Carnegie Mellon University

Master of Science in Music and Technology

A Music Player for the Human Computer Music Performance Project

---

Author:
Dalong Cheng

Supervisor:
Roger Dannenberg
Richard Stern
Richard Randall

April 12, 2013

Abstract

Computer that coordinate or interact with human musicians exist in many forms and is used in music performance for many years, from computer instruments to computer accompaniment, from compostion system to conducting system. Human Computer Music Performance (HCMP) project aims to create a more autonomous "artificial performer" that does not require human interference. To develop a practice of HCMP, we need to develop a HCMP Player which is able to flexbily change the tempo and communicate with other components of HCMP components.

The HCMP Player is designed and implemented with 2 goals in mind. The first goal is to design a midi player which has a both time and space efficient tempo changing algorithm, it need to deal with frequent tempo changing during performance within bounded time. The second goal is to clearly define a set of programming interface in the HCMP player, which can be easily extended and used to cooperate and coordinate with other components of HCMP project. In this paper, I describe the whole development process of HCMP player, from design logic of code structure to general software architecture, from implementation to complete software testing plan. Challenges, Problems during development and Key design logic behind implementation is also discussed and explained in the paper.

Acknowledgements

I would like to express my appreciation to my advisory committee: Prof. Roger B. Dannenberg, Prof. Richard Stern and Prof. Richard Randall. Thanks for giving me the opportunity to be part of CMU Music Technology program. Special thanks to Professor Dannenberg for his time, patience, and understanding. Professor Dannenberg, it always has been my honor and pleasure to work with you.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Overview

Computer has been used widely in performance of music for many years. Some advanced system use computer as a independent module to replace role of the accompanist in traditional Western art. In popular music field, computers have gained great success through different purpose of music systems and there are a lot of opportunities for innovative applications of highly intelligent and coordinated computer music systems. With more powerful algorithms and more advanced sensing devices, computer music system in near future could fill in for missing musicians and take place of tradictional band player, and ultimately to inspire new musical directions based on new capabilities and concepts from new technologies.

Live popular music offers a wealth of opportunities for computing and music research. We term the integration of computers as independent autonomous performers into popular live music performance practice, "Human-Computer Music Performance" (HCMP). In HCMP, computers become more than instruments and are, in some degree, seen as performers. To bring HCMP into the realm of popular music performance, ceratin problems need to be solved. One problem is that popular music is organized around a tight synchronization to beats and computer cannot reliably and efficiently adapt to human tempo variations. Another significant problem is that HCMP project is a large complex system which contains a lot of subcomponents, with each independent components play its own function, it rely on the cooperation of several different components to complete the task.

The motivation behind HCMP Player is to provide a good solution to above problems. Therefore HCMP Player should be able to play different representations of music, work as accompaniment and quickly adjust its tempo to follow the performer. A clearly defined programming interface is also required in HCMP Player in order to communicate and cooperate with other components within the system. To create such a player, we must coordinate time among different media. We need at least two functional components, one component worked as "backend" and responsible for coordinatting and scheduling different music events, the other worked as "frontend", which receive command from user or other component and dynamicly adjusti play sequence and tempo etc based on those requests.

This paper begins by presenting the role of the HCMP Player in the project of HCMP, the whole "picture" of HCMP as well as its different kinds of componet will be given, and question like how does the HCMP Player fit into the whole picture will be anwsered. In the following 3 chapters, specific problems and challenges regarding to the HCMP Player will be presented and discussed. These 3 chapters together talk about varisou design issues in the HCMP Player, though ecah chapter explains the problem from a different perspective and has its own emphasis. Chapter 2 describes the design of Graphic User Interface (GUI) of HCMP Player together with an detailed explaination of all the GUI components' usage and function. Chapter 3 descibes the "backend" of the HCMP Player – the player engine, and illustrate how player engine cooparate and communicate with GUI to complete external request. Chapter 4 address the network communication problem between multiple players, a set of unified API is defined to make HCMP Player compatible with other components in HCMP project. Chapter 5 mainly describe the implementation detail of the HCMP Players, some pseucode snippet is drawwd in order to give a clear explaination. Chapter 6 mainly focus on the evaluation of HCMP Player, a ful list of feature points will be given and a clear success criteria will be defined to help evaluate HCMP Player. In chapter 7, I make a summary of all my work in this project, some possible future works and extension of the project will also be discussed.

1.2 Background

Before making a formal introduction to the background of HCMP Player. I would like to make short review of some existed human computer music systems. The most common use of computer in music performance is through computer instruments, typically keyboards. These, and other electronic instruments, are essentially substitutes for traditional instruments and rely upon human musicians for their control and coordination with other musicians. Many composers of interactive contemporary art music use computers to generate music in real time, often in response to live performers.

Another meaningful use of computer in music is computer accompaniment system (7), whcih solves the synchronization problem by assuming a pre-determined score (music notation). During the performance, a performer expressively played music score, while the accompaniment system "listen" and analysis, then follows the performer in the score and synchronizes an accompaniment.

In live popular music peformance, computer is quite useful too. The computer has had a significant impact on popular music through drum machines, sequencers and loop-based interfaces, but one can argue that popular music has adapted to new technology rather than the other way around. The sound and beat of drum machines seems stiff, mechanical, and monotonous to many musicians, but that became the trance-like foundation of club dance music and other forms . Similarly, the inability of sequencers and other beat-based software to listen to human musicians has led to performances with click tracks in fixed media or simply a fixed drum track that live musicians must follow. Ableton Live (8) is an example of software that uses a beat, measure, and section framework to synchronize music in live performance, but the program is not well suited to adapting to the tempo of live musicians. Robertson and Plumbley (9) used a real-time beat tracker in conjunction with Ableton Live software to synchronize pre-recorded music to a live drummer. This extension could be considered HCMP. The table 1.1 summarize some of existed computer music systems and their usage.

Computer Instruments	Direct physical interaction with virtual instruments
Interactive Contemporary Art Music	Composed interactions; often unconstrained by traditional harmony or rhythm.
Computer Accompaniment	score following synchronizes computer to live performer.
Fixed Media	Many musical styles and formats. Live performers synchronize to fixed recording.
Conducting System	Synchronize live computer performance by tapping or gesturing beats. Best with expressive traditional/classical music.

Table 1.1: Computer Music Systems Summary

1.3 Objective

The objective of HCMP (1) is to create an autonomous "artificial performer" with the ability of a human-level musical performance. Hopefully, HCMP does not require a human operator's interfere, the system itself is designed to be adaptive and responsive. With sophisticated listening and sensing component, HCMP is able to adjust related system parameters in real-time during performance. From functional perspective, we can divide HCMP into two main categories: music preparation and music performance. With music preparation part aims to work and understand with multiple music representations. Music performance part is to deal with various issues regarding to question how to play music.

An important component of the music performance part is the HCMP Player, which is able to flexibly adjust and respond to changes of music signal. Figure 1 illustrates relationship between scheduler, conductor and HCMP Plyaer. During performance, the HCMP Player will be controlled by conductor and constantly receive control messages from its conductor. In this project, I will design, implement the HCMP midi player for the HCMP project that is able to fully cooperate with conductor and scheduler.

