A Framework for Multi-Channel Telecommunication

Applied to TeleCare Application

Author: Yifan Ruan
Supervisor: Danny Weyns
Examiner: Danny Weyns
Date: 2015-6-4
Course Code: 5DV00E, 30 credits
Subject: Software Technology
Level: Master
Department of Computer Science
Abstract
This thesis is deriving from a telemedicine project "TeleCare" of constructing software for remote medical diagnosis between the doctor and the patient. The software has to fix the problem of managing local and remote media information. This thesis presents a telecommunication framework for synchronizing multiple media channels, following research methodology, from problem description, iterative and incremental development to prototype finalization. For the framework, I have described the framework requirements and corresponding architecture design and implementation. From the evaluation result of "TeleCare" software developed above it, I can conclude the framework has reached the problem.

Keywords: framework, communication, synchronization, channel
Preface

Nowadays, the population of elderly people is growing fast around the world. So it is a nice idea to provide better social care with assistive technologies, such as "TeleCare" software. With "TeleCare", the patient and the doctor can communicate with each other like face-to-face meeting. In addition, with some state-of-the-art device, touchscreen monitor, they can express themselves through drawing on the screen. For example, the doctor can draw an arrow representing the patient should raise his arm to a certain height, and the patient can follow the instruction when seeing it on his screen. Image a scenario that if an elderly person lives far away from the hospital, one day he broke his leg, it was inconvenient for him to go to hospital for reexamination again and again. Why not let the doctor diagnose the recovery condition of the patient remotely? "TeleCare" software is created for such case.

To realize remote communication functions, the software must have the ability of sending and receiving media information, like video or audio data, which has already been accomplished by the framework researched in this paper. A framework is a set of generic functionality can be used by the specific application. I have designed and implemented the framework from scratch, above which, the software "TeleCare" was created.

When the software was ready, I validated the functionality through simulating some application scenario. I acted as a patient and another partner as a doctor. We interacted with each other via the software and it worked well as expected. In the coming months, the "TeleCare" will be involved in a larger scale testing. More elderly people and doctors will participate in the test. Then we can improve the software depending on the analysis of feedbacks. We hope the software can solve some actual problems and help the elderly to live better.

Finally, thank the supervisor to guide me in the right research direction and the partner to help me test the application.
Contents

1 Introduction 1

2 Existing Frameworks 3

3 Methodology 4

4 Theory Background 6
   4.1 Communication Model 6
   4.2 NAT Traversal 6
      4.2.1 NAT Terminology 6
      4.2.2 NAT Classification 7
      4.2.3 UDP Hole Punching 7
      4.2.4 Standard Protocol 9

5 Research Case 10

6 Framework Requirements 11

7 Framework Design 12
   7.1 Overview of Framework 12
   7.2 RUDP Communication Fundamental 12
      7.2.1 RUDP 13
      7.2.2 Reactor Pattern 14
   7.3 Channel Controller 14
      7.3.1 Multiple Controller 15
      7.3.2 Hooking Mechanism 15
      7.3.3 Relay Server 16
   7.4 Wrap-up Layers 17

8 Framework Implementation 19
   8.1 Implemented Structure 20
   8.2 Implemented Classes 21

9 Framework Evaluation 23
   9.1 TeleCare Construction 23
      9.1.1 With Inherent Channel Controller 23
      9.1.2 Without Inherent Channel Controller 24
   9.2 TeleCare Test 25
      9.2.1 Function Test 26
      9.2.2 Network Performance Test 28

10 Conclusion 31

A Framework Guideline 1
   A.A Introduction 1
   A.B Getting started 1
      A.B.1 Web 1
      A.B.2 Local 1
   A.C Source code 3
C TeleCare Application Report for Preparation

C.A Description of project ........................................... 5
C.B Technology ............................................................. 5
  C.B.1 Hardware ........................................................... 5
  C.B.2 Software ........................................................... 8
C.C Related projects ....................................................... 8
  C.C.1 AGNES ............................................................. 9
  C.C.2 ALICE (Advanced Lifestyle Improvement system & new Communication Experience) ........................................... 9
  C.C.3 Yoom ............................................................. 10
  C.C.4 EXCITE (Enabling Social Interaction Through Embodiment) .......... 10
  C.C.5 MOTION ........................................................... 10
  C.C.6 NITICS ........................................................... 12
1 Introduction

Telecommunication is a set of technologies for exchanging information between two entities at a distance, applied in different mediums, radio, telephone, television, Internet and the like. This thesis only focuses on telecommunication software, how to transmit formatted information over communication channels in the Internet. The channel is not the physical medium carrying signal, but the logical connection for sharing information simultaneously. Nowadays, such telecommunication software becomes an essential module in more and more applications. For example, the doctor can collect the patient’s health data remotely to be further analyzed. For another instance, a group of people can chat with each other at a distance with web conferencing software. However, telecommunication construction is a bit tedious, so it is convenient to apply existing communication frameworks, most of which can be classified into three kinds with different issues.

- **Work focusing on basic communication.** Arcademis [1], as Java-based framework for object oriented communication middleware development, can be highly configured to meet specific requirements of different application domains. For example, programmers can select transport protocol or determine specific authentication algorithm. An event-drive framework [2] mainly addresses two inter-user communication problems.

- **Work restricted to some particular application domain.** A real-time communication framework [3] for massive multiplayer online games, is to guarantee communication efficiency and high readability. A secure framework [4] for video conferencing system is created to protect audio and video streams with encryption algorithms.

- **Work for providing generic functionality.** With communication protocol WebSocket, Controller Application Communication framework [5] is to collect and store structured data from a set of controller devices, Kinect or Will, to web server, where requesting applications can gain and then process received data.

The problem researched in this thesis is "synchronizing media communication channels". Synchronization is the information coordination between two connected entities. Different applications can employ their own media channels. And the entities involved in the application scenario are equality, not one acts as a server and another is a client. So we cannot directly use or refine above communication frameworks to solve the problem. We have to create our own communication framework for synchronizing multiple media channels, which is describe in this thesis. Although default media types supported by the framework are video, audio and touch information, it can be further extended for supporting more interesting channels. In addition, the framework is lightweight, without dependence of other third-part middleware, and is totally constructed with Java.

Above the framework, I have built the application "TeleCare", allowing the doctor and the patient to interact with each other over distances. With "TeleCare", the doctor can diagnose the patient remotely and the patient doesn’t have to go to hospital. "TeleCare" is part of projects “Push the Line!” and “BoConnect”. "Push the Line!” project aims to provide assistive services to deal with challenges of daily life when the elderly people live alone at home. "BoConnect" project mainly focuses on how to provide better home care for those elderly discharged home from hospital with some assistive technologies. And this project plans to deploy large scale testing when the product is ready. Both projects are for the elderly to live safely and independently with better social care and less

All in all, my contributions of the thesis are framework design, implementation and final validation testing with the application "TeleCare". In the rest of this thesis, section 3 describes how to make the research and section 4 discusses the theory background. Section 5 overviews the problem case and describes the general technology issues, from which I propose the framework requirements in section 6. Architecture design is explained in section 7 and the implementation is in section 8. Finally, assessment of the framework with the practical example "TeleCare" is in section 9.
2 Existing Frameworks

Nowadays, there are some popular libraries or frameworks helpful for building concrete communication applications, such as "libjingle" and "Moho".

- **libjingle**: a library providing list of functions to build multi-user communication applications for voice chat, video conferencing, live music streaming and file sharing. It can handle current network connections and exchange data from multiple entities. The company "Google" uses this library to build its own talk application.

- **Moho**: a Java framework for developing multi-channel communication applications. It applies SIP (Session Initiation Protocol) [6] for managing multi-media communication sessions, which defines transported message type.

