THINC Client in JAVA

Young Jin Yoon <yy2223@columbia.edu>
Department of Computer Science
Columbia University

Table of Contents

Chapter 1. Objectives

Implement THINC Client for PDA in JAVA with full functionality. Based on THINC protocol specification, it should handle all possible frame buffer, video, cursor, and sound messages from the server. Moreover, it should send proper messages to the server when an event from user occurs, such as mouse move and click, resize the window and key press.

Chapter 2. Development Environment

2.1. For computer

CPU: Intel Pentium Mobile 1.73Ghz

Memory: 1GB

OS: Microsoft Windows XP Home edition Version 2002 Service Pack 2

Language: JAVA

JVM: Sun J2SE 1.5.0_08

IDE: Eclipse SDK version 3.2.0

Library used: PNG decoder(com.sixlegs.png)

2.2. For PDA

Model : DELL AXIM X51v

CPU: Intel PXA270 (StrongARM)

Memory: 49.43MB

OS: Microsoft Windows Mobile Version 5.0

Chapter 3. Why not PDA?

First of all, I couldn't implement THINC Client for PDA in JAVA. In This Chapter I will try to describe why I could not implement it, it what I have done.

3.1. Feasibility Study

I searched various areas to find the proper Java Virtual Machine for the PDA described on Section 2. In JAVA, there are three platform specified by Java community. For PDA and cell phone, Java Platform, Micro Edition (J2ME) is suitable to use. In J2ME, there are profile and configuration, which described as below by Sun:

> • *Configuration*. A J2ME configuration defines a minimum platform for a "horizontal" class or family of devices, each with similar requirements on total memory budget and processing power. A configuration defines the Java language and virtual machine features and minimum libraries that a device manufacturer or a content provider can expect to be available on all devices of the same class.
> • *Profile*. A J2ME profile addresses the specific demands of a certain "vertical" market segment or device category. The main goal for a profile is to guarantee interoperability in a certain vertical device category or domain by defining a standard Java platform for that market. Profiles may include libraries that are far more device category specific that libraries provided in a configuration. Profiles are implemented on top of a configuration.

Because configuration denotes the type of operated devices, I had to choose the configuration between CLDC and CDC. Because CLDC denotes Limited devices such as cell phone, it seems to choose CDC better. However, because Java Virtual Machine for CDC has not completely configured and settled as such a good JVM package, choosing CLDC seems like the only solution for J2ME. The goal of CLDC is supporting limited devices described below:

> Connected, Limited Device Configuration (CLDC)
> - 160 kB to 512 kB of total memory budget available for the Java platform
> - a 16-bit or 32-bit processor
> - low power consumption, often operating with battery power
> - connectivity to some kind of network, often with a wireless, intermittent
> - connection and with limited (often 9600 bps or less) bandwidth

There are various JVM which supports CLDC or CDC configuration. I described pros and cons for each Java Virtual Machines on Table 1.

JVM name	Publisher	Advantage	Disadvantage
J9 JVM	IBM	Supports a lot of different environment such as Windows mobile, Pocket Linux, etc. Supports both CDC and CLDC.	Commercial Product, No PNG library
CrEme	NSICOM	CDC support for recent version, Supports good AWT interfaces	No PNG library
Personal Java	Sun	Most stable, made by Sun	Not maintained anymore, limited functionality, No PNG and ZIP library
Mysaifu	Haneta (Personal)	Almost Full J2SE support, GPL licensed	Unstable, Out of memory error
SuperWaba		Good support for PDA devices, PNG and ZIP support	Commercial for Additional feature, such as PNG library, Not following Standard Java specification
Jeode	Esemertec	Supporting Dell Axim X5	Commercial and does not providing SDK, No PNG and ZIP library
Kaffe	Open Source	Open Source JVM.	Implementation is not complete and contains bugs, Linux only

Table 1 Various JVM for PDA

To implement PDA version of THINC Client, It should supports PNG and ZIP library. Because I do not have much time to make PNG and ZIP library from the scratch, I cannot implement PDA version of THINC. If JVM supports those libraries, however, it has some commercial license to pay about 100 dollars, which cost a lot. Even I can have the money to buy them, there is no promise that I can complete implementation with those Commercial JVM.

Because of that, I tried to use Mysaifu JVM, which is suitable to use PNG library from

J2SE. However, it did not work because of absurd out of memory error. Even I tried to minimize the size of the code and memory occupancies, it did not work, still.

As a result, I couldn't make any THINC client for PDA in JAVA. To make this, we have to negotiate available compression library between client and server, which is slightly different from current implementation of THINC. I will describe proposal of revised THINC protocol on Chapter 8.

3.2. Not for PDA, but for J2SE

From the provided THINC Client source code in JAVA, It does not have any cache, video, resize, and SSL implementation. Moreover, the color of image from network seems to process improperly from the client. It seems like well object-oriented from first draft, however, it does not maintained properly because it has hard-coded handshaking stages, and improperly added PNG library features. It also does not have any comments for next developer.

For that reason, I decided to rewrite the source code of THINC Client in JAVA, which should be object-oriented, well-commented with full functionality. First of all, the reason of improper image color is caused by ARGB format itself. Because Java BufferedImage class prefers to use ABGR rather than RGBA or ARGB format from the server, I had to make convert methods to convert RGBA and ARGB into ABGR pixel format. Next, I redesigned the whole class diagram for THINC Client in JAVA. I designed ThincHandler, ThincMsg, ThincMsgFlag and ThincVideoFormat classes, and try to derive every class in THINC Client from those classes. Based on these classes, I rewrote entire source code for speed of THINC Client and implemented Video, SSL, resize and cache in new version of THINC Client. For detailed information about redesign for THINC Client, I will explain in Chapter 5.

I also concentrated to add a comment for next developers. Because it is only one semester project for me, there would be another student who will improve or make another THINC Client for PDA or normal client version of JAVA. For that reason, it is very important to add comments for maintaining the source code.

Chapter 4. Project Plan

In this chapter, I will describe what my project plan was, and what is changed as time goes by.

4.1. First Project Plan

Before I received the code, I thought it works properly. Because I set my first project plan beyond of that assumption, it seems to be awkward for now. However, it is described as Table 2.

