Note: *This is a preliminary Technical Manual – it does not contain the UML diagrams and other visual aides nor any functioning code at the present moment but will once all details have been finalised – this is just a guide to how the project MAY be tackled*

1. Introduction and Summary of constraints

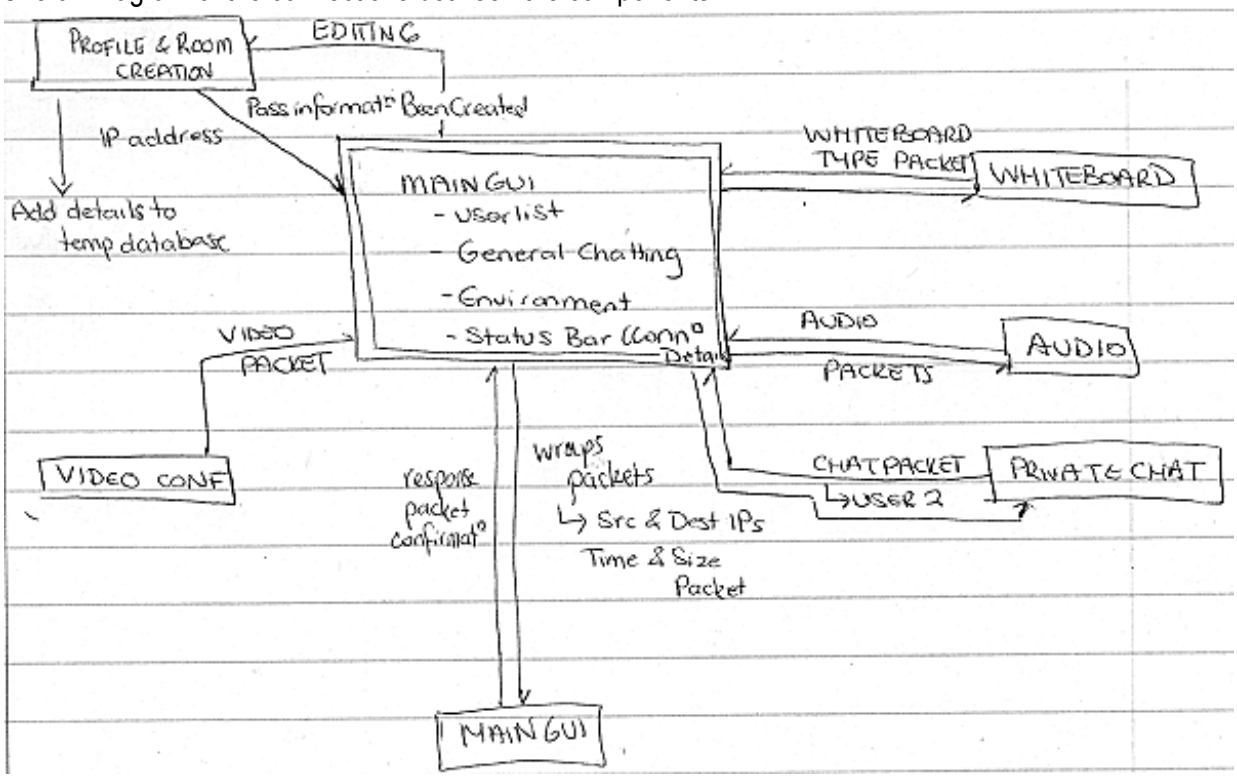
a. jSummit will be a Peer-to-Peer program that will use Threaded connections and a central server to help the program run smoothly. Programs will connect to one another using TCP/IP protocol and will send packets with various wrappers to one another
b. Their are a few Risks and Uncertainties at this point in time.
   - Risks: backwards compatibility, reliance on TCP connections, being thread safe, mutexes, semaphores, etc
   - Uncertainties: OS operations, using APIs and other modules, performance
c. The user requires: -easy to understand GUI
   - functionality [see User Manual]
   - reasonable speed
   - resources of computer to be used effectively
   - flexibility
   - password security

