). JamMo components are presented in Figure 4.

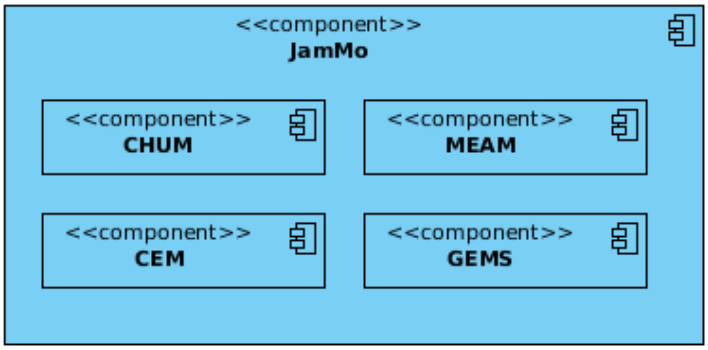

Figure 4: JamMo Components

CHUM contains the user interface (UI) including various elements and the local application logic including games and mentor. Tangle [50], developed in University of Oulu, and Clutter [9] Graphical User Interface (GUI) libraries are used extensively in CHUM. Most sub-components of CHUM are GObject classes while others are procedural C code and use GLib.

MEAM consists of music making components. GStreamer [18], a streaming media framework using GObject, is used extensively in MEAM. GStreamer hides complexity from application programmers by encapsulating functionality inside elements. GStreamer elements are supplied in plugin packages. They can also be programmed by deriving from

GStreamer element classes. MEAM consists mainly of GObject classes that control GStreamer elements. Readily available and created GStreamer elements do the actual processing when they are linked to each other.

GEMS consists of networking components. GEMS provides group management, authentication and authorization, profile management and communications. It uses PeerHood [44] to establish connections. PeerHood is an open source ad hoc connectivity and peer-to-peer service framework library developed in Lappeenranta University Technology. It provides seamless connectivity between networking technologies and an Application Programming Interface (API) for providing services. GEMS consists of procedural C code and uses Glib.

CEM is used to log user actions and other events for researchers. Logging provides important information for social inclusion and musical learning research. For example, it can be traced with whom the children have collaborated with and for how long. Also musical learning can be evaluated based on logged data. CEM consists of procedural C code and uses Glib.

2.2 JamMo component relationships

JamMo component relationships are presented in Figure 5. All components use CEM for logging. CHUM uses GEMS for networking including communications within a group during multiplayer games and getting information about group related events, such as peers leaving or joining a group. CHUM uses subcomponents of MEAM for music making. GEMS uses data structures from MEAM to transfer musical information over network.

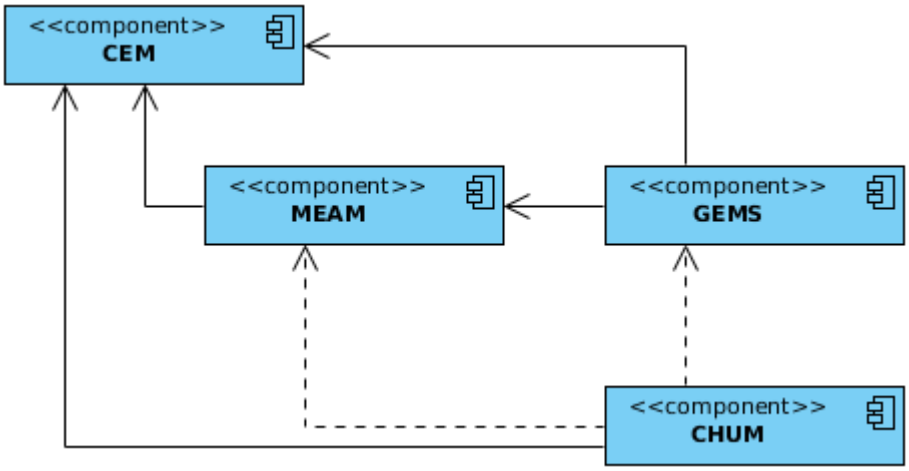

Figure 5: JamMo Component relationships

Dependencies (dashed arrows) in Figure 5 show that MEAM and GEMS are linked to CHUM when building JamMo. All components are required for building JamMo. Directional arrows indicate visibility, e.g. CHUM is able to use functions from GEMS. Communication from GEMS to CHUM is implemented with callback functions to overcome the limited visibility.

2.3 Nokia N900

Nokia N900 (see Figure 6) is the latest Internet Tablet by Nokia. It was released in 2009. Unlike its predecessors (N810, N800, N770) it also has mobile phone functionality. A summary of specifications of N900 are presented in Table 1.

Figure 6: Nokia N900[42]

Table 1: Nokia N900 specifications [43]

Feature	**Description**
Processor	ARM Cortex – A8 processor running at 600 MHz
Random Access Memory (RAM)	256 MB
Internal storage	32 GB
Graphics	3D graphics accelerator with Open Graphics Library (OpenGL) support
Operating system	Maemo 5
Connectivity	Bluetooth v2.1, High Speed Packet Access (HSPA) 3.5G, Universal Serial Bus (USB) 2.0, Wireless Local Area Network (WLAN) 802.11 b/g
Display	3.5” Wide Video Graphics Array (WVGA) touch-screen, 800 × 480 resolution
Size	110.9 mm × 59.8 mm × 18 mm
Other features	Global Positioning System (GPS), 5-megapixel digital camera, QWERTY keyboard

As of 2010 Nokia N900 is still a powerful and versatile mobile phone based on its specifications. It runs Maemo operating system which is Linux-based and derived from

Debian GNU/Linux. Lately Nokia has been developing MeeGo platform based on Maemo and Moblin by Intel. In addition to Maemo also MeeGo can be run on N900. JamMo can easily be built for MeeGo which is important for the future of JamMo.

2.4 Communication technologies

Available communications technologies of Nokia N900 include Bluetooth, 3.5G, USB and WLAN. Bluetooth and WLAN are introduced in this section as they are suitable technologies for the context of this thesis. USB was not considered feasible because it requires connecting devices with a cable while 3.5G was not regarded fitting for the purpose due to heavy infrastructure considering the scenario and high latency.

2.4.1 Bluetooth

Bluetooth is a short range communications technology developed by the Bluetooth Special Interest Group (SIG) which consists of thousands of telecommunications, computing and automotive companies, such as Ericsson, Intel, Lenovo, Microsoft, Motorola, Nokia and Toshiba. The original purpose of Bluetooth was to replace cables. It operates in the 2.4 GHz unlicensed Industrial, Scientific and Medical (ISM) radio range in 79 bands of 1 MHz. The bands are used with Frequency Hopping Spread Spectrum (FHSS) technology. [4]

Over the years different versions of Bluetooth have been released. The original specification was released in 1999. Subsequent version have improved many aspects, such as interoperability, data rates, resistance to interference and energy consumption. Version 2.0 specified Enhanced Data Rate (EDR) technology which reached 2 Mbps data rate whereas 2.1 improved the data rate to 3 Mbps. Moreover, version 3.0 specified up to 24 Mbps data rate using Institute of Electrical and Electronics Engineers (IEEE) 802.11 for data transfers. Version 4.0 added Bluetooth low energy which reduces power consumption on short ranges and low data rates. [4, 5]

Bluetooth consists of layers of protocols for different purposes. The protocols range from link management and control to service discovery and standardized interface between hosts

and controllers. Moreover, the protocols include more specialized protocols for serial port emulation, IP traffic, audio and video control and distribution and telephone control. [4, 6]

Bluetooth devices form piconets. In a piconet there is one master device and up to 7 active slave devices. Piconet devices use the same frequency hopping pattern which enables communication. Slave devices are only allowed to send when master polls them. Several piconets can be joined through slaves to form a larger, logical scatternet which enables communication between multiple piconets. Different piconets of a scatternet are not frequency synchronized. Furthermore, they do not use the same frequency hopping pattern. [5]

2.4.2 WLAN

WLAN was introduced as a wireless replacement for 802.3 Ethernet in 1997 by IEEE. WLAN has used 2.4 GHz and 5 GHz ISM radio ranges and infrared. Various modulation techniques such as FHSS, Direct Sequence Spread Spectrum (DSSS) and Orthogonal Frequency Division Multiplexing (OFDM) have been used in WLAN. Lately the use of Multiple-Input Multiple-Output (MIMO) has been introduced in WLAN to improve performance. [23, 55]

Several amendments have been released for 802.11. Some amendments specify modulation techniques whereas others enhance the technology otherwise, such as Quality of Service (QoS) support in 802.11e and security in 802.11i [22]. The original standard specified 1 and 2 Mbps data rates [12]. The latest modulation amendment 802.11n enables data rates up to 600 Mbps [55].

The Wireless Fidelity (Wi-Fi) alliance certificates WLAN devices that conform certain standards of interoperability. It was found in 1999 by 3Com, Aironet, Intersil, Lucent Technologies, Nokia and Symbol Technologies to improve interoperability. Today there are over 300 members in Wi-Fi alliance. [54]

WLAN utilizes the standard 802 Logical Link Control (LLC) and is indistinguishable from

wired 802 to higher networking layers. The issues related to wireless communication are taken care of on the Media Access Control (MAC) sublayer. [24]

3 REAL-TIME PAIR IMPROVISATION

In this chapter the real-time pair improvisation scenario presented in chapter 1 is defined in the viewpoint of technology. The used concepts are introduced, requirements elicited, constraints considered and finally, the scenario is discussed.

3.1 Terminology

The important terms in real-time pair improvisation are presented in Table 2.

Table 2: Terms in real-time pair improvisation

Term	**Definition**
Control data	Information, e.g. MIDI, needed to produce desired audio by sound synthesis.
Event	A unit of control data. The same as a MIDI message in MIDI protocol.
Local latency	The delay between receiving a local or network event and production of sound. Includes processing inputs from UI or network, audio processing and buffering latencies.
MIDI	A protocol for controlling musical hardware and software.
Network latency	The delay between sending and receiving an event on application layer.
Note-off event	An event that defines the ending of a particular musical note.
Note-on event	An event that defines the beginning of a particular musical note.
Sequencer	A software component that controls many tracks. Enables simultaneous playback of various audio sources.
Sound synthesis	" -- numerical algorithms that aim at producing musically interesting and preferably realistic sounds in real time." [52] Includes the use of existing audio files (sampling synthesis).
Total latency	The sum of network and local latency.
Track	A software component used in a sequencer responsible for producing and/or storing audio with a virtual instrument or audio files for example.
Virtual instrument	A software component that produces audio from control data.

3.2 Requirements

Real-time pair improvisation is an extension to standalone virtual instrument improvisation. There was implemented functionality for producing sound according to user actions and storing the created music. Also device discovery and group management were already implemented in the networking components. In terms of processing power simultaneous playback of two virtual instruments and a backing track was required. The most important new requirements were real-time communications and initializations, which were not already implemented in JamMo. The communications also required some processing power.

Improvisation is extremely latency critical. When playing a virtual instrument latency is the time between pressing a key or giving any other kind of an input and hearing a sound. It consists of UI latency, audio latency of the sound card and possibly various buffering latencies between the two. The term local latency is used in this thesis to distinguish it from network latency. In UMSIC local latency should have been less than 10 ms according to the requirements of the project. The musical components, however, were in active development during this thesis and it was unknown whether this requirement could be satisfied. However, it was assumed that 10 ms local latency is achievable on Nokia N900.

In real-time musical collaboration over a network there is always an additional network latency for remote peers. According to Zimmerman et al. a total latency as low as 15 ms is required for a rhythmically demanding piece and less than 50 ms is usually acceptable [57]. A shorter latency enables more complex performances [57]. As local latency consists of various processing and buffering operations it was assumed that 10 ms latencies were present in both communicating devices. Thus, network latency should have been at most 30 ms but as little as possible was targeted to provide the best possible performance.

Real-time pair improvisation can be implemented by transferring either audio or control data (such as MIDI) between peers. Both require a low latency network but bandwidth is significantly lower in control information transfer. Standard MIDI stream is 31.25 kb/s [37] whereas 44 100 Hz 16 bit mono raw audio is approximately 690 kb/s. On the one hand control data transfer requires less bandwidth which makes meeting the latency requirement

easier. On the other hand audio transfer has some other advantages. Both peers would not need to have the same audio material that is needed in sound reproduction. Also it would not be necessary to negotiate the instruments beforehand.

In JamMo a small logical subset of MIDI messages are used. The used data formats are slightly different in order to make implementation easier. In JamMo only Note-on and Note-off events are used. Note-on indicates that a specific note should start playing, whereas Note-off is the opposite. Excluded MIDI messages include control changes, synchronizations and many others [38].

As the number of messages is greatly reduced even smaller data rate than in MIDI protocol can provide similar performance. JamMo is intended for children not professional musicians which makes higher latencies acceptable. Moreover, the data rate can be reduced even more as long as the total latency requirement is met. With a 10 ms sending interval and 10 byte messages the bandwidth for application layer data would have been 1000 b/s. This number, however, does not include the protocol overhead of the lower networking layers which are unavoidable when using network connections instead of MIDI cables.

Typically in real-time applications, e.g. an audio or video stream, data is relevant only for a short period of time. In such cases a lost or corrupted packet affects the application only for a short period of time, e.g. a short missing video segment. In real-time improvisation implemented with control data transfer a lost or corrupted packet may ruin the whole performance. A single lost Note-off event may cause a note to play the full duration of the improvisation which alters the performance dramatically (see Figure 7). Furthermore, the data has to be processed in correct order. It is possible to get a stuck note also if later Note-off event is processed before the corresponding Note-on event due to network connection (see Figure 8). Thus, control data transfer has to be reliable.

In Figure 7 and Figure 8 circles represent received Note-on events and crosses Note-off events. A rectangle in Figure 7 is a lost Note-off event which is never received and causes a note to play the remaining duration of the improvised performance which was not intended.

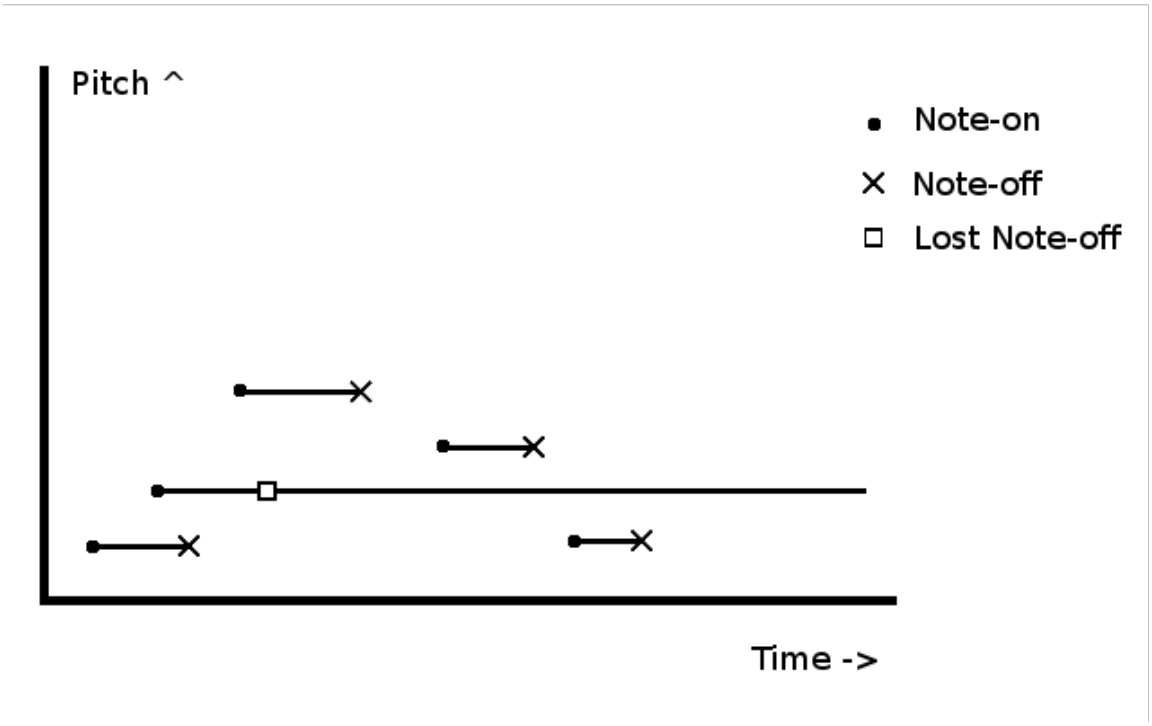

Figure 7: Lost Note-off event

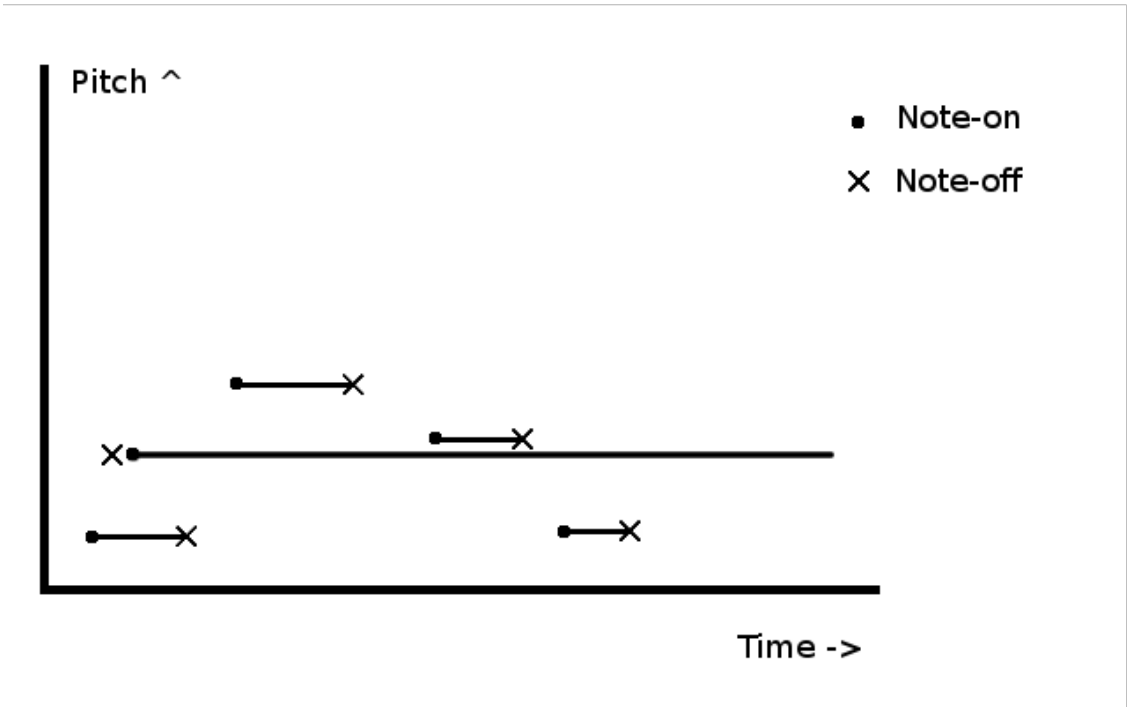

Figure 8: Note-on and Note-off events received in the wrong order

In pair improvisation implemented with audio transfer lost or out of order packets would not

be as critical as with control data transfer. The effect of a lost or out of order packet is limited to the duration of audio it contains, for example 10 ms. Out of order packets should be ordered when possible without causing subsequent packets to be delayed later than their playback time.