Start from	End to	Issues	Description
1.Sep.06	16.Sep.06	Feasibility study	Search the PDA version of JVM as many as possible
9.Sep.06	23.Sep.06	Feasibility study	Test code for Network, Image in PDA simulator
17.Sep.06	30.Sep.06	Feasibility study	Analyzing source code of THINC Client in JAVA
17.Sep.06	30.Sep.06	Feasibility study	Analyze the THINC Protocol specification
1.Oct.06	16.Oct.06	PDA Assigning	Assign the PDA
1.Oct.06	16.Oct.06	PDA Feasibility	Find the PDA supported JVM
7.Oct.06	16.Oct.06	PDA Feasibility	Test code for Network, Image in PDA
11.Oct.06	16.Oct.06	PDA Feasibility	Try to send some init messages to server
17.Oct.06	6.Nov.06	Class Design	Class Design PDA Version of THINC
7.Nov.06	17.Nov.06	Implementation	Implement PDA Version of THINC
10.Nov.06	20.Nov.06	Function Test	Functional test for each messages
21.Nov.06	1.Dec.06	Integration Test	Test with full functionality
7.Nov.06	10.Dec.06	Debugging	Should be with function/integration test
10.Dec.06	13.Dec.06	Write a Report	Write a final report
10.Dec.06	15.Dec.06	Preparing presentation	Make example case for presentation

Table 2 First draft of Project plan

4.2. Revised Project Plan

As soon as I found the source code is not working properly, I decided to maintain both normal JAVA client and PDA client. Figure 2 is the revised version of my Project Plan, which considers both PDA and normal client for JAVA. Bolded letter represents refactoring issues for my revised Project Plan.

Start from	End to	Issues	Description
1.Sep.06	16.Sep.06	Feasibility study	Search the PDA version of JVM as many as possible
9.Sep.06	23.Sep.06	Feasibility study	Test code for Network, Image in PDA simulator
17.Sep.06	30.Sep.06	Feasibility study	Analyzing source code of THINC Client in JAVA
17.Sep.06	30.Sep.06	Feasibility study	Analyze the THINC Protocol specification
23.Sep.06	**7.Oct.06**	**Refactoring**	**Redesign THINC Client in JAVA**
1.Oct.06	16.Oct.06	PDA Assigning	Assign the PDA
1.Oct.06	16.Oct.06	PDA Feasibility	Find the PDA supported JVM
7.Oct.06	16.Oct.06	PDA Feasibility	Test code for Network, Image in PDA
11.Oct.06	16.Oct.06	PDA Feasibility	Try to send some init messages to server
17.Oct.06	30.Oct.06	Class Design	Class Design PDA Version of THINC
31.Oct.06	**6.Nov.06**	**Refactoring**	**Rewrite the source code of THINC Client in JAVA**
7.Nov.06	17.Nov.06	Implementation	Implement PDA Version of THINC
10.Nov.06	20.Nov.06	Function Test	Functional test for each messages
21.Nov.06	1.Dec.06	Integration Test	Test with full functionality
7.Nov.06	10.Dec.06	Debugging	Should be with function/integration test
10.Dec.06	13.Dec.06	Write a Report	Write a final report
10.Dec.06	15.Dec.06	Preparing presentation	Make example case for presentation

Table 3 Revised draft of Project plan

4.3. Actual works

I followed my Project Plan until October except feasibility study for PDA. I tried to find another way to implement PDA without PNG library. First I tried not to use PNG or ZIP library for PDA version. I made some test code for that, however it didn't work because the server tells what compression algorithm will be used without any negotiation. Next I tried to modify the open source of PNG library. Because it is totally based on standard J2SE, I cannot modify it into J2ME version because of complexity. Last, I tried to make my own PNG library from the specification from libpng.org. However, it failed because it is too complicated to implement all options in PNG format. So the best way to do was just keep doing normal client version with feasibility study for PDA.

Finally, I found mysaifu JVM which support almost every specification of J2SE. I thought I can maintain two versions (PDA version and normal version) with only one code if I use those mysaifu JVM. I tested java swing components and JPanel implementation of mysaifu JVM with some image. Moreover I tested socket operation by using SocketChannel class in J2SE. However, when I try to draw initial frame buffer onto the screen, mysaifu JVM causes the out of memory error.

The unimplemented feature of previous THINC Client in JAVA contributes to delay my project progress. Because the resize and video stream messages were little bit different from THINC Protocol specification I received, I have to figure out with the server side. Even I did not test the sound because it is not configured well on the server side. Basically, it is not a problem from maintainer; however, the sound seems like not configured easily on the server side.

As a result, I could make THINC Client in JAVA, but not suitable for PDA. I think we have to modify the handshaking stages in THINC Protocol to make THINC useful to limited devices such as PDA. Based on my experiences, because J2ME does not have good supported PNG or ZIP library currently available, it should be disabled by the client side with negotiation.

Chapter 5. Design and Implementation

In this Chapter, I will explain the design of THINC Client in JAVA. First I will show the entire class diagram for THINC Client, and then I will explain part by part including what is revised.

5.1. Class Diagram

Figure 1, 2 and 3 are the Class diagram for final implementation of THINC Client, in JAVA. Because its classes are more than 100, It cannot be shown in this document. I attached <dia.jpg> for entire class diagram. If you have UML tool, you can see it by opening < (default package).uml> file.

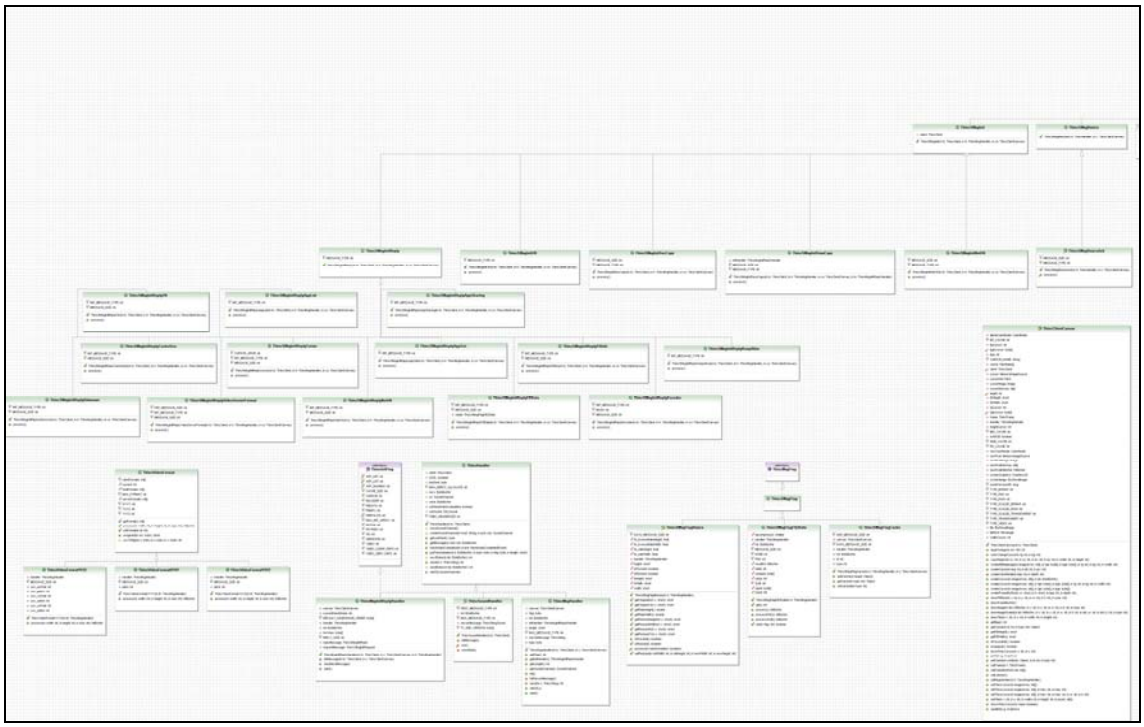

Figure 1 Class diagram for THINC Client in JAVA (part 1)