2. Recommended Approach

a. Constantly testing and revising the program including the development of prototypes in stages. We will be returning to lower priority requirements once the main functionality has been reached
b. Development environment will be Sun's Java runtime environment with the aid of SWING, AWT, JMF for all media streams (Video and Audio) using media capture and streaming media
c. Software tools will be using: Java SDK 1.5.0 runtime development environment along with IDE jCreator, Sun's JMF (Java Media Framework) – though the end user will need a Java compiler on their computer in order to run the program
d. Target Hardware/Software will be PC's including those running Windows and Linux and will port to Mac users at a later date

3. Implementation

Overall Diagram of the connections between the components:

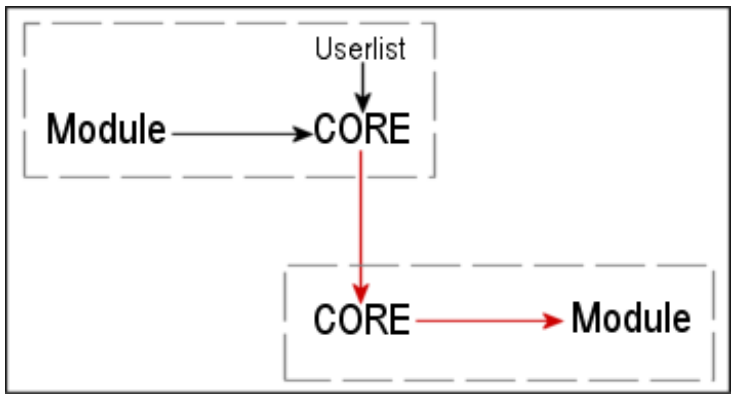


a. Main CORE

Connection to a Room

Upon Connection to another CORE module – the new client receives Room Details including the current User List – the CORE then processes this user list and sets up the display panels with the information - the Server then sends the already connected users the details of this new user and they all update their displays as well – this would be a “NEW User Packet”

Module interactions



The above diagram shows how the wrapping of data occurs where the Module wraps its data in its own packet then the CORE wraps it in a CORE packet (details below) – this information is then sent through the Server program which has connections to each of the clients and passes the packet through – the Server does not wrap the packet at all it just passes it on. What happens is that a Module wraps its packet and puts a Username destination – the core reads where the packet is from and wraps it in its own structure with the following parameters

```
Inetaddr        UserIP        //fetched from the User list details
string          ModuleID      //sent from the module itself
int             Size          //the total size of the packet to be sent
Time            Time          //time packet has been sent by CORE
Object          DataPacket    //Packet already wrapped by the Module
```

The Packet will then be sent via a threaded server to the desired location though the up running TCP connection – a response will then be received from the recipient and if an error response is received then the packet is resent depending on the type of packet (i.e. Chat packets will be resent whilst, Video streams won't).

Video and Audio streaming generally needs its own connection because it relies on UDP, the CORE will terminate a connection to a client if the client's CORE drops out – i.e. if a CORE drops out the server CORE will shut down all other connections to this user

b. Profile and Room Selection/Generation

Start-up

Initial Start-up

1. Load configuration → spawn server thread
2. Setup display (keep it hidden)
3. Display logo and progress bar
4. Show join/create options
5. Proceed to end
6. End

When a Client Starts:

1. Multicasts a broadcast query for any available servers (may be able to accept a direct IP connection)
2. Waits 3-4 seconds for response – no response → spawns server thread
3. If there is a response, or several, records IP's then requests to the rooms that are hosting and discovers list of members and if password protected
4. Initiate a connection with the room to join it
5. Once connected continue until finished – log out to disconnect

Profile Creation and Editing

- Details will be stored of previous profiles in a flat file that will be opened and exampled upon initiation of function

The data will be stored in XML format:

```
<PROFILE><USERNAME>InputName</USERNAME>
    <LOCATION>Location</LOCATION>
    <COMPANY>Company</COMPANY>
    <REALNAME>Real Name</REALNAME>
    <EMAIL>email@jsummit.net</EMAIL>
    <CONNECTION>56Kb</CONNECTION>
    <VIDEO>T</VIDEO><AUDIO>T</AUDIO><WHITEBAORD>T</WHITEBOARD>
</PROFILE>
```