Needed initializations for pair improvisation include the selection of instruments and a backing track. In JamMo there are multiple instruments and backing tracks available and both devices have to produce the same audio. Although desirable, user interfaces for selections were not considered as important as means to negotiate instruments and backing track over a network connection.

The requirements for new components of real-time pair improvisation are summarized in Table 3. Some requirements are different for control data and audio transfer. Tested column shows in which part of this thesis the requirement was tested.

Table 3: Summary of requirements for real-time pair improvisation

ID	Requirement	Description	Tested
1	Processing power	Playback of two simultaneous virtual instruments and a backing track plus processing required for communications	See section 3.3 Constraints and 3.4 Discussion
2	Transfer of musical data	Musical data has to be transferred to peer over a network connection.	See section 4.3 Application layer
3	Bandwidth	Bandwidth for audio: 690 kb/s Bandwidth for control data: Max. 31 kb/s	See section 3.3 Constraints
4	Latency	Maximum 50 ms in total (including network latency, sound processing and playback) [57]. Network latency approximately 30 ms.	See section 3.3 Constraints
5	Reliability	Control data: Note-off event may not be lost. Events can not be received in wrong order. Audio data: Ordering of out of order packets without causing subsequent packets to be delayed.	See section 3.3 Constraints
6	Initializations	Means to negotiate instruments and a backing track over the network.	See section 6.1 Used components

3.3 Constraints

In the UMSIC project there were many constraints which were derived from Nokia N900 and Maemo operating system. Constraints of N900 include communication technologies and processing power whereas Maemo and selected libraries in the UMSIC project introduced software constraints. The implications of hardware and software constraints are discussed in this section.

3.3.1 Communication

Although WLAN and Bluetooth are widely used technologies neither has been used

extensively in transferring real-time musical information. Thus, it was important to find out whether real-time improvisation was possible with the given communication technologies. There were a few commercial wireless MIDI systems available [8, 10, 29, 34] but most companies did not specify the used technology. Kenton used Ultra High Frequency (UHF) [29]. Some research have been done in the field of MIDI over IP with wired connections [56, 57].

For real-time improvisation latency and bandwidth requirements had to be met. Both WLAN (802.11g 54 Mbps) and Bluetooth (2.1 3 Mbps) met the bandwidth requirements (see Requirement 3) [4, 20] which made latency the most important issue. Bluetooth, however, left less room for non-ideal throughput in audio transfer.

It has been measured that Bluetooth has an average latency of 20 ms with a protocol stack optimized for low latency [3]. The network jitter, however, was dependent on the piconet role of the sending node. For piconet slave the jitter was approximately 5 ms and for master 1 ms [3]. Moreover, the perceived maximum latencies were 25 to 70 ms and 22 to 27, respectively [3], highlighting the asymmetric nature of communications. The average latency was already majority of the maximum latency specified in Requirements section (3.2) leaving little room for network jitter and retransmissions of lost packets in time. Bluetooth has not been used in commercial MIDI systems. Kenton considered using Bluetooth in its wireless MIDI system but soon noticed it was not possible due to high latencies [29].

The asymmetric latency issue of Bluetooth is not present in WLAN. In Bluetooth a slave device has to be polled by the master in order to be able to send [3]. In WLAN, however, an optional Request To Send (RTS) / Clear To Send (CTS) procedure is used instead making the communications symmetric.

Less than 10 ms latency has been achieved in MIDI transfer over WLAN [35] which is less than the required latency (see Requirement 4). Streaming audio over WLAN had been done [13] but not in as latency critical setting as improvisation with a peer. When streaming audio for listening there is no interaction and thus, big buffers which create long latencies are acceptable and desirable to hide the unreliable nature of wireless communications. Real-time

improvising, however, is highly interactive as peers are creating music together in real-time.

A series of measurements were conducted to ensure the suitability of WLAN on N900 in pair improvisation. The purpose was to ensure that low enough latency can be achieved in application layer communications over WLAN and especially with Nokia N900s. For comparison computers were also used.

In the measurements two networking devices measured Round Trip Time (RTT) over WLAN. A WLAN router was used. One device was located approximately one and half meters away from the router and the other approximately five (see Figure 9). There was line of sight (LOS) propagation between one device and the router and no LOS between the other device and the router. No LOS propagation was selected to ensure that similar or better performance can be achieved in real application use.

The used devices were on default retransmission settings. RTS / CTS was used. The performance using default settings was interesting as the end users of JamMo are not expected to be able to configure the devices for optimal performance.

RTT measurements were performed using simple application layer client-server C programs. RTT performance was measured using User Datagram Protocol (UDP), Transmission Control Protocol (TCP) and TCP without Nagle's algorithm to rule out the effect of transport layer. Nagle's algorithm concatenates data packets when sending in order to reduce protocol overhead which may cause additional delays. Disabling Nagle's algorithm has been found to improve MIDI over Internet Protocol (IP) performance [56].

One thousand measurements were performed for each program in series of 100 measurements to even the changing amount of noise. There was a 5 ms delay between each measurement to simulate data streams. Measurements were performed sequentially and the latency of one measurement did not affect the others in any way as would have happened with a continuous stream of events. On the one hand it enabled accurate deviation measurements of packets. On the other hand it made the measurements slightly unrealistic. It was expected that the performance in real application use would be slightly worse due to

latency peaks which affect subsequent packets as well. Packet sizes on application layer were 11 bytes and 441 bytes to simulate control and audio data transfers respectively. The amount of data in audio packets is equal to 5 ms of 44.1 kHz 16-bit mono audio. Due to low data rates it was assumed that there is no need to send multiple packets at the same time.

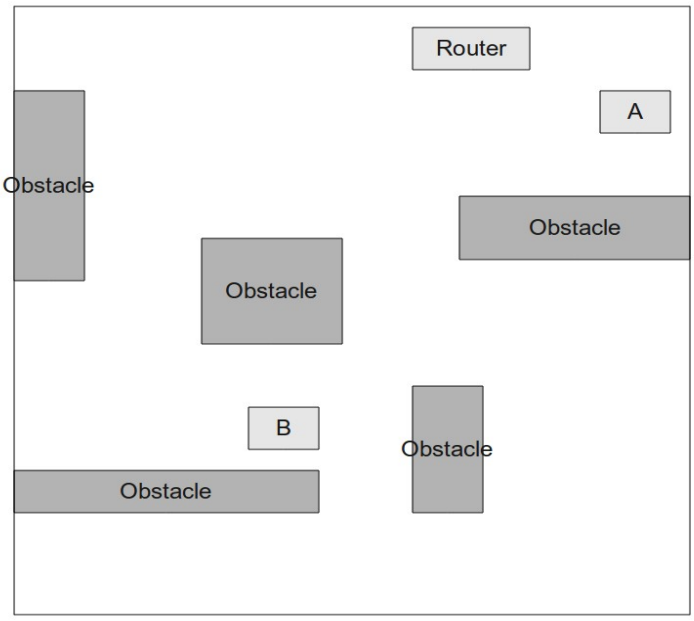

Figure 9: RTT measurement setup

The first measurements were performed using two Nokia N900's. WLAN power saving was on maximum on both devices. The results are presented in Table 4 and Table 5.

Table 4: Control data measurements on N900s with full power saving

	UDP	**TCP**	**TCP without Nagle**
Average (µs)	79793	194067	98207
Standard deviation (µs)	169214	271976	213934
Average absolute deviation (µs)	102078	247584	151438
Maximum (µs)	617157	615876	609436
Minimum (µs)	2411	2960	2808
Median (µs)	3662	4212	3662

Table 5: Audio data measurements on N900s with full power saving

	UDP	**TCP**	**TCP without Nagle**
Average (µs)	69769	199877	136797
Standard deviation (µs)	127399	270868	249332
Average absolute deviation (µs)	72262	246960	205748
Maximum (µs)	609375	610046	610168
Minimum (µs)	4028	4089	4090
Median (µs)	5554	5433	4791

Averages and deviations are significantly greater than 60 ms and show that it is impossible to improvise with two N900s with full WLAN power saving (see Requirement 4). Minimum and median values show correlation between packet size and latency. Averages, deviations and maximum values suffer from aggressive power saving. TCP without Nagle's algorithm performed better than TCP. The same measurements were conducted without the power saving. The results are presented in Table 6, Figure 10, Table 7 and Figure 11.

Table 6: Control data measurements on N900s without power saving

	UDP	TCP	TCP without Nagle
Average (µs)	3550	3815	3801
Standard deviation (µs)	2198	990	1676
Average absolute deviation (µs)	663	416	563
Maximum (µs)	44586	22217	27039
Minimum (µs)	2564	2929	2625
Median (µs)	3174	3601	3540

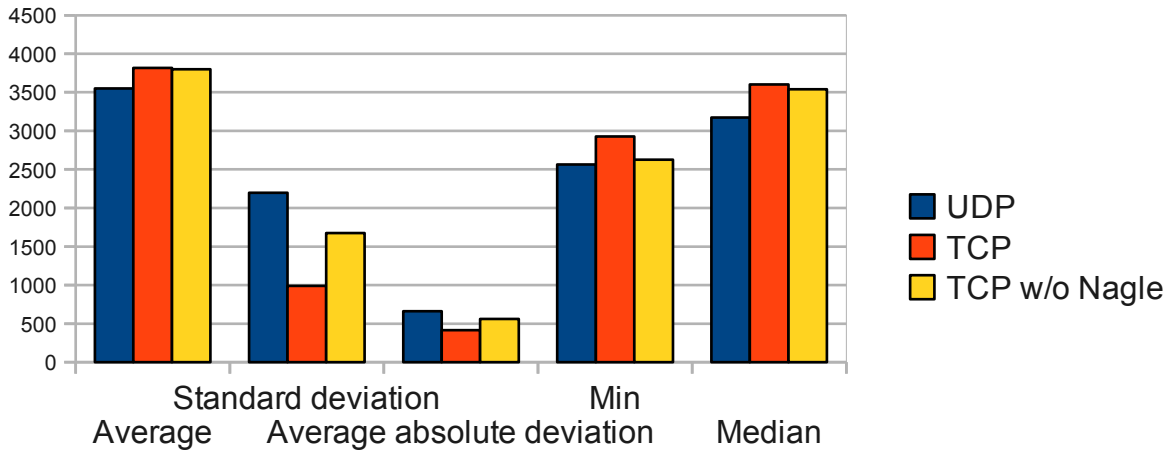

Figure 10: Control data measurements on N900s without power saving

Table 7: Audio data measurements on N900s without power saving

	UDP	TCP	TCP without Nagle
Average (µs)	4510	4977	5153
Standard deviation (µs)	1724	2346	2689
Average absolute deviation (µs)	499	566	922
Maximum (µs)	37140	53650	38361
Minimum (µs)	3693	3997	3876
Median (µs)	4242	4700	4669

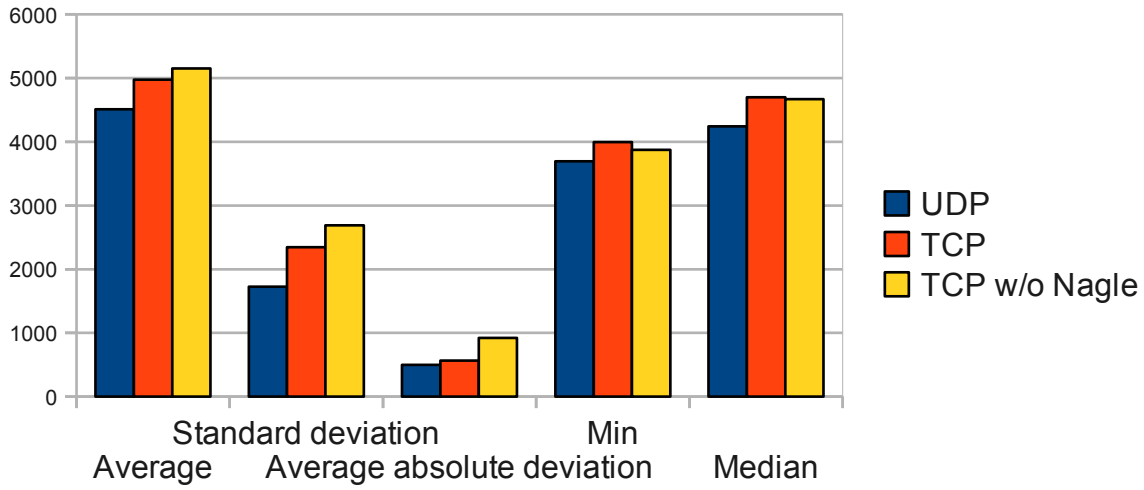

Figure 11: Audio data measurements on N900s without power saving

The results show great performance improvement in average and deviation times although minimum RTTs remained approximately the same. The assumption that few packets are waiting on the link layer to be transmitted in real application use seems to be correct. In addition to minimum and median times also averages show correlation between packet size and latency. TCP without Nagle's algorithm did not provide advantage over TCP unlike with power saving which could be due to link layer delays. Maximum times suggest that it may still be possible to encounter greater RTT than the required network latency (see Requirement 4) as almost double of the required latency was reached. It was assumed that temporary latency peaks can be tolerated in the scenario as long as average latency is less than required.

In general N900 is capable to achieve acceptable network latencies using WLAN. Without power saving, however, battery is used fast and the device gets warm. The drawbacks can not be fully avoided in the pursuit of satisfactory performance. Intermediate power saving option on N900 was not used in measurements as the device was already found capable of low latency communications.

For comparison the performance of a desktop and a laptop computer were measured in a similar setting. WLAN power saving was on default settings on both computers. The results are presented in Table 8, Figure 12, Table 9 and Figure 13.

Table 8: Control data measurements on two computers

	UDP	TCP	TCP without Nagle
Average (µs)	3240	3954	3684
Standard deviation (µs)	14339	18283	11033
Average absolute deviation (µs)	3571	4621	3920
Maximum (µs)	390726	386142	211954
Minimum (µs)	781	821	819
Median (µs)	922	948	1194

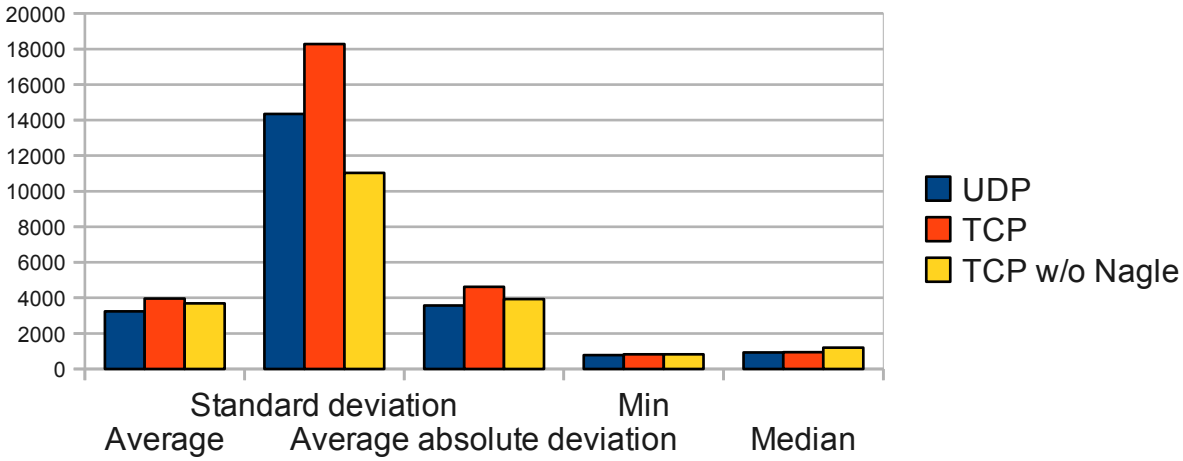

Figure 12: Control data measurements on two computers

Table 9: Audio data measurements on two computers

	UDP	TCP	TCP without Nagle
Average (µs)	5917	8315	8800
Standard deviation (µs)	3926	20709	22447
Average absolute deviation (µs)	913	4922	5781
Maximum (µs)	116187	230398	268531
Minimum (µs)	1176	1300	1175
Median (µs)	5667	5808	5807

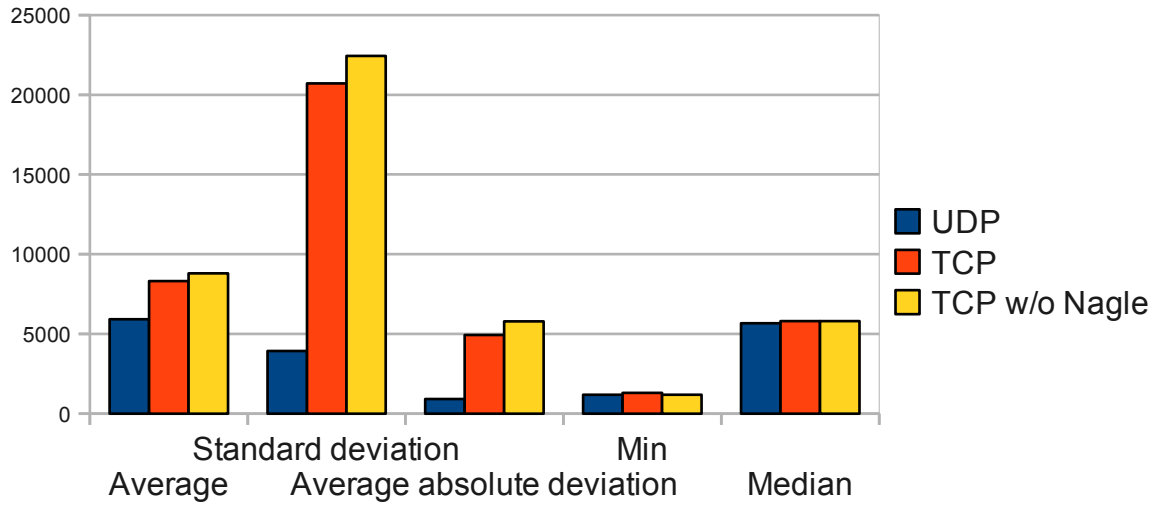

Figure 13: Audio data measurements on two computers

On Nokia N900 WLAN the minimum RTTs were higher than on computers. The difference could be due to unconfigurable power saving on hardware, WLAN chip quality or different link layer configurations.