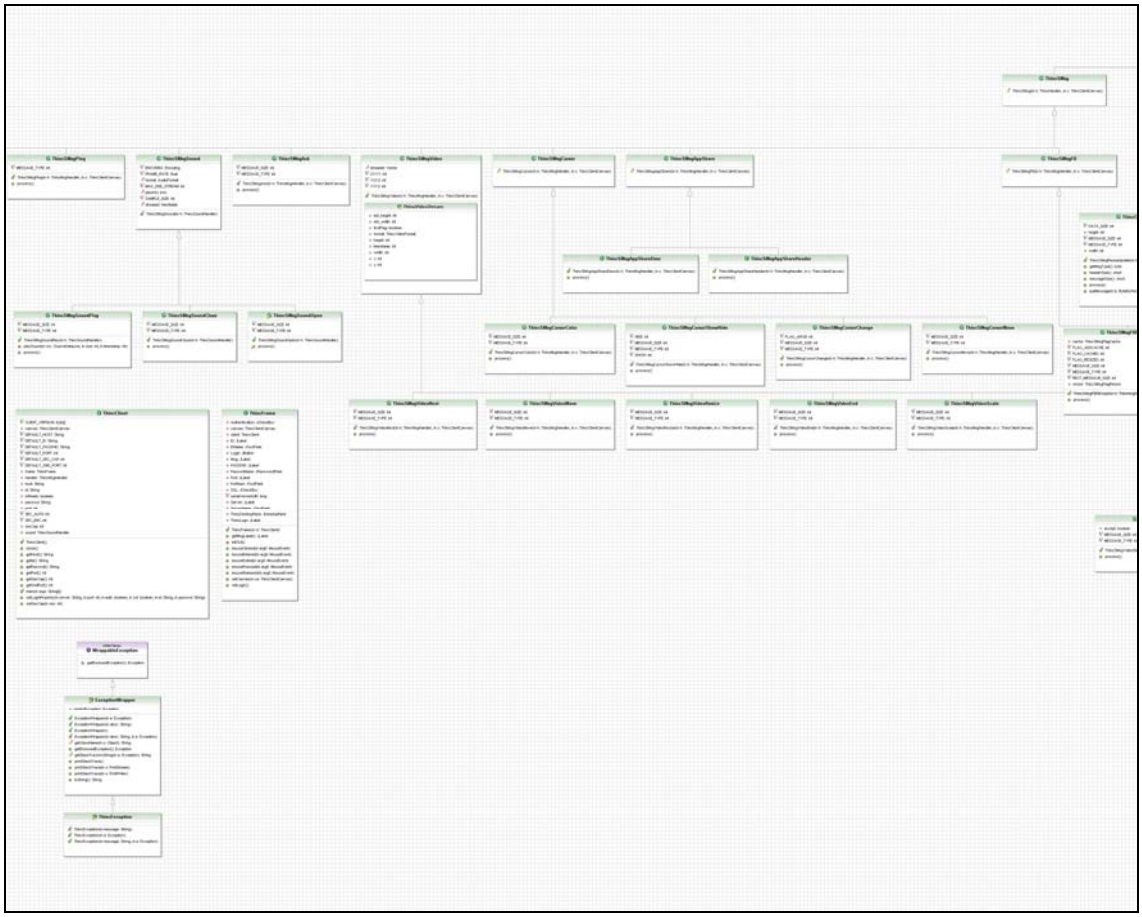

Figure 2 Class diagram for THINC Client in JAVA (part 2)

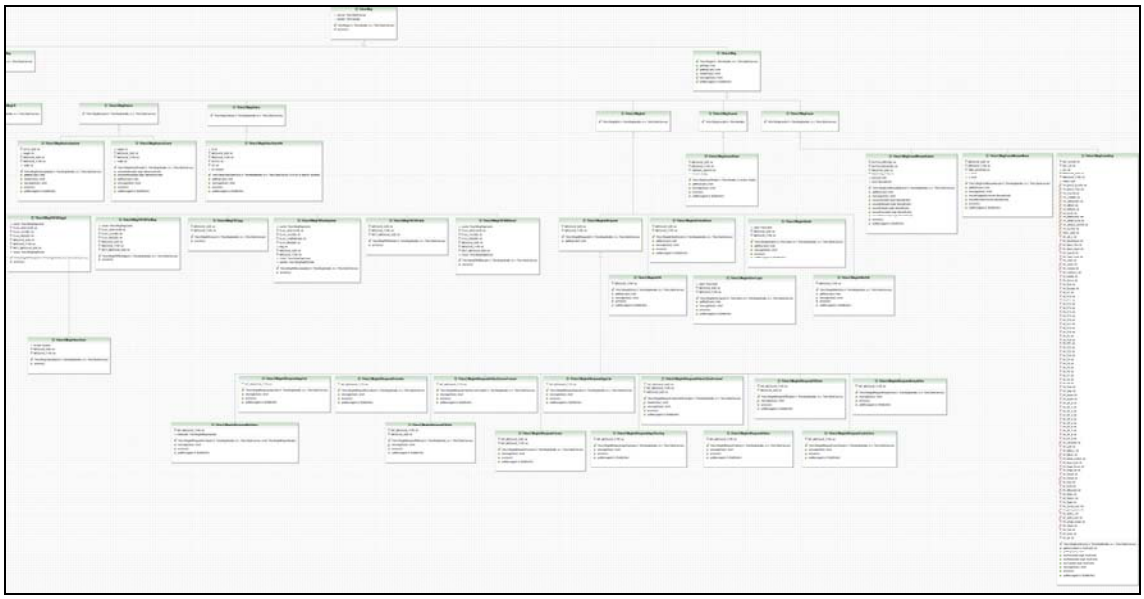

Figure 3 Class diagram for THINC Client in JAVA (part 3)

It seems that hard to understand. However, the entire structure for THINC Client in JAVA does not hard as expected. Simple diagram to describe how it works is in Figure 4.