- The records will be parsed and added to an in-memory database where the user can select which one to use – any modifications will be written back to the flat file – the file will be rewritten completely if any changes are made.
- Once a record has been chosen this will be stored as part of the Profile class that consists of:

```
class Profile {
    public Profile();
    public Profile( stUsername, stCompany, stLocation, stRealName, boVideo,
boAudio, boWhiteboard, iConnection);
    private String Username;
    private String Company;
    private String Location;
    private String RealName;
    private bool Video;
    private bool Audio;
    private bool Whiteboard;
    private int Connection;
    //appropriate Get and Set functions will also be here
}
```

Room Creation and Use

-If creating the room the room will be stored in a similar like structure to the profile though it will contain a pointer to the user list (rest of the design will be completed once we get the CORE written)
- add Password field to the class – though it can be empty – this room creation details are analysed once a connection is made and the details are packaged and sent back to the client

User List

User list will probably be a hash table or a vector list that is a mapping of UserName to IP Address – every client will have a copy of it and they will use this list to direct where packets will be going – though it will be a global variable so all modules will be able to see it

c. Video

Introduction

Underlying infrastructure of the video conferencing will be using the JMF API provided for Java by Sun Microsystems.

Connection Setup

When a chat room is setup, the video module retrieves the list of chat participants in the room.

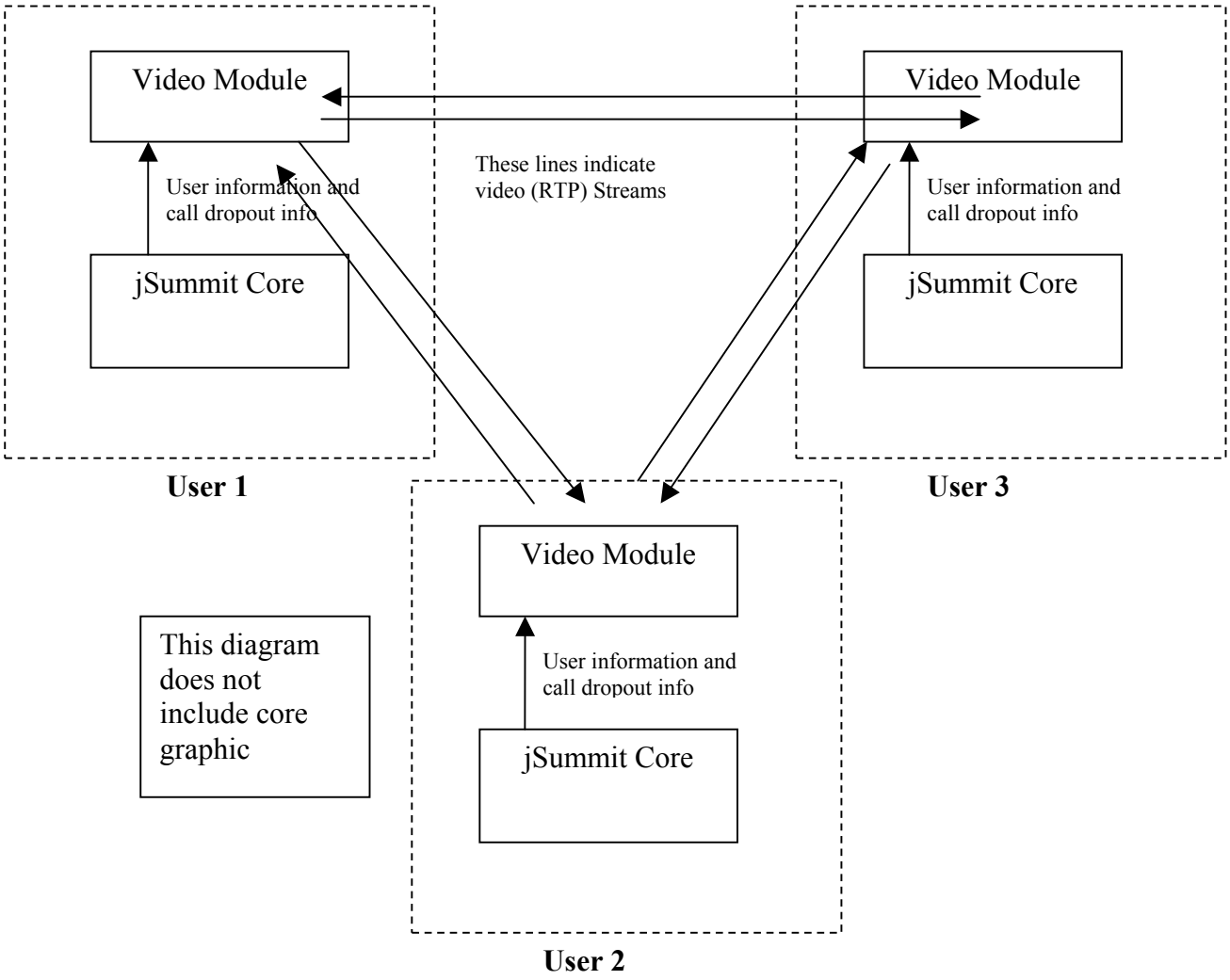
RTP connections will be then established between the users, this will make use of the RTPSession, SendStream, and ReceiveStream classes of the JMF API