The results show that in the used setting the performance of different transport layer protocols was relatively similar on N900 and the used computers. While UDP was slightly faster than TCP with or without Nagle's algorithm all protocols provided rather similar performance.

With TCP the deviations increased with the increased amount of data. With UDP, however, the deviations decreased which highlights the better performance over TCP in real-time applications. The improved deviations can be explained by increased averages, minimums and medians. Although the latencies were more stable they were higher in general.

Average latencies from all measurements are summarized in Table 10. The results show that the power saving of N900 is too aggressive to be used in real-time pair improvisation. Link layer performance is much more important than the selection of transport layer protocol. Without power saving and on computers there was a correlation between the amount of data and latency. Furthermore, TCP performed slightly worse than UDP and Nagle's algorithm

had little effect.

Table 10: Summary of average latencies

	UDP	**TCP**	**TCP without Nagle**
N900 power saving / Control data	79793	194067	98207
N900 power saving / Audio data	69769	199877	136797
N900 / Control data	3550	3815	3801
N900 / Audio data	4510	4977	5153
Computers / Control data	3240	3954	3684
Computers / Audio data	5917	8315	8800

3.3.2 N900, Maemo and libraries

In previous work by Gynther Nokia N810, the predecessor of N900, was found capable of running virtual instruments [20]. Due to different libraries in Maemo and JamMo the performance of N900 had to be evaluated. In addition to performance of N900 also the selected communications library added constraints.

Maemo 5 and selected libraries were a serious performance constraint. Measuring the actual effects were out of the scope of this thesis as the most processor intensive components were not developed in this thesis.

Improvisation requires a lot of processing power (see Requirement 1) which may be a problem with Nokia N900. MEAM components use GStreamer exclusively. GStreamer uses heavily threaded approach which degrades performance on Nokia N900 probably due to mutex implementation [11]. Especially achieving low latency was found difficult [11]. In the requirements elicitation of UMSIC context switching, e.g. a change of a thread or a process, on ARM processor architecture was found drastically slower compared to desktop computers.

In Maemo 5 PulseAudio sound server was added. In previous versions lightweight Advanced

Linux Sound Architecture (ALSA) library was used. PulseAudio is advanced but more complex and heavier than ALSA. PulseAudio supports simultaneous audio streams with different sample rates, networked sound and various other features whereas ALSA is a simple API for sound input and output. In low latency audio work on Linux Jack Audio Connection Kit (JACK) is typically used on top of ALSA. JACK can be run on Maemo but it requires disabling PulseAudio and it is not fully supported as there is no real-time scheduling in Maemo kernel. Disabling PulseAudio also disables the phone functionality of N900 which was not desired in UMSIC project. Furthermore, the use of JACK requires detailed audio configuration by the user.

Playing a virtual instrument was found a major task for N900. Measurements were done with a software tool which measures processor usage of processes (top). When playing a virtual instrument on N900 PulseAudio sound server used 15 to 30 % of the processing power. A simplistic test program running a GUI and a single virtual instrument consumed 35 to 40 % of processing power when playing calmly. Intense playing caused the sound to stop which was a sign of lack of processing power. Compared to results on N810 with Advanced Linux Sound Architecture (ALSA) library API instead of PulseAudio and GStreamer [20] the performance was dramatically worse. On N810 several simultaneous virtual instrument sounds were created using different algorithms [20] which is equivalent to running multiple simultaneous virtual instruments. The musical components of JamMo, however, were under intense development and it was unclear whether they would be more efficient. Moreover, the measured results showed that the processing power of N900 would not be insufficient in an order of magnitude. Slightly more powerful devices could meet processing power requirement (see Requirement 1) easily. The processing power of full JamMo when playing a virtual instrument could not be evaluated as the user interface was under development.

In addition to local processing also communications consume processing power. In providing low latency real-time communications cause a lot of system calls including small reads and writes. In addition the user space processing has to be done in small chunks of data. Processing small amounts of data is unavoidable in the pursuit of low latency.

In addition to processing power also the communications were constrained. PeerHood

supported only TCP connections which ruled out UDP based network traffic. In the previous section TCP was not seen as a problem for real-time pair improvisation. However, in a real data stream a delayed packet may cause significant delay in subsequent packets as well because TCP reorders the data.

3.3.3 Constraint discussion

The resources of Nokia N900 were found limited for real-time pair improvisation. Heavyweight software libraries are used in JamMo and the local processing left little resources for communications. In this thesis the musical components were not considered and thus it was not possible to free resources for communications.

During the constraints evaluation it was unclear if the performance of N900 would be sufficient. It was decided that if the processing power of N900 would not have been sufficient for real-time improvisation more powerful devices, e.g. desktop and laptop computers may have been utilized. Many JamMo components outside the scope of this thesis used a lot of processing power. It was also understood that N900s could have also been used to produce control data and audio could have been created with a more powerful device, such as a laptop. Fast context switching combined and greater processing power of a modern desktop or laptop it was clear that there would not be performance constraints on a computer. Using a different device instead of changing the scenario was favored.

WLAN was found more suitable than Bluetooth. WLAN provided better performance in bandwidth and in latency when power saving was not used. Although the use of Bluetooth was not found impossible it was considered more feasible to utilize only WLAN due to limited time. Requirement for latency (see Requirement 3) was met.

The transfer of control data provided better latency performance than audio due to smaller amount of data. TCP suffered more from the increased amount of data than UDP. However, acceptable latency was achieved with UDP and TCP protocols in audio and control data transfer. Nagle's algorithm should be used as disabling it failed to provide advantage on N900 without power saving. The use of PeerHood library constrained JamMo to use only

TCP. Although the performance of TCP was worse than UDP it was still acceptable.

Larger audio packets could be used on the expense of local latency. It was assumed that also network latency per packet would be increased with a larger packet size although there would be less packets in total. On the other hand there would be less protocol overhead. Network bandwidth, however, was not seen as a limited resource.

In the communications temporary latencies due to power saving or other matters should be taken into account. It should not be assumed that QoS is available in the used devices. Timestamping the sent data would provide the receiver valuable information. Based on this information late data can be discarded in order to avoid musical confusion. Furthermore, the improvisation could be stopped altogether should the network latency turn out to be unbearable.

A summary of constraints and their origins are presented in Table 11. How to overcome column indicates how the problematic constraints were overcome.

Table 11: Summary of constraints

ID	Constraint	Origin	How to overcome
1	Limited processing power	Hardware (performance not fully comparable to different architectures), Operating system, selected libraries	See section 3.4 Discussion
2	WLAN	Better performance than with Bluetooth. Lack of time to study Bluetooth.	Not problematic
3	No guarantee of low latency	Temporary latency peaks possible. QoS may not be available.	See section 4.3 Application layer
4	TCP	PeerHood	Not problematic

3.4 Discussion

The constraints of Nokia N900 presented in the previous section were partly conflicting with the requirements for JamMo. Moreover, the hardware and software constraints were partly conflicting as heavyweight libraries are used on a mobile device.

It was not clear if N900 would have been suitable for real-time pair improvisation in the terms of processing power. Furthermore, it was not clear if N900 could handle even the local processing of pair improvisation without any communications at all. As JamMo can be easily built and run on a computer, the most resource intensive components were not part of this thesis and the research questions focused on design and implementation using two computers was regarded as a back-up environment. Using computers was seen as a way to overcome Constraint 1 and to fulfill Requirement 1 without changing any other technologies or software.

The other constraints limited design and implementation but were not considered problematic. In order to provide understanding beyond the constraints also other transport layer protocols than TCP were discussed in design. Due to limited time Bluetooth or other link layer technologies were not considered.

4 COMMUNICATIONS DESIGN

In this chapter the effects of different protocols, design decisions and configurations are considered. The real-time requirements of pair improvisation had on effect on various networking layers of the TCP/IP protocol stack, see Figure 14. Moreover, different protocols and configurations affect other layers. Transport layer is discussed first as it affects higher and lower layers the most. Also algorithm for synchronizing the devices for improvisation is considered.

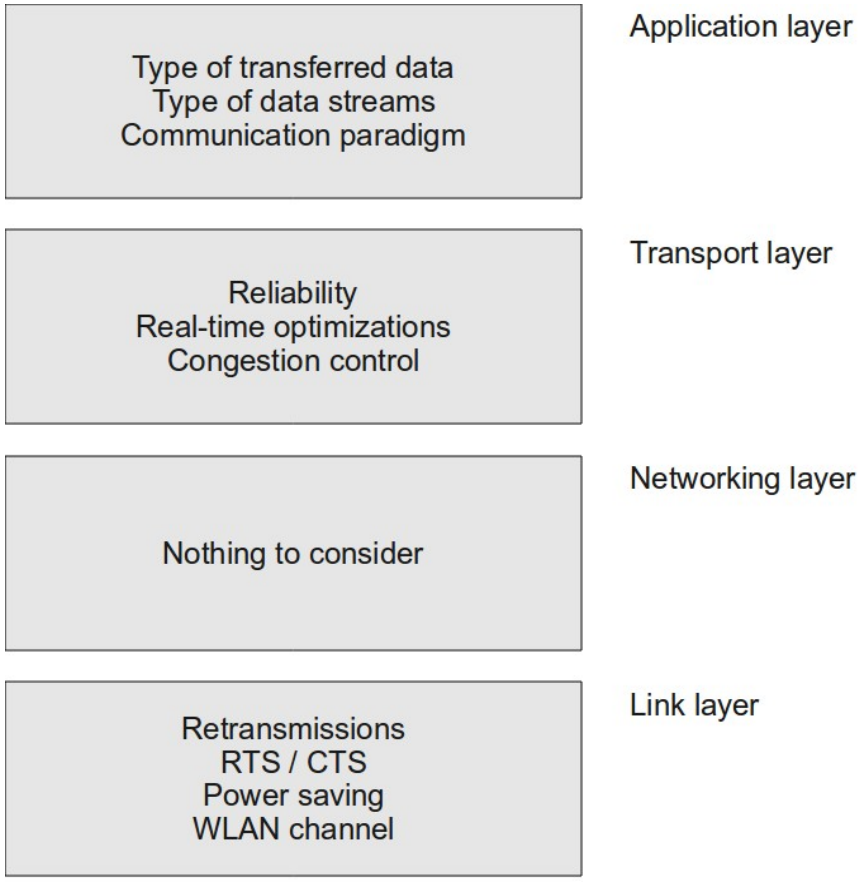

Figure 14: The effect of real-time requirements on different networking layers

Only WLAN was considered for the link layer due to limited time (see Constraint 2). As the link layer performance affects upper layers, the same constraint was also considered in transport and application layer sections.

4.1 Transport layer

The most important transport layer considerations were reliability, real-time optimizations and congestion control. There are various transport layer protocols available and they can be categorized into reliable and unreliable.

Although the implementation of real-time pair improvisation was constrained to the use of TCP (see Constraint 4) other protocols were considered as well. The evaluation of different transport layer protocols and their consequences helped to provide understanding of real-time communication and an answer to Research Question 1.

User Datagram Protocol (UDP) is unreliable which means that it works without guarantees of receiving in sending order or receiving at all. Any required reliability has to be implemented on the higher networking layers if UDP is used. Furthermore, UDP does not provide any congestion control. Since there is minimal overhead in UDP it can provide fast performance.

Transmission Control Protocol (TCP) is reliable meaning that the application layer will receive all data. The delivery is also ordered which means that data will be received in the sending order. If packets are lost on a lower layer any subsequent data can not be delivered to the application before the lost packets are transmitted successfully. Reordering may cause significant delays which can be harmful for real-time applications.

TCP slow-start algorithm used to avoid congestion causes problems in wireless networks. In wireless environments the assumption that packet loss is always caused by congestion is incorrect. Various radio effects cause temporary packet loss or connectivity problems which trigger the congestion avoidance algorithm. Variants for TCP congestion avoidance algorithm have been suggested. TCP Congestion Control introduced Fast Retransmit and Fast Recovery in which recovery from congestion avoidance caused by lost segments was faster than in the original algorithm [1]. TCP Selective Acknowledgement (SACK) made possible to specify which segments were received or not for more efficient retransmissions [36]. TCP New Reno specified a careful version of Fast Recovery in order to avoid multiple

retransmissions when TCP SACK is not available [14]. In TCP Westwood (TCPW) sender monitors acknowledgement stream from receiver to be able to distinguish congestion from random packet loss [7]. TCPW significantly increased throughput over a wireless link [7] but the effect on single segment latency was not measured. TCP SACK was used in the kernel of Maemo 5 by default. TCP Westwood was not available. Several other TCP congestion avoidance algorithms have also been suggested, such as TCP Hybla [51]. These were not considered in this thesis due to unsuitable assumptions in the algorithms, e.g. long RTTs over satellite links.

Although the effect of congestion control algorithm has a significant effect on throughput the effect on latency in pair improvisation is likely low. Bandwidth is sparsely used and RTT is low which mean that most of the time there are few packets in transmission. The congestion control algorithm, however, can make a difference when packets have been lost on the link layer and there are many packets to be sent. If retransmitted packets are lost randomly congestion control may seriously degrade the transfer speed. Measurements with different congestion control algorithms in pair improvisation were considered out of the scope of this thesis.

Transmission Control Protocol - Real Time Mode (TCP-RTM) is a modified version of TCP. It allows skipping late data which improves real-time performance but also makes it only partially reliable [33]. Thus, it is not immune to the lost Note-off event problem presented in Requirements chapter. It was not implemented in modern versions of Linux kernel which is why it could not have been used in the scope of this thesis. Moreover, implementing would have required changing the running environment of JamMo which was not desired in the UMSIC project.

Stream Control Transmission Protocol (SCTP) is a message oriented (like UDP) transport layer protocol [49]. It is reliable but can provide unordered data transfer to overcome the delays of TCP [49]. A single SCTP connection may be used to transfer multiple streams [49]. SCTP is also multi-homed, meaning packets can be sent to multiple destinations at once [49].

Datagram Congestion Control Protocol (DCCP) is a message oriented unreliable transport

layer protocol like UDP [30]. In contrast it provides two congestion control mechanisms: one is similar to original TCP congestion control and the other minimizes abrupt changes in sending rate [30].

Also application layer protocols implementing transport layer functionality, such as Real-time Transport Protocol (RTP), have been used on top of UDP and TCP. Typically RTP is used on top of UDP for better performance. Multiple RTP variants on top of TCP and TCP implementations have been studied in Local Area Networks (LAN) and Wide Area Networks (WAN) in transferring MIDI over IP [56]. The results show that a multi-threaded TCP implementation performed better than RTP on top of TCP or the default TCP implementation.

The presented transport layer protocols are compared in Table 12. More aspects of transport layer protocols need to be considered with control data traffic than with audio. SCTP would have suited control data transfer very well. It provides reliability, which diminishes the burden on application layer, and faster performance than TCP. Still, the order of packets would have had to be considered on application layer. SCTP, however, was not implemented in Linux kernel. If an unreliable transport layer protocol is used in control data transfer reliability has to be provided by the application layer. At least the lost Note-off and Note-on and Note-off in wrong order problems have to be considered. Also retransmissions of lost events should be considered when possible with a reasonable delay.

TCP would have suited audio transfer the worst. The other protocols do not always reorder data which allows the application to get the the latest data in time. TCP-RTM would reorder data that is received in time. In others it would have to be done on the application layer. With unreliable protocol retransmissions of lost data should be considered when possible within the playback time of the data.