Although these codes are useful, they still have some limitations and cannot cope with all scenarios.

- For "libjingle", it only concentrates on the basic communication mechanism. Applications can use network connections maintained by the library for exchanging data. So developers have to synchronize required media information by themselves with provided interfaces.

- For "Moho", it depends on the third-party middleware SIP Servlets, which is a server-side component performing SIP protocol. SIP Servlets is responsible for handling incoming requests and returning corresponding responses. In other words, applications built above the framework must run in services supporting SIP Servlets. So it is not flexible and developers have fewer selections for deployment. Moreover, developers have to read a lot of technical documentations for configuring the framework, which increases the learning cost.

The approach in this thesis aims to reduce those problems. The researched approach is a lightweight framework, without dependency on the third-party middleware. And it is easy to be configured and deployed. It isolates the basic communication mechanism, meaning that developers can directly use media information synchronized by the framework and know nothing how it is exchanged, which accelerates the development.
3 Methodology

Research methodology, in terms of the research process, whose steps are described in Figure 1. In brief, this research was driven by the practical project "TeleCare", aiming to provide communication functions between the patient and the doctor. Then I generalized the research problem "synchronizing media communication channels" and constructed the framework for this problem. For verification I built and evaluated the "TeleCare" application above the framework.

The whole research rests on the application "TeleCare", allowing the doctor and the patient to communicate with each other remotely for medical diagnosis. So the application has to transmit essential media information between two entities, like video or audio. Furthermore, with some state-of-the-art devices, like touchscreen monitor, it has to support sharing touch information when one use’s finger moving on the screen. To satisfy those requirements and not be restricted to some particular application domain, I generalized the research problem of constructing a telecommunication framework for synchronizing multiple channels.

Next I made the research investigation. I studied architecture designs of current communication frameworks and learnt internal workflows of supported communication functions. With the paper of analyzing popular chatting software, like Skype, I understood some technology issues and corresponding solutions. To support features of video conferencing and touching, a camera and a touchscreen monitor are essential hardware components. So I made a report comparing different kinds of cameras and touchscreen monitors in the market to select the most cost-effective products. (The report is attached in the appendix.)

To construct the framework, I followed the iterative and incremental development [7] of design, implementing and review. Iterative and incremental development is the popular developing method of receiving feedbacks on time and improving the product step by step. However, it is also a challenge to ensure the fundamental architecture can support continuous modifications.

First of all, I determined the technology issues for the research problem and refined them to several functional requirements. Then I weighted those requirements and sorted

![Figure 1: Research Methodology](image-url)
from high to low priority. After that, I began the development.

- For **design**, I used UML (Unified Modeling Language) to describe design models of the framework. Models mainly focus on two aspects, structure and behavior. The structure model references the organization of the system and the relationship among inherent subsystems. If necessary, a subsystem can be divided into several smaller parts. The behavior model is used for illustrating activities among different components for different use cases.

To fix some design problems, I applied design patterns [8], which are the formalized solutions for general problems. However, design patterns cannot be directly applied to the system, which require to be modified, because they are only the descriptions or templates of ideas to solve problems, which are suitable for most cases.

In the iteration, I selected at least one use case to develop. If the current iteration was not the first one, I had to refine the previous model to add new functions and meanwhile maintain previous functions. Apart from documenting those models, I also explained them and recorded useful information for implementation, like design decisions or related modifications.

- For **implementing**, following design documentations from the last step, I programmed identified use cases with Java. The code should be well commented and easy to be read and understood. Then before taking the next step, I tested the current code to find bugs and fix them, to guarantee the program running without any errors.

- For **review**, according to use cases in the current iteration, I defined some test cases to verify the code implemented as designed. Moreover, I checked with previous test cases to ensure previous functions were not broken after modifications. If use cases were remaining, I would return to the design step and begin the development again.

After several developing iterations, the framework was ready to be used. I constructed the prototype application "TeleCare" with interfaces provided by the framework. Then "TeleCare" was deployed in the real environment. And I evaluated functional features supported by the application to verify the research has solved the problem.
4 Theory Background

This section is to provide the theoretical background of the research. Network communication, in terms of sending and receiving network messages, is the basic module to construct the framework. So I should discuss different communication models and select a proper one. Furthermore, some problems may appear along with a particular model. So I describe the corresponding solution and explain supported theory.

4.1 Communication Model

In general, there are mainly two kinds of communication models, P2P (Peer-to-Peer)[9] and C/S (Client/Server)[10]. P2P is a decentralized model, where two peers can directly communicate with each other. In contrast, for C/S model, a server is responsible for delivering network messages to target client when receiving requesting messages from another one.

Because P2P mode can effectively avoid network delays, which is a key factor for communication, this framework employs P2P as the basic data transmission mode, like famous software Skype[11], however, in the real Internet environment, sometimes direct communication between two host machines will fail if at least one machine deployed behind a NAT (Network Address Translation)[12] device.

4.2 NAT Traversal

Next I introduce terminologies about NAT and related classifications. Then I can explain the reason why P2P communication will fail under a certain condition and the proper solution.

4.2.1 NAT Terminology

A host machine in the Internet should have its own network identifier IP (Internet Protocol) address for communication with others. There are two versions of IP standards, IP Version 4 (IPv4)[13] and IP Version 6 (IPv6)[14]. Currently most machines use IPv4 to be connected in the Internet. However the number of public IPv4 address is limited, it is impossible to assign a unique address to every device. To cope with this challenge, scientists propose a solution of network address translation. If possible, several devices can be grouped as a private network. Inside that network, those internal devices can send and receive message with private IPv4 address[15]. If some device wants to communicate with external one in the Internet, the private IPv4 address should be translated to another public address, which let several internal devices share one same public address. The device that responsible for translating address is named NAT device.

Host machines in the private network have to be deployed behind the NAT device for monitoring outgoing and incoming messages. And it is well known that a message contains its source address where it comes from so that the receiver can response a message. So when a NAT device receiving an outgoing message from an internal machine, it will modify the private source address to the public and store a map of the private and public addresses. And when receiving an incoming network message from an outside machine, it will look up the destination in the map and deliver it to the target internal machine.
4.2.2 NAT Classification

NAT devices can be simply classified to four types with behavior[16], Full-cone NAT, Address-restricted-cone NAT, Port-restricted cone NAT and Symmetric NAT. Next I will take an instance to explain these four NATs.

![Figure 2: NAT scenario](image)

In Figure 2, Peer-1 is behind a NAT device, whose private endpoint (combination of IP address and port) is iAddr:iPort. Server’s endpoint is sAddr:sPort and another host machine Peer-2’s endpoint is hAddr:hPort. Endpoint iAddr:iPort is mapped to eAddr:ePort stored by the NAT device, when Peer-1 communicates with Server.

**How can Peer-1 receive messages from Peer-2 if Peer-1 knows Peer-2’s endpoint hAddr:hPort and Peer-2 only knows Peer-1’s public endpoint eAddr:ePort?**

- **Full-cone NAT**
  Peer-2 can send a message to the endpoint eAddr:ePort, then NAT will deliver the message to Peer-1 automatically;

- **Address-restricted-cone NAT**
  Peer-1 can receive a message from Peer-2 only if Peer-1 has previously sent a message to the address hAddr with any port;

- **Port-restricted-cone NAT**
  Peer-2 can send message to Peer-1 successfully only if Peer-1 has previously sent message to the endpoint hAddr:hPort;

- **Symmetric NAT**
  If Peer-1 sends a message to Peer-2, NAT will use another public endpoint eAddr’:ePort’ to map iAddr:iPort, Peer-2 has to send a message to the new endpoint eAddr’:ePort’. It is impossible for Peer-1 to receive message from Peer-2 with the endpoint eAddr:ePort.