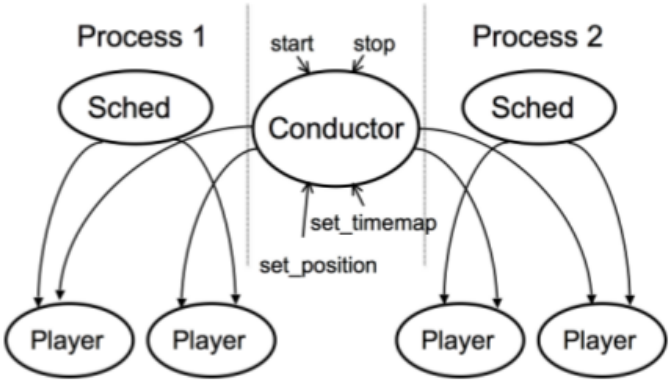

Figure 1.1: Key Components of HCMP

1.4 Architecture

1.4.1 HCMP Architecture

Figure 1.2 shows architecture of the whole HCMP project. it mainly contain 5 subsystems. A brief description of each subsystem and its function will be given in following paragraphs.

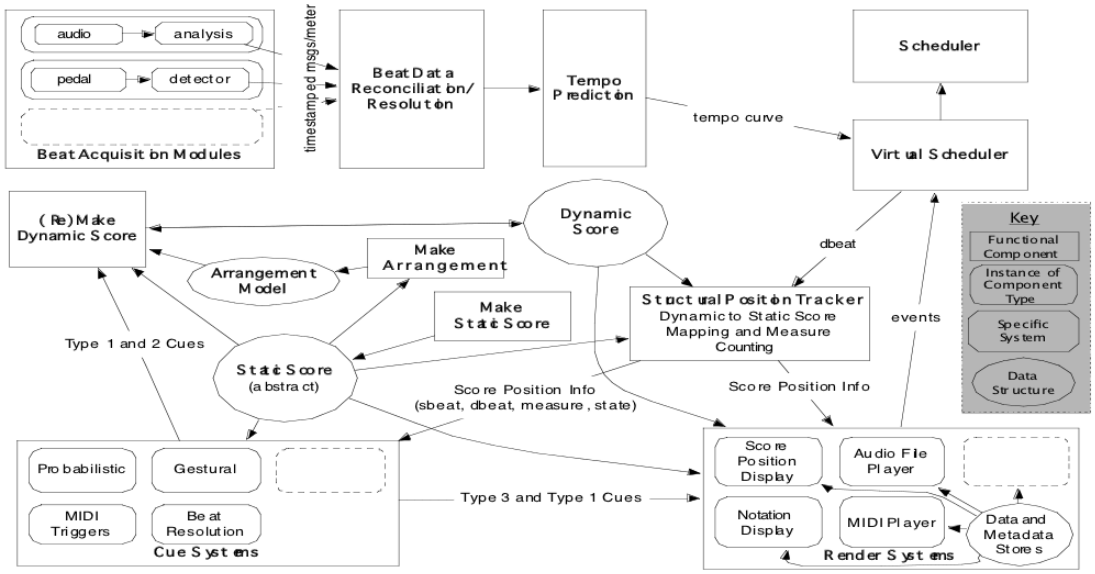

Figure 1.2: Whole Picture of the HCMP Project

1.4.1.1 Real-Time System

Real-time components are needed to keep an HCMP system coordinated with the human musicians in an ensemble. Real-time synchronization aspects are handled by components such as beat and tempo tracking systems.

1.4.1.2 Abstract-Time System

Abstract time components are needed to manage and schedule score events in the context of the performance. The virtual scheduler and its associated systems are concerned with the abstract time aspects of the system. The virtual scheduler should reschedule events scheduled on a nominal time trajectory by warping the event times according to the incoming tempo data from the tempo prediction system. Events are then passed to an actual scheduler for real-time scheduling.

1.4.1.3 Score and Arrangement System

Score management is handled by the functional components in the center box of the diagram. These systems will allow a human musician to encode, manage, and

arrange scores for performance.

1.4.1.4 Cueing System

Cueing systems are required to allow the computer system to react to high-level structural and synchronization changes during performance (e.g. additional repetitions of a chorus). There are mainly three types of cues:

1. Static Score Position Cue. This cue is necessary when synchronization with the static score is lost. Issuing it will cause the dynamic score to be re-made accordingly.

2. Intention Cue. This cue is needed to inform the computer of the intended direction of the current performance (e.g. exiting a vamp section or adding an additional chorus). Issuing it (e.g. using a MIDI trigger, gesture recognition or other method) will cause the future dynamic score to be remade.

3. Voicing/Arrangement Cue. This cue is needed to allow control over the voicing of a section (e.g. it may be desirable to prevent a particular instrumental group from playing on the first time through a repeat but allow them to play on the second). Issuing this type of cue affects only the render system to which it is issued.

1.4.1.5 Render System

Render systems are responsible for providing different kindos multi-media output at the appropriate time. Render system will be designed as some kind of "add-on" for the HCMP, so that we could flexibly change, remove and update the render system. With unified programming interface, each reander system is provided similar raw data and each system should also define its own way of parsing the raw data. Sometimes, metadata is required to link these data elements to their appropriate static score position (and thus to their appropriate scheduling as the dynamic score is played). A standardized format will be required for this to relate static-score measure-numbers and the dynamic score position and context to the properties of the format concerned. This provide render systems more flexibility to determine how they need to deal with beat-level information or simply use the measure-level data, for example, a score display system might map a measure to

image information or an audio render system might represent audio at the beat level.

1.4.2 HCMP Player Architecture

From system architectures' perspective, the interal HCMP Player is mainly made up of two threads, we name the thread that taking care of user input, maintaining graphic state GUI thread. And the thread worked as player engine is called performer thread, each thread is a independent module and executed in a separated thread space. Timer will be external source which periodically invoke the user in do some task. A timer usually come with a language library, so its implementation is within the category of this paper.

During the performance, the HCMP Player will act as the server for the conductor component, which is constantly receiving control messages and responding accordingly.

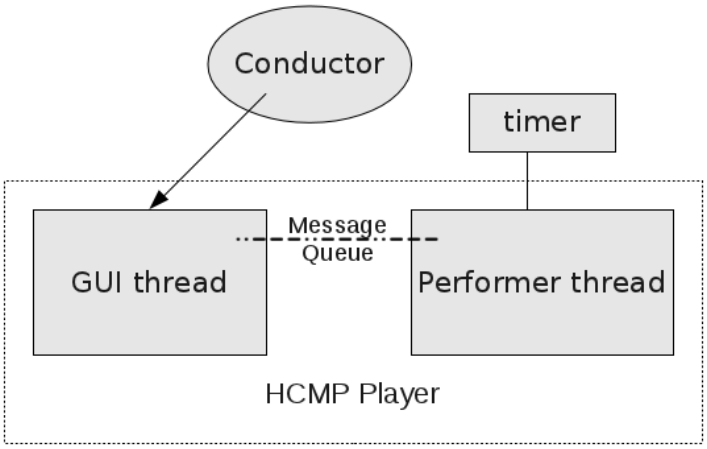

Figure 1.3: Architecture of the HCMP Midi Player

Internally, the Player will have two threads, with one thread for GUI interactive control (GUI thread) and the other thread for performing music data (performer thread). The two threads will communicate with each other through a shared message queue, we can assume the message queue is large enough to avoid blocks for both caller and callee threads. The performaner thread will handle time critical operations, and there will be a timer setup before this thread is created. The

goal of the timer is to wake up the performer thread periodically. Everytime the performer thread's timer callback function is invoked, it will check the message queue and process any command from the control thread. Figure 2 illustrates the overall structure of the Player.

2

Graphic User Interface Design

2.1 Overview

In chapter 2, I mainly describe the GUI design of the HCMP Player, focus will be on how GUI design meet the requirement of HCMP Player, and each GUI components' usage and function. Basicly, the GUI of HCMP Player contains 5 components, a detailed explaination of each component will be given in following sections. Figure 2.1 is a screenshot of GUI of HCMP Player.