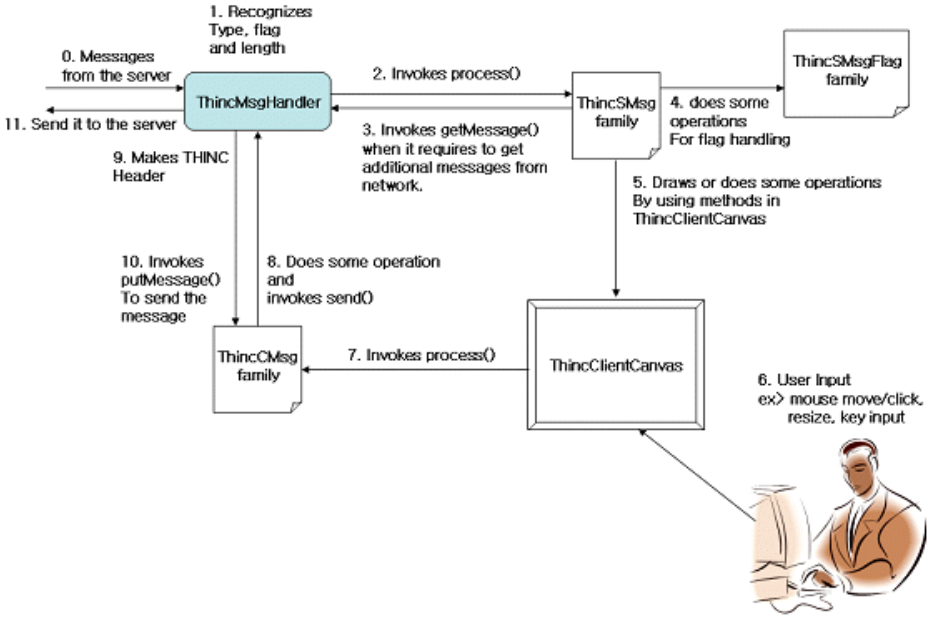

Figure 4 how THINC Client in JAVA works?

When the message comes from the server, first ThincMsgHandler receives the message and recognize the type, flag and length of the message. And then it passes the control to proper ThincSMsg derived classes (call it ThincSMsg family in figure). When ThincSMsg derived classes need to get more messages, it simply invokes getMessage method in ThincMsgHandler class. If it has flags to process, it is processed by ThincMsgFlag family. When it is ready to draw images on frame buffer, or modify the information of the cursor, it calls various methods in ThincClientCanvas.

When user inputs valid events, it calls process method in proper ThincCMsg family by ThincClientCanvas. In process method, it processes the event and get the information from the event. Whenever ready to send, it invokes send method in ThincMsgHandler class. And then ThincMsgHandler puts proper THINC header automatically then calls putMessage method in ThincCMsg family to put proper message followed by. After finished, it sends the message to the server.

Message for sound works similar. However, because there is no such ThincCMsg for sound, ThincSoundHandler only handles the message from the server and yield the control into proper ThincSMsg family.

5.2. ThincHandler family

ThincHandler family is for handling all socket operation. It masks socket implementation from ThincMsg family for easy implementation. Moreover, it also analyzes THINC header from the message and yield the control to a proper ThincSMsg derived class. It provides implemented methods for socket operation. Methods provided by ThincHandler are:

createSocketChannel()	Creates socket connection
closeSocketChannel()	Closes socket connection
send()	Send ThincCMsg family to the server
putThincHeader()	Put THINC header to send (only used as private)
sendBytes()	Primitive method for sending message (used when initial handshake and in send() method)
getMessage()	Get message from the network (used in ThincSMsg)
recvBytes()	Primitive method for receiving message (used when initial handshake and in getMessage() method)
getLastSent()	Get THINC type of last sent
setSSLSocketChannel()	Set socket as SSL mode
handshakeCompleted()	Only invoked when handshake for SSL is completed. (defined as private type)

Detailed Class diagram for ThincHandler is described in Figure 5.

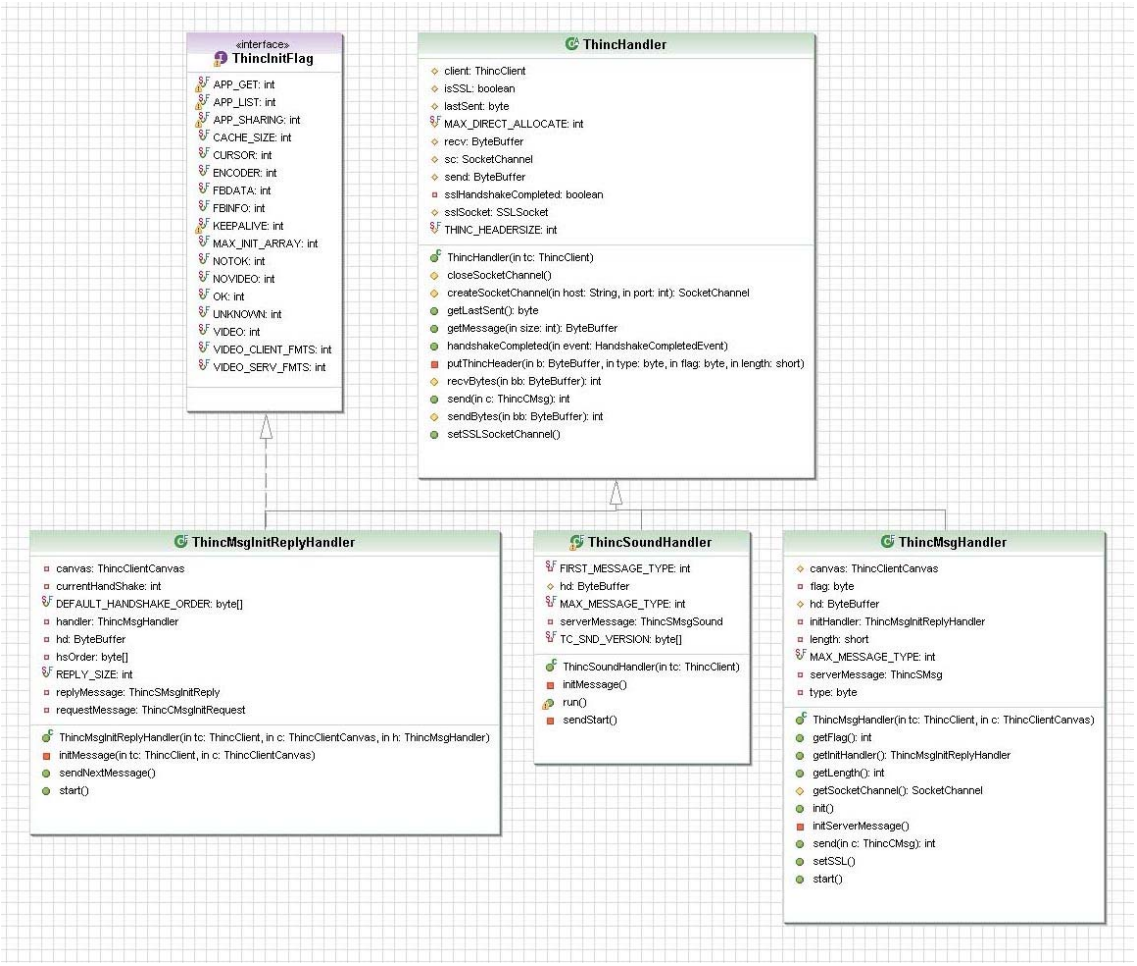

Figure 5 Class diagram for ThincHandler family

5.3. ThincMsg family

ThincMsg family consists of ThincCMsg family and ThincSMsg family. ThincCMsg family is for send the message from the client; while ThincSMsg family is for receive the message from the server. It provides abstract process() method for each derived classes. Usage for that method is:

process()	Processes the actual message in here. In case of ThincSMsg, it is invoked by ThincMsgHandler. For ThincCMsg, it is invoked by ThincClientCanvas, generally.

5.3.1. ThincCMsg family

ThincCMsg family is for creating the message and sending it to the server. It provides abstract messageSize(), putMessage(), getMsgType(), headerSize() and getFlag() methods for each derived classes. Usage for each method is:

messageSize()	Returns the total message size when it generates message.
putMessage()	Put the message into the buffer to be sent.
getMsgType()	Returns the type of message when ThincMsgHandler generates THINC header.
headerSize()	Returns the header size of the message(not THINC header) when it required in ThincCMsgInitReply.
getFlag()	It is only for future implementation. It returns the flags when ThincMsgHandler generates THINC header. Currently it is set to 0 as default.

Lists and works for each ThincCMsg family except ThincCMsgInitRequest are as follows:

ThincCMsgEventKey	Create the message from the key input of user
ThincCMsgEventMouseButton	Create the message from the mouse click/release of user
ThincCMsgEventMouseMove	Create the message from the key move of user
ThincCMsgInitAuth	Create the message from the id/password of user
ThincCMsgInitClientDone	Create the message that handshake is done
ThincCMsgInitOK	Create the message for ACK of client
ThincCMsgInitNotOK	Create the message for NAK of client
ThincCMsgInitSecCaps	Create the message for security capability of client
ThincCMsgResizeEvent	Create the message for resize event of client
ThincCMsgResizeUpdate	Create the message for resize update region of client

5.3.1.1. ThincCMsgInitRequest family

ThincCMsgInitRequest family is for creating the initial handshake request message and sending it to the server. Because init request have additional header for its request, it is much easier to make each class for each additional header.
Lists and works for each ThincCMsgInitRequest family are as follows:

ThincCMsgInitRequestCacheSize	Create the message for initial cache size request
ThincCMsgInitRequestCursor	Create the message for initial cursor request
ThincCMsgInitRequestEncoder	Create the message for encoder request
ThincCMsgInitRequestFBData	Create the message for initial frame buffer data request
ThincCMsgInitRequestFBInfo	Create the message for initial frame buffer information request
ThincCMsgInitRequestNoVideo	Create the message for supporting no video
ThincCMsgInitRequestVideo	Create the message for supporting video
ThincCMsgInitRequestVideoClientFormat	Create the message for supported video client format
ThincCMsgInitRequestVideoServerFormat	Create the message for supported video server format request

5.3.2. ThincSMsg family

ThincSMsg family is for handling the message from the server and drawing frame buffer, operating with cursor, and processing for video stream. It does not provide any method for derived class because process() method in ThincMsg can handles server message processing.

Lists and works for each ThincSMsg family except ThincSMsgInitReply are as follows:

ThincSMsgAck	Handle the message for ACK from the server
ThincSMsgCursorChange	Handle the message for cursor change from the server
ThincSMsgCursorColor	Handle the message for changing cursor color from the server
ThincSMsgCursorMove	Handle the message for cursor move from the server
ThincSMsgCursorShowHide	Handle the message for cursor show/hide from the server
ThincSMsgFBCopy	Handle the message for frame buffer copy from the server
ThincSMsgFBFillBilevel	Handle the message for filling bitmap pixel to the frame buffer from the server

ThincSMsgFBFillGlyph	Handle the message for filling glyph bitmap pixel to the frame buffer from the server
ThincSMsgFBFillPixMap	Handle the message for filling pixmap pixel to the frame buffer from the server
ThincSMsgFBFillSolid	Handle the message for filling solid to the frame buffer from the server
ThincSMsgFBRawUpdate	Handle the message for filling raw image to the frame buffer from the server
ThincSMsgInitNotOK	Handle the message for NAK for handshake from the server
ThincSMsgInitOK	Handle the message for ACK for handshake from the server
ThincSMsgInitSecCaps	Handle the message for security capability from the server
ThincSMsgInitSessCaps	Handle the message for security capability ACK from the server
ThincSMsgSoundClose	Handle the message for sound stream close from the server
ThincSMsgSoundOpen	Handle the message for initial sound config from the server
ThincSMsgSoundPlay	Handle the message for playing sound from the server
ThincSMsgVideoEnd	Handle the message for video stream close from the server
ThincSMsgVideoMove	Handle the message for moving the location of video from the server
ThincSMsgVideoNext	Handle the message for playing video from the server
ThincSMsgVideoResize	Handle the message for resizing the size of video from the server
ThincSMsgVideoScale	Handle the message for video scaling from the server
ThincSMsgVideoStart	Handle the message for video stream open from the server

5.3.2.1. ThincSMsgInitReply family

ThincSMsgInitReply family is for handling the initial handshake reply message and sending it to the server. Because init request have additional header for its request, it is much easier to make each class for each additional header.
Lists and works for each ThincSMsgInitReply family are as follows:

ThincSMsgInitReplyCacheSize	Handle the reply for initial cache size from the server
ThincSMsgInitReplyCursor	Handle the reply for initial cursor information

	from the server
ThincSMsgInitReplyEncoder	Handle the reply for encoder type from the server
ThincSMsgInitReplyFBData	Handle the reply for initial frame buffer data from the server
ThincSMsgInitReplyFBInfo	Handle the reply for initial frame buffer information from the server
ThincSMsgInitReplyNotOK	Handle the reply for initial request NAK from the server
ThincSMsgInitReplyOK	Handle the reply for initial request ACK from the server
ThincSMsgInitReplyUnknown	Handle the reply for initial request is unknown from the server
ThincSMsgInitReplyVideoServerFormat	Handle the reply for server video format from the server

5.4. ThincMsgFlag family

ThincMsgFlag family is for receiving message related to flag in THINC header. When there is a flag in THINC header, the following message should contain some information about flags before handling actual THINC messages. It is made for handling those additional messages. It does not provide any method for derived class because it is too vary to handle all flag types in one method. Detailed Class diagram for ThincMsgFlag family described in Figure 6.