4.2.3 UDP Hole Punching

The essential prerequisite factor of P2P communication model is that both sides know public network address of each other. But because of the existence of NAT, the address is dynamically changed, which cannot be acquired in advance. To solve this problem, I employ “Hole Punching”[17] method.
UDP (User Datagram Protocol) [18] is the core transparent layer protocol, along with TCP (Transmission Control Protocol)[19]. UDP defines datagram packet as network message carrying on actual data to be delivered to target machine. It is the most simple network protocol of sending messages, without any complex mechanism to guarantee message delivery. UDP hole punching helps two host machines to send and receive UDP datagram packets with each other.

To apply UDP hole punching, a third-part rendezvous server is necessary, which records the network endpoints of machines. To discuss how UDP hole punching works, I have to introduce several participants, two clients Peer-1 and Peer-2, and one rendezvous server RS. Both Peer-1 and Peer-2 don’t know the external network topology environment, whether they are behind one or more NAT device or not. Before communication, both should send message containing their private endpoint to RS. RS also extracts the public source endpoint from the message and stores them both. Both peers have to take following steps for building a connection.

1. Peer-1 sends a request message for connection with Peer-2 to RS;
2. RS sends Peer-1’s private and public endpoints to Peer-2, meanwhile sending related endpoints of Peer-2 to Peer-1;
3. Both Peer-1 and Peer-2 send messages with private endpoints firstly, if no response received, try public endpoints;
   - As Figure 3 shows, Peer-1 and Peer-2 are in the same LAN (Local Area Network), they can directly communicate with private endpoints.
   - However, as Figure 4 shows, Peer-1 and Peer-2 are in different LANs, so it is impossible to be connected with private endpoints. Before Peer-2’s first message to Peer-1’s public endpoint has been through the NAT-2, when NAT-2 receiving message from Peer-1, it will consider this message to be unsolicited and drop it. But after Peer-2’s first message has crossed the NAT-2, NAT-2 will
Figure 4: Peers behind different NATs

accept the message from Peer-1 and deliver it to Peer-2. The whole process above will also repeat again for NAT-1. Finally the hole through NAT-1 and NAT-2 for connecting between Peer-1 and Peer-2 is established.

4. Since UDP is a connectionless protocol and it will not maintain the connection by itself, when there is no message through the hole, the NAT device will eliminate the previous recorded map of private and public address automatically. So it is necessary for both peers to send messages periodically to keep the current connection alive.

From the description of NAT types in section 4.2.2, we can find that if two peers behind different NATs and one NAT is symmetric, they cannot connect with each other using UDP hole punching. Further, in the real network environment, the behavior of NAT device is more complexity, mixing several types. So we have to enable another relay server to exchange network messages if UDP hole punching method fails.

4.2.4 Standard Protocol

Because UDP hole punching is the common method to cross the NAT device, there have already been standard protocols STUN[20] (Session Traversal Utilities for NAT) and TURN[21] (Traversal Using Relays around NAT) to describe how it works formally. STUN protocol defines set of services to allow host machines to get endpoints from a rendezvous server; TURN protocol defines services provided by a relay server to help relay messages between host machines. In the framework, I use terms of STUN and TURN to define servers, only to reflect construction intention, but not realize the complete services described in standard protocols.
5 Research Case

This thesis is driven by the software "TeleCare", belonging to the field of telemedicine. Telemedicine[22] provides remote health care with a set of telecommunication and information technologies, which can be classified into two types, pre-recorded and real-time interactive services. Pre-recorded telemedicine software records the medical data of the patient in advance, and then transfers it to the doctor when necessary. People on both sides can be offline at the same time. With real-time telemedicine software, the patient and the doctor can communicate with each other remotely.

With the application "TeleCare", the doctor can diagnose the patient over distances, particularly convenient for the elderly. For example, if an old person’s arm has fractured by accident, he has to go to hospital for regular reexaminations after the first treatment. Sometimes the examination is simple, the doctor only let the patient shake or raise his arm to a certain height. And ask some questions to know the patient’s feeling about the recovery. So it is unnecessary to be in the hospital, which can be done at a distance with some software.

The certain problem researched in this thesis is "synchronizing media communication channels". In detail, when two applications establish a network connection, they have to exchange media information to realize some functions, whose process is the synchronization. The framework in this thesis aims to provide generic functionalities of synchronizing media information, acting as middleware to be applied for different application scenarios.

To achieve this object, I divided the problem to two main technology issues.

- A reliable communication mechanism with UDP for sending and receiving messages
- Controllers responsible for synchronizing media channels depending on the above communication mechanism

With these issues fixed, I can construct the communication framework, above which the "TeleCare" software was developed.
6 Framework Requirements

I refined the technology issues introduced in the last section to describe functional requirements to be satisfied by the framework.

- Requirement 1: the framework shall provide reliable and unreliable methods of sending and receiving messages with UDP for developers to select for use.

- Requirement 2: the framework shall allow developers to configure parameters of the reliable communication mechanism.

- Requirement 3: for network performance, the latency of messages in the framework shall be less than 300 milliseconds.

- Requirement 4: the framework shall provide application interfaces for developers to use synchronized media information for specific application scenarios.

- Requirement 5: the default media channels provided in the framework are video, audio and touch channels for developers to use.

- Requirement 6: the framework shall allow developers to construct their own custom controller for synchronizing custom media channel.

- Requirement 7: the framework shall solve NAT traversal problem with UDP hole punching technology for developers.

The first technology issue has "Requirement 1", "Requirement 2" and "Requirement 3", and remaining requirements belong to the second issue. With these requirements, I followed the iterative and incremental development, firstly design, then implementation, finally review. In next section, I explain how to design the framework.
7 Framework Design

This section concentrates on the architecture design, the central part of framework construction. First of all, it displays the overview of layer structure and illustrates relationship among layers. Then it describes inherent modules of each layer and explains specific design patterns, along with rationales. In the end, it wraps up layers to demonstrate the design has accomplished the goal.

7.1 Overview of Framework

The framework consists of two layers, “RUDP Communication Fundamental” and “Channel Controller”. A concrete application is built above the framework, as Figure 5 shows.

![Figure 5: Framework layers](image)

In detail, “RUDP Communication Fundamental” is a realization of RUDP (Reliable UDP), with routines of sending reliable and unreliable messages. “Channel Controller” is responsible for synchronizing multiple media channels. “Central Controller” and “Relay Server” help to set network communication connection between two sides, whether they are in P2P or C/S communication model. “Video Controller”, “Audio Controller” and “Touch Controller” manage local or remote media information. The whole framework works in Java runtime environment. Next I explain each layer with design models.

7.2 RUDP Communication Fundamental

This layer provides message delivery for the upper layer to use, designed for "Requirement 1", "Requirement 2" and "Requirement 3". I introduce the necessity of applying reliability mechanism to UDP and explain how to realize that. With reactor pattern, the framework decouples responsibilities of layers for treating received messages. This layer only needs to consider dispatching a message to a proper handler provided by the upper layer, without knowing how to handle it.
7.2.1 RUDP

Unlike TCP, UDP is a unreliable protocol, in terms of no guarantees to ensure successful message delivery. When one UDP message has been sent from a host machine to another, no way to know it is lost or reaches the destination. If the message is lost, it will not be sent again. However, in some scenario, reliability is essential. For example, after one application sending connection invitation request to another, it should wait the response of accept or decline before it takes the next step. However, if the request or response message is lost, the sequential workflow will terminate.