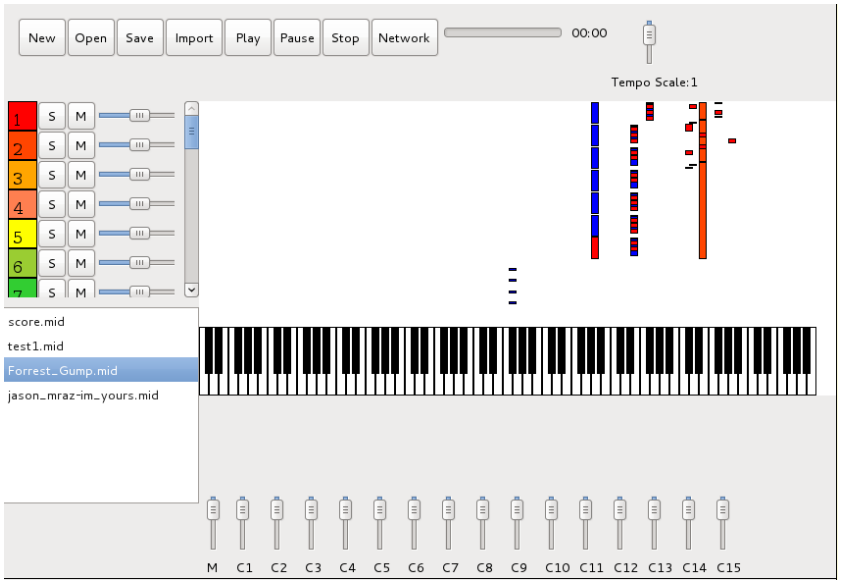

Figure 2.1: HCMP Player GUI Screenshot

2.2 Layout

Figure 3 is layout of the GUI of HCMP Player. The whole area cover 800 * 600 pixels. The top part is the menu panel with 800 * 200 in size. Track panel and library panel share the middle-left part, ecah is 200 * 200 in size. The middle-right part is 400 * 600 in size, which covers the most part of GUI, will be a midi keyboard and data display component. The bottom part is the channel panel. The bound every component to its own area, I put a container panel at most outside layer. The container panel define the boundary GUI of HCMP Player. Any component out of this boundary is invible to users.

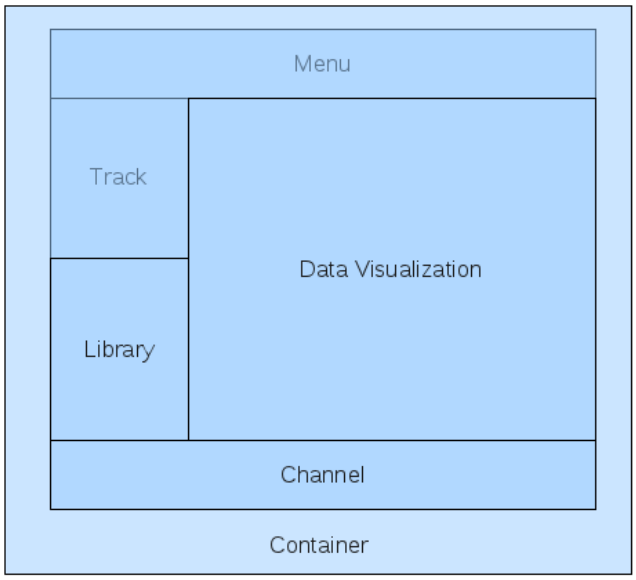

Figure 2.2: HCMP Player Layout

2.3 GUI Component

2.3.1 Menu Panel

Figure 2.3 is the screenshot of menu panel. The menu panel integrate most of control functions for HCMP Player. New, open, save buttons form a group, the group is used to manipulate configuration file. Play, pause, stop buttons form another group, the group is used to directly control the HCMP Player. The function of

each button just like its name suggest. The network button is for HCMP Player network function, after clicking the button, the player will try to automatically initiate a connection to a remote server, the IP address and port number pair is predefined in configuration file and can be easily changed. Once connection is successfully built, all the control will then be transferred to the remote server. This function will be explained in more detail in following chapters. Apart from those buttons, menu panel also contain a slider to change tempo, the slider inidicate the scale coefficient of tempo value.For example, if the original tempo is 90 BMP, with slider set to scale coefficient 1.5, the tempo will be 1.5 * 90 BMP when playing the file. The import button is used to import a midi file into the HCMP Player, once a midi file is successuflly imported into the HCMP Player , track panel, library panel and data display component will all be updated. The path of track will also be recorded into the configuration file, so that the user will not bother to import the same file again.

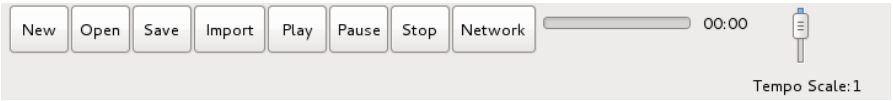

Figure 2.3: HCMP Player Menu Panel

2.3.2 Track Panel

Figure 2.4 is the screenshot of track panel. The track panel is used to control each track's parameter, these parameters are volume, solo, mute. Color block at the front of panel indicate its color in data display component . The track panel will automatically be populated once a midi file is successfully loaded into HCMP Player.

Figure 2.4: HCMP Player Track Panel

2.3.3 Midi Keyboard and Data Display

Figure 2.5 is the screenshot of data display component. It is made up of a virtual keyboard and a midi note canvas. After successfuly loading a midi file, the midi note canvas will draw each midi note to this area. When playing a midi file, midi canvas will automatically update the position of each midi note.

The virtual keyboard has 9 octaves and 108 keys. Both the white key and black key are drawn by canvas. Every midi note will be highlighted in the virtual midi keyboard based on its pitch while playing. To avoid asynchronized speed of canvas update and playing midi note problem. The canvas update speed is coordinited with the player engine. Player engine is responsible for beat update and it periodically send synchronization information to GUI. After receiving update information from player engine, the GUI will update all the midi note in canvas.

Each color of midi note represent a track inside a midi file, in current version, color for each track's midi note is predefined. Midi canvas scroll speed is linear to the tempo of midi. Whenever user make a change to the tempo, the scroll speed of midi canvas will also be adjusted accordingly. When a user mute certain track, its color midi note in midi canvas will temporarily be set to invisble. The canvas update frequency is about 20Hz. To make logic easier, everytime update event happened, the whole midi canvas will be redrawed. The midi canvas is about 400 * 600 in size, and these pixels are update 20 times every second, which is approximately 20 frames per second. The performance is not a major concern. There is canvas buffer data structure to store all the "visble" midi notes. Usually,

the size of a midi file is no more than 10MB, the overall performance of using redraw method is acceptable.

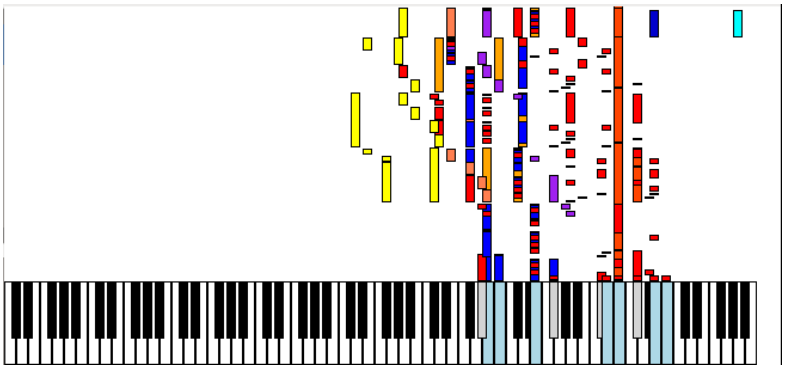

Figure 2.5: HCMP Player Data Display Component

2.3.4 HCMP Player Library

Figure 2.6 is the screenshot of midi library panel. This simple midi library is used to remember all the midi file that has ever been successfully loaded . A configration file of HCMP Player store all the file name as well as its path, so that next time when user launch HCMP Player again, it is able to load these recently openned midi files immediately. Everytime a user click any file in the libary, The GUI looks up a hash-like data structure and fetch its file path (if exist), it then is automatically loaded into the HCMP Player. If the midi file failed to load, the library and its configuration file will remove the obsolete entry and then update the library panel agian.