Table 12: Comparison of transport layer protocols

Protocol	**Advantages**	**Disadvantages**
UDP	• Fast	• Unreliable, application layer has to provide the needed reliability
TCP	• Reliable • Data is ordered	• Slow, although there are differences between implementations • Ordering of data causes delays when a packet is lost • Congestion control may cause inefficient use of link layer
TCP-RTM	• Lost data does not cause delays to reading received data	• Partially reliable, late data may be lost. Application layer has to provide the needed reliability
SCTP	• Faster than TCP (reordering of data can be disabled) • Reliable	
DCCP	• Fast	• Unreliable, application layer has to provide the needed reliability

Transport layer protocol has a significant effect on lower and higher layers. Reordering and reliability in transport layer simplify application layer a lot although reordering may compromise performance. Reliability also affects optimal link layer configurations because discarding late data on the link layer will only lead to retransmissions on the transport layer.

4.2 Link layer

Various link layer settings have an effect on real-time performance. Retransmissions of data that is already late are not desired with an unreliable transport layer protocol. Request To Send (RTS) / Clear To Send (CTS) handshake increases latencies but helps in hidden node problems. WLAN power saving (see Constraints section 3.3) and bad selection of WLAN channel can compromise real-time performance.

Also changes to link layer implementations, such as packet proxies for TCP connections, have been suggested for improvements in throughput [2]. The proxies store packets and utilize selective retransmissions to improve performance over lossy wireless links [2]. The effect on latency is likely low in pair improvisation as there is supposed to be only one (if any) WLAN access point between the devices. Changes to link layer implementations, however, were considered out of the scope of this thesis.

As configuring the link layer requires root access in Linux JamMo can not make changes to configurations. When administrating the used network and communicating devices many optimizations can be done. With changes to configurations significant performance improvements can be achieved.

With an unreliable transport layer protocol retransmissions should be adjusted such that late packets are dropped. On some WLAN devices both retransmission count and time can be adjusted. Retransmissions could be adjusted according to application layer sending interval for optimal performance. With reliable protocols relatively high retransmission counts and times should be utilized to avoid transport layer timeouts because of temporary packet losses and lost connectivity. TCP timeouts use exponential back-off which may lead to long periods of inactivity on link layer even when the connectivity has been regained.

By default 802.11 utilizes RTS / CTS handshake to overcome hidden node problem. In the hidden node problem nodes that can not sense each other (e.g. A and C) try to send data to another node (e.g. B) at the same time. The receiving node can not receive data from either sending node because the radio signals get mixed on the shared transfer medium. RTS / CTS

mechanism is presented in Figure 15.

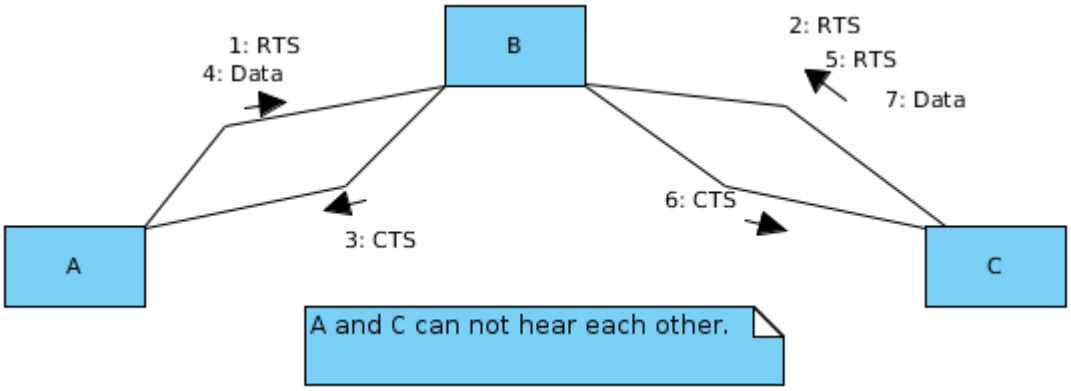

Figure 15: RTS / CTS mechanism

RTS / CTS handshake degrades performance when there are no hidden nodes but with the presence of hidden nodes RTS / CTS improves performance [27]. In real-time pair improvisation RTS / CTS should be turned off for better performance if the used network is dedicated to improvisation and the hidden node problem is unlikely. Otherwise RTS / CTS should be used.

The power saving of 802.11 has been found inefficient in throughput and power usage and other mechanisms have been suggested [28, 32]. In Constraints section (3.3) the WLAN power saving of N900 was found insufficient for pair improvisation. A short Beacon Period (BP) would improve the packet delay with 802.11 power saving [32]. At the start of each BP the devices wake up and decide whether they need to stay awake for the rest of the BP based on control communication with the other devices [28].

In 802.11g there are 13 channels in use in most parts of the world. One channel uses 20 MHz spectrum in 2.4 GHz range and the channels are overlapping. Only three of 13 channels can be used simultaneously without sharing any spectrum. The used channel should be selected such that there is as little interference as possible from other 802.11, Bluetooth, ZigBee and other devices. Interference has a significant effect on latency and throughput [19].

802.11e provides Quality of Service (QoS) which would improve latency performance

especially when there is other traffic in the same network. A high priority (voice) QoS setting should be used in real-time pair improvisation when possible.

4.3 Application layer

On application layer there are many things to consider. These include the type of transferred data, type of data streams and a communication paradigm meaning the model of roles that the two software instances play. Application layer synchronization, required in some circumstances, was separated as a section of its own for better readability.

Improvisation can be implemented by transferring control data or audio. Control data (such as MIDI) needs less bandwidth but lost and out of order data may cause problems (see Requirements section 3.2). It consists of information what kind of audio should be produced and when without any actual audio data. Thus, control data needs less bandwidth (see Requirements chapter for more details).

There are a few protocols and implementations available for transferring MIDI over IP networks. Real-time Transport Protocol (RTP) Payload Format for MIDI utilizes RTP for transferring MIDI [31]. It was published as an Internet Engineering Task Force (IETF) Request for Comments (RFC). RTP Payload Format for MIDI or RTP do not specify a means synchronizing two or more devices [31, 48] which is needed in pair improvisation. A commercial MIDI over IP solution called MIDIoverLAN Cross-platform (CP) exists for Windows and Macintosh Operating System X (Mac OSX) [39] and is available for purchase. NetJack [41] is able to transmit MIDI over LAN. In NetJack slave machines are synchronized to the audio clock of the master [41].

When utilizing an unreliable transport layer protocol a lost control event that stops a note from playing may ruin a performance completely (see Requirements chapter) whereas lost audio segment only causes short disturbance in audio stream. In some circumstances a lost audio segment can be effectively approximated based on other audio segments by an audio concealment algorithm [13].

When control data is transferred with an unreliable transport layer protocol application layer procedures are needed for required reliability. RTP Payload Format for MIDI addresses this issue by a recovery journal [31]. The recovery journal is sent in all packets and it contains the history of the stream since the previous checkpoint [31]. It would also be possible to utilize acknowledgements and retransmits for all or the most critical events. Acknowledgements and retransmissions would save bandwidth but the latency performance would be worse as there would be more transmissions over the wireless link.

Both control and audio data streams could utilize variable bit rates. With control data transfer empty events, signaling that the stream is alive but there is nothing to send, are necessary. Some empty events can be omitted on the expense of monitoring accuracy of the stream. In MIDI protocol, however, this is not specified. In audio transfer silence could be transferred with a special message omitting most of the data. Utilizing audio compression would also be possible. In this thesis it was assumed that audio compression and decompression would consume at least most of the latency gain on the wireless link. Moreover, compression and decompression would require processing power which was seen very limited in Constraints section (3.3).

Two models of roles which fulfill Requirement 2 were considered. In Master and Slave model master pulls the data from slave and sends the created audio stream to slave (see Figure 16). This way devices do not need to be synchronized before starting improvisation as they are both synchronized to the audio clock of Master. A similar model is used in NetJack [41], a networked extension of JACK Audio Connection Kit (JACK) sound server [26]. In NetJack, however, the slave machine is not playing audio through a sound card. Playback on slave could create noise due to differences in clock rates on master and slave. Local audio concealment techniques could be used to solve this problem.

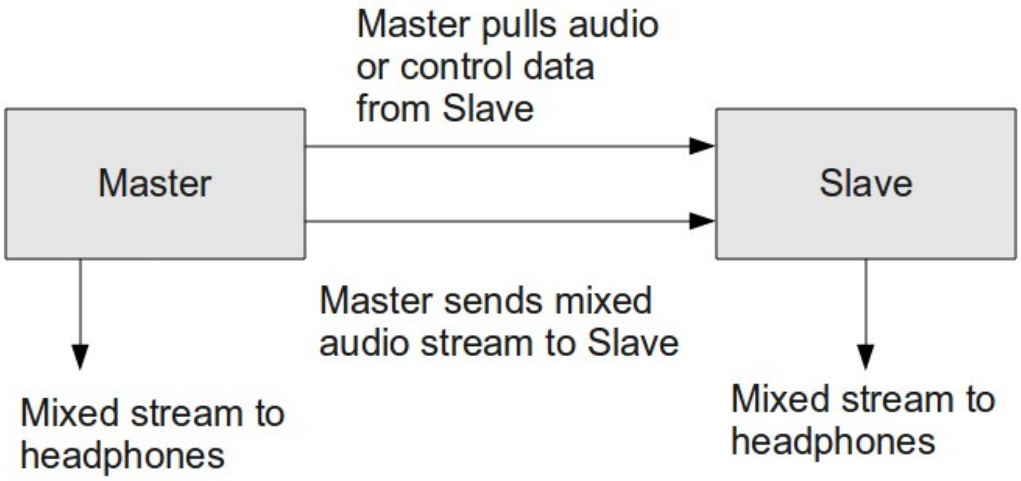

Figure 16: Master and Slave model

In Peer-to-Peer model both devices create their audio locally and only send control data or audio of their own track to the other peer (see Figure 17). This makes workload balanced and if using control data transfer also bandwidth requirement smaller. Furthermore, some network bandwidth is saved and latency avoided due to less hops over the wireless link.

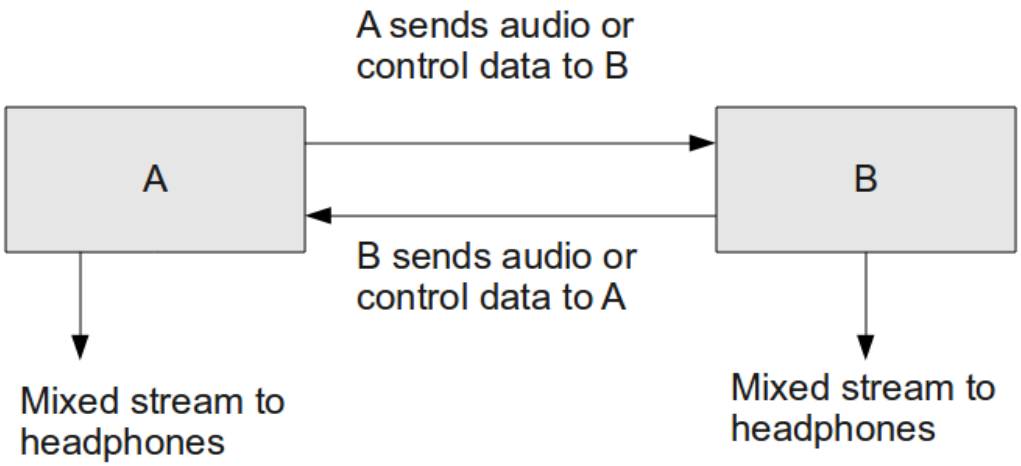

Figure 17: Peer-to-Peer model

The Master and Slave and Peer-to-Peer models presented in Figure 16 and Figure 17 are compared in Table 13.

Table 13: Comparison of models

Model	**Advantages**	**Disadvantages**
Master and Slave	• Synchronization not required	• Audio transfer required • Complicated communications • Master does more processing
Peer-to-Peer	• Simple communications • Control data transfer possible • Balanced workload	• Synchronization required

On the application layer it can not be assumed that link and transport layer configurations are optimal and data is always received shortly after sending. RTP Payload Format for MIDI addresses this issue by timestamping data [31] which gives valuable information to receiver. It is possible to receive application layer data much later than current playback time. In pair improvisation it may be better to skip the late data instead of e.g. playing a sound a second later than it was intended (see Constraint 3).

4.4 Synchronization

In Peer-to-Peer model presented in the previous section the devices used in pair improvisation must be synchronized before the performance may start. Synchronization provides a means to start playback at the same wall clock time on both devices.

Synchronization of clocks over IP networks is a well researched area. Simple and complex protocols, such as Daytime Protocol [45], Time Protocol [46] and Network Time Protocol

(NTP) [40], have been created. These protocols and definitions address synchronization of clocks over a multi-hop network with varying degree of precision. The synchronization is usually done gradually over a long period of time if high precision, such as one millisecond, is desired.

In pair improvisation, however, the needs for synchronization are different. Clock times on the devices do not need to be the same as long as playback starts at the same time with as high precision as possible. In addition the synchronization process should be fast.

Simultaneous playback can be started by letting one peer decide the start time relative to current time and informing the other by a message sent over a network. Peers are able to start playback at the same time if one peer is able to evaluate the network delay and compensate the starting time. With a short RTT compared to audio latency it is possible to neglect the effect of network delay completely.

NTP takes RTT into account but does not offer means to measure or evaluate one way delays. Thus, it was assumed that one way delays are symmetric. With the previous assumption synchronization can be done with Master and Slave roles as follows:

- Master measures RTT by sending a message to Slave and waiting for a reply.
- Measurement is done multiple times. If RTT is low enough for improvisation Master sends a message containing the relative start time, e.g. one second after the present time.
- Slave starts after the time Master specified.
- Master starts after the time specified plus RTT divided by two which is the assumed start time of Slave.

During improvisation remaining clock differences or clock skews can be corrected by adjusting the clocks. Adjustments to clock can be done by seeking the sequencer to the correct location.

4.5 Discussion

Due to limited time it was not possible to implement multiple options presented in this chapter. It was compulsory to select one combination of options for implementation.

In JamMo it was compulsory to use TCP on transport layer but without using PeerHood others would have been possible too. UDP or RTP would have suited better to the real-time nature of communication as all late data would not have been retransmitted automatically. DCCP would not have improved the performance of UDP as congestion would not have been an issue on low data rates. SCTP would have provided reliability and fast operation because of unordered delivery. As TCP had to be used there was no risk of lost events and Requirement 5 was satisfied. Furthermore, a RTP Payload for MIDI style recovery journal was not needed. Moreover, TCP also makes sure that there is no need to synchronize events with peer after improvisation for saving the performance.

TCP has not been used extensively in real-time applications because its performance is not optimal due to reordering of packets. Moreover, the performance of TCP over wireless connections has been found non-ideal. The measurements in Constraints section (3.3), however, show that in the scenario of this thesis the performance of TCP is not considerably weaker than UDP. Due to lack of time it was not possible to measure the performance of different congestion avoidance algorithms in pair improvisation. In Maemo 5 SACK is used by default and it was assumed feasible.

It is possible to develop real-time pair improvisation on top of TCP but it should be ensured that it is unlikely to end up in situations where TCP performs badly. This can be done by monitoring RTT before starting improvisation because packets lost on link layer will increase RTT. Still, it is possible to encounter performance problems because of TCP. Reordering of data may cause long delays in application level data stream if multiple packets are lost. To avoid hanging notes all playing notes should be turned off if nothing has been received in a while. It was assumed that silence is more user-friendly than hanging notes.

Peer-to-Peer model was preferred. It is simpler than Master and Slave and enables control

data transfer. NetJack, utilizing Master and Slave model, is intended at wired LANs where latencies are significantly lower than in wireless networks. Synchronization procedure, which was required in Peer-to-Peer model, was presented in the previous section.

Control data transfer was preferred as it was easier to implement in JamMo. The API of virtual instruments supported real-time input of control data but there was no audio streaming functionality implemented. Moreover, lower latency could be achieved with control data transfer as shown in Constraints section (3.3).

An existing MIDI over IP protocol should not be used. Firstly, there was no library implementation for RTP Payload Format for MIDI. Secondly, RTP over TCP performed worse than TCP on a study [56]. Thirdly, JamMo did not utilize MIDI standard. In a custom protocol it is possible to utilize the same data types as in JamMo to simplify implementation.

The idea of timestamping in RTP Payload for MIDI was found helpful. The timestamps could have been used in decision making. Although because of TCP there was no risk of receiving out of order data, timestamps could have been used in monitoring the data stream. It was considered more user-friendly not to play a note received much later than it was played on the peers device. Furthermore, major clock differences could have been detected when a timestamp ahead of the current clock time was received.

The summary of communications design is presented in Table 14.

Table 14: Summary of design decisions

Design aspect	**Decision**	**Justification**
Transport protocol	TCP	Constraint 4
Application layer model	Peer-to-Peer	Simpler than Master and Slave.
Type of transferred data	Control data, custom protocol	Easier to implement than audio data. Lower network latency.
Synchronization	Custom algorithm based on RTT measurements	Required for control data transfer. NTP does not offer a more sophisticated way to estimate network latency.

The same communications could have been used for other scenarios as well, such as All JamMos Real-time scenario. In the scenario N900s are used as controllers only. Audio is generated on a server and played back from speakers.

5 COMMUNICATION PROTOCOL

The protocol for real-time control data connections in pair improvisation is presented in this chapter. Messages, timers and constants are defined. Finally, the operation of the protocol is presented.

The protocol enables sending and receiving JammoMidi events between two peers. By sending JammoMidi playing of one peer can be transferred over a network. Playing is transferred as Note-on and Note-off events. Note-on indicates that user has started playing a sound, e.g. pressed a key of the keyboard on the touch screen of a mobile device. Note-off means that user has stopped playing a previously started sound, e.g. released a key of the keyboard.

There are two types of communication modes: one-way and two-way. In both modes there are two roles: sender and receiver. In one-way communication one entity acts as a sender and the other as a receiver. One-way communication can be utilized in using one device as a controller only, e.g. playing a virtual instrument on N900 and generating sound on a computer. In two-way communication both entities play the sender and receiver roles which is intended in real-time pair improvisation. Two-way communication enables more efficient stream monitoring and thus it should be implemented even when there is nothing to be sent to one direction.