![Figure 6: Message definition](image)

**Message Definition**  Figure 6 defines the structure of a RUDP message with six fragments. "isReliable" signs the message reliable or unreliable, which will be applied different mechanisms to deal with; "messageId" is the unique message identifier. It is impossible two different messages from the same sender have the same message id; "senderId" points to who sends the message; "code" classifies the message as “GENERAL”, "ACK", "PING" or "REPLY". If "ACK" or "REPLY", "repliedMessageId" will be set replied message id; "content" is the actual media information to be delivered.

![Figure 7: Class RUDPImpl description](image)

**RUDP Realization**  RUDPImpl class is the realization of Reliable UDP, including two routines for message delivery. "sendMessage" is an unreliable method, which is used for sending not critical information. For example, for video conferencing, both sides should transmit messages carrying image data and both screens can display received images. It is acceptable to lose some messages, which only influences on the video quality. But it is unacceptable to retransmit messages again when messages are lost, which will cause network delays.

In contrast, inside "sendReliableMessage" routine, I apply reliable mechanism, whose workflow is illustrated in Figure 8. In detail, when invoked "sendReliableMessage" function, RUDPImpl will wait the ACK (acknowledgement) response after the message is sent. If waiting timeouts, it will try sending the previous message again until the number of attempts exceeds the maximum. Once having received ACK response, it will switch to waiting the reply if necessary. Otherwise it will only notify the success. After receiving the reply, it will use registered handler to deal with this message and return corresponding result. If unfortunately, the message is lost and the end user will be notified the failure.
In addition, for "Requirement 2", I design class `ReliableConfiguration` to set parameters of maximum waiting time for acknowledge message and reply, and attempts number for retransmission when the message is lost, which can be configured by developers. I also set the formal definition of the result returned from "sendReliableMessage" function. There are three types of the result for the sent message, "RECEIVED", "REPLIED" and "TIMEOUT". In detail, "RECEIVED" notifies the message has reached the destination; "REPLIED" signs the result has the response from another side; "TIMEOUT" shows the function has not completed the task successfully, maybe the message is lost or the expected reply is not received.

7.2.2 Reactor Pattern

Reactor pattern [23] is a design pattern for dispatching incoming messages to proper handlers to handle depending on the message event. This pattern includes three participants, "Event", "Handler", and "Reactor". "Event" is the identifier of a message, representing which media it belongs to; "Handler" is the registered object to deal with a message; "Reactor" is the dispatcher that selects a handler for received messages.

With this pattern applied to the framework, shown in Figure 9, channel controllers in the upper layer can register their own event and handler. Then incoming messages containing different "content" ("content" consists of "event" and "payload", shown in Figure 6) will be dispatched to proper handlers, where certain actions will be taken.

7.3 Channel Controller

For media channel synchronization, I design multiple channel controllers for video, audio and touch. The controller is that manages media information no matter in local or
remotely. In addition, I introduce how to use synchronized information by specific application.

7.3.1 Multiple Controller

To employ reactor pattern introduced in section 7.2.2, different controller messages should have their own events. From Figure 10, the fragment "content" inside a message consists of "event" and "payload" fragments. According to different message types, "event" can be "CONNECT", "STUN", "TURN", "VIDEO", "AUDIO" or "TOUCH". As for "payload", the value is "data" or the combination of "flag" and "data" ("data" is the actual media data, "flag" represents particular action, like "ADD" or "CONFIG").

This layer contains two types of controllers, "Central Controller" and media channel controllers. "Central Controller" is designed for managing registered media controllers. It uses "STUN" message to register and obtain network endpoint of another side, for establishing or terminating bidirectional connection with "CONNECT" message. When everything is ready, it configures RUDP communication settings of registered controllers. "Video Controller", "Audio Controller" and "Touch Controller" use messages with event "VIDEO", "AUDIO" and "TOUCH" separately. More specifically, "Video Controller" is for image data captured by a camera; "Audio Controller" is for audio data received by a microphone; "Touch Controller" is mainly for touch path information, drawn by one finger of a user on the screen.

7.3.2 Hooking Mechanism

There is an issue that controllers know nothing about how to make use of the media data by the specific application. For example, after receiving remote image data, some application may display it. However, another one may modify and store it. Even both
have to show the image, they mostly like to use different graphic libraries. So I have to apply the hooking mechanism, one kind of interceptor pattern [24], to extend behavior of handlers from controllers to handle messages, described in Figure 11.

![Figure 11: Class diagram of hooking mechanism](image)

A hook is functionality that created by the upper specific application but executed by the channel controller in this layer. Owing to messages with different events, controllers must define their own hook templates and how to execute hooks. For example, AudioController has two lists of AudioHook’s instances, one is for incoming audio data extracted from messages and another is for outgoing audio data. However, TouchController only has hooking lists for incoming touch path information, for different intentions "CONFIG" and "ADD". List of hooks in the same controller will be called one after another when incoming or outgoing data is ready. So it is convenient to realize particular application scenarios through constructing proper hooks.

### 7.3.3 Relay Server

This module is designed for applying "UDP hole punching" technology to solve NAT traversal problem, to ensure communication between two entities. "Relay Server" contains two independent UDP servers, "STUN Server" and "TURN Server". "STUN" and "TURN" names are from standard protocols, as introduced in section 4.2.4.

- "STUN Server" acts as a rendezvous server in "UDP hole punching" method, providing services for discovering endpoints. It has three services, "REGISTER" is for registering private and public endpoints of the requesting entity, "GETINFO" is for getting endpoints of a specific entity and "UNREGISTER" is for removing the record of endpoints of the requesting entity.

- "TURN Server" is responsible for relaying messages among entities. It has two services, "RELAY" is for notifying the server to relay messages for the requesting entity and "UNRELAY" is for stopping the relay.

As illustrated in Figure 12, in the beginning, both sides try P2P communication model depending on "STUN Server". If direct communication fails, current communication will switch to C/S model, which uses "TURN Server" as intermediate to exchange messages.
7.4 Wrap-up Layers

In above sections, I introduced layers of the framework and discussed specific patterns to design them. Now I wrap up layers to describe outgoing and incoming workflow for media information.

There are two cases for outgoing workflow, one requires user action but another not, as Figure 13 shows. For example, as described in Figure 13(a), when one user draws a line on the screen, touch events are triggered and related routines in "Touch Controller" are invoked. Then the controller will record the touch path information and send "TOUCH" message to connected application. In contrast, as Figure 13(b) shows, without user intervention, when the current communication connection is established, "Audio Controller" will receive audio information continuously, and send "AUDIO" message to another side. Meanwhile, it will invoke inserted hook of sounding from the default speaker.

Incoming workflow is described in Figure 14. For instance, both sides have already been connected and listen for incoming messages. When receiving "VIDEO" messages, the reactor in "RUDP Communication Fundamental" layer will dispatch messages to as-
signed handler, registered by "Video Controller". Then this handler will extract image data from received messages. Finally it will invoke the hook of displaying images on the screen.

In conclusion, the combination of incoming and outgoing workflow makes up the complete activity for synchronizing multiple media channels. After a connection is built, synchronized media information is ready to be used by specific applications. The requirements listed in section 6 have already been satisfied ("Requirement 6" will be demonstrated in section 9).

Figure 14: Incoming workflow diagram
8 Framework Implementation

The framework is implemented by Java8 [25], with new features of “Functional Interface” and “Lambda Expressions”. "Lambda Expressions", aliased as "Closures", is an important characteristic of functional language, treating a function as an object to be passed into a method as a parameter. "Functional Interface" is an interface with only one abstract method, attached with "@FunctionalInterface" annotation. This special interface can be instantiated with "Lambda Expressions". With these new features, developers haven’t to create too many useless classes and code programmed becomes short and clear.

For example, I define an interface VideoHook. The object instantiated from VideoHook can be used by an instance of VideoController.

```java
@FunctionalInterface
public interface VideoHook{
    public void execute(BufferedImage bufferedImage);
}
```

In previous, for different use of image data, I have to create related classes, implementing the interface VideoHook. Those classes are very similar.

```java
// create classes with different specific functions
public class DisplayHook implements VideoHook{
    @Override
    public void execute(BufferedImage bufferedImage){
        // display image
    }
}
public class StoreHook implements VideoHook{
    @Override
    public void execute(BufferedImage bufferedImage){
        // store image
    }
}
public class EditHook implements VideoHook{
    @Override
    public void execute(BufferedImage bufferedImage){
        // edit image
    }
}
// instantiate above classes
DisplayHook displayHook=new DisplayHook();
StoreHook storeHook=new StoreHook();
EditHook editHook=new EditHook();
// use above objects to do something
```

But now, with "Lambda Expression" feature, it is simple to create objects with particular functions to deal with image data.

```java
// create different objects for different purposes
VideoHook displayHook=(bufferedImage)->{
    // display image
};
VideoHook storeHook=(bufferedImage)->{
    // store image
};
VideoHook editHook=(bufferedImage)->{
    // edit image
};
// use above objects to do something
```

From above, it is obvious to see the code becomes clear and easy to understand with features provided by Java8.
8.1 Implemented Structure

The realization of the framework consists of two parts, the "local" and the "web", separated from running environments. The package information of both parts and the relationship among classes in packages are shown in Figure 15.

![Framework package diagram](image)

Figure 15: Framework package diagram

The "local" is helpful for making local desktop application, containing five packages, "network.address", "network.protocol", "network.assist", "network" and "controller". And the "web" is the construction of "Relay Server", which can be directly started without any modification. It shares packages "network.address", "network.protocol" and "network.assist" with the "local".
### 8.2 Implemented Classes

The introduction of classes in the "local" is in Table 1. Classes in the package "network" are the implementation of "RUDP Communication Fundamental" and controller classes in the package "controller" are for synchronizing media channels.

<table>
<thead>
<tr>
<th>package</th>
<th>class</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network.address</td>
<td>Endpoint</td>
<td>combination of address and port, destination for delivering messages</td>
</tr>
<tr>
<td></td>
<td>NetworkInfo</td>
<td>combination of public and private endpoints</td>
</tr>
<tr>
<td>network.protocol</td>
<td>Event</td>
<td>message type identifier</td>
</tr>
<tr>
<td></td>
<td>Message</td>
<td>message protocol definition, describing inherent fragments</td>
</tr>
<tr>
<td></td>
<td>Payload</td>
<td>inherent message fragment, containing actual media data</td>
</tr>
<tr>
<td></td>
<td>STUNFlag</td>
<td>flag to represent actions for data in STUN message</td>
</tr>
<tr>
<td></td>
<td>TURNFlag</td>
<td>flag to represent actions for data in TURN message</td>
</tr>
<tr>
<td></td>
<td>TouchFlag</td>
<td>flag to represent actions for data in TOUCH message</td>
</tr>
<tr>
<td></td>
<td>ConnectFlag</td>
<td>flag to represent actions for data in CONNECT message</td>
</tr>
<tr>
<td>network.assist</td>
<td>Serialization</td>
<td>convert between object and byte array</td>
</tr>
<tr>
<td></td>
<td>HTTPClient</td>
<td>client for &quot;post&quot; and &quot;get&quot; HTTP services</td>
</tr>
<tr>
<td></td>
<td>STUNServerClient</td>
<td>client for STUN services</td>
</tr>
<tr>
<td></td>
<td>TURNServerClient</td>
<td>client for TURN services</td>
</tr>
<tr>
<td>network</td>
<td>Handler</td>
<td>interface with single routine &quot;handle&quot; for incoming messages</td>
</tr>
<tr>
<td></td>
<td>Reactor</td>
<td>responsible for dispatching incoming messages to proper handler depending on specific event</td>
</tr>
<tr>
<td></td>
<td>Result</td>
<td>reliable message result</td>
</tr>
<tr>
<td></td>
<td>RUDPImpl</td>
<td>reliable UDP implementation, with routines &quot;sendMessage&quot; and &quot;sendReliableMessage&quot;</td>
</tr>
<tr>
<td>controller</td>
<td>AbstractController</td>
<td>abstract channel controller, hiding fundamental communication information</td>
</tr>
<tr>
<td></td>
<td>Hook</td>
<td>generic hook to be inserted to controllers</td>
</tr>
<tr>
<td></td>
<td>CentralController</td>
<td>controller for building connection and managing controllers, responsible for CONNECT, STUN and TURN messages</td>
</tr>
<tr>
<td></td>
<td>VideoHook</td>
<td>specific hook dealing with image data</td>
</tr>
<tr>
<td></td>
<td>VideoController</td>
<td>video channel controller, synchronizing VIDEO messages</td>
</tr>
<tr>
<td></td>
<td>AudioHook</td>
<td>specific hook dealing with audio data</td>
</tr>
<tr>
<td></td>
<td>AudioController</td>
<td>audio channel controller, synchronizing AUDIO messages</td>
</tr>
<tr>
<td></td>
<td>PathAddHook</td>
<td>specific hook dealing with touch path data to be added</td>
</tr>
<tr>
<td></td>
<td>PathConfigHook</td>
<td>specific hook dealing with touch path data to be configured</td>
</tr>
<tr>
<td></td>
<td>TouchController</td>
<td>touch channel controller, synchronizing TOUCH messages</td>
</tr>
</tbody>
</table>

Table 1: Classes in the "local"

The "web" is a part of realization of "UDP hole punching" technology. Class RelayServer creates and runs two instances of UDPServer, one acts as "STUNServer" providing services for discovering requested endpoints, and another is "TURNServer" for relaying
messages. The detailed introduction is in Table 2.

<table>
<thead>
<tr>
<th>package</th>
<th>class</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network</td>
<td>ServerHandler</td>
<td>abstract class with routine to handle incoming messages</td>
</tr>
<tr>
<td></td>
<td>STUNServerHandler</td>
<td>subclass of &quot;ServerHandler&quot; to provide methods for discovering endpoints</td>
</tr>
<tr>
<td></td>
<td>TURNServerHandler</td>
<td>subclass of &quot;ServerHandler&quot; to relay messages</td>
</tr>
<tr>
<td></td>
<td>UDPServer</td>
<td>responsible for listening and sending UDP messages</td>
</tr>
<tr>
<td></td>
<td>RelayServer</td>
<td>instantiate and start &quot;STUNServer&quot; and &quot;TURNServer&quot; extending from &quot;UDPServer&quot;</td>
</tr>
</tbody>
</table>

Table 2: Classes in the "web"

The detailed information of variables and methods in classes can be viewed from source code, which is not mentioned here. Finally, the framework is exported into two jar files, "MCTF.local.jar" and "MCTF.web.jar". In next section, I will introduce how to use these files to build "TeleCare" software.
9 Framework Evaluation

Now the framework has already been constructed and it is time to use it. TeleCare is created for medical diagnosis between the patient and the doctor, through video, audio and touch information, built above the framework.

9.1 TeleCare Construction

TeleCare system consists of three subsystems, "TeleCare Local Application", "TeleCare HTTP Server" and "Relay Server". "Relay Server" can be directly used from "MCTF.web.jar" file. "TeleCare HTTP Server" is responsible for managing registered patients and doctors, providing functions of sign in and sign out. Owning to the tutorial[26], I created the server with web framework "express" by JavaScript. "TeleCare Local Application" is programmed with "MCTF.local.jar" file, which is explained below.