Figure 2.6: HCMP Player Library Panel

2.4 Player Mode

The HCMP Player has two mode, stand-alone mode and network mode. In stand-alone mode, HCMP Player can be used as normal midi player , user could load and play midi file. In network mode, the HCMP Player will connect to a remote server, in HCMP project, the server usually contains a conductor, and this conductor is responsible for communication and coordination with all the instance of HCMP Players. For example, a typical workflow of conductor is to read cue from external resource, then issue command all the connected instances of the HCMP Players.

2.4.1 Stand-alone Mode

When lauching the program, the HCMP player is default in stand-alone mode, all the GUI component's function is described like above. User could use HCMP Player to play midi file, change tempo, change volume, etc.

2.4.2 Network Mode

After launching the HCMP Player, if user click the network button, the HCMP player will try to enter into the network mode. At the beginning, it initiate a connection request to remote server, the address and port number will stored in the HCMP Player configuration file. If receving no reply within certain timeout period, the HCMP player "rollback" to its stand-alone mode. If receving confirm signal from remote server, the 2-way connection between the HCMP Player and remote server is successfully built up. Then, the HCMP Player and remote server is able to communicate with each other. The control of HCMP Player will partially be "transfered" to remote server. In this mode, player control buttons like play, pause , stop still work as normal, to avoid conflict problems, other buttons will be temporarily disabled, which means binding method for these buttons is replaced by another set of functions. The detailed network API information will be explained in the chapter 5.

3

Player Engine Design

3.1 Overview

Chapter mainly describe the player engine HCMP Player and its design, usage, and programming interface. Focus will be on how player engine communicate with GUI component, what kind "service" player engine provides. In real implementation, player engine is executed in a independent thread, namely perfermer thread. The message queue is like unix-style "pipe" between two programs. The player engine is periodically invoked up by a external timer, execute some predefined task. For example, check and message queue and handle pending request. The timer period is about 10ms, so any user input will immediatelly passed to player engine through message queue and be handled, the latency is trivial and can be ignored.

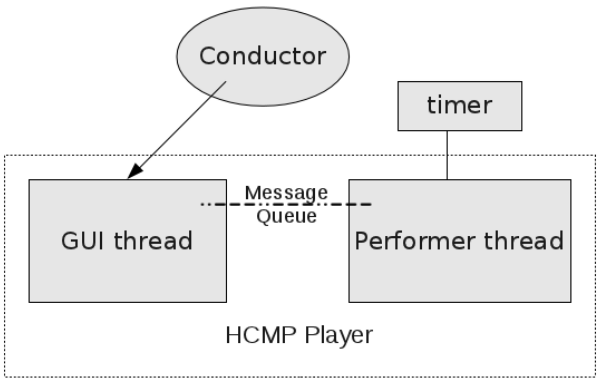

Figure 3.1: GUI Thread and Performer Thread

The player engine server two major purposes, the first purpose is to receive message from GUI component and process it, the second purpose is to schedule and play midi event at desire time, sometimes, player engine also need to reschedule those scheduled events. In concolusion, we can basically divide design of player engine into two parts, communication and synchronization. Communication refer to the way player engine and GUI work together. Synchronization refer to media synchroization, which involv how schedule midi note, how to resposne to the change of tempo.

3.2 Communication

Figure 3.2 illustrate commuication between GUI component and player engine, in this figure, the GUI component receive request and worked as "front-end" part of HCMP Player, and player engine handle the requeset and worked as the "back-end" part. In stand-alone mode, the GUI component receive request from user-generate events like clicking button or adjustting slider bar. These events are all captured by GUI component, processed and evnentualy passed to player engine. In network mode, the request is captured through network interface, the GUI is responsible for handle it when request first arrived by network. Form player engine's perspective, it has no idea whether the request is from a user motion or a remote server. Based on this scenaio, a unified API between GUI component and player engine is defined to clearly illustrate the responsiblity boundary of each module. Note that the API between GUI component and player engine is different from those of network API, the former is an internal API between two module, the latter is defined to coordiate between remote server and multiple instance of HCMP players.

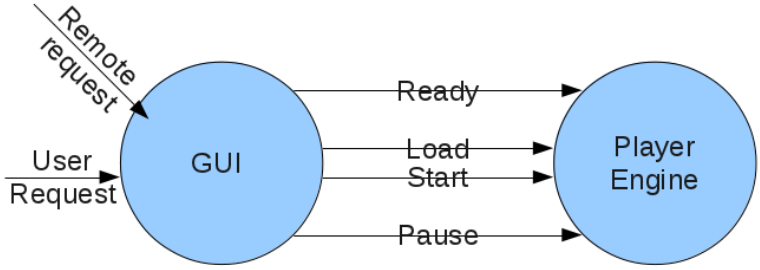

Figure 3.2: GUI and Player Engine

3.2.1 Graphic User Interface Thread

Table 3.2 list the GUI component and its request, in stand-alone when user click those buttons, it will send request to player engine.

GUI component	Send Request
load button	load_file
play button	play
pause button	pause
stop button	stop
network button	send_con_req
tempo slider	change_tempo
volume slider	change_volume

Table 3.1: GUI Component & Request

3.2.2 State Transition Diagram

Table 3.2 and 3.3 are state transition diagrams for player engine. Column is state, Row is method, each (column, row) pair in table means that, given current state if we invoke certain method, what player engine state will transit to.

The player engine only receive and accept control messages from GUI component. It immediately process the message upon receiving. In real implementation, the player engine has a hash like data structure to store all the invoking methods, the key is a pair of current state and request, the value is method which transit the current state to new state, it is quite similar to previous state transition diagram. When the player engine thread firstly created, it also initialize a timer to periodically invoke itself after initialization. Note that to avoid dropping request problem from happening, we need make sure that all the routinues can be executed and completed within bounded time.

3.3 Media Synchronization

To accurately schedule the midi note and apply to any tempo chage. The player engine requires two scheduler, real-time scheduler and virtual-time scheduler. Usu-

Method / State	load	play	pause	stop
uninitialize	ready	undefine	undefine	undefine
ready	ready	pause	ready	undefine
playing	ready	playing	pause	ready
pause	ready	playing	pause	ready
network	con_ready	undefine	con_suspend	undefine

Table 3.2: Player Engine State Transition Diagram

Method / State	change_tempo	change_volume
uninitialize	undefine	undefine
ready	undefine	undefine
playing	apply_change	apply_change
pause	apply_change	apply_change
network	undefine	undefine

Table 3.3: Player Engine State Transition Diagram

ally, a computation will perform some action needed at the present time, followed by the scheduling of the next action. The real-time schedulers role is to keep track of all pending actions and to invoke them at the proper time, thus eliminating the need for Players to busy wait, poll, or otherwise waste computer cycles to ensure that their next computation is performed on time. Virtual-time scheduler is built upon the real-time player, in most of popular music, we can treat synchronization happened at beat level. The purpose of virtual-time scheduler is to schedule everything at beat level, and the huge advantage is that once tempo changed, either due to internal parameter of midi file or external signal, there is no need to reschedule everything. With virtual-time scheduler, we only need to rearrage limited number of midi notes.