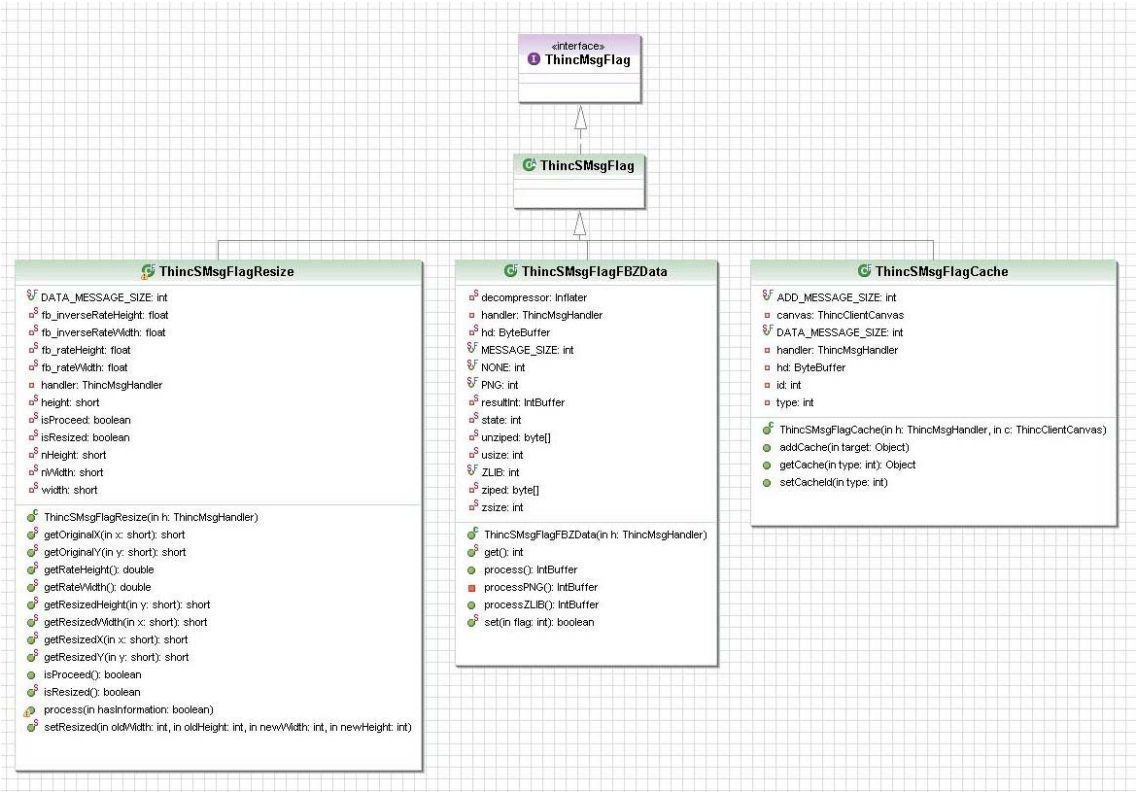

Figure 6 Class diagram for ThincMsgFlag family

5.4. ThincVideoFormat family

ThincVideoFormat family is for handing video stream in THINC messages. Because each video stream has there own video format to process, THINC Client should handle each video stream differently. Currently it support YV12, UYVY and YUY2 format by ThincVideoFormatYV12, ThincVideoFormatUYVY, and ThincVideoFormatYUY2, respectively. Detailed class diagram for ThincVideoFormat family is described in Figure 7.

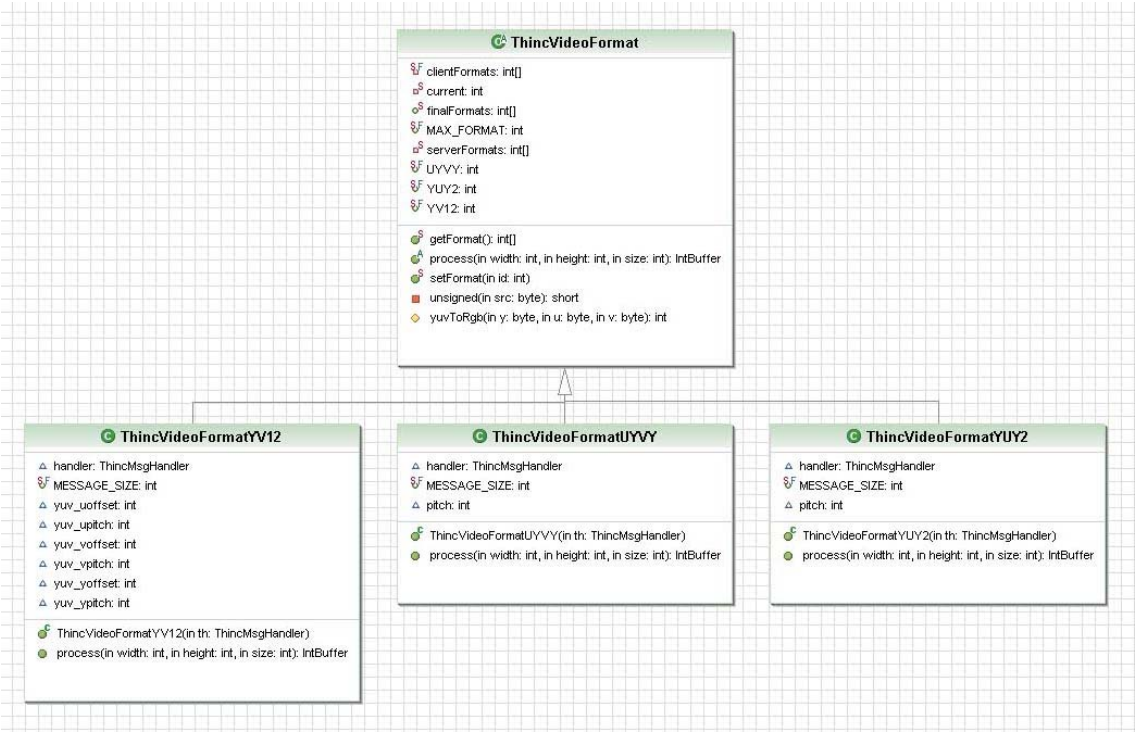

Figure 7 Class diagram for ThincVideoFormat family

5.5. ThincClientCanvas

ThincClientCanvas is for drawing actual image to the screen, for cache handling and for cursor handling. It masks complicated JWT Framework operation from ThincMsg family. It generally derived from JPanel in standard JWT Frameworks which replaces Canvas in AWT. Name of the class originated from Canvas in AWT and it works as expected. It draws frame buffer onto the screen periodically and handles all information about cursor, key input, mouse movement and cache for various image types. Detailed class diagram is described in Figure 8.

Figure 8 Class diagram for ThincClientCanvas

5.6. ThincFrame

ThincFrame is derived class from JFrame in standard JWT Frameworks which replaces Frame in AWT. It contains ThincClientCanvas object and login window as Figure 9. When user puts the login information to this, it passes the information to the ThincClient class, and then ThincMsgHandler, ThincSoundHandler will use that information from ThincClient class. Detailed class diagram is described in Figure 10.

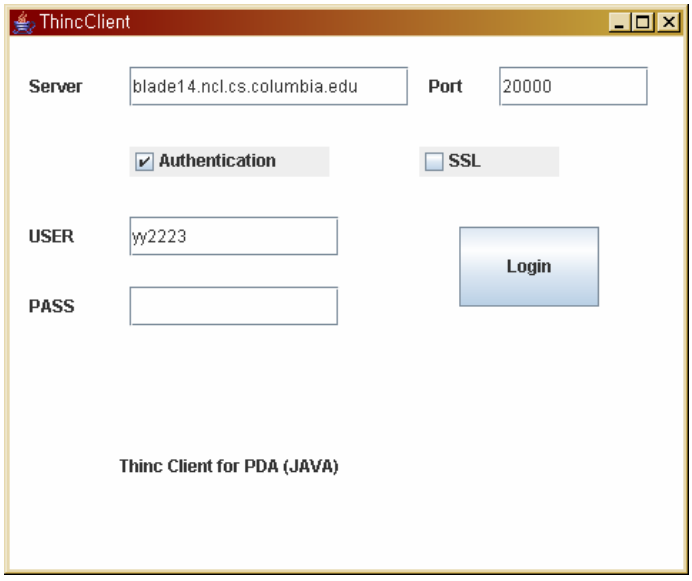

Figure 9 Login Window for THINC

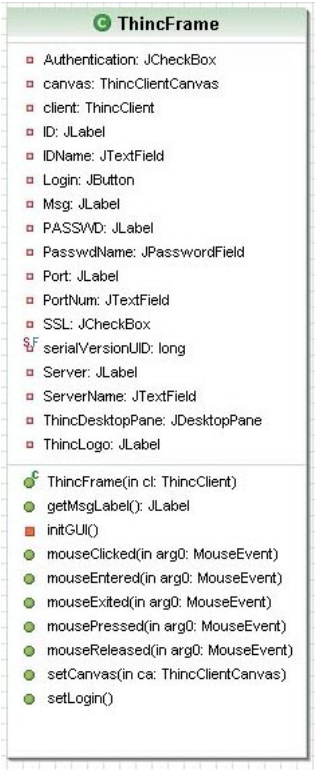

Figure 10 Class diagram for ThincFrame

5.7. ThincClient

ThincClient is the class which contains main function. It also has various user and client information including handshake information. Detailed class diagram is described in Figure 11.

Figure 11 Class diagram for ThincClient

Chapter 6. Lessons Learned

First of all, I was amazed that I can play some video stream via network by using remote desktop environment. Because playing video has such a heavy network load, I thought that anything cannot play video in remote desktop environment at the same time. However, THINC does this very well by sending messages from low-level.

Moreover, I learned a lot of video related knowledge from that project. Because I did not take Computer Graphics course in previous institution, I am not good for how video works. (Even I know something about 3D graphics manipulation by Media processing class, I did not know about 2D graphics and its displaying mechanism from the bottom). By implementing THINC Client, I realize how video works by analyzing each messages coming from the server.

However, I couldn't implement PDA version of THINC Client because of lack of supported library. Even Personal Profile in CDC in J2ME supports almost all of functionality of JAVA, I cannot get the PNG library for J2ME, which required to process image resource from memory. By modifying standard PNG library for J2SE, I thought it could be implemented with modified PNG library; however, it is too complicated to modify all source code of PNG library suitable to JAVA.

In conclusion, it was good experience to increase my background knowledge for video. Even I could not implement PDA version of THINC, It was enough to study the mechanism of video.

Chapter 7. Revising THINC Protocol

In this Chapter, I will introduce how to revise the THINC protocol so that client can be implemented in limited device and for better performance. In each section of this chapter, I will try to explain what should be revised and propose newly designed THINC protocol.

7.1. Handshaking

The entire handshaking stage is as followed:

1. Server sends its version string of THINC without header
2. Client replies with its version string of THINC without header
3. Server sends security capability such as Authentication and SSL availability
4. Client respond security capability
5. Server sends session security capability as an ACK or sends NAK.
6. If SSL is enabled, do some handshake for SSL.
7. If authentication is required, send authentication information to the server
8. If authentication information comes from client, server replies with ACK or NAK
9. Client requests various initialization options
10. Server replies client's request.
11. Client sends done message when handshaking is done.

In client request, there is no absolute sequence of request. However, it seems that there is preferred sequence defined by enum keyword in C structure.

```
typedef enum {
        T_REQ_FBINFO=T_REPLY_LAST       /* basic framebuffer info */
        , T_REQ_CURSOR                  /* cursor information and data */
        , T_REQ_FBDATA                  /* contents of framebuffer */
        , T_REQ_ENCODER                 /* how is image data encoded */
        , T_REQ_CACHESZ                 /* size of various caches */
        , T_REQ_VIDEO                   /* server supports video? */
        , T_REQ_NOVIDEO                 /* client telling server that it
                                           can't support video (see below) */
        , T_REQ_VIDEO_SERV_FMTS         /* video formats supported by server */
        , T_REQ_VIDEO_CLIENT_FMTS       /* client informs server of its video
                                           formats */
        , T_REQ_KEEPALIVE               /* client supports or not keepalives
                                           does server support them? */
        , T_REQ_APP_SHARING             /* does the server support application
                                           sharing? */
```

```
        , T_REQ_APP_LIST                /* list of available applications */
        , T_REQ_APP_GET                 /* request a particular application.
                                           client may send this message
                                           multiple times */
} thinc_clientReqs
```

Table 4 prefered sequence of THINC handshaking

7.1.1. Revising encoder type in handshaking stage

Table 4 seems awkward because encoder type should be in front of frame buffer data because frame buffer data contains a lot of compressed raw image. Even frame buffer has its own flag for notifying what kind of encoding type is by using that, we can eliminate those flag if encoder handshaking is in prior to frame buffer data handshaking. So, I suggest making some sequence for each handshaking request, at least between encoder and frame buffer data.

Moreover, because encoding type is only sent by the server, if there is no library for PNG or ZIP compressed image, the client will not work because of the encoding type from the server. For more flexibility, I suggest following negotiation algorithm for encoding type.

First, new encoding stage looks like as followed:

1. Client sends request to the server to receive the server's encoding type.
2. Server sends its available encoding type to the server
3. Client responds its available encoding type based on server's type
4. Server decides to choose encoding type and send it to the server based on client response.

New encoding protocol is designed in Table 5.

```
typedef struct thinc_reqServEncoder_s {
        t_req_head;
} thinc_reqServEncoder;

#define T_REQ_SERV_ENCODER_SIZE  0

typedef struct thinc_replyServEncoder_s {
        t_reply_head;
        int which        :8;     /* zlib, png, none. see below */
        int reserved     :24;    /* reserved for future use */
} thinc_replyServEncoder
#define T_REPLY_SERV_ENCODER_SIZE 4

typedef struct thinc_requestClientEncoder_s {
        t_req_head;
        int which        :8;     /* zlib, png, none. see below */
        int reserved     :24;    /* reserved for future use */
} thinc_replyClientEncoder
#define T_REPLY_ CLIENT_ENCODER_SIZE      4
```

```
typedef struct thinc_replyClientEncoder_s {
        t_req_head;
        int which       :8;     /* zlib, png, none. see below */
        int reserved    :24;    /* reserved for future use */
} thinc_replyClientEncoder
#define T_REPLY_CLIENT_ENCODER_SIZE       4

#define T_ENCODER_NONE          0x00
#define T_ENCODER_PNG           0x01
#define T_ENCODER_ZLIB          0x02
```

Table 5 Revised design of THINC encoder handshaking

7.1.2. Revising Cache size handshaking

Because cache consumes some memories, cache should be negotiable from the client for limited devices. I proposed new cache size handshaking method because of that purpose.

First, new cache size stage looks like as followed:

1. Client sends its mostly available(maximum) cache size to the server
2. Server responds prefer cache size based on client's size.
3. Client sends ACK for server's response.

New cache size negotiation protocol is designed in Table 6.

```
/****** CACHESIZE ***************/
typedef struct thinc_reqCacheSz_s {
        t_req_head;
        /* all cache sizes are in bits */
        int             img:8;
        int             bit:8;
        int             pix:8;
        int             reserved:8;
} thinc_reqCacheSz;

#define T_REQ_CACHESZ_SIZE       0

typedef struct thinc_replyCacheSz_s { // client sends OK or NotOK.
        t_reply_head;
        /* all cache sizes are in bits */
        int             img:8;
        int             bit:8;
        int             pix:8;
        int             reserved:8;
} thinc_replyCacheSz;

#define T_REPLY_CACHESZ_SIZE     4
```

Table 6 Revised design of THINC cache size handshaking

7.2. Frame Buffer operation

For frame buffer message incoming from server, it is so solidified that it seems nothing will be changed. However, there is something I found to making easier implementation.

7.2.1. Flag option for frame buffer images

In Protocol specification of THINC, it says that each flag option is different for each message. However, by unifying those flag to one we can more reusable code in client side. Because the size of flag in THINC header is suitable to unify the flag, I propose unifying flag option in Table 7 into Table 8.

```
/* flags */                             /* order */
#define T_FB_RUPDATE_COMPRESSED  0x01    /* 3 */
#define T_FB_RUPDATE_RESIZED     0x02    /* 2 */
#define T_FB_RUPDATE_CACHED      0x04    /* 1 */
#define T_FB_RUPDATE_ADDCACHE    0x08    /* 1 *

#define T_FB_PFILL_RESIZED       0x01
#define T_FB_PIXMAP_CACHED       0x02
#define T_FB_PIXMAP_ADDCACHE     0x04

/* flags for glyph and bilevel */
#define T_FB_BITMAP_CACHED       0x01    /* 1 */
#define T_FB_BITMAP_ADDCACHE     0x02    /* 1 */
#define T_FB_BITMAP_RESIZED      0x04
```

Table 7 Previous design of THINC frame buffer flags

```
/* flags */                       /* order */
#define T_FB_COMPRESSED  0x01     /* 3 */
#define T_FB_RESIZED     0x02     /* 2 */
#define T_FB_CACHED      0x04     /* 1 */
#define T_FB_ADDCACHE    0x08     /* 1 *
```

Table 8 Revised design of THINC frame buffer flags

7.2.2. Resized frame buffer messages.

In resized frame buffer messages, the client should calculate the new coordination of each message. However, because calculation result is not as same as different implementation, I suggest that the server sends resized x, y, width rather than original coordinates. Current implementation for each resized frame buffer operation is in Table 9.

Copy Pixmap Solid Bitmap Video	Client should calculate resized x, y, width and height
RawUpdate	Client should calculate resized x and y

Table 9 resized calculation from THINC client

If everything in Table 9 is resized, the client could draw same image no matter what kind of language the THINC client is implemented by.

7.4. Audio Stream

I have not tested my audio because of lack of supporting server environment. However, I heard that audio stream only have one uncompressed stream type. By adding another compressed stream type for sound, the bandwidth for audio stream over the network can be solved somehow. Even though it is only proposal and does not have specific implementation of compressed stream type, it can be solved by adding one type specifier in struct thinc_sndOpen. I propose some specifier(flags) for each type in struct thinc_sndOpen in Table 9, which enables sound play in another type.

```
typedef struct thinc_sndOpen_s {
   int type       :8;
        /* for synchronization */
   int rate       :16;
   int bits       :8;
   int channels    :8;
   int endian     :8;
   char pad[2];
} thinc_sndOpen;

#define T_SND_OPEN_SIZE   6

/* flags in type*/
#define T_SND_TYPE_WAV   0x01
#define T_SND_TYPE_MP3   0x02
```

Table 10 Revising THINC sound

By using ACK or NAK messages provided by THINC Protocol, the client simply denies or accepts following format. And then server tries to send same stream by using same format, or we can make some negotiation stages just like handshake in THINC Client.

Appendix A. References

THINC Protocol specification, Ricardo Barrato and Leo Kim

From Java Community Process Program (http://jcp.org)

JSR 30: J2ME Connected, Limited Device Configuration, Antero Taivalsaari

JSR 36: Connected Device Configuration, Jon Courtney

JSR 37: Mobile Information Device Profile for the J2ME Platform, Mark VandenBrink

Appendix B.

implementation note
for THINC Client in JAVA
(javadoc for THINC Client)

Postscript: Because THINC Client has huge amount of javadoc, I attached Tree of the javadoc only in here. You can see the entire javadoc in doc directory of source code.