To make the graphic user interface of "TeleCare", I use Java graphic library "JavaFX" [27], which has rich interesting features and plenty of graphic components. As for the synchronization of media channels, some have already been done by the framework, but some should be created following some principles.

9.1.1 With Inherent Channel Controller

The framework has default channel controllers for synchronizing image, audio and touch path data. And the left work is to create hooks about how to use that media data and insert it to a instance of proper controller.

For example, I want to realize a specific scenario, when the doctor draws an arrow on the screen, the screen of the patient will show the same arrow in the same position. It is clear that the arrow sharp belongs to touch path information, which is managed by "Touch Controller".

1. Register $\text{touchController}$ to $\text{centralController}$, instantiated from class $\text{CentralController}$;

   ```java
   TouchController touchController = new TouchController();
   centralController.registerController(touchController);
   ```

2. Create and insert an instance of $\text{PathAddHook}$ to $\text{touchController}$;

   ```java
   PathAddHook addHook = (points) -> {
   // display remote touch path information on the screen
   touchController.insertInPathAddHook(addHook);
   ```

3. Let $\text{touchController}$ register the handler to handle "TOUCH" message.

   ```java
   touchController.registerControllerHandler();
   ```

After above work has been done, when touch events triggered in one application, it will send "TOUCH" message to another connected application. Then the screen of another will display the same touch path information.
9.1.2 Without Inherent Channel Controller

Besides the default channel controllers, I can also create custom controller easily. For instance, it is common to allow the patient and the doctor to send text messages. For that purpose, I construct and employ `TextController` with following steps.

1. **Construct class `TextController` with particular hook class `TextHook`, which extends `AbstractController`**;

```java
public class TextController extends AbstractController{
    private List<TextHook> inHooks=new ArrayList<>(); // list of inserted hooks
    // insert a hook to list
    public void insertInHook(TextHook hook){
        this.inHooks.add(hook);
    }
    // register the handler to handle "TEXT" message
    @Override
    public void registerControllerHandler() {
        Handler handler=(message)->{
            String text=(String) Serialization.deserialize(message.getPayload()); // extract string payload from a message
            inHooks.forEach(inHook->inHook.execute(text)); // loop to execute hooks
            return null;
        };
        Event.registerCustomEvent("TEXT"); // register custom message event "TEXT"
        /** register handler to handle messages with "TEXT" event */
        "registerHandler" from "AbstractController" hide the information of
        * registering handler to the reactor from instance of class RUDPimpl
        *
        this.registerHandler(Event.customEvent("TEXT"), handler);
    } // send reliable "TEXT" message to another side
    public void sendMessage(String text){
        this.sendReliableMessage(Event.customEvent("TEXT"), Serialization.serialize(text), null);
    }
    // particular "TEXT" hook with specific "execute" method
    @FunctionalInterface
    public interface TextHook{
        public void execute(String text);
    }
}
```

2. **When `TextController` is ready, then do similar work as last section mentions.**

```java
// create instance of TextController and register to centralController
TextController textController=new TextController();
centralController.registerController(textController);
// instantiate object of TextHook
TextHook textHook=(text)->{
    Platform.runLater(new Runnable() {
        @Override
        public void run() {
            // add text message to graphic user interface
        }
    });
    // insert particular hook to textController
    textController.insertInHook(textHook);
    textController.registerControllerHandler();
}
```

Therefore, it is convenient to create custom channel controller and apply this controller for specific application scenario. The complete framework guideline of getting started is in the appendix.
9.2 TeleCare Test

When the original prototype was ready, the supervisor organized a workshop and demonstrated functions of the application. There were three stakeholders involved in the meeting, two therapists and one mid-manager of Växjö municipality. In the meeting, they expressed strong interest for the application and predicted such application could save a lot of time and effort for medical care, particularly for the elderly. They pointed that in Växjö municipality about 12 physiotherapists and 25 occupational therapists responsible for average 90 rehabilitation programs with elderly and about 50% of these programs could be applied remote service. Meanwhile, they also presented some valuable feedback about how to improve the application.

- providing function of text messaging
- displaying an avatar, which can be used to explain some action more explicitly by the doctor
- identification for specific group of elderly for whom the technology is applicable
- essential to alignment with other related stakeholders

With these suggestions, I refined the "TeleCare" application. The current one has functions of video conferencing, drawing, text messaging and screenshot. "TeleCare HTTP Server" is deployed in Openshift Cloud Application Platform (the official website is https://www.openshift.com/) and "Relay Server" is in an free instance of Amazon EC2 (the official website is http://aws.amazon.com/ec2/), which as Figure 16 shows. For end user, he should have a computer running windows 7, a touchscreen monitor and a camera. Next, I test the "TeleCare" from two aspects, functions and performance, with hardware "Dell P2314T (the touchscreen monitor)" and "Logitech C930e (the camera)".

Figure 16: Deploy diagram of TeleCare
9.2.1 Function Test

We simulated the medical diagnosis scenario, one person acting as a patient, whose arm was broken and another as a doctor, who would exam the recovery.

1. We started the local program "TeleCare Local Application" and signed in with registered name and password, as shown in Figure 17.

![Figure 17: Sign in](image)

2. We saw a list of reservations in Figure 18. Each reservation item includes information about reserved user and date, and a button for communication inviting, representing currently the user was online or not. If a reserved user were online, the button would change from disable to enable. The doctor clicked the button to send invitation to the patient.

![Figure 18: List of reservations](image)
3. After the patient accepted the invitation, we could communicate with the same graphical user interface, as illustrated in Figure 19. We chatted and typed text messages with each other.

![Graphical user interface for communication](image1)

Figure 19: Graphical user interface for communication

4. After asking some questions, the doctor wanted the patient to raise his arm. So he told the patient and typed the message "raise the arm". To highlight the height to be reached, he draw an arrow representing the height on the screen and the patient took the action as expected, which described in Figure 20.

![Instruction with drawing](image2)

(a) Before directing  (b) After directing

Figure 20: Instruction with drawing

It is obvious to see that the application "TeleCare" has expected functions, supported by the telecommunication framework. Now it’s time to know the framework performance.
9.2.2 Network Performance Test

Network performance measures the quality of services provided by communication products, covering several factors, bandwidth, latency, loss rate and the like. I only focus on two measurements, the latency and the loss rate. The latency references the time interval between a message sent and received, in terms of network delay. The loss rate is the ratio of messages reaching the destination occupied the total being sent.

For calculating the time difference between timestamps of messages sent and received by two computers, I should synchronize both internal clocks. I put them under the same LAN network and configured one as a NTP (Network Time Protocol) server, and then another adjusted its clock basing on the time fetched from the server. I also created specific classes for logging messages with timestamps to a file, used in the framework. However, the result of computing the latency is still not precise. Because there were still some time offset between clocks in two computers. And the "TeleCare" is a multi-threaded application, it is difficult to determine the actual time to send and receive messages. Moreover, it cost time to record messages to the file. Owing to both sides are equal, I only consider the network performance of one side.

I tested the framework under two conditions, in the same LAN and in different LAN environments. With logged information from two applications, I analyzed those messages of different media channels and calculated the corresponding latency and loss rate. To describe clear, I generated the graphic chats for the message latency.

As Figure 21 and Figure 22 show, I describe the latency of video and audio messages with line chats, because they were synchronized continuously. However, touch and text messages were discrete, so I use scatter graphs. In the graph, the x-axis is the time latency, whose unit is “ms” (milliseconds) and the y-axis is the time, lasting from the first message sent to the last message.

- When two applications in the same LAN, they applied P2P as basic communication model to send and receive messages. For all media messages, the loss rate is zero. From Figure 21(a) and Figure 21(a), we can see the latency of most video and audio messages is around 20 ms.

- When two applications in different LANs, they used C/S communication model. Because messages were not directly sent and received between applications, required to be exchanged by the relay server. It is possible for some messages to lose on the road. So in Table 3, the loss rates of video and audio messages are 3.48% and 3.33% separately. In Figure 22(a) and Figure 22(a), it costs about 50 ms for video and audio messages to reach the destination.

<table>
<thead>
<tr>
<th></th>
<th>Video Message</th>
<th>Audio Message</th>
<th>Touch Message</th>
<th>Text Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lost Num</td>
<td>12</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total Num</td>
<td>345</td>
<td>120</td>
<td>40</td>
<td>12</td>
</tr>
<tr>
<td>Loss Rate</td>
<td>3.48%</td>
<td>3.33%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Note: I consider those messages whose latency are larger than 2 seconds as lost messages.

Table 3: Message loss rate with C/S communication model

From above, we can see that different communication models influence on the network performance, the latency and the loss rate. With C/S communication, delayed time
for transmitting messages is about twice larger than that with P2P communication. In addition, some messages were not delivered to the target machine. That’s the reason why I design applications try P2P communication in the beginning, then C/S communication if the direct communication fails. Both cases satisfy "Requirement 3", the latency of messages is less than 300 milliseconds.

Further more, for touch and text message delivery, there are no messages lost. Because I applied reliable method for sending them and it worked as expected. The end user can see lines draw on the screen and the text messages from another side. As for video and audio messages, it is acceptable to lose some messages, because end users still can view the image and the display would be updated with the next received messages.

![Latency Graphs](image)

(a) Video message latency
(b) Audio message latency
(c) Touch and text message latency

Figure 21: Message latency with P2P communication model
Figure 22: Message latency with C/S communication model
Conclusion

In above sections, I have raised the research problem "synchronizing media communication channels". In other words, the research is to construct a communication framework for providing synchronized media information, used in different application domains. The problem was generalized from the practical project "TeleCare", aiming to provide remote medical diagnosis between the doctor and the patient, particularly helpful for the elderly live alone at home.

The problem is divided to two technology issues, which further extended to several functional requirements. In general, the framework must have controllers for synchronizing multiple media channels with inherent communication functions. To satisfy those requirements, I designed the framework separated to three layers, with design patterns "Reactor Pattern" and "Hooking Mechanism". Then I implemented the design with Java8 and developed "TeleCare" software above the framework. The construction of "TeleCare" has proven that synchronized media information provided by the framework can be used for different application scenarios. And the framework has the extensibility of creating custom media channel to manage interesting media information.

From the evaluation, the "TeleCare" has expected functions and the framework has reached the problem for synchronizing multiple media channels. However, there are some remaining functional and non-functional limitations of the framework.

- Currently, the "Relay Server" is deployed in an instance of Amazon EC2, whose type is t2.micro, with limited capability for handling incoming requests. In the testing, when two applications involved with C/S communication model, the average latency of messages is about 50 ms, along with some messages lost. When more people participate, "Relay Server" may crash when the number of requests at the same time larger than the maximum it can afford. So it is obvious that current deployment is not applicable for the real scenario.

  Because the "Relay Server" consists of two independent UDP servers, "STUN Server" and "TURN Server", the former is for discovering network endpoints and the latter is for exchanging messages. And "TURN Server" is responsible for handling most requesting messages. With supporting from the framework, deployers can separate and deploy them in different servers. In addition, they can also employ more powerful servers to run both applications.

- As for the framework itself, developers have few options to configure the default channel controllers. For example, the Java library to capture images from a camera works fine in the Windows or Linux systems, but it fails in Mac OS. The developer cannot change to other libraries for capturing images in the video controller. However, he can build his own custom video controller following steps explained in section 9.1.2.

- Moreover, the framework lacks security methods to protect transmitted data. Health data of the patient is the most personal sensitive information, should not be fetched by the unauthenticated user. It is better for the framework to provide functions for encrypting that data, which can be applied by the developers.

- There is no authentication mechanism for the "Relay Server". With public services, any application can register and obtain requested endpoints from the "STUN Server" and asks the "TURN Server" for delivering its messages, which occupies
network resources and reduces the network performance for our applications. So it is a nice idea to design a proper mechanism to check requesting applications and only accept those have already been authenticated.

In the following months, the software "TeleCare" will be tested in a large scale, and more elderly people and doctors will participate in the testing. With those feedbacks, the "TeleCare" will be added more features and improved gradually, to achieve the object of helping the elderly with better services and less cost.
References


A Framework Guideline

A.A Introduction

The framework is built for synchronizing media channels of bidirectional communication. The default supported media types are video, audio and touch path information, which managed by corresponding controllers. According to application scenarios, developers can create their own custom controllers to deal with custom media information. Above the framework, different applications can be developed, video conferencing, telemedicine system and the like.

A.B Getting started

The framework is programmed with Java and exported to files, "MCTF.web.jar" and "MCTF.local.jar". Required Java developing environment is Jdk 1.8, because it uses some new features of Java8.

A.B.1 Web

"MCTF.web.jar" provides implementation of "Relay Server", which can be directly run with below command.

```bash
$ java --jar MCTF.web.jar
```

A.B.2 Local

With "MCTF.local.jar", developers can construct their own local applications. (Note: DefaultController references inherent constructed channel controller VideoController,AudioController or TouchController; CustomController references the controller not in the framework.)

1 Initialize CentralController

```java
// initialize centralController
CentralController centralController=new CentralController(RELAY_SERVER_ADDRESS,
STUN_SERVER_PORT, TURN_SERVER_PORT, USER_ID, HOST_PORT);

// define and set the hook when receiving "CONNECT" messages for establishment
Hook establishHook=(args)->{
    // should return true or false, true means accept, otherwise means decline
};
centralController.setInEstablishHook(establishHook);

// define and set the hook when accepting the invitation
Hook connectHook=(args)->{
    this.centralController.setConnectHook(connectHook);
}

// define and set hook when receiving "CONNECT" message for termination
Hook terminateHook=(args)->{
    centralController.setInTerminateHook(terminateHook);
}

// register the handler for received "CONNECT" message
centralController.registerControllerHandler();

/**
 * The below code is for querying users have registered to STUNServer or not
 * if a user has registered to STUNServer, he is online, otherwise offline
 * developer can create his own way to query user online or not, without below code
 */
// set related list of queried remote users
centralController.setRemoteUserIds(reservedUserIds);
```
2 Configure default channel controller

(a) Initialize an instance defaultController of DefaultController;
(b) register defaultController to centralController;
(c) define and insert a hook to defaultController;
(d) register the handler to handle messages with corresponding event.

For example of VideoController

```java
// step (a)
VideoController videoController=new VideoController();
// step (b)
centralController.registerController(videoController);
// step (c)
// define and set a hook when receiving "VIDEO" messages
VideoHook hook=(bufferedImage)->{
  Image image = SwingFXUtils.toFXImage(bufferedImage, null);
  bufferedImage.flush();  // display image in gui
};
videoController.insertInHook(hook);
// define and set a hook before sending "VIDEO" messages
VideoHook hook=(bufferedImage)->{
  Image image = SwingFXUtils.toFXImage(bufferedImage, null);
  bufferedImage.flush();  // display image in gui
};
videoController.insertOutHook(hook);
// step (d)
videoController.registerControllerHandler();
// start video communication
videoController.start();
```

3 Construct custom channel controller

CustomController should extend from AbstractController, with the method "registerControllerHandler" to be overridden. If necessary, it can define its own hook class.

For example of TextController, responsible for synchronizing text messages.

```java
public class TextController extends AbstractController{
    private List<TextHook> inHooks=new ArrayList<>(); // list of inserted hooks
    // insert a hook to list
    public void insertInHook(TextHook hook){
        this.inHooks.add(hook);
    }
    // register the handler to handle "TEXT" message
    @Override
    public void registerControllerHandler() {
        Handler handler=(message)->{
            String text=(String) Serialization.deserialize(message.getPayload()); // extract string payload from a message
            inHooks.forEach(inHook->{inHook.execute(text);}); // loop to execute hooks
            return null;
        };
        Event.registerCustomEvent("TEXT"); // register custom message event "TEXT"
        /** register handler to handle messages with "TEXT" event */
        * "registerHandler" from "AbstractController" hide the information of
```
4 Configure custom channel controller

Similar as default controller configuration, take the same example of TextController.

```java
// create an instance of TextController and register it to the centralController
TextController textController=new TextController();
centralController.registerController(textController);
// instantiate an object of TextHook
TextHook textHook=(text)->{
    Platform.runLater(new Runnable() {
        @Override
        public void run() {
            // add text message to gui
        }
    });
};
textController.insertInHook(textHook);
textController.registerControllerHandler();
```

5 Start up

When everything is ready, methods "connect" in centralController can establish a connection, and "disconnect" can terminate the current connection.

A.C Source code

The framework implementation is from scratch, with Java8. Dependent libraries are introduced below.

- commons-lang3 to serialize object to byte array and deserialize byte array to object
- imgscalr to rotate captured image
- webcam-capture to capture image from camera
- slf4j to log information, which is essential dependence for library "webcam-capture"

Classes and methods in the framework have detailed comments and it also has Javadoc about public interfaces, which can be checked through source code.
**B  TeleCare Guideline**

TeleCare application consists of three parts, "TeleCare HTTP Server", "Relay Server" and "TeleCare Local Application". The former two servers should be deployed in public cloud platform or public server to be visited by the latter one.

Now "TeleCare HTTP Server" is configured and deployed in Openshift Cloud Application Platform and run with command.

```
$ node server.js
```

"Relay Server" is deployed in Amazon EC2 and started with command.

```
$ java -jar MCTF.web.jar
```

End user should have a computer running windows 7 or above, a touchscreen monitor and a camera. Hardware, which has already been tested, is "Dell P2314T (touchscreen monitor)" and "Logitech C930e (camera)". With Java8 execution environment, end user can start the runnable file directly with command.

```
$ java -jar TeleCare.jar [IP address of Relay Server]
```

There are three default users that can sign in to "TeleCare HTTP Server", represented by "name/password". They are "patient/patient", "doctor1/doctor1" and "doctor2/doctor2".
C TeleCare Application Report for Preparation

C.A Description of project

The whole system consists of two similar parts, one for the elderly and another for the doctor. In one part, there are camera, touchscreen monitor and system control unit. The system control unit is software program, which can display image on screen, read gestures from screen and communicates with another control unit.

Figure 23: Simple system description

For now, I should select proper technology for the project.

C.B Technology

C.B.1 Hardware

<table>
<thead>
<tr>
<th>Monitor</th>
<th>Size/inches</th>
<th>Touch Points</th>
<th>Connectivity</th>
<th>Price/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acer T232HL</td>
<td>23</td>
<td>10</td>
<td>USB3.0(3), HDMI, VGA, DVI</td>
<td>300</td>
</tr>
<tr>
<td>Dell P2314T</td>
<td>23</td>
<td>10</td>
<td>USB2.0(3), HDMI, VGA, DisplayPort</td>
<td>310</td>
</tr>
<tr>
<td>Dell S2240T</td>
<td>21.5</td>
<td>10</td>
<td>USB2.0(1), DVI, VGA, HDMI</td>
<td>245</td>
</tr>
<tr>
<td>Planar PCT2485</td>
<td>24</td>
<td>10</td>
<td>USB(2), HDMI, Display Port</td>
<td>385</td>
</tr>
</tbody>
</table>

Information from "Top Ten Reviews" and "PC Magazine"

Table 4: Comparison of touchscreen monitors

I select touchscreen monitors relying on the popularity and proper size and all monitors can be compatible well with Window, but not compatible with Mac OS or Ubuntu because there are no corresponding official drivers. But maybe we can find the driver from third part. We can use monitor arm in Figure 28 to change the monitor orientation to vertical. From Figure 24 to Figure 27 are the pictures of different monitors.

I select proper camera that has wide angle and is compatible with Windows, Mac OS and Ubuntu. The comparison of different cameras is in Table 5, and related pictures are in Figure 29.
Figure 24: Acer T232HL

Figure 25: Dell P2314T

Figure 26: Dell S2240T
Table 5: Comparison of cameras

<table>
<thead>
<tr>
<th>Camera</th>
<th>Connectivity</th>
<th>Resolution</th>
<th>Angle/°</th>
<th>Price/€</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logitech C930e</td>
<td>USB</td>
<td>1080p</td>
<td>90</td>
<td>95</td>
</tr>
<tr>
<td>Genius F100</td>
<td>USB</td>
<td>1080p</td>
<td>120</td>
<td>42</td>
</tr>
<tr>
<td>Logitech C615</td>
<td>USB</td>
<td>1080p</td>
<td></td>
<td>63</td>
</tr>
</tbody>
</table>
C.B.2 Software

The program has to read touch signals on different operating systems, Windows, Mac OS or Ubuntu, which have advantages and disadvantages.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>compatible with most monitors, MS provides nice touch API</td>
<td>require machine running Windows</td>
</tr>
<tr>
<td>Mac os</td>
<td>available machine running Mac os</td>
<td>no official driver, require finding appropriate way to send and receive touch signals</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>open source, not require high configuration machine</td>
<td>no official driver</td>
</tr>
</tbody>
</table>

Table 6: Comparison of programming on different operating systems

After touchscreen monitor is ready, we should test whether it can work well with Ubuntu, Mac OS or Window. **For these systems, I prefer to use Ubuntu as the programming environment.**

C.C Related projects

On AAL (AMBIENT ASSISTED LIVING) website, I have found five related projects, which are described below.
C.C.1 AGNES

With the home-based ICT (Information and Communications Technology) and social networks, AGENS project provides a platform to connect the elderly with their relatives or friends.

![AGNES System Description]

Figure 30: AGNES system description

C.C.2 ALICE (Advanced Lifestyle Improvement system & new Communication Experience)

At home, people connect the set-top-box machine to their TV set. Then depending on a set of ICT based services, the elderly can enjoy the social interaction and communicate with friends or relatives, like sharing photos, video calling or so on.

![ALICE System Description]

Figure 31: ALICE system description
C.C.3 Yooom

With dual cameras, Yooom is a video communication tool that can display full body gestures of both sides. And people can use the lower screen to replace of the mouse or keyboard to interact with the device.

Figure 32: Application scenery of Yooom

Figure 33: Yooom description

C.C.4 EXCITE (Enabling SoCial Interaction Through Embodiment)

EXCITE project creates a robot, with which people can virtually enter home with computer via the Internet. Robot can move freely around the house, and people can chat with their family members through video communication.

C.C.5 MOTION

For remote physical training guiding of the elderly, MOTION project provides an ICT-based service to connect coaches and people.
Figure 34: Application scenery of EXCITE robot

Figure 35: MOTION system description
C.C.6 NITICS

To monitor and navigational support for the activity of the elderly and provide collecting data to related people, NITICS project designs and implements a platform depending on a set of devices.

Figure 36: NITICS system description

NITICS system collects data from sensors or other devices at home, then processes data and offers APIs to other services.

Figure 37: System architecture