3.3.1 Tempo Prediction

Assumming most popular music forms has common structure of beats and measures across all instruments. Thus time is measured in beats. The basis for synchronization is a shared notion of the current beat and the current tempo. Beats are represented by a floating point number, hence they are continuous rather than integers or messages such as in MIDI clock messages. Also, rather than update the beat number at frequent intervals, we use a linear mapping from time to beat. This mapping is conveniently expressed using three parameters (b0, t0, s):

$$b = b_0 + (t - t_0) * s \tag{3.1}$$

where tempo s is expressed in beats per second, at some time in the past beat b_0 occurred at time t_0, the current time is t, and the current beat is b.

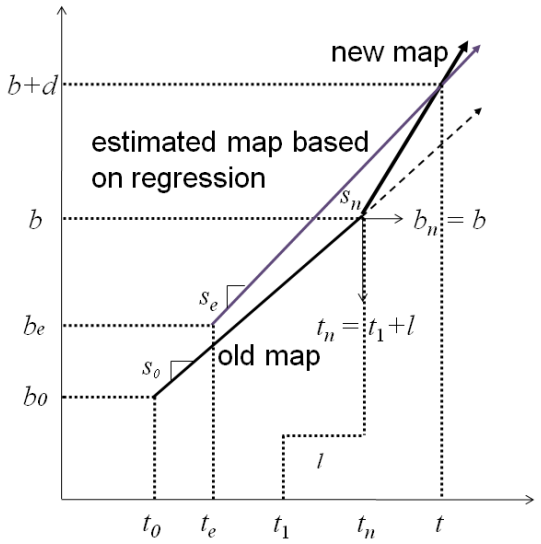

Figure 3.3: Midi data display integrated with vitural keyboard

One advantage of this approach is that it is almost independent of latency. One can send (t_0, b_0, s) to another computer or process and the mapping will remain valid regardless of the transmission latency. When parameters change, there can be a momentary disagreement in the current time among various processes, but this should be small given that tempo is normally steady.

Media players schedule computation to affect the output at specific beat times. For example, an audio player may begin a sample playback at beat 3, or a MIDI player may send a note-on message at beat 5. The current beat time b in equation 3.1 refers to the beat position of media which are being output currently. Since audio output buffer contains 0.01s of audio, then computation associated with beat b should be performed 0.01s earlier than b. Thus, given a player-specific latency l, we need to compute the real time t at which to schedule a computation associated with beat b. The following formula is easily derived:

$$t = t_0 + (b - b_0)/s - l \tag{3.2}$$

We simply map the beat position b according to (b_0, t_0, s), and then subtract the latency l to get the computation time t.

Our current schedueIr use a simple method to predict the beat . Basically, a linear regression over recent taps is used to estimate the mapping from beat to

time.

Once the tempo and beat phase is established, there is a way to determine an offset from the arbitrary beat number to the beat number in the score. This might be determined by a external signal that tells when the system should begin to play. The important point here is that some mechanism estimates a local mapping between time and beat position, and this mapping is updated as the performance progresses.

3.3.2 Scheduling

Schedulers in the HCMP Player accept requests to perform specific computations at specific times in the future. Sometimes, the specified time can be a "virtual" time in units such as beats that are translated to real time according to a tempo, as in equation 3.2. An important idea is that all pending events can be sorted according to beat time and everytime the player engine just pick up the earliest event and check its time-stamp. If the tempo changes, only the time of this earliest event needs to be recomputed. When event times are computed according to equation 3.2, the earliest pending event can change when tempo changes.

3.4 Player Engine Programming Interface

In this section, I list all the API that GUI used to communicate with player engine. The GUI and player communicate with each other using strings and all the parameters are also of type string. A typical string message will be like, `"method_name;argument1;argument2;argument3"`, each argument is separated by a `";"` character,. When passing in the parameter string, the player engine use a `split`-like function to parse the string and store all elments into an array. After parsing the request string, the first element (the request name) together with player engine current state form a pair argument. This pair will be used as a argument to state transition matrix, then fetch the method, which transit the current state to a new state. These API maybe different from the those APIs defined in next chapter, while most of them has quite similar function, the only different lies in that these APIs are defined only for the communication purpose of GUI and palyer engine

Player control related APIs

- `load - "load;file_path"`
- `play - "play;void"`
- `pause - "pause;void"`
- `stop - "stop;void"`

Audio control related APIs

- `change_volume - "change_volume;track_num;velocity"`
- `change_tempo - "change_tempo;tempo_scale_coefficient"`

Connection mode related APIs, detail define in following chapter.

- `ini_connection - "ini_connection;void"`

4

Network Design

4.1 Overview

Chpater 4 mainly discuss the network design of the HCMP Player, focus will be on illustrating design decision behind the network function of HCMP Player, A full list of network API will be given, by which any other HCMP componets can use to communicate with HCMP Player through network. Apart from network API, chapter 4 also discuss how HCMP player establish a connection with remote server at first place. In real implementation, the HCMP Player use zeromq library to facilitate network devleopment. Zeromq library integrate several different ways for communication between nodes within network range. We will briefly discuss each method, together with its pros and cons, and its usage inside the HCMP Player.

4.2 Conductor and Player

In network module of HCMP Player, 3 kinds of design pattern is mainly used. The request & reply design pattern is the most commonly used one, it is used in building connection between remote server and HCMP Player. After connection has been established, the remote server and HCMP Player will adapt another kind of pattern, observer pattern.

4.2.1 Request & Reply Pattern

The request and reply pattern is the most basic pattern used in network communication. Just as its name suggests, the server send request to client, after receving the request, client handle the request and send reply to server. This pattern is heavily used in remote procedure call (RPC) programming model, for example, the caller object can invoke some methods from callee object regardless of callee object is stored in local machine memory or memory space in a remote machine. The most typical use is in web programming, where user press a button, this event eventually invoke a method in server-side, after some computation, the server reply with result to webpage. In most cases, the request & reply pattern rely on the feature of TCP to reliably deliver the message.

Figure 4.1 indicate how this pattern works in HCMP Player. In HCMP Player, when user try to establish the connection with remote server, it will send request and wait for its reply. A timeout limit is set to avoid waiting for too long. If connect to a remote server failed, the HCMP Player will "rollback" to stand-alone mode.

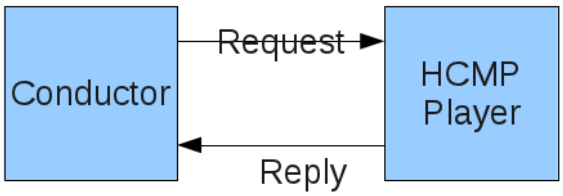

Figure 4.1: Request-Reply Pattern

4.2.2 Observer Pattern

In observer pattern, there is an subject object, the subject object maintains a list of its dependents, which are called observers, and notifies them automatically of any state changes, usually by calling one of their methods. Figure