The video source will make use of JMF's DataSource class, this class allows java to open video devices on the client machine.

Note

Unlike other modules, this module communicates directly with the users, this simplifies the packets between the cores. If a users computer crashes then the core will notify the video modules that the user has left/timed out, and to close the streaming session to the users.

Diagram of communications



d. Audio

YET TO BE FINIALISED Will work in a similar way to Video – once we get video going we will analyse the Audio section as well

e. Poll

Suggested Protocol & Ideas for Vote / Poll Implementation

Firstly ideas surrounding the vote and poll implementation centres around it being a module, capable of being plugged in, enabled, disabled e.t.c. This goes along with the idea of our central core of the project being a central point of focus for all interaction of clients within the conference / summit and its capability of despatching tasks / packets of data to the relevant handlers.

The voting and polling capabilities that are to be implemented in the final product include the following:

- A single selection vote / poll (Casting a vote, where you can select one only).
- A selection vote where you must prioritise ALL of the options (give them a rank from most preferred to least, generally with numbers).
- A multi selection vote which defines the following (this one differs from the above, as you are not required to prioritise ALL of the selections, although it can be done if so desired) –
  - Able to select at least one or more of the options in no particular rank (multi select – no preference)

- Same as above, but being able to select based on a preference or priority (multi select – preference / priority)

Ok, starting with some sort of basis of implementation I have the following 'structures' (or classes) to help implement the basis of the voting and polling system.

- Please note this is preliminary and may not be functional or operational, also actual code implementation has not been done, just some of the basic function declarations and such -

```
class Vote {
      public static final int VOTE_SINGLE = 0,
                        VOTE_ALL = 1,
                        VOTE_MULTI_NOPREF = 2,
                        VOTE_MULTI_PREF = 3;

      private String    descStr; // A description string for the vote.
      private Vector    vOptions;
      private Vector    vSelected;
      private int       selected;
      private int       voteType; // e.g. VOTE_SINGLE e.t.c.
      private int       keyID; // Unique identifier for this vote.

      public String getDescription();
      public Vector getOptions(); // Returns vector of vote options.
      public int getVoteType(); // Returns an integer to describe type.
      public boolean setSelected( Vector vSelectedSlots ); // Will return a true for a
valid vote, false otherwise.

      private Boolean isValidVote( Vector vSelectedSlots );
};
```

That is a very basic layout of the class to contain a vote structure (it is by no means yet complete, just a base to work on).

As for transmission using a protocol over the net it will basically use a standard TCP/IP socket (or in our case be wrapped within a core data packet). It will contain possibly a transmission object like so: -

```
class VoteQuestionPacket {
      int         keyID; // A unique identifier so the client can respond to more than one vote
simultaneously.
      // The following variables will basically be taken to be the same as in the 'Vote' class.
      String      descStr;
      Vector      vOptions;
      int         voteType;
};

class VoteAnswerPacket {
      int         keyID; // This will be the same as the keyID of the originating vote.
      Vector      vSelected; // This will contain multi results.
      int         selected; // This will work for a vote that needs only one response.
};
```

The originator of the packet (person who is organizing the vote) will send out the VoteQuestionPacket, and the person/s sending back responses will use the VoteAnswerPacket structure to respond.