The protocol is described in five sections. Details of messages (5.1), timers (5.2) and constants (5.3) are presented first. Operation (5.4) defines the operation of sender and receiver. Finally, the implications of different aspects of the protocol are considered in Discussion section (5.5).

The protocol is an application layer protocol intended to be built on top of TCP or other reliable protocol with ordered delivery of data. The communication is supposed to happen over a wireless link with few hops, e.g. two devices communicating through an access point or ad hoc. Due to the wireless link it is not assumed that data is always received without

disturbing delays.

5.1 Messages

In this section messages are defined on a high level of abstraction. Byte level encoding of messages presented in this section are defined in Appendix 1. The Error message presented in Table 15 is used in improvisation and synchronization phases.

Table 15: Error message

Message	**Description**	**Contents**
Error	There has been an unrecoverable error and connection should be closed.	Message type

The messages presented in Table 16 are used in synchronization phase of real-time connection.

Table 16: Synchronization messages

Message	**Description**	**Contents**
SyncInit	Slave contacts master.	Message type
SyncRTT	Master requests slave to send a SyncRTTreply message. Master measures RTT.	Message type
SyncRTTreply	Slave responds to SyncRTT from master.	Message type
SyncStart	Master specifies how many milliseconds slave should wait before synchronization is finished. Includes the number of milliseconds to the start time of Improvisation phase.	Message type Time
SyncStartOK	Slave responds to SyncStart from master.	Message type
SyncRTTaverage	Master reports the average RTT to slave. In future versions of this protocol this value could be used for calibrating timers and constants. Includes the average RTT in microseconds.	Message type Average RTT

The messages presented in Table 17 are used in real-time communication after synchronization.

Table 17: Messages after synchronization

Message	Description	Contents
Note-on	User has started playing a note. The message includes a note value similar to MIDI standard and a timestamp in nanoseconds.	Message type Note value Timestamp
Note-off	User has finished playing a note. The message includes a note value similar to MIDI standard and a timestamp in nanoseconds.	Message type Note value Timestamp
EmptyEvent	Empty events indicate that the data connection is working. The message includes a timestamp in nanoseconds.	Message type Timestamp
Resync	The clocks of peer do not appear to be synchronized. The difference is intolerable. A new synchronization is needed.	Message type
ClockDiff	The clocks of peer do not appear to be synchronized. The difference is tolerable. ClockDiff specifies the difference in milliseconds to enable more accurate stream monitoring.	Message type Clock Difference

5.2 Timers

The timers utilized in real-time connection protocol are presented in Table 18. Example values used in the implementation are specified in Appendix 2.

Table 18: Timers

Timer	Description
RT_DELAYQUIT	The time that connection is considered valid if no events are received.
RT_EMPTY_INTERVAL	The interval of sending Empty events.
RT_ENDTHRESHOLD	The time that needs to be waited after reaching the end of connection. During this time peers have are supposed to have sent all their events.
RT_MAXWAITFORSLAVE	The maximum time Master waits for slave to contact in synchronization.
RT_NOTEHANGDELAY	The time that connection is considered real-time if no events are received.
RT_SLAVETIMEOUT	The time that slave waits for messages from Master in synchronization.

5.3 Constants

The constants utilized in real-time connection protocol are presented in Table 19. Example values used in the implementation are specified in Appendix 3.

Table 19: Constants

Constant	Description
RT_DIFFSENDTHRESHOLD	The threshold value for correcting clock difference or sending clock difference to peer.
RT_LATESTSYNCPOINT	The point of performance after synchronization is not done anymore. The time of the latest synchronization point is calculated by multiplying the duration with RT_LATESTSYNCPOINT.
RT_LATETHRESHOLD	The time that Note-on events are considered valid.
RT_MAXSYNCHRONIZATION	The maximum number of synchronizations.
RT_RTT_COUNT	The times to monitor RTT before and after specifying the start time.
RT_RTT_THRESHOLD	The maximum acceptable RTT time in synchronization.
RT_SYNCTHRESHOLD	The maximum time peer clock is allowed to be ahead without considering the previous synchronization failed and requesting a new synchronization.

5.4 Operation

Operation is divided into three phases which are synchronization, improvisation and ending. Prior to synchronization initializations must have been carried out by out-of-band means. Initializations include connection creation and negotiations of used instruments and backing track. The used backing track specifies the duration of connection.

In this section many messages, timers and constants are mentioned but not discussed in detail. Refer to previous sections for definitions. Messages are defined in section 5.1. Timers and example values are specified in section 5.2 whereas Constants and example values are defined in section 5.3. Any other received data than the specified messages should be discarded.

During improvisation sender sends timestamped events to receiver. Timestamps have two purposes. Firstly, they indicate the original time of the event which is needed in storing the playing. Secondly, timestamps are used in monitoring networking latency. For the second purpose the clock difference of peers is important information. Although the devices are synchronized it can not be assumed that the clocks are perfectly synchronized.

Several variables are needed. Required variables are presented in Table 20. Other variables may also be needed depending on implementation.

Table 20: Required variables

Variable	**Purpose**
Diff	To measure the current clock difference of peer. Only peer ahead difference is measured because it can be distinguished from the network latency. If clock difference has changed more than RT_DIFFSENDTHRESHOLD from the previous sent value it should be sent to peer as a ClockDiff message.
PeerDiff	To calculate the network latency from timestamps and clock. The value is updated whenever a ClockDiff message is received from peer.
PreviousDiff	To notice changes in clock difference. Used in comparison with *Diff* in decision whether to send the difference to peer. The latest sent difference should be stored in this variable.
RecvTime	To monitor the data stream. The time of received message during Improvisation is stored to this variable.
SyncFails	To count the number of failed synchronizations.

Synchronization and Improvisation phases use messages of their own with the exception of common Error message. Whenever Error is received communication should be ended and the connection terminated.

5.4.1 Synchronization

Before actual improvisation devices need to synchronize to ensure that improvisation will start at the same time on both devices. Synchronization is based on RTT measurements and there are two roles: master and slave. Master measures RTT and if RTT is feasible (less than RT_RTT_THRESHOLD) it should tell slave when to start. Start time is specified in milliseconds from the current time. Slave should start when specified and master tries to estimate the start time of slave. The starting time specified should be regarded as the actual starting time. If starting the playback takes time due to implementation it should be taken into account.

While waiting for the starting time master should measure RTT to make sure that it is still feasible. If RTT is more than RT_RTT_THRESHOLD master should cancel the improvisation by sending Error message. A successful synchronization procedure is presented in Figure 18.

The roles for synchronization are selected as follows. For the first synchronization the roles are selected by out-of-band means. In later synchronizations the peer that requested synchronization is master and the other is slave.

It is possible that synchronization fails. Whenever a synchronization fails the value of *SyncFails* should be incremented. Slave may fail to send a proper message or it may take too long. If improper messages are received synchronization should be considered failed. Synchronization should be tried again with the same roles if it fails. During the whole connection synchronization should be tried at least RT_MAXSYNCHRONIZATION times if synchronizations keep failing. If RT_MAXSYNCHRONIZATION is reached Error message should be sent, communication ended and connection closed.

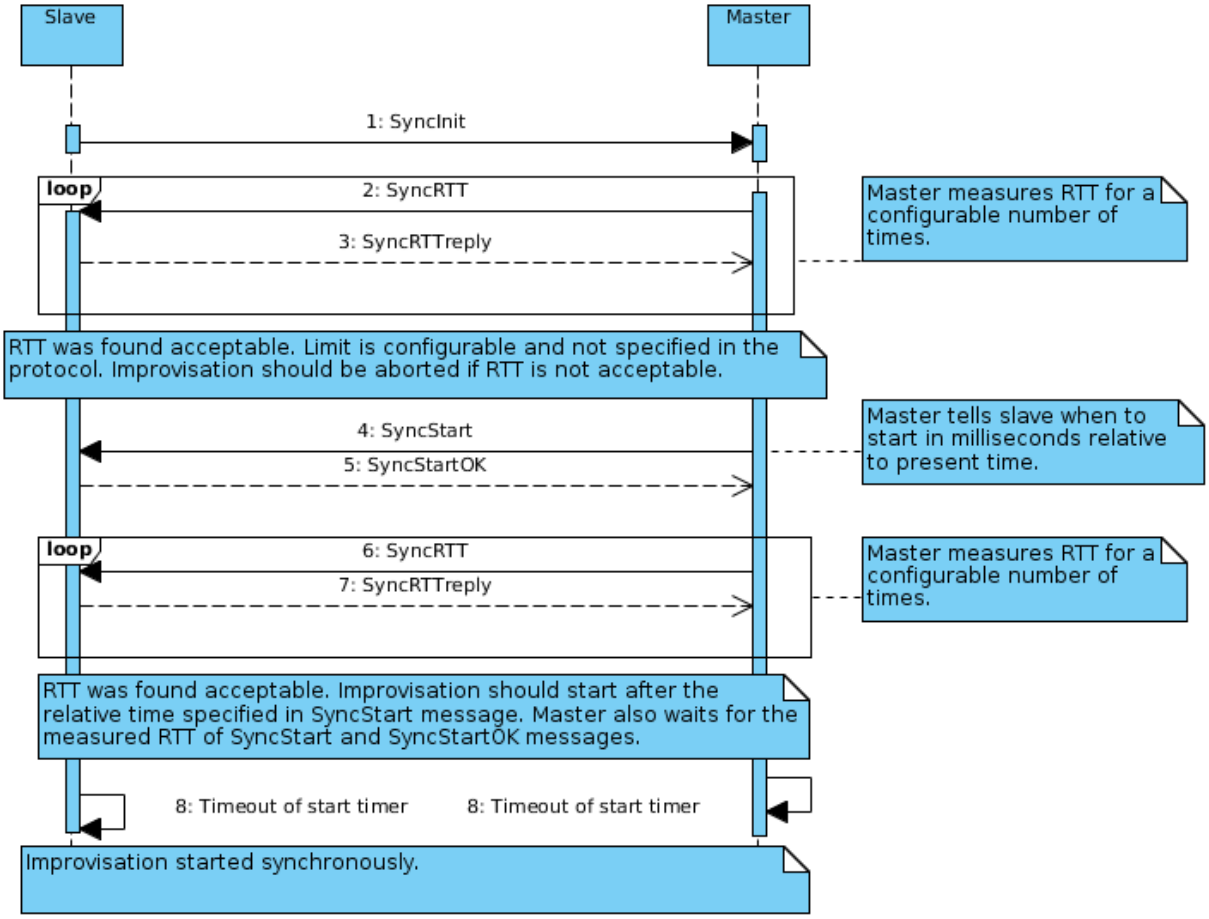

Figure 18: Sequence diagram of synchronization

The state machine of master node in synchronization is presented in Figure 19. In the figure states and messages are explained. For better readability all timeouts are not included. If no message is received in RT_MAXWAITFORSLAVE the synchronization should be considered failed and synchronization procedure started again from the beginning.

In Count state Master approximates the starting time of slave by adding half of the RTT of SyncStart and SyncStartOK messages to count time. If any other message is received it should be ignored and synchronization considered failed. The synchronization procedure, however, does not need to be started again.

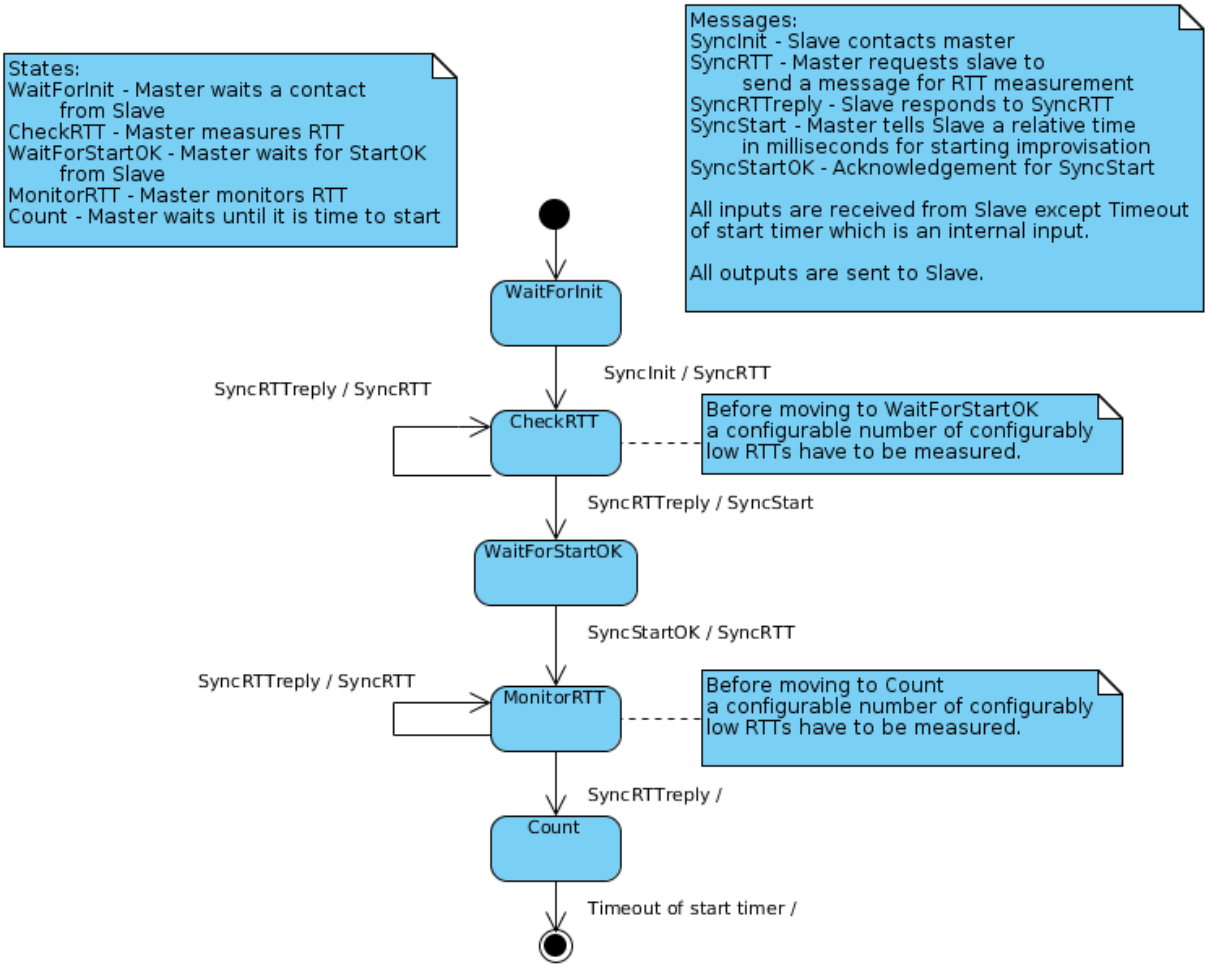

Figure 19: State machine of master node in synchronization

The state machine of slave node in synchronization is presented in Figure 20. Similarly to state machine of master node all timeouts are not presented. If no message is received in RT_SLAVETIMEOUT the synchronization should be considered failed and started again from the beginning.

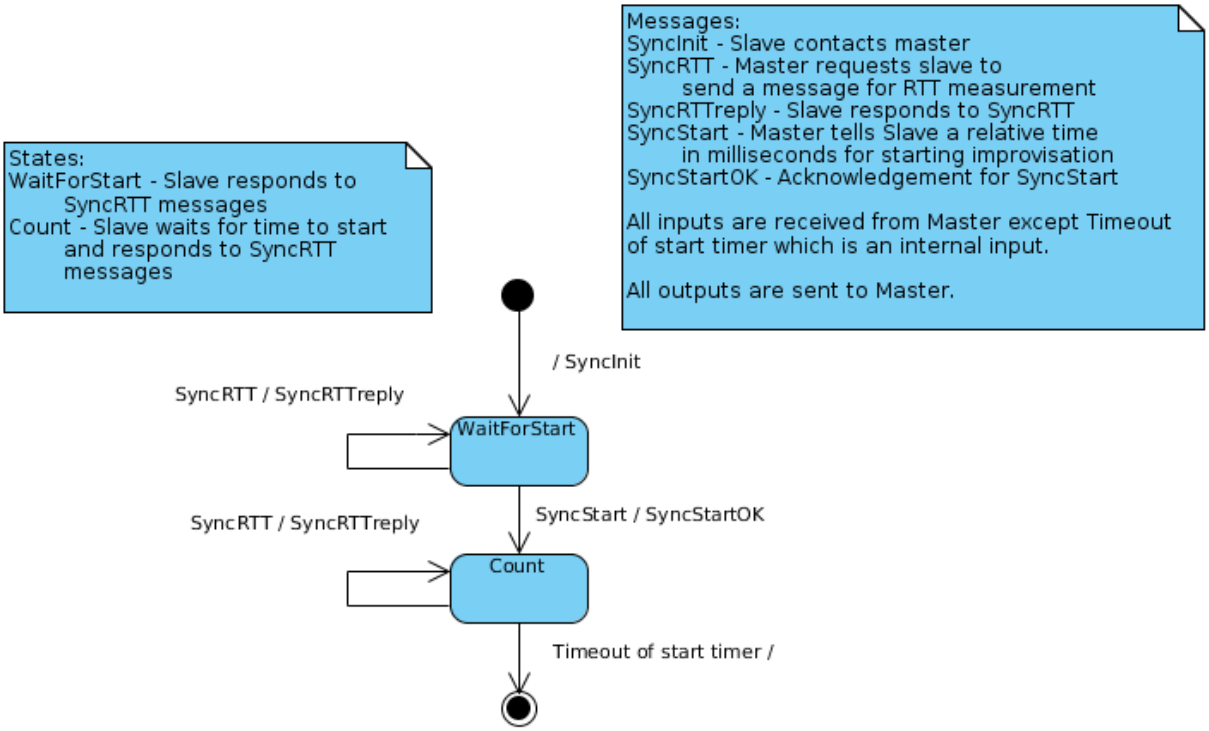

Figure 20: State machine of slave node in synchronization

If any other data than the messages specified for synchronization are received the data should be discarded and the synchronization considered failed. Any other received message should be ignored and synchronization considered failed. The synchronization procedure, however, does not need to be started again.

Synchronization should also be considered failed if desired precision is not achieved. Clock differences are monitored in Improvisation phase (see the next section).

5.4.2 Improvisation

During improvisation Note-on and Note-off events are sent according to playing of users. In one-way only one device sends. In two-way both devices send. Events should be sent immediately when asked to be sent to avoid any excess delays. EmptyEvents are needed for discovery of any possible network problems, such as temporary latency. EmptyEvents should be sent periodically after RT_EMPTY_INTERVAL of previous sent event. All sent and received Note-on and Note-off events should be stored for later editing and listening.

Peers need to have a clock. Sender needs it for timestamping events. Sent timestamps should always be the current clock value. Receiver needs the clock for discovery of networking latency and synchronization issues. The clock should be started from 0 immediately after synchronization has finished. Precision of one microsecond is preferred but not required.

Six different messages are used during improvisation. The reaction to each message is different. The reactions to different received messages are presented in Table 21. Any message specifies a common reaction and is not a message of its own.

Table 21: Reactions to received messages

Received message	**Reaction**
Any message	The local clock time of receiving should be stored. This makes monitoring the stream possible.
Note-on	The receiving time should be compared to the sum of timestamp of the received message, *PeerDiff* variable and RT_LATETHRESHOLD. If receiving time is smaller or equal the event should be passed to virtual instrument.
Note-off	The event should be passed to virtual instrument.
EmptyEvent	No need to do anything message specific.
Resync	A new synchronization should be done. Receiving peer is slave in synchronization.
ClockDiff	*PeerDiff* variable should be updated according to the value in the received message.
Error	Communication should be ended and the connection terminated.

Receiver should monitor the connection (see Figure 21). It should store the receiving time of last received event. If receiver does not receive any events in RT_NOTEHANGDELAY it should assume that there is a problem in communications and should turn all playing notes off to avoid stuck notes. If receiver does not receive any events in RT_DELAYQUIT an error

message should be sent and connection closed, In one-way communication it is not possible to send an error message to peer.

In all figures of this section event is either a Note-on, a Note-off or an EmptyEvent. The actual type is not specified when the type does not make a difference.

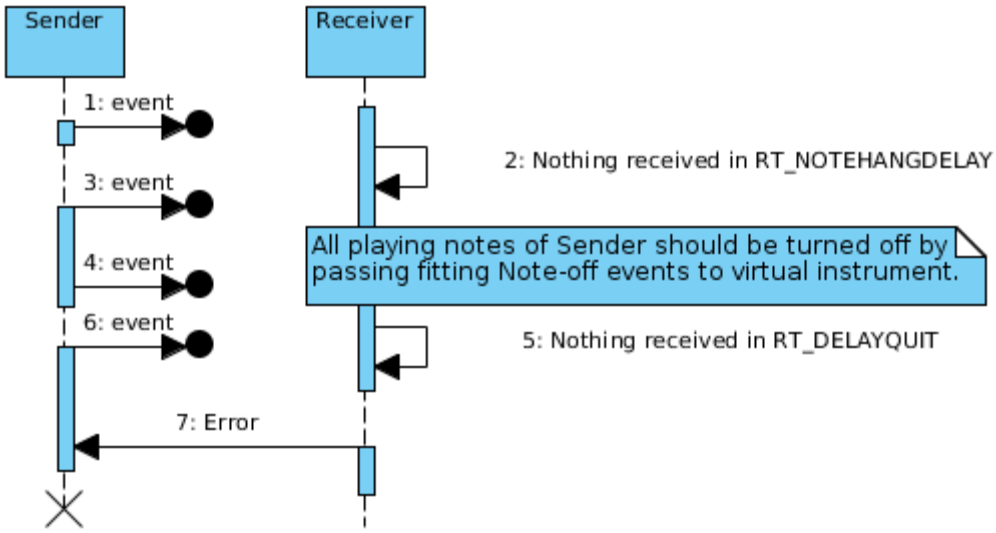

Figure 21: Monitoring communication errors

Receiver should pass all received Note-off events to a virtual instrument. Note-on events that have an earlier timestamp than current time plus *PeerDiff* plus RT_LATETHRESHOLD should be passed to a virtual instrument (see Pseudo code 1). Note that all values should be converted to the same time unit. Timestamp is sent as nanoseconds and clock difference in milliseconds.

Pseudo code 1:

```
if clock <= timestamp + PeerDiff + RT_LATETHRESHOLD then
     pass Note-on to virtual instrument
else
     do not pass Note-on to virtual instrument
```

Resynchronization should be requested if it is apparent that peers are not synchronized well

enough and it is not possible to change the clock time. Synchronization problems can be discovered by comparing the timestamps of received events and the current clock time. If the timestamp value is ahead more than RT_SYNCTHRESHOLD the previous synchronization should be considered failed, the value of *SyncFails* incremented and a new synchronization requested with a Resync message (see Figure 22). After synchronization the improvisation should be started from the beginning.

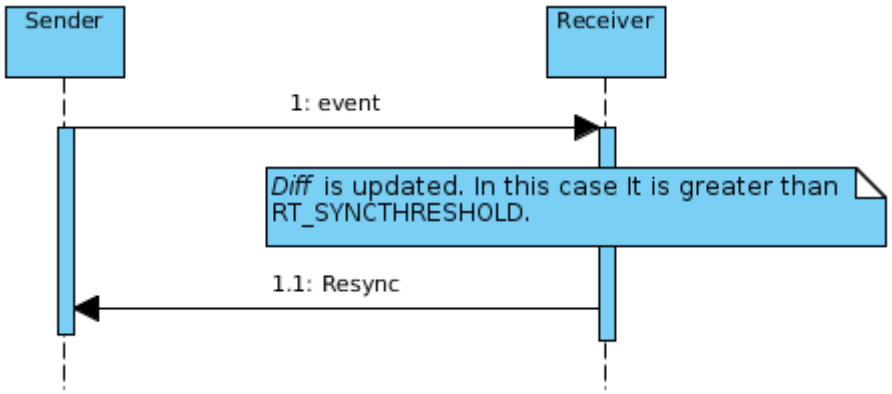

Figure 22: Requesting a new synchronization

If the timestamp value of a received message is ahead less than RT_SYNCTHRESHOLD the clock time should be changed to the timestamp value. If possible the backing track should be seeked to the timestamp value. If seeking is not possible the difference of the received timestamp and the clock should be stored to the *Diff* variable. When a new value for *Diff* is calculated it should be compared to *PreviousDiff*. If the absolute difference of these variables is more than RT_DIFFSENDTHRESHOLD is should be sent to peer as a ClockDiff message (see Figure 23). The sent value should be stored to *PreviousDiff* variable.

In one way communication the discovery of synchronization issues is not always possible. It is impossible to distinguish clock differences from network delay when the clock of receiver is ahead.

A new synchronization should not be requested after the time of RT_LATESTSYNCPOINT. If a Resync message is received after the time of RT_LATESTSYNCPOINT an error message should be sent and connection closed.

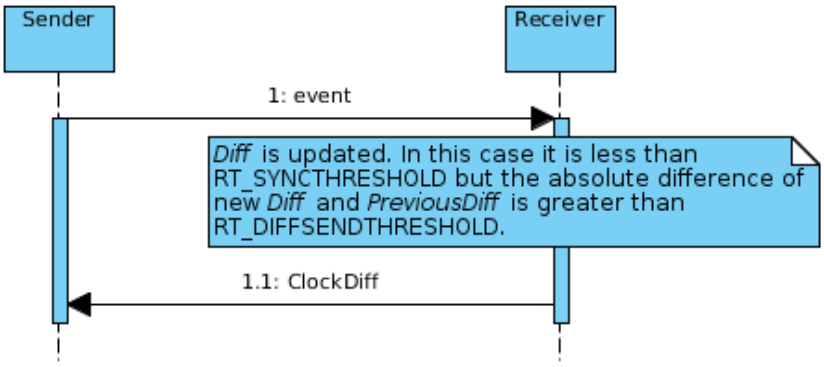

Figure 23: Sending clock difference to peer

5.4.3 Ending

After pair-improvisation is over due to finished duration or an error sender should stop sending events. To indicate finishing sender should send one Empty event with a greater timestamp than the duration of improvisation. When all events have been received receiver may close connection. Finally all playing notes of sender should be switched off by receiver with Note-off events. In one-way communication sender should wait for RT_ENDTHRESHOLD after finishing the improvisation.

5.5 Discussion

Timeout and constants values were not specified in this protocol. They have to be tailored for each scenario. Example values were given to illustrate the magnitude of suitable values. Future work of this thesis includes finding out if there are values that perform better than others. It would also be important to study what values work in what kind of situations. Evaluations with actual users would provide important information about the values for timers and constants.

Fixed length packets waste some bandwidth but make implementation easier. Due to lower layer protocols the excess overhead is not significant, e.g Internet Protocol (IP) version 4 header is at least 20 bytes long [25] and there are more headers on other layers.

Resynchronization is not possible in one-way communication. It is possible to implement two-way communication with only empty events in one direction if possibility to discover synchronization issues and resynchronization are needed.

6 IMPLEMENTATION

In this chapter the implementation of real-time pair improvisation in JamMo is considered. Source code of the implementation is released under GPL version 2. It is available in a fork of JamMo at Gitorious [21].

Real-time pair improvisation required implementation of three new components (see Figure 24). JammoCollaborationRTpairImprovisation component is a GObject class which controls the used components in pair improvisation. JammoCollaborationGameSelection is a simple menu for creating and joining pair improvisations. It consists of a C code file and a JavaScript Object Notation (JSON) file that defines the user interface. Gems_rt_communication implements the communication protocol presented in the previous chapter.

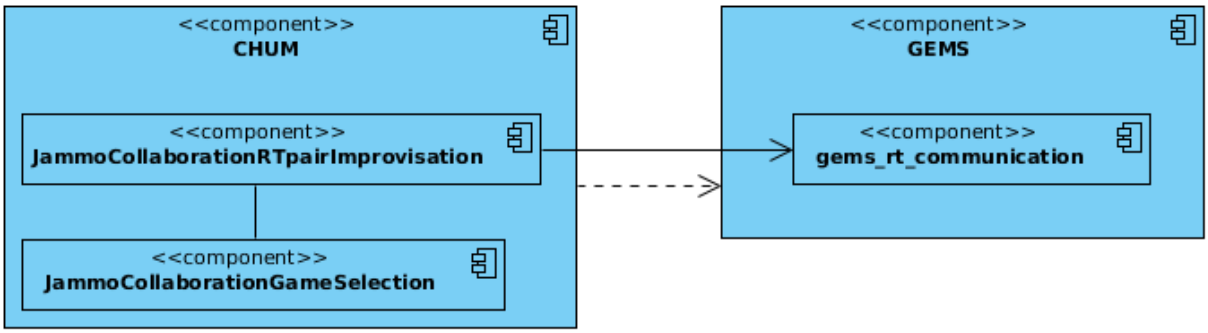

Figure 24: Implemented components

The structure of the chapter is the following. Firstly, the components used in real-time pair improvisation are presented. Secondly, the implemented components are described. Thirdly, the performance of the implementation is considered. Finally, the implementation is discussed.

6.1 Used components

Real-time pair improvisation used many other components in addition to implemented

components (see Figure 25). The used components include musical components, GUI and the main networking components of JamMo. MEAM components were required for audio production. CHUM components are related to local application logic and UI. GEMS components were utilized in creating and joining a group, creating the dedicated PeerHood connection and initializations. To keep Figure 25 simple relationships between used components are not visible. The purpose is to illustrate what components are used in the implementation. White components were created for pair improvisation and dark gray components were modified.

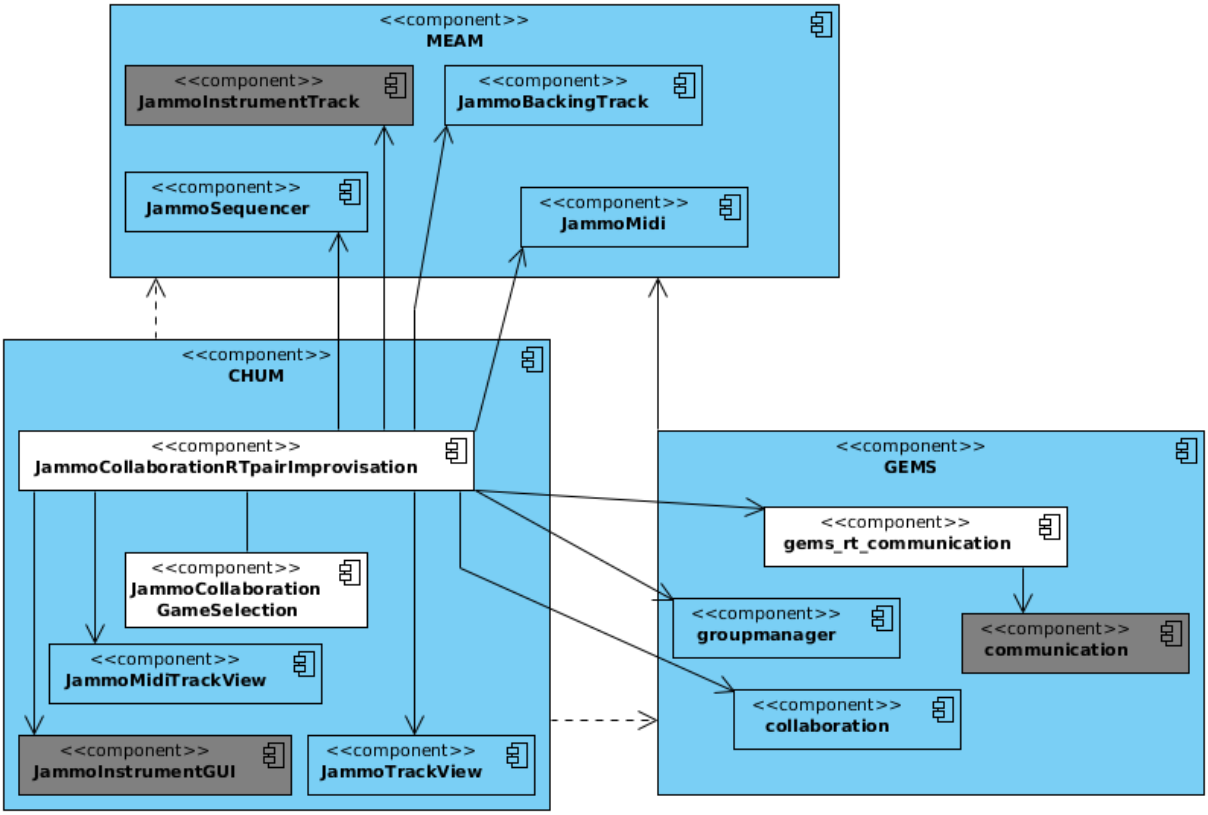

Figure 25: Used components

The used CHUM components consist of JammoMiditrackView, JammoInstrumentGUI and JammoTrackView. JammoMidiTrackView is an abstraction of a virtual instrument track controlling a GUI element and a musical component. It is derived from JammoTrackView which is also needed because base class functionality is used. Track views are required by JammoInstrumentGUI which is the GUI for playing a virtual instrument. Depending on the used instrument JammoInstrumentGUI shows a piano keyboard or drum pads. Some

modifications were needed in JammoInstrumentGUI. Many buttons that normally are used for changing views, e.g. to MIDI editor, were disabled in pair improvisation mode.

MEAM components consists of JammoSequencer, JammoInstrumentTrack, JammoBackingTrack and JammoMidi. JammoSequencer controls all the tracks. JammoInstrumentTrack controls a virtual instrument. JammoBackingTrack plays the selected backing track. JammoMidi contains a C language structure representing an event and functions for using lists of events.

Slight modifications were also needed to a MEAM component. A callback system was added to JammoInstrumentTrack which allows reporting of JammoMidi events to JammoCollaborationRTpairImprovisation. These events are then stored to a list for file creation and sent to peer. Direct function calls to CHUM were not available because MEAM can not see CHUM component.

GEMS components consist of groupmanager, collaboration and communication. Groupmanager is responsible for creating and joining groups which are needed in all collaboration games in JamMo. Collaboration handles messages and callbacks needed in collaboration games. Communication is responsible for creating connections between the devices. The functionality to create a dedicated connection for real-time data was added to it. When a group for real-time pair improvisation has been created both devices try to create a new PeerHood connection to each other utilizing a timed function. Before each new connection attempt it is checked if a connection initiated by peer already exists. This way creating multiple connections is avoided.

Initial information needed in pair improvisation (see Requirement 6) were transferred in a normal GEMS message called Song Info prior to creating a dedicated connection. These initializations included the types of instruments and the location of the backing track. The instruments were specified using the same integer values as in MEAM. Backing track was specified with the relative location of the file to the audio folder. Thus, it is required to have the same data files available which was the situation during this thesis. Data files are downloaded during the installation of JamMo and different versions refuse to communicate.

Moreover, it was not possible to create new backing tracks in JamMo yet. The format for the Song Info message was not final because it was not specified by the time of implementation.

During improvisation the playing of a child is transferred to peer as single events. In the following explanation devices are called A and B. A child starts pressing a key of the keyboard on A (see Figure 26). JammoInstrumentGUI (GUI in Figure 27) registers the pressing and informs JammoInstrumentTrack (InstrTrack) of it. JammoInstrumentTrack passes a Note-on event to JammoCollaborationRTpairImprovisation (RTpairImpro). Next, the event is passed to gems_rt_communication (gems_rt_com) which sends the event to B over a network connection. Gems_rt_communication component receives the event from A and passes the event to JammoCollaborationRTpairImprovisation which in turn passes it to JammoInstrumentTrack. Finally, the event is passed to a virtual instrument (Instr) for playback. This sequence is illustrated in Figure 27. On one device events are passed from one software component to another using function calls. In addition to sending an event to B the sound is also played on A which is omitted from Figure 27 for readability. Note-off events are transferred similarly when a key is released.

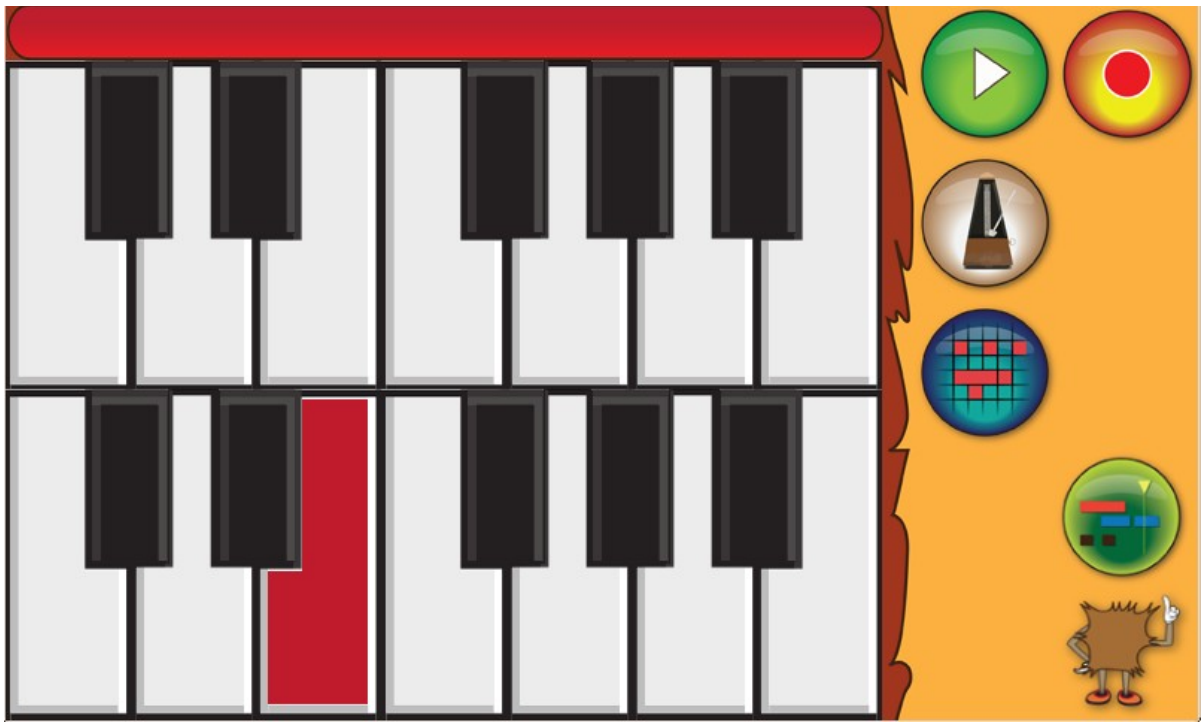

Figure 26: Playing a sound in improvisation

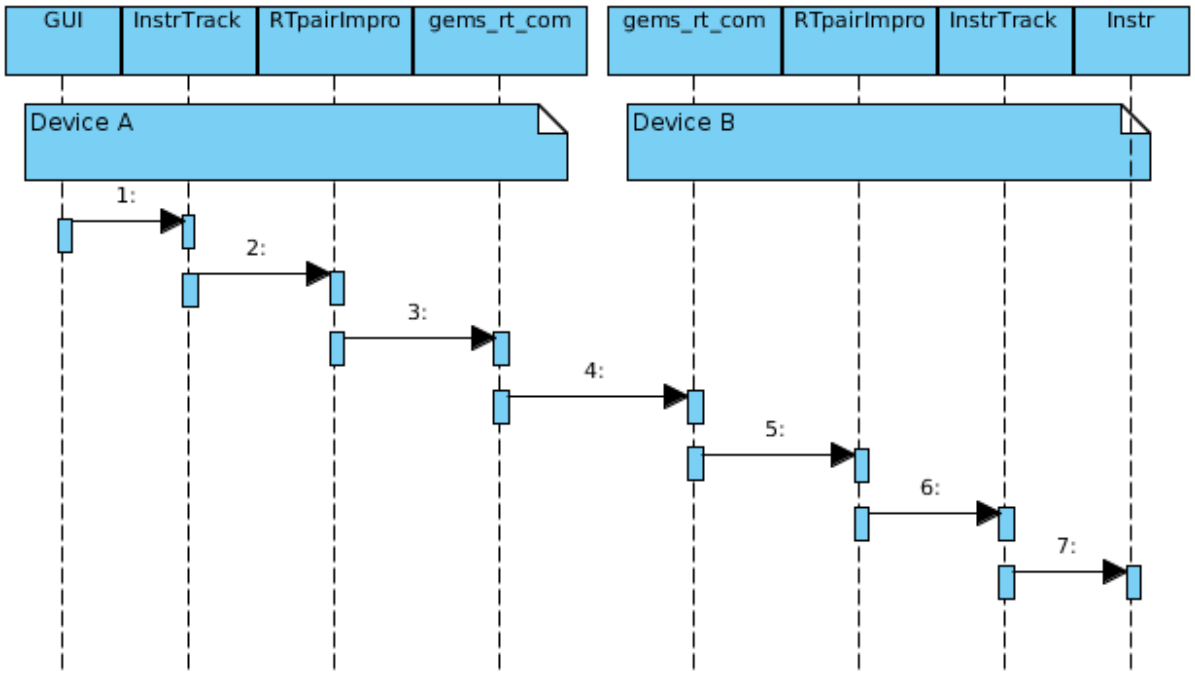

Figure 27: Transferring events in the implementation

6.2 Implemented components

Real-time pair improvisation could not use normal communication provided by GEMS. By default GEMS uses encryption to protect the privacy of users which causes communication and processing overhead. The data transferred during pair improvisation does not contain personal information and transferring it without encryption was considered suitable. Gems_rt_communication uses a dedicated PeerHood connection. In addition to avoiding communication and processing overhead also message buffering in JamMo was avoided by a dedicated connection.

Gems_rt_communication utilizes callback functions for communication with JammoCollaborationRTpairImprovisation because GEMS component can not see CHUM. Additionally, the callback structure makes communications component re-usable because it does not need to be aware of the component that is using it. The callbacks are registered using C language function pointers and consist of information of the data stream and

received events. Messaging system of GObject could not be used as GEMS components are procedural C code. The API of gems_rt_communication is presented in Table 22.

Table 22: Gems_rt_communication API

Function name	**Parameters**	**Description**
gems_rt_communication_init_send_only	Peer user id Get time function Stream info function Duration Master or slave	Used in initializing sending only.
gems_rt_communication_init_recv_only	Peer user id Get time function Stream info function Duration Master or slave Event callback	Used in initializing receiving only.
gems_rt_communication_init_send_and_recv	Peer user id Get time function Stream info function Duration Master or slave Event callback	Used in initializing sending and receiving.
gems_rt_communication_send_event	Peer user id Event type Note Timestamp	Used in sending an event to peer.
gems_rt_communication_end	Peer user id	Used in ending communication before duration is reached.
gems_rt_communication_set_duration	Peer user id Duration	Used in setting duration for the communication.

In gems_rt_communication API peer user id is used in differentiating connections. Simultaneous connections to different peers can exist but only one connection with a particular peer is allowed. Get time function is called whenever the current time of improvisation is needed. Stream info function is called when JammoCollaborationRTpairImprovisation needs to be notified of changes in connection, such as starting playback or turning all events off due to problems in communication. Event callback function is called when an event is received.

Gems_rt_communication uses g_timeout functions extensively. Creating dedicated PeerHood connections, receiving events, sending EmptyEvents and starting synchronizations are timed with g_timeout.

The callback functions presented in Table 22 are described in Table 23. Game object is the JammoCollaborationRTpairImprovisation object that uses the connection. The object is always given as a parameter when a method is called from a GObject. The Game object is stored as a GObject instead of JammoCollaborationRTpairImprovisation because GEMS components can not see JammoCollaborationRTpairImprovisation which is located in CHUM.

Table 23: Callback functions

Function	**Parameters**	**Description**
get_time_func	Game object	Used in querying the time from the game object. The function returns the current time.
stream_info_func	Game object Type	Used in reporting incidents during real-time communication. Type specifies the incident.
event_callback	Game object Note Event type Timestamp Received on time or not	Used in informing of a received control event. Received on time or not is used in decision whether to play a sound or not when a Note-on event is received.

The possible types for stream_info_func consist of the incidents presented in Table 24. The types are defined in gems_rt_communication as numeric values. Event types of event_callback consist of Note-on and Note-off and are defined in JammoMidi component in MEAM. Events received from peer are stored to a list during improvisation for later use. Also the events from local user are stored to a list.

Table 24: Stream info types

Type	**Description**
SYNCHRONIZATION_FAILED	Synchronization has failed for the maximum number of times and improvisation should not be continued.
SYNCHRONIZATION_DONE	Synchronization is successfully done and playback should be started. Peers should start sending events to each other.
ERROR	An unrecoverable error has been encountered in communications. Improvisation should be ended.
FINISHED	Connection has been finished successfully.
GOT_SYNCHRONIZATION_REQUEST	Peer requested synchronization. Playback should be stopped and a new synchronization waited.
REQUESTED_SYNCHRONIZATION	A new synchronization has been requested due to problems in communications. Playback should be stopped and a new synchronization waited.
TIMEOUT	There has been temporary problems in communications. All playing notes are turned off to avoid hanging notes.
CONNECTION_LOST	Connection has been lost completely. Improvisation should be ended.

JammoCollaborationRTpairImprovisation implements the local application logic in real-time pair improvisation. It initiates gems_rt_communication, controls musical components during improvisation and stores improvisation after it has finished. Real-time pair improvisation can

be initiated by creating a new real-time pair improvisation with jammo_collaboration_rtpair_new function and by calling jammo_collaboration_rtpair_start. When jammo_collaboration_rtpair_start is called a group is created or joined depending on parameters. Jammo_collaboration_rt_start requires parameters as follows:

- A pointer to real-time pair improvisation object
- Whether a new game should be created or an existing game joined
- Group id if a group is joined
- Group owner id if a group is joined

JammoCollaborationRTpairImprovisation has several states which are presented in Table 25. States are changed when stream information function is called by gems_rt_communication (see Table 26) or the used JammoSequencer stops (see Table 27).

Table 25: States of JammoCollaborationRTpairImprovisation

State	**Description**
RTPAIRSTATE_INIT	Waiting for starting.
RTPAIRSTATE_STARTED	Improvising with peer.
RTPAIRSTATE_FINISHED	Improvisation has finished. Try to store the improvisation.
RTPAIRSTATE_GENERATING_MIX	Storing improvisation as audio and JammoMidi files.
RTPAIRSTATE_ERROR	Unrecoverable error encountered. Improvisation will be ended.

When gems_rt_communication calls stream information function the state of JammoCollaborationRTpairImprovisation is typically changed to another state regardless of the current state. RT_COMMUNICATION_ERROR and RT_COMMUNICATION_CONNECTION_LOST are exceptions. They are only taken into account in RTPAIRSTATE_INIT and RTPAIRSTATE_STARTED states. However, this is

trivial as communications are not utilized in later states.

Table 26: Reactions to stream information types

Stream information type	**Reaction**
SYNCHRONIZATION_FAILED	Change state to RTPAIRSTATE_ERROR. Stop sequencer.
SYNCHRONIZATION_DONE	Change state to RTPAIRSTATE_STARTED. Start sequencer.
ERROR	If improvisation has been long enough to be saved change to state RTPAIRSTATE_FINISHED. Otherwise change state to RTPAIRSTATE_ERROR. Stop sequencer.
FINISHED	Change state to RTPAIRSTATE_FINISHED and stop sequencer.
GOT_SYNCHRONIZATION_REQUEST	Change state to RTPAIRSTATE_INIT, delete stored control event lists and stop sequencer.
REQUESTED_SYNCHRONIZATION	Change state to RTPAIRSTATE_INIT, delete stored control event lists and stop sequencer.
TIMEOUT	Turn all playing notes off.
CONNECTION_LOST	If improvisation has been long enough to be saved change to state RTPAIRSTATE_FINISHED. Otherwise change state to RTPAIRSTATE_ERROR. Stop sequencer.

Based on stream information sequencer is stopped in many situations. It should be noted that when sequencer is stopped further state changes may take place. This makes implementation simpler and more robust. Storing improvisation is initiated only when sequencer has stopped and the state is RTPAIRSTATE_FINISHED although sequencer may have been stopped because the backing track had ended or communication had ended. Furthermore, real-time pair improvisation is finished only after sequencer has stopped in

RTPAIRSTATE_GENERATING_MIX or RTPAIRSTATE_ERROR states.

Table 27: State changes when sequencer stops

Old state	**New state**	**Description**
RTPAIRSTATE_INIT	RTPAIRSTATE_INIT	Wait for sequencer to start.
RTPAIRSTATE_STARTED	RTPAIRSTATE_GENERATING_MIX	The end of the backing track has been reached.
RTPAIRSTATE_FINISHED	RTPAIRSTATE_GENERATING_MIX	Gems_rt_communication has informed that the communication has been finished.
RTPAIRSTATE_GENERATING_MIX	Final	Improvisation has been stored.
RTPAIRSTATE_ERROR	Final	Unrecoverable error has been encountered.

When improvisation is stored the events from both peers are stored from lists to text files and an audio file is generated. Event files enable later editing whereas audio file contains the improvisation as it took place on the devices. In the audio file generation the events from lists are used with their original timestamps. This way no network latency is present in the generated audio file. During improvisation events from peer are played slightly later than on the peers device.

JammoCollaborationGameSelection shows currently available real-time pair improvisation groups. A group can be joined by touching the image of the group. It is also possible to create a new group. When a new group is created or an existing group joined a JammoCollaborationRTpair object is created. The available pair improvisation groups are stored in a UI container. The available groups are periodically queried from GEMS and updated to the UI. The updating procedure is taken care of by a g_timeout function.

The statistics of implementation are presented in Table 28. The JSON file of JammoCollaborationGameSelection is not included in Lines of code field.

Table 28: Statistic of implemented components

Component	**Lines of code**	**Functions**
JammoCollaborationRTpairImprovisation	831	36
gems_rt_communication	1250	19
JammoCollaborationGameSelection	154	4
Total	**2235**	**59**

6.3 Evaluation

Real-time pair improvisation worked on Nokia N900 although it was uncertain during earlier stages of this thesis. The performance, however, was worse than on computers. Local latency of 10 ms could not be achieved as from time to time latency was easy to notice by playing. GStreamer and PulseAudio automatically used a longer latency. Manually adjusting the latency was not considered feasible as there were already audible glitches in audio playback from time to time. Using a lower latency would have degraded the audio quality more. The perceived latency was suitable for playing slowly. Latency measurements were not considered as musical components were not part of this thesis. During improvisation approximately 80 % of processing power was used which explains the problems in audio playback. When a background process required processor time all audio processing could not be done in time.

The described performance was achieved when JamMo was started from the applications menu or standard output and error streams were forwarded to the null device (/dev/null). Otherwise JamMo printed a lot of log messages to terminal which caused audio playback and synchronization problems.

The communications worked well on N900. The events from peer were received smoothly and network latency could not be distinguished from the local latency. Network delay peaks were rare and there were no long periods of missed notes. The implementation of pair

improvisation was considered successful.

The synchronization never succeeded when one N900 and one computer were used. This was due to the processing the musical components required for starting playback. The problem could have been overcome by estimating the required wall clock time for starting and subtracting that from the waiting time in synchronization. The estimation was not implemented due to lack of time. When similar hardware, e.g. two computers or two N900s, were used the synchronization procedure succeeded.

On two computers pair improvisation worked well. Local latency and network latency were acceptable and the communications robust.

6.4 Discussion

There were two possible ways to receive the JammoMidi events produced by the user. The events could have been received from the JammoInstrumentTrack class of MEAM or the instrument GUI of CHUM. Both options would have required additions to the components. JammoInstrumentTrack was chosen for better re-usability. The implemented callback system can be used on any JammoInstrumentTrack in any or even without a GUI view whereas using GUI would have restricted receiving events to that view.

Gems_rt_communication was not implemented as a GObject class for consistency. Other GEMS components are procedural C code. Using GObject in gems_rt_communication would have made signaling between objects easier and creating a callback system unnecessary.

A few problems were encountered in the implementation. They were all related to synchronization of devices. Firstly, synchronization could not be implemented as a polling function run periodically in main loop of JamMo due to limited accuracy of timing. A specified thread was not used as in the development of JamMo avoiding threads was instructed. Instead, the synchronization procedure was implemented as two functions (one for master, the other for slave) that do not allow any other functions of JamMo to be run

during synchronization. In other words UI does not respond and normal GEMS communication does not happen during synchronization.

Secondly, starting of playback takes a variable time caused by GStreamer pipeline initialization. This issue makes synchronization less accurate. Thirdly, clock differences could not be adjusted without a new synchronization. Seeking GStreamer pipelines with virtual instrument was not fast, accurate and robust enough. Instead a new synchronization was required. The outcome of the playback starting and seeking problems was a need to synchronize devices a couple of times before the playback could start. Also means to measure and inform peer of clock differences were added to the communication protocol presented in chapter 5 due to seeking problems.

7 CONCLUSION

This thesis was done in UMSIC project. UMSIC aims at measuring and increasing social inclusion of children by collaborative musical activities. In the project a musical software, JamMo, is developed. JamMo is an open source musical collaboration and learning tool for children aged 3 to 12. JamMo is targeted at Nokia N900 mobile phone but it can be run on computers as well.

In this thesis a scenario of UMSIC was designed, implemented and integrated into JamMo. In the scenario two children improvise with virtual instruments on N900 in real-time. Both children can hear themselves, each other and a common backing track. Children use headphones to minimize noise and to enable multiple simultaneous musical activities in a classroom.

The use of headphones in the scenario required transferring musical information in real-time over a wireless connection. The latency and bandwidth requirements for data transfer were evaluated. Also other requirements were elicited for the scenario.

Research Question 1 was how should the devices communicate in real-time pair improvisation scenario. Different transport layer protocols were measured and evaluated. Link layer configurations and technologies were considered with performance issues in mind. Different application layer communication paradigms were discussed. Transferring different types of data was considered. The effects of decisions on one layer to the other layers were considered.

It was found out that pair improvisation can be implemented in different ways. Constraints on one layer can be taken into account on the other layers of TCP/IP protocol stack. Link layer was seen as the most important layer in terms of performance. Both audio and control data could be transferred over different transport layer protocols in real-time with suitable link layer configurations. Reordering of packets on transport layer was seen as undesirable as it can prevent application layer from receiving already transferred data on time. Control data

transfer had slightly lower latency and less possible problems, such as audible effects of clock differences and lost audio segments due to low latency streaming. Control data transfer, however, required reliability. A lost or out of order Note-off event may alter a performance significantly. A communication protocol for control data transfer was designed with the link and transport layer constraints in mind.

Many aspects of Research Question 1 were not considered in detail due to limited time. The timers and constants in the communication protocol were not fine tuned based on measurements. Furthermore, a protocol for audio transfer was not developed at all. Moreover, WLAN was the only thoroughly considered link layer technology. Thus, Research Question 1 was not specific enough to be fully answered in the scope of this thesis.

Research Question 2 was how to implement the real-time pair improvisation scenario in JamMo. Real-time pair improvisation was implemented according to the module structure of JamMo. Local application logic was encapsulated in a CHUM component whereas communication was located in GEMS. The GEMS component used a dedicated PeerHood connection for real-time data stream to avoid overhead as much as possible. It implemented the communication protocol designed in this thesis. Available software libraries and communication technologies were utilized in the implementation. Only slight changes and additions were made to existing JamMo components.

The results of this thesis show that it is possible to develop real-time musical applications for mobile devices. Moreover, wireless and mobile technologies are suitable for real-time communication even with low latency requirements. The performance on Nokia N900 was not as robust as desired. The performance issues were due to processing required by musical components and limited resources of N900. However, it was shown that the implementation described in chapter 6 was sufficient. On computers there were no signs of performance problems. It can assumed that with future mobile devices the performance will be similar to computers today.

Many things were left undone because of the limited scope of this thesis. The future work of this thesis includes testing, measurements and extensions. The communication protocol

could be improved.

Field testing was not done. It is unknown whether children aged 7 to 12 find the achieved performance satisfactory. Furthermore, it is unknown whether the performance achieved with a low amount of network traffic can be achieved in the classroom when a wireless access point gets more congested. Particularly if the required performance can not be achieved it would be important to study the performance of dedicated ad hoc networks over one centralized access point. Ad hoc WLAN networks could be created for pair improvisation on a free WLAN channel. Afterwards the centralized access point could be rejoined for creating and joining new improvisation sessions. Moreover, the effect of WLAN signal strength on latency performance was not measured.

Pair improvisation could be extended in multiple ways. In this thesis only one type of two instruments of JamMo were used. In addition to JammoSampler based instruments also JammoSlider instruments could be added in pair improvisation. With JammoSlider one can play various slides and slurs not typical to Western music. For JammoSlider support the communication protocol would have to be extended because slider uses floating point frequencies instead of integer note values.

User experience of pair improvisation could be improved. Menus for instrument and backing track selection could be created. Furthermore, the avatar of peer could be shown while improvising. While waiting for a peer to join a improvisation session a game could be played or at least a waiting image could be shown. Also a picture could be shown during synchronization.

Improvisation could be extended to group improvisation of three to four peers. For group improvisation the communications should be considered once again. The performance of multicast traffic should be compared with point-to-point connections. With multicast the amount of traffic from peers to access point would be greatly reduced.

The constants and timers were left unspecified in the communication protocol. Performance measurements should be conducted in order to find out guidelines for values. It would be

important to find out if some values perform better even when other variables are changed. The protocol should be evaluated in other scenarios, such as using N900 as a controller only, as well.

REFERENCES

[1] Allman, M., Paxson, V., Stevens. W. TCP Congestion Control. 1999. http://www.ietf.org/rfc/rfc2581.txt. Retrieved 2010-10-27.

[2] Balakrishnan, H., Padmanabhan, V.N., Seshan, S., Katz, R.H. A comparison of mechanisms for improving TCP performance over wireless links. 1997. IEEE/ACM Transactions on Networking (TON), Volume 5 Issue 6.

[3] Bartolomeu, P.; Fonseca, J.; Rodrigues, P.; Girao, L. 2006. Evaluating the Timeliness of Bluetooth ACL Connections for the Wireless Transmission of MIDI. ETFA '06. IEEE Conference on Emerging Technologies and Factory Automation, 2006.

[4] BLUETOOTH SPECIFICATION Version 2.1 + EDR. 2007. http://www.bluetooth.com/Specification%20Documents/Core_V21__EDR.zip. Retrieved 2010-10-19.

[5] BLUETOOTH SPECIFICATION Version 4.0. 2010. http://www.bluetooth.com/Specification%20Documents/Core_V40.zip. Retrieved 2010-10-21.

[6] Bluetooth specification documents. http://www.bluetooth.com/English/Technology/Building/Pages/Specification.aspx. Retrieved 2010-10-21.

[7] Casetti, C., Gerla, M., Mascolo, S., Sanadidi, M.Y., Wang, Ren. TCP Westwood: End-to-End Congestion Control for Wired/Wireless Networks. 2002. Wireless Networks. Volume 8, Issue 5. September 2002.

[8] Classical MIDI Works, Wireless MIDIjet Pro International, Wireless MIDI system, http://www.midiworks.ca/products/details/189/8/wireless-midi-products/wireless-midijet-pro-international. Retrieved 2010-10-15.

[9] Clutter. Open source graphical user interface library. http://www.clutter-project.org/. Retrieved 2010-10-21.

[10] CME WIDI-X8, Wireless MIDI system, http://www.cme-pro.com/en/product-detail.php?product_id=8. Retrieved 2010-10-15.

[11] Contreras, F. GStreamer, embedded and low latency are a bad combination. 2010. http://felipec.wordpress.com/2010/10/07/gstreamer-embedded-and-low-latency-are-a-bad-

combination/. Retrieved 2010-10-21.

[12] Crow, B.P., Widjaja, I., Kim, J.G., Sakai, P.T. IEEE 802.11 Wireless Local Area Networks. 1997. IEEE Communications Magazine. September 2007.

[13] Floros, A., Avlonitis, M., Vlamos, P. 2007. Frequency-domain stochastic error concealment for wireless audio applications reconstruction of lost packets in pcm stream. MobiMedia '07: Proceedings of the 3rd international conference on Mobile multimedia communications.

[14] Floyd, S., Henderson, T., A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. 2004. http://www.ietf.org/rfc/rfc3782.txt. Retrieved 2010-10-27.

[15] Gitorious project hosting service. JamMo Project. http://gitorious.com/jammo. Retrieved 2010-10-11.

[16] GLib Reference Manual. http://library.gnome.org/devel/glib/. Retrieved 2010-09-16.

[17] GObject Reference Manual. http://library.gnome.org/devel/gobject/. Retrieved 2010-09-16.

[18] GStreamer. Open source multimedia framework. http://gstreamer.net/. Retrieved 2010-09-16.

[19] Gummadi, R., Wetherall, D., Greenstein, B., Seshan, S. Understanding and Mitigating the Impact of RF Interference on 802.11 Networks. 2007. SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications.

[20] Gynther, M. 2008. Avoimen lähdekoodin musiikkiteknologia (Open Source Musical Engineering). Bachelor's Thesis. Lappeenranta University of Technology. https://oa.doria.fi/handle/10024/43266?locale=len&author=. Retrieved 2010-10-11.

[21] Gynther, M. 2010. Real-time Musical Pair Improvisation source code at Gitorious. http://gitorious.org/~mgynther/jammo/mgynther-jammo/commits/rtpair. Retrieved 2010-11-16.

[22] IEEE 802.11 WORKING GROUP PROJECT TIMELINES. http://www.ieee802.org/11/Reports/802.11_Timelines.htm. Retrieved 2010-10-21.

[23] IEEE Std 802.11g-2003. http://standards.ieee.org/getieee802/download/802.11g-2003.pdf. Retrieved 2010-10-19.

[24] IEEE Std 802.11-2007. http://standards.ieee.org/getieee802/download/802.11-2007.pdf. Retrieved 2010-11-09.

[25] Internet Protocol version 4. 1981. Information Sciences Institute, University of Southern California. http://www.ietf.org/rfc/rfc791.txt. Retrieved 2010-10-11.

[26] JACK Audio Connection Kit. http://jackaudio.org/. Retrieved 2010-10-15.

[27] Jasani, H., Alaraje, N. Evaluating the Performance of IEEE 802.11 Network using RTS/CTS Mechanism. 2007. IEEE International Conference on Electro/Information Technology, 2007.

[28] Jung, E., Vaidya, N.H. Improving IEEE 802.11 power saving mechanism. 2008. Wireless Networks, Volume 14 Issue 3.

[29] Kenton MidiStream. Wireless MIDI system. http://www.kentonuk.com/products/items/wireless/midistream.shtml. Retrieved 2010-10-15.

[30] Kohler, E., Handley, M., Floyd, S. Datagram Congestion Control Protocol (DCCP). 2006. http://www.ietf.org/rfc/rfc4340.txt. Retrieved 2010-10-29.

[31] Lazzaro, J., Wawrzynek, J. 2006. RTP Payload Format for MIDI. RFC 4695, IETF Proposed Standard Protocol. http://www.rfc-editor.org/rfc/rfc4695.txt. Retrieved 2010-10-11.

[32] Lee, J., Rosenberg, C., Chong, E.K.P. Energy efficient schedulers in wireless networks: design and optimization. 2006. Mobile Networks and Applications, Volume 11 Issue 3.

[33] Liang, S., Cheriton, D. TCP-RTM: Using TCP for Real Time Multimedia Applications. 2002. http://www-dsg.stanford.edu/sliang/rtm.pdf. Retrieved 2010-10-19.

[34] Limex Wireless 3. Wireless MIDI system. http://www.limexmusic.com/cgi-bin/content.pl?q=bD1lbmdsaXNoJmlkPWFra29yZGVvbg&section=produkte&eintragid=4531. Retrieved 2010-10-15.

[35] Maekawa, T., Nishimoto, K., Mase, K., Tadenuma, M. A wireless, networked musical environment consisting of wearable MIDI instruments. 2003. 10th International Conference on Telecommunications, 2003. ICT 2003.

[36] Mathis, M., Mahdavi, J., Floyd, S., Romanow, A. TCP Selective Acknowledgment Options. 1996. http://www.ietf.org/rfc/rfc2018.txt. Retrieved 2010-10-27.

[37] MIDI Media Adaptation Layer for IEEE-1394. 2000. MMA/AMEI RP-027. Version 1.0. http://www.midi.org/techspecs/rp27v10spec(1394).pdf. Retrieved 2010-10-15.

[38] MIDI Messages. http://www.midi.org/techspecs/midimessages.php. Retrieved 2010-10-18.

[39] MIDIoverLAN CP. A commercial implementation of a custom MIDI over IP protocol.

http://www.musiclab.com/products/rpl_info.htm. Retrieved 2010-10-28.

[40] Mills, D., Delaware, U., Martin, J., Burbank, J., Kasch, W. Network Time Protocol Version 4. 2010. http://www.ietf.org/rfc/rfc5905.txt. Retrieved 2010-10-20.

[41] NetJack, Realtime Audio Transport over a generic IP Network. http://netjack.sourceforge.net/. Retrieved 2010-10-15.

[42] Nokia. Maemo website. N900. http://maemo.nokia.com/n900/. Retrieved 2010-10-07.

[43] Nokia. N900 specifications. http://maemo.nokia.com/n900/specifications/. Retrieved 2010-10-08.

[44] PEERHOOD SUBSYSTEM SPECIFICATION version 0.2. 2004. https://www2.it.lut.fi/svn/public/peerhood/trunk/PeerHood_documentation/. Retrieved 2010-10-21.

[45] Postel, J. Daytime Protocol. 1983. http://www.faqs.org/rfcs/rfc867.html. Retrieved 2010-10-20.

[46] Postel, J., Harrenstien K. Time Protocol. 1983. http://www.faqs.org/rfcs/rfc868.html. Retrieved 2010-10-20.

[47] Rantalainen, A. 2010. JamMo website. http://jammo.garage.maemo.org/. Retrieved 2010-10-11.

[48] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V. RTP: A Transport Protocol for Real-Time Applications. 2003. http://www.ietf.org/rfc/rfc3550.txt. Retrieved 2010-10-28.

[49] Steward, R.Stream Control Transmission Protocol. 2007. http://www.ietf.org/rfc/rfc4960.txt. Retrieved 2010-10-29.

[50] Tangle Toolkit. http://gitorious.org/tangle. Retrieved 2010-10-21.

[51] TCP Hybla Homepage. http://hybla.deis.unibo.it/. Retrieved 2010-11-22.

[52] Tolonen, T., Välimäki, V., Karjalainen, M. Evaluation of Modern Sound Synthesis Methods. 1998. Report no. 48 / Helsinki University of Technology, Department of Electrical and Communications Engineering, Laboratory of Acoustics and Audio Signal Processing. TKK, Otaniemi. http://www.acoustics.hut.fi/publications/reports/sound_synth_report.pdf. Retrieved 2010-09-16.

[53] Usability of Music in Social Inclusion of Children. http://www.umsic.org/. Retrieved 2010-09-08.

[54] The Wi-Fi Alliance. Organization. http://www.wi-fi.org/organization.php. Retrieved 2010-10-21.

[55] The Wi-Fi Alliance. Wi-Fi CERTIFIEDTM n:Longer-Range, Faster-Throughput, Multimedia-Grade Wi-Fi® Networks. 2009. http://www.wi-fi.org/register.php?file=wp_Wi-Fi_CERTIFIED_n_Industry.pdf. Retrieved 2010-10-21.

[56] Williams, J. P., Chapman, R. O.: A musical duet performance MIDI over IP system. 2005. Journal of Computing Sciences in Colleges. Volume 21 Issue 2.

[57] Zimmermann, R., Chew, E., Ay, S. A., Pawar, M. 2008. Distributed musical performances: Architecture and stream management. Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP). Volume 4 Issue 2.

APPENDIX 1: Byte level encoding of protocol messages

All messages are 10 bytes long. All integers are in network byte order. See sections Operation and Abstract Messages in chapter 5 for the usage and purpose of messages. Integers are signed unless explicitly specified unsigned. Error message is used in Synchronization and Improvisation phases. All other messages are specific to Synchronization or Improvisation.

Error

Message Type (1 byte)	Padding (9 bytes)
5 (1 byte integer)	

The following messages are used in Synchronization.

SyncInit

Message Type (1 byte)	Padding (9 bytes)
10 (1 byte integer)	

SyncRTT

Message Type (1 byte)	Padding (9 bytes)
11 (1 byte integer)	

(continued)

SyncRTTreply

Message Type (1 byte)	Padding (9 bytes)
12 (1 byte integer)	

SyncStart

Message Type (1 byte)	Count Time (4 bytes)	Padding (5 bytes)
13 (1 byte integer)	4 byte integer	

SyncStartOK

Message Type (1 byte)	Padding (9 bytes)
14 (1 byte integer)	

SyncRTTaverage

Message Type (1 byte)	Average RTT (4 bytes)	Padding (5 bytes)
15 (1 byte integer)	4 byte integer	

(continued)

The following messages are used in real-time connection after synchronization.

Note-on

Message Type (1 byte)	Note value (1 byte)	Timestamp (8 bytes)
0 (1 byte integer)	1 byte unsigned integer. Values 0-96.	8 byte unsigned integer.

Note-off

Message Type (1 byte)	Note value (1 byte)	Timestamp (8 bytes)
1 (1 byte integer)	1 byte unsigned integer. Values 0-96.	8 byte unsigned integer.

Empty event

Message Type (1 byte)	Padding (1 byte)	Timestamp (8 bytes)
2 (1 byte integer)		8 byte unsigned integer.

Resync

Message Type (1 byte)	Padding (9 bytes)
3 (1 byte integer)	

ClockDiff

Message Type (1 byte)	Padding (1 byte)	Clock Difference (8 bytes)
4 (1 byte integer)		8 byte unsigned integer.

APPENDIX 2: Timer values in implementation

Timer	Example Value
RT_DELAYQUIT	5 s
RT_EMPTY_INTERVAL	100 ms
RT_ENDTHRESHOLD	100 ms
RT_MAXWAITFORSLAVE	2 s
RT_NOTEHANGDELAY	150 ms
RT_SLAVETIMEOUT	2s

APPENDIX 3: Constant values in implementation

Constant	Example Value
RT_DIFFSENDTHRESHOLD	5 ms
RT_LATESTSYNCPOINT	0.5
RT_LATETHRESHOLD	50 ms
RT_MAXSYNCHRONIZATION	20
RT_RTT_COUNT	5
RT_RTT_THRESHOLD	50 ms
RT_SYNCTHRESHOLD	40 ms