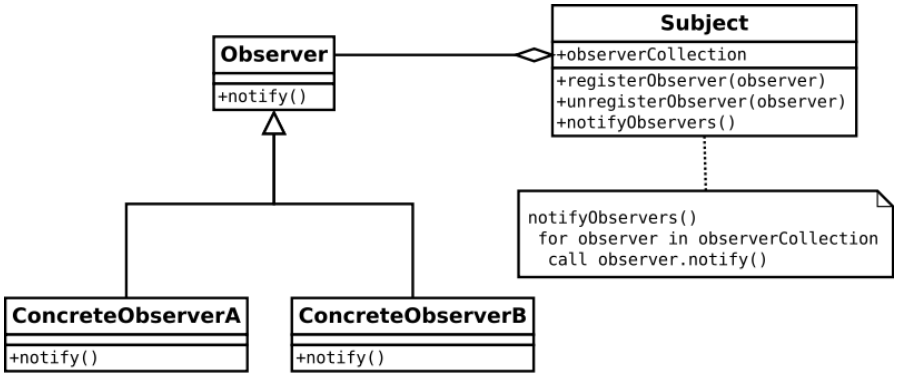

Figure 4.2: UML Diagram of Observer Pattern

After connection has been built, the remote server and the HCMP Player form a observer-pattern relationship, where remote server is the subject and HCMP Player is dependent. Firstly, the HCMP Player try to use a RPC call to register into remote server dependents list, then it wait for reply from the remote server. In real implementation, the conductor is in remote server part and resposible for synchronizing all the HCMP Players. Each conductor will maintain list of HCMP Players , this list will be updated once old player become obselete, or new player join in. Figure 4.3 illustrate observer pattern in HCMP Player.

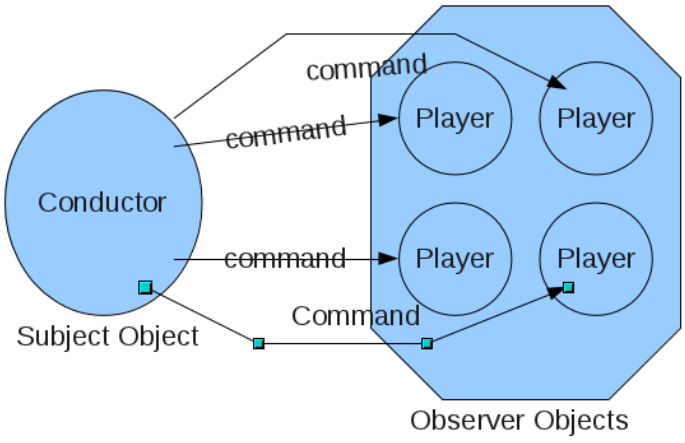

Figure 4.3: Observer Pattern

4.2.3 Push & Pull Pattern

In order to monitor the status of each HCMP Player, each registered HCMP Player need to periodically push "heart beat" message to remote server. The remote server set a count down timer for each HCMP Player currently connected to server. It then periodically pull message and then reset according HCMP Player's timer. If there exist a timer reach zero, then remote server will further send confirm message, temporarily deregsiter this HCMP Plyaer from dependent list. Figure 4.4 indicate how push & pull pattern used in HCMP Player.

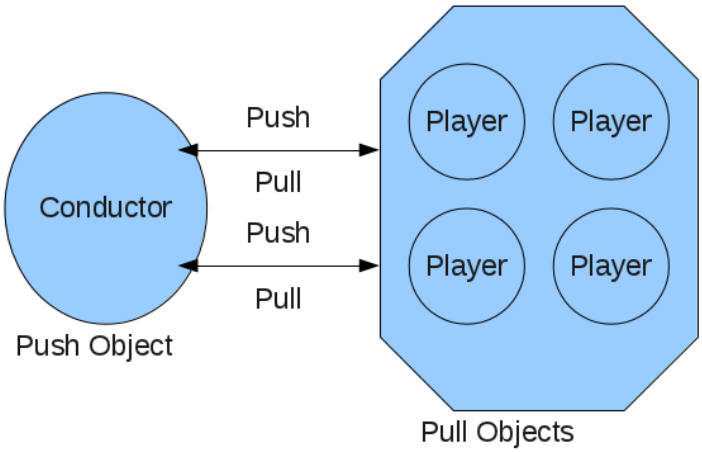

Figure 4.4: Push & Pull Pattern

4.3 Network Programming Interface

The programming interface defined below is used for communication between remote server and HCMP Player. Any external exponent that follows these APIs is able to communicate and take control of the HCMP Player. The GUI component is responsible for receving all the network request. Imagine a situation where remote server try to start command to all HCMP players. The remote server send start request to all HCMP players in its dependent list. Once the request is sucessfully delivered, the GUI of HCMP player will receive and handle the request, after parsing the request string, it map and invoke the internal API, then send the request to player engine through message queue. Once player engine read from message

queue and begin to schedule the first midi note, the HCMP Player start playing.

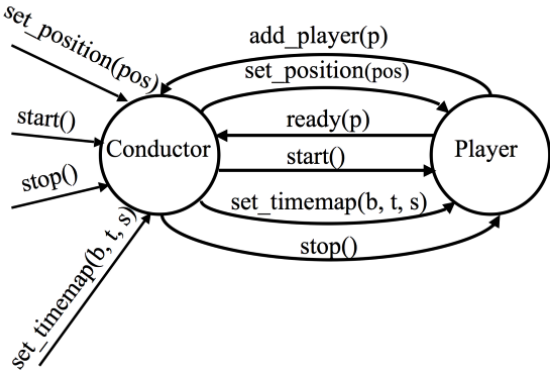

Figure 4.5: HCMP Player Network API

Figure 4.5 illustrate how conductor coordiate with HCMP Player, all the related APIs are list below

From remote server to HCMP Player

- `play - start playing`
- `stop - stop playing`
- `pause - pause current playing`
- `update_time_map - send new time map to player`
- `set_position - set play position to the given parameter`

From HCMP Player to remote server

- `play_all - inform the remote server to play`
- `stop_all - inform the remote server to stop`
- `ready - inform the remote server is ready to play`
- `position - inform remote server to begin from specified position`

5

Implementation

5.1 Overview

Chapter 5 mainly describe the implemenation detail of HCMP Player. Some pseudocode snippets are given to illustrate the relationship between different components. Focus will be on discussing some implementation related issues.

5.2 Development Environment

The HCMP Player is developed using script language serpent(4), serpent is similar to python, which has a simple, minimal syntax , dynamic typing and support for object-oriented programming. The best part of serpent is that it has many convenient builtin functions for midi messages processing, which greatly facilitate developing music-related applications. As for the GUI part, wxWidgets (5) library provides one of best solution, some core functions and objects has been integrated into the serpent library. The zeromq library is used to develop network related functions, it integrate with common network commmunication functions. The whole project is developed in Linux 32-bit environment, serpent provide a cross-platform solution, so the HCMP Player should also be run in Windows or Mac platform.

5.3 GUI Implementation

Table 5.1 illustrate the general code structure of GUI component, it firstly parse the configuration file, if there is no configuration file in current working directory, then the HCMP Player will create a new one. The configuration is stored at current working directory with name "`.player_config`", and contains all the HCMP Player related settings, for example, the default remote server address and port number, recently openned file path etc. Configuration is written with plain ASCII text, user could easily modify the configuration. GUI component has 2 operations, new and save to create a new configuration file and save current configurations to the file.