This is still preliminary and some bugs, methods need to be ironed out.

Also on the originating clients end we will need a storage structure to encapsulate and keep track of votes that have been returned. An idea for the container would be as follows:

```
class VoteResults {
      int          keyID;  // Unique identifier for the vote responses to match to.
      Vector       vResultTally;  // Count of votes in slots.
}
```

- Still working on this bit. -

f. Whiteboard

The Whiteboard module implements an arbitrary number of shared whiteboards, where vector-based graphics can be drawn. Although the user may specify a small region to view, the whiteboard has no fixed size, with the coordinate (0, 0) being at the top left of the whiteboard, and extending in each direction as far as the precision of an unsigned integer will allow.

Whiteboards may be classified as public (readable and writeable by all), semi-public, (readable by all, writeable by a select group), and private (only readable and writeable by selected users). For security, semi-public whiteboards have an associated owner, and a client will only accept information from the owner. Authorised members submit data packets to the owner, who broadcasts them.

Each client maintains a personal list of whiteboards available to them, since they also maintain their own copy of the whiteboard. The server also maintains a list of used whiteboard ID numbers and is responsible for generating and allocating new ones, to prevent collisions.

The preliminary whiteboard packet is described below:

```
WBPacket:
inetaddr       author          //Generally, the sender of the packet
unsigned int   objectID        //Sender-generated identifier for the object or command
string         timestamp       //Timestamp for packet
unsigned int   whiteboardID    //Whiteboard number
enum           commandID       //Command ID
object         data            //Command arguments
```

For a specific operation (operations may require multiple packets to complete) author and objectID form a unique identifier.

Commands are mapped onto command codes, and their arguments are given as the "data" object. A client receiving a packet will be able to determine the structure of the data object by the command code.

Preliminary command list:

`request_whiteboard_list();`
Request a listing of all public/semi-public whiteboards. Usually this command is sent by a newly joined client to the Summit, however, if directed at another peer, the listing will include the private whiteboards that the requestor is part of.

`send_whiteboard_list(WBID whiteboard, enum public, inetaddr member, bool owner);`
(see "List Handling" below)
Sends the information of one whiteboard. If the whiteboard is public, the member part will be null, and owner disregarded. If the whiteboard is semi-public or private,

`request_whiteboard_dump(WBID whiteboard);`
Request a complete dump of a whiteboard, useful for a newly joined client to the Summit.

`send_whiteboard_dump(WBPacket object);`
(see "List Handling" below)
Sends a complete listing of packets (one packet at a time) that make up the requested whiteboard.

`end_of_list();`
(see "List Handling" below)
Signals the end of a list.

`request_whiteboard();`
Sent to the server to reserve a Whiteboard ID, which will be used to create a new whiteboard.

`alloc_whiteboard(WBID newWBID);`
Sent by the server, this is a newly generated and reserved whiteboard ID.

List handling:
1. Requestor submits request for a list/dump with unique objectID
2. Requestee sends individual list elements with requestor as author and requestor's objectID as objectID
3. Requestee sends end_of_list() command with client's author/objectID

Drawing commands:
Some of the drawing commands share a few common attributes:

`(enum) brush:`
Defines the shape used to paint on the whiteboard with. This is an enumeration, with the following values
CIRCLE
SQUARE
HLINE (horizontal line, 1 pixel tall)
VLINE (vertical line, 1 pixel wide)

`(unsigned int) size:`
Used with "brush", this is the diameter or width of a circle or square brush, or the height or width of either of the line brushes.

`(color) colour, colour1, colour2:`
The colour of the drawn object. When two are defined, colour1 is the colour of the border of the object, and if "fill" is set, colour2 is the colour it is filled with (otherwise it is ignored)

`(bool) fill:`
An indicator of whether a shape is filled with colour2

`(unsigned int) x, x1, x2, y, y1, y2`
These are x- and y-coordinates of the drawn object, when two pairs are defined, they form opposite corners of the rectange that the shape is drawn to fit inside.

`dot(colour, brush, size, x, y);`
Places a single instance of the brush shape/size at the coordinates specified. If the mouse is dragged, multiple dots are drawn. To help avoid network congestion, dots are not drawn for every pixel the cursor is dragged over. Instead, the mouse location is sampled several times per second.

`highlight(colour, brush, size, x, y);`
Same as dot, but drawn with a fixed translucency (i.e. drawing over a previously highlighted area will not darken the highlight)

`line(colour, brush, size, x1, y1, x2, y2);`
Draws a line from the first pair of coordinates to the second.

`ellipse(colour1, colour2, size, fill, x1, y1, x2, y2);`
Draws an ellipse within the rectangle defined by the coordinate pairs.

`rect(colour1, colour2, fill, x1, y1, x2, y2);`
Draws a rectangle between the specified coordinates

`graphic(graphic, x1, y1, x2, y2);`
Inserts a graphical object "graphic", resized to fit in the rectangle defined by the coordinates

`erase(author, objectID);`
Erases the object defined as (author, objectID)

`textbox(x, y, font, colour, size, string text);`
Defines the top left corner of an area of text.

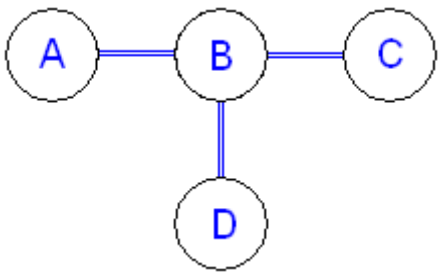
`laserpointer(colour, x, y);`
A "laser pointer" function, which is a fixed-size translucent brush, which is not saved as part of the whiteboard, but can be moved in realtime (i.e. as long as the mouse button is held). Used to point at areas of the whiteboard.

`laserpointer_off();`
Stops the laser pointer from being drawn.

g. Chatting and File Sharing

General Chatting
One ‘host’ runs and owns the general chat room. As such all chat is done through this node. Upon a person join, the host will store the person into a dynamic array of char[]’s.

This data will be used to direct traffic. For example,



This is what a typical chat service would look like, with B hosting the chat room. If A would like to send a message to the chat room, then it sends a chat packet (see below) to B. B then sends this packet to all the people, via its array. Hence every node will receive the message.
The messages sent by people, are sent through the network in the following type:

```
class General_Chat_Packet {
      private String username;
      private String message;
}
```

This packet structure gets sent to the core, which will then pack it up, and send it to the chat room host, who will then distribute the message to the other people.

Private Chatting

Private Chatting is done in the same way as General Chatting, however there is no host to host a room. It is simply one node connected to the other node, and it will use the following structure

```
class Private_Chat_Packet {
      private String desitination_username;
      private String source_username;
      private String message;
}
```

Whether it be private of general chatting, in either case, the packets are handed to the core, and the core holds the user list with the IP's, matches the user/nicknames to the IP's and sends them off appropriately.

File Sending

File sending is built into the Chat module, via selecting the 'send a file' button. This will be done with `RandomAccessFile()`, and `FileWriter()`.

On side that is sending the file, a RandomAccessFile object is created. From this, the entire file is read into an array, which is sent to the core for sending. One the receiving side, the complete opposite occurs. It receives the object, unpacks it, checks the length, creates a file of that length, and then writes the bytes to the file, resulting in a copy of the file. The packet that will hold this file is as follows:

```
class file_packet {
  private RandomAccessFile aFile;
      private String desitination_username;
      private String source_username;
}
```

and on the receiving end, it will simply accept this object, and then create its `FileWriter()` object, and begins the write the file.

Note: All of the Implementation ideas above are subject to change and probably most will be changed.

4. Implications

a. Project products: Functioning Peer-to-peer program that has video, audio conferencing, chatting, whiteboards and a poll function that uses TCP connections and a usable 'funky' GUI. It will be a free alternative that combines functionality
b. Will be free to modify and upgrade once the final project has been delivered and completed

[TO BE ADDED]
- A Bunch of the code – exact and more specific details on the protocols that will be used once some code has been trailed and explored
- UML class diagrams and sequence diagrams to help add a visual element to the design
- Maybe even some formal descriptions
- In-depth usage of the JMF functions
- A link to the full Source Code because it is GPL
- Index and groovy side icons for each of the sections