After the configuration parsing and initialization job complete, the the GUI begins to create each subcomponents. Each subcomponent has defined its own object construction function. Note that, to make data accessing easier between different components, we pass in "this" (object itself) to each subcomponents.

```
def GUI {
  data_init() //parse configuration file
  create_menu_panel(this) // initialize all the sub-components
  create_track_panel(this)
  create_data_panel(this)
  create_library_panel(this)
}
```

Table 5.1: GUI Pseudocode Snippet

5.3.1 Binding Method

Table 5.2 is a serpent code snippet, which shows a simple way for GUI object to bind with a method, we bind play button a method, which send a play request to player engine, note that the argument of binding method is a string with method name and parameters separated by `;`. The binding function's type is predefined inside serpent, which has 4 prefined arguments. The first argument is object

itself, the second arugment is event type, thrid and fourth argument is event position, note that not all of the event argument is valid, some events provide 4 arguments, some events may only need 2 arguments. In serpent, all the object event is dispatched through the same user defined fucntion, which give user full power to develop its own customized handlers.

```
Class PlayBtn(button)
  def init()
    super.init()
    this.method = 'send_play'

  def send_play(obj, event, x, y)
    send_player_engine('play;void')
```

Table 5.2: Binding Method

5.3.2 Drawing Canvas

Canvas object define an specified area to be drawable. In serpent, the canvas use the wxWidget canvas object, the canvas object has a full-size bitmap, the drawing method first draw on the bitmap, then the whole bitmap is copied to canvas. The code has been special optimized to deal with bitmap-copy, so the efficiency is not big issue. In serpent, a class need to fulfill two conditions in order to be drawable. First condition it the class need to be a subclass of canvas class, second condition is the class need to define its own paint method. Table 5.3 illustrate the user defined `Keyboard` class, which is to draw a keyboard on canvas.

In HCMP Player, there are two classes which adapts canvas class, `Keyboard` and `MidiNotes` class, the former draws a 108-key keyboard, the latter draws all midi notes into canvas. Each class has a buffer to store all the objects need to be drawn. The play engine periodically send "synchronization" signal, everytime, GUI receive the signal, it will redraw the whole area.

```
class Keyboard(Canvas)
    def init(parent, x, y, w, h, b, c)
      super.init(parent, x, y, w, h)

    def paint(x)
      draw_octave()
```

Table 5.3: Serpent Canvas

5.4 Player Engine Implementation

5.4.1 Create New Thread

Player engine runs inside a separate thread space, when GUI thread complete all the subcomponets' initialization work, it will create a new thread, the newly spawned thread will execute the player engine code. In table 5.4, we use the `thread_create` builtin function of serpent standard library to create new thread, it has 3 arguments, the first argument is to set a period by which newly spawned will wake up and call callback-style function, the mechanism can be used to set a message channel between two threads for communication purposes. The second argument specify which script file to execute for the new thread, the third argument specify the memory size for the new thread.

```
g_object = GUI() //complete constructing GUI object
thread_create(2, "player_engine.srp", 0) // create new thread
```

Table 5.4: Create New Thread

5.4.2 Thread Communication

Serpent has a very primitive multi-thread interface, the thread concept is different from how thread works in C, though serpent thread exist in one process, it does not share any variables and the only communication channel is a message queue. Ser-

pent library provides two builtin functions `thread_send` and `thread_receive` for message passing. Both `thread_send` and `thread_receive` require `thread_id` argument, which represent the caller thread Id. Note that the thread 2 need to define a `portime_callback` function, in order to periodically check the message queue. Table 5.5 illustrate how two threads send and receive request via `thread_send` and `thread_receive` functions.

```
//thread 1 send request
Class PlayBtn(button)
  def init()
    super.init()
    this.method = 'send_play'

  def send_play(obj, event, x, y)
    thread_send(thread_id, "play;void")

------- separte line -------

//thread 2 receive request
def porttime_callback(ms)
  cmd = thread_receive(thread_id)
  engine.process_cmd(cmd) //process command string
```

Table 5.5: Push & Pull Pattern

5.4.3 Player Engine

Table 5.6 is a pseudocode snippet of player engine, its main job is to read from GUI component through a message queue, parse the command string(the command string structure is defined chapter 3), and then look up a state-method matrix(refer to table 3.2 and 3.3) fetch the method to transit from current state to new state. Note that this segment of code snippet is also embeded within the `portime_callback` function.

```
def player_engine {
  cmd_str = read_from_GUI() //Player engine read request from GUI
  cmds = parse_cmd_str(cmd_str) // parse command string into an array
  method = cmds[0]
  parameters = cmds[1]
  apply_function(method, parameters)
  cur_state = update_state(cur_state, method)
}
```

Table 5.6: Player Engine Pseudocode

5.5 Network Implementation

5.5.1 Initiate Connection

Table 5.7 is a pseudocode snippet that illustrate how HCMP Player and Conductor build connection. Basically, it is a remote procedure model. With Conductor in client side and HCMP Player in server side, HCMP Player initiate the connection and wait for reply from Conductor. The underlying connection related function is provided inside zeromq library.

5.5.2 Maintain Connection

Table 5.8 is the pseudocode snippet of observer pattern, which is used by HCMP Player and remote server for network communication. The code has two part, subject and dependent, in real implementation, conductor is subject and HCMP Player is dependent. To begin communication , subject need to bind to certain port of local machine first, once a dependent register into the list, the subject begin to “publish” messages. In this situation, dependent just wait for comming messages and handle it.

```
// client side
def server {
  srv = socket (context, ZMQ_REP);
  srv.bind ("tcp://*:5555");
  srv.send(con_request);
}

------- separate line -------

// client side
def client {
  cli = socket (context, ZMQ_REP);
  cli.bind ("tcp://*:5555");
  con_req_str = cli.recv();
  reply = engine.process_req(con_req_str);
  cli.send(reply);
}
```

Table 5.7: Initiate Connection Request

```
// subject thread
def subject {
  context = zmq_ctx_new ()
        rc = zmq_bind (subject, "tcp://*:5556");
        rc = zmq_bind (subject, "ipc://HCMP_Player.ipc");

  while (1) {
    get_data_from_conductor();
    s_send (subject, update);
  }
  zmq_close ();
}

// dependent thread
def dependent {
  context = zmq_ctx_new ();
  subscriber = zmq_socket (context, ZMQ_SUB);
  rc = zmq_connect (dependent, "tcp://localhost:5556");

  rc = zmq_setsockopt (dependent, ZMQ_SUBSCRIBE)
  cmd_string = s_recv (dependent);
  send_player_engine('cmd_str')

  zmq_close ();
}
```

Table 5.8: Observer Pattern Pseudocode

6

Evaluation

6.1 Overview

Chapter 6 mainly focus on evaluation of the project. In this chapter, we define the successful criteria of the project and discuss how to meet those requirements. A complete list of required features is listed. At last part of chapter, we also make brief performance test and discuss some potential performance issues.

6.2 Functional Evaluation

Based on the project design from previous chapters, the evaluation of the project mainly contains two parts. To reach the successful criteria, the project not only need to satisfy the functional requirement but also all of performance requirements. The functional requirement cover all the basic function of the HCMP Player, each functional requirement will be either manully tested or automatically tested by scripts. As for the performance test, some code segment will be added to original program to test the general performance issues.

6.2.1 Stand-alone Mode Requirement

1. User should be able to load and play normal midi files.

2. User should be able to play, pause, stop while playing midi file.

3. User should be able to freely adjust tempo, while plying the midi file.
4. Data visualization component should correctly show the midi note and synchronize with audio stream while playing the midi file.
5. Each track component should match to color on data visualization component.
6. Each track component should be able to mute, solo and adjust volume.
7. The player should contain a simple midi library to manage midi files. The library should automatically remember recently imported midi files.
8. The player should provide a complete configuraion interface for the user, all user's setting should be saved automatically.
9. The configuration of HCMP Player should be saved into a file.

6.2.2 Network Mode Requirement

1. Each instance of HCMP Player runs in its own process space, one player should not effect other player's function or performance.
2. Each HCMP Plyaer should be launched separately by user. After launching, the midi player should automatically connect to a conductor based on its configuration.
3. If HCMP Player failed to establish connection with remote server, it should be rollback to stand-alone mode without crashing.
4. The Conductor communicate with midi player through a predefined API, the API is defined in Chpater 5. After establishing connection with remote server, HCMP Player should register into Conductor's dependent list. Once received command, the Conductor send message to all HCMP Player that previously registered in the dependent list.
5. Multiple instances of midi player should be able to connect to one conductor.
6. After establishing connection with conductor, the midi player should be able to receive tempo informaiton from conductor and synchronize with it.

7. The conductor should be able to explicitly command certain midi player to load specified configuration file or it could directly send configuration information through network to that midi player.

8. The HCMP Player is compatible with ZeroMQ (3) api, and 4 core functions will be like

    - play - ask a player to start playing
    - stop - stop a player from playing
    - update time-map - send new time-map to the player
    - set position - set position to certain beat

6.2.3 Performance Requirement

1. The HCMP Player GUI part should always response to user input without significant delay, and the GUI thread should not hang there.

2. The HCMP Player should response to conductor without significant delay.

3. The HCMP Player should use no more than 100MB memory.

4. The HCMP Player should use reasonable amount of CPU while playing midi file.

5. Hopefully, the midi player shoud not crash without any reason.

6.3 Test Plan

6.3.1 Functional Test

Functional test aims to test all the features list above, some of key test cases are list below.

- Test case 1 – load 10 different midi files, each has different length, including a ill midi file, play, pause and stop.
- Test case 2 – load 10 different midi files, change tempo from high to low, and then from low back to high, change the tempo every 0.5 second.

- Test case 3 – load 10 different midi files, use track panel to mute from first track to last track, then do the similar action with volume adjust.
- Test case 4 – load 10 different midi files in HCMP Player library, click each track to import, then delete all the files in disk, and clikc each track again.
- Test case 5 – create a virtual conductor and create a HCMP Player use network function to connect to conductor. Conductor issue command to test each network API and make sure its work properly.
- Test case 6 – create a virtual conductor and create 10 HCMP Players. All connected to conductor. Conductor issue command to test each network API and make sure its work properly.
- Test case 7 – create a virtual conductor and create a HCMP Player to connect to conductor. While HCMP Player playing the file, cut the network connection, and HCMP Player behave properly.
- Test case 8 – create a virtual conductor and create 10 HCMP Players, all connect to conductor. Conductor explicitly ask each HCMP Player to play with certian configuration file.

6.3.2 Perfermance Test

Performance test aims to test the general performance of HCMP Player. Two important aspects for performance test are canvas redraw test and midi event test.

As described in previous chapters, canvas redraw the whole bitmap everytime the beat update, the bitmap will be updated approximately 20 times per second. Even though the bitmap size is relatively small, it still possible to become a bottelneck for the whole system, any inefficiency in the code will lead to GUI lag or hang there without responding to user input. The test case 1 try to measure the general accumlative time spent on redrawing.

Another issue is about midi event, the player engine use real-time scheduler to contantly poll and execute event, ideally, the event is dispatched at its schedule time, or at least we should control so that event dispatched within certain limited time, the test case 2 aims to measure the overall latency between ideal schedule time and actual dispatch time.

6.3.2.1 Performance Test Case

- Test case 1 – measure the time between canvas redraw segment code, and add each redraw time together to gain total time spend on drawing, calculate the ratio of redraw time and total time.
- Test case 2 – In player engine, after the event has been dispatched, check the ideal time stored in that event, and get current time, calculate the latency. change the tempo, and measure the latency again.

7

Conclusion

7.1 Future Work

This paper discusses various kinds of issue regarding to the development of HCMP Player, the HCMP Player itself is part of a larger HCMP project that facilitates music representation, preparation, and performance, and there are some components can be built upon HCMP Player and some can be directly intergrate with HCMP Player. Two features can be added to further extend the HCMP Player's function.

7.1.1 Integrate with Score Following

HCMP has a score following component which is also developed with serpent. Figure 7.1 is a screenshot of score following component. The score following component can be used to notate music score, turning "pages". The original score following component has a builtin player function, and it can easily replaced by HCMP Player. With predefined API of HCMP Player, score following component can be easily extended to cooperate with HCMP Plyaer. The HCMP Player will synchronize with score following componet by constantly receiving command through network.

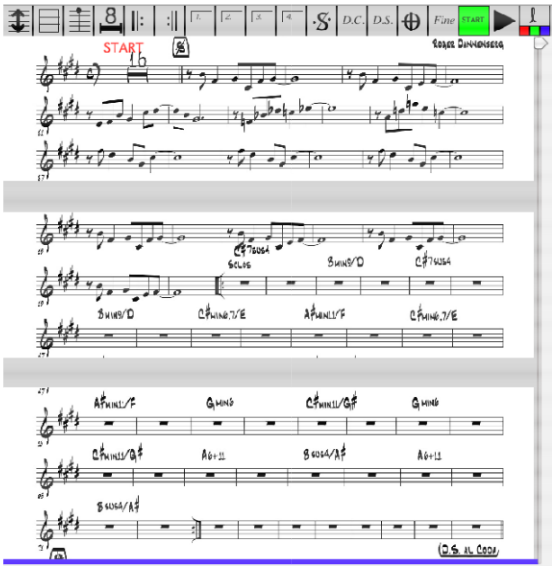

Figure 7.1: HCMP Score Display Component

7.1.2 Integraet with Midi Database

Another interesting work is about midi database(6) which is based on Dawen Liang's previous work. The basic idea of midi database is to create a tool by which user is able to quickly record, organize and retrieve audio information from various sources. The midi database define a full set of API for its clients. One possible extention of HCMP Player is to integrate midi database into HMCP Player library function, so that user can easily use HCMP Player to search and play midi segement from previous history database.

7.2 Conclusion

From HCMP project's perspective, HCMP Player is a building block of a larger project, upon which more powerful component can be built, it can also cooperate with other components to complete complex job, or be used by other components. From developer's perspective, HCMP Player provides a complete set of API which can be extended and tailored to fit into more sophiscated project. The HCMP Player can be split into 3 parts, GUI, player engine and network. The design and usage of each part is explained in detail in previous chapters. I also briefly decribe

implementation of each part, as supplementary, some pseucode segment is provide to illustrate logic. At the end, complete HCMP Player features is listed and a successful criteria is defined to help evaluate the project.

Bibliography

[1] Framework for Coordination and Synchronization of Media, D.Liang, G.Xia and R.Dannenberg, NIME 2011. 4

[2] http://www.zeromq.org

[3] http://sourceforge.net/p/livedisp/wiki/HCMP_messagingi_protocol/ 41

[4] http://www.cs.cmu.edu/ music/aura/serpent-info.htm 31

[5] http://www.wxwidget.org 31

[6] Segmentation, Clustering, and Display in a Personal Music Database for Musicians, G.Xia, D.Liang, R.Dannenberg, ISMIR 2011. 46

[7] Dannenberg, R. Real-time scheduling and computer accompaniment. In Current Directions in Computer Music Research, edited by Max. V. Mathews & John R. Pierce, MIT Press, Cambridge, MA, 1989, pp.225-261. 3

[8] Ableton. Ableton reference manual (version 8). http://www.ableton.com/pages/downloads/manuals (2011). 3

[9] Robertson, A. and Plumbley, M. D. B-Keeper: A beat tracker for real time synchronisation within performance. Proceedings of New Interfaces for Musical Expression (NIME 2007), New York, NY, USA, (2007), pp 234-237.

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; This paper has not previously been presented in identical or similar form to any other foreign examination board.

The thesis work was conducted from 02/14/2013 to 05/15/2013 under the supervision of Prof. Roger B. Dannenberg at Carnegie Mellon University.

Signiture: