Providing videoconferencing software to ZETA Operating System:

Whisper

by Oliver Ruiz Dorantes

3.2. A preview of the implementation and architecture of BeNet:

We will make a brief summary about the implementation. There are three basic parts in the implementation: the GUI, the Network stack, and the sound Engine. Of course, the GUI follows the BeApi concretely the Interface_kit.

About the sound engine we noticed that it is based on a application offered as source example by Be Inc, named SoundCapture (afterwards we realized also that the BeOS port of openh323 also contains source examples from Be Inc).

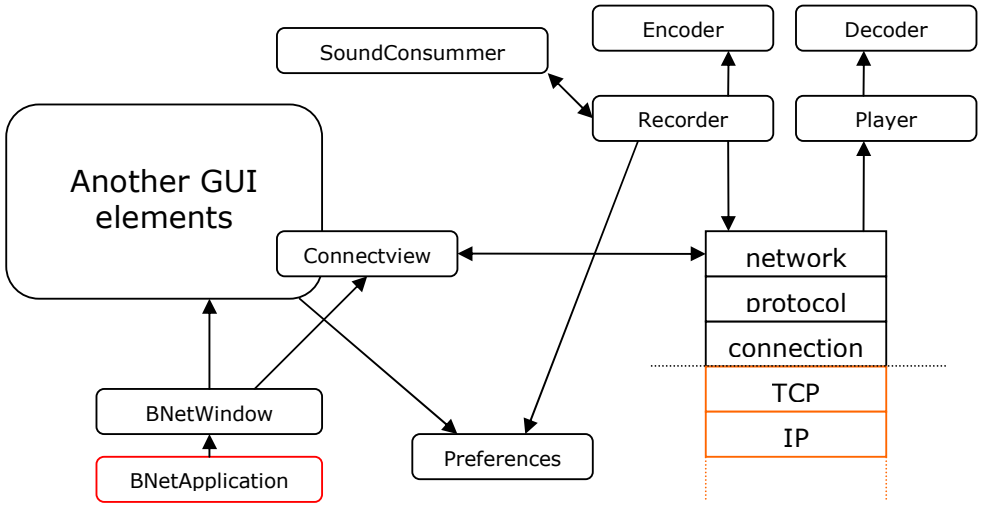
The most important is the SoundConsumer object derived from the BBufferConsumer class. Following the media_kit, this object is connected to the audio_input invoking a hook function to give the buffers of sound. We can see that this hook will be the one that sends this buffer to the network by the network object we are going to explain now. These buffers are encoded with the g721 encoder.

Also a good feature of BeNet is that before encoding the sound buffer it pass thought a filter that evaluates if the sound received is useful. It means this filters detects if there is silence in the stream. This is a reason to not sending the frame. Note that usually in a conversation not both persons are talking at the same time. One is asking while the other hears the question and after the other replies while the other part is listening to the answer. Just 50% of the data is needed to be sent, it is rarely a full duplex traffic.

The voice packets around the network are sent using the RTP/RTCP protocol. Note that these protocols are also used in H323 in order to transfer the media streams. The Network stack is composed by 3 objects: connection, protocol and network, even we could put these 3 layers in the top of the TCP/IP stack to understand it better:

Basically the connection just makes calls to the sockets: setting them, receiving, and sending the packets through the network.

The Protocol object manages these packets, carries the task to build the packet to be able to be sent through the network the connections object and also dissembles the packet giving the most important information to the network object. [... STACK HERE...]



The network is the most important object. It contains 4 threads 2 for each protocol one to receive the data and another to send it. Also contains the interface with the sound engine to reproduce or to send the audio buffers. Moreover has a interface to communicate with the GUI, which will invoke his methods and by the other hand the

network object will send BMessages to the GUI to be able to have a provide a feedback to the user of the actions.

3.3. Bug fixing comments:

Some of the issues in the application are coming directly that it was designed and implemented in BeOS R4.5 (dated 1999). Zeta is considered the release 6 of the operating system.

Several changes in the Network environment and in the media kit are done. Most of these problems are already in the release 5 of the system (dated 2000). This could be one of the reasons they abandoned the project. Some things just cannot be done as they wanted it to be done, or are done going in a unsafe way.

Other problems are traditional bugs: pointers not checked, ignored returned errors, exceeding sizes working with buffers

The biggest problem was without doubt the audio retrieving. This could be as important as all the others together. Not only retrieving this but sending them in a way that the other peer could understand, unpack and reproduce the data in a proper way.

3.3.1. First Stage: Compilation:

As we feared, the sources we received were not compilant in Zeta, maybe they were in R4.5. This is common looking applications coming from R5 some documented changes should be done. Moreover this application comes from R4.5.
Its being commented that this is a problem we will probably have if we try to compile applications from R5 or R4.5. So this chapter could be very useful to developers who want to provide their R5/4.5 applications to Zeta

Linking libraries:

As Zeta is using the new BeOS network environment, BONE, libraries as libnet.so libnetapi.so are deprecated as they belong to the net_server (present until R5). By other hand we must use libbind.so and libsocket.so. The first is used to DNS calls connected functions like gethostbyname, gethostbyaddress etc, and the second is needed to work with sockets (creating, setting, sending and receiving).

As a new feature in Zeta, we need to link libzeta.so to not to get the undefined references to the: "find_directory_r(directory_which, BPath *, bool, BVolume *)" function.

Posting messages:

We will find this compilation problem almost in all applications wanted to be brought in Zeta, it is due to the Application kit changes. If the application makes use of passing BMessages between the of its GUI elements it is sure we will have these problems. As BeNet uses it, around all the code we will comment this only once. So always we have something like:

```
pointer_to_some_gui_element->PostMessage(BMessage*, NULL);
```

We should change this for

```
BMessenger(pointer_to_some_gui_element).SendMessage(BMessage*);
```

There changes in the application kit were done to have more security and avoid confusion sending the messages around the application.

Keeping relation, a quite popular BMessage to send is B_QUIT_REQUESTED which usually is sent to the BApplication object to tell that something or someone asked the application to quit and finish. Until BeOS R5, we did this by:

```
be_app->PostMessage(B_QUIT_REQUESTED);
```

But in Zeta by the same reasons as before we need to do:

```
be_app_messenger.SendMessage(B_QUIT_REQUESTED);
```

These changes are deeply commented in the yellowTAB.com developed corner.

Compiling Errors:

Obviously this is an incorrect cast: Fixed in BitmapCatalog.cpp

```
#106: int32 * cBit = (int32)result->Bits();
#106: int32 * cBit = (int32*)result->Bits();
```

Window.h is not included in ScopeView.cpp so these calls are not defined in the scope:

```
#62: if( Window()->LockWithTimeout(100000) == B_OK )
#65: Window()->Unlock();
```

Idem with SoundControlsView.cpp

```
#51:_Window()->MessageReceived(msg);
```

The macro timersub was added in network.cc to avoid a implicit declaration error

```
#ifndef timersub
#define timersub(a, b, result) \
 do { \
   result)->tv_sec = (a)->tv_sec - (b)->tv_sec; \
  (result)->tv_usec = (a)->tv_usec - (b)->tv_usec; \
  if ((result)->tv_usec < 0) { \
   --(result)->tv_sec; \
   (result)->tv_usec += 1000000; \
  } \
 } while (0)
#endif
```

- In *client.cpp* was needed to be included *#include <arpa/inet.h>* to be able to use *inet_addr()* function and *#include <unistd.h>* to apply *close()* over the socket

The function closesocket() is not provided anymore so needed to use :

```
#192: close(m_sStream)
```

These headers were changed:

```
#60: char* TClient::GetFirstNameByIP(const char *ip){
```

```
#69: char* TClient::GetLastNameByIP(const char *ip){
#78: char* TClient::GetEmailByIP(const char *ip){
```

As working with threads by this way:

```
#237: my_thread = spawn_thread(&Cmd, "Cmd Thread", 10, this);
```

Brought us problems we defined a helped method:

```
#237:_my_thread = spawn_thread(Cmd_helper, "Cmd Thread", 10, this);
```

Which uses the cmd() method which is private in this context. So this method was moved in the header to the public scope.

```
#218: static int32 Cmd_helper(void* data)
#219: {
#220:  ((TClient*)data)->Cmd();
#221: }
```

- In connection.cc moreover including these headers due to the BONE environment:

```
#10: #include <sys/socket.h>
#11: #include <unistd.h>
#12: #include <netdb.h>
```

The following were nedded in the header:

```
#11: #include <netinet/in.h>
#12: #include <arpa/inet.h>
```

Skipping the deprecated:

```
#9: #include </boot/develop/headers/be/net/socket.h>
#10: #include </boot/develop/headers/be/net/netdb.h>
```

We had to change the function to close the sockets (closesocket per close) as before.

- In the header network.h, we had to change the type of the returned value.

```
#53: const char*     GetRemoteIP()     {return protocol->GetRemoteIP();}
```

Something similar we could see in the headers of protocol and connection.

In the GUI elements it can be talked about

- In *ShrinkView.cpp* and *BeNetButton.cpp* the header to *Window.h*, as before there are calls to the function *Window()* who returns us the Window that contains the *BView* or *BView* derived object.

```
#14: #include <interface/Window.h>
```

- *BubbleHelper.h* class as we explained shows a helping yellow tip when the mouse is moved above a control. This must be applied to the BView reference so we need this:

```
#28: #include <interface/View.h>
```

- BeNetWindow.cpp need #include <app/Roster.h> to these function:

```
#223: be_roster->Launch(&ref);
```

Concerning all the MailIP classes:

- The BeNetMailIPWindow.cpp makes of a BMessenger to send messages so we need to add the Application header:

```
#14: #include <app/Application.h>
```

Also using a BEntry object in :

```
#171: BEntry entry(pTemp->pzFile);
#172: entry.Remove();
```

So will need the header

```
#15:_#include <storage/Entry.h>
```

- In the EMailIPManager.cpp we will se a example how to define and spawn correctly a thread without getting annoying warnings. It is simple, just use in the spawn_thread() function a method that accomplish a int32 as returned value and a void pointer as parameter:

Spawning:

```
m_iThreadID = spawn_thread(EMailCheckThread, "EMailCheck", B_LOW_PRIORITY, NULL);
```

The thread function definition:

```
int32 EMailIPManager::EMailCheckThread(void* data) {
return value;
}
```

Note that the header of this class had to be changed.

- *BeNetMailIPWindow.h* needs *Window.h* header because it inherits from the BWindowclass
- *EMailIP.h* we can see that almost all the method uses the *BFile* object. So we have to declare it.

```
#7: class BFile;
```

Concerning the BookmarkTools:

- in PostDispatchInvoker.cpp needed the header Looper.h when calling this function:

```
#37: Looper()->DispatchMessage(message, *target);
```

- CLVListItem.h we can see that almost all the method uses the BBitmap object. So we have to declare it.

```
#8:_class BBitmap;
```

3.3.2. Second Stage: Avoiding potential crashes:

The bitmaps:

We could notice a different behavior running the application behind the Terminal and another different behavior behind the Tracker. In the first we could see how the

Graphical interface is drawn before the crash; in the second one we could not even see the creation of the window. The reason is that running from the Terminal, the work path (usually) is the path were is the binary with its folder, with the help html documents, and finally, the Bitmaps used in the application. So the code can access the bitmaps just with:

```
/*      BFile file(pzFileName, B_READ_ONLY);
        BTranslatorRoster *roster = BTranslatorRoster::Default();
        BBitmapStream stream;
        BBitmap *result = NULL;
        if (roster->Translate(&file, NULL, NULL, &stream, B_TRANSLATOR_BITMAP) < B_OK)
                return NULL;
        stream.DetachBitmap(&result);
*/
```

But what happens running behind the Tracker? The current path is set to " / " and not where the application was pleased to run. As we saw before in the code, evidently this path appended to the root path does not point to the path where actually the Bitmaps are stored. So this provokes the crash, the translation kit cannot find the Bitmaps to build the GUI, after creating the BFile object(which points the Bitmap) and the call to the BTranslatorRoster(which makes the format media conversion), no return errors are tested.

This was solved with this line of code commenting the others:

```
BBitmap *result = BTranslationUtils::GetBitmapFile(pzFileName);
```

It doesn't mind where the current path is. Somewhere in the Implementation its looks in the BApplication object which the real path of the application can be retrieved. Moreover we can see the we changed 7 lines of code with just 1, With the setback that we had to include TranslationUtils.h.

Connected with the current Path...

It is not a potential bug, event it was not one of the firsts bug fixed. But it has connection with the last one, so here is the best place to comment it. It is when the user asks for the help. As again, the current path is the root and not the path where the help is stored. When the user clicks the help it is correctly showed when BeNet is being ran in the Terminal and not when it is being ran with a simple click in the Tracker. Something similar has happened with the previous explained bug. When Help is required, as the help is provided in HTML format, BeNet launches the default webrowser:

```
if (get_ref_for_path("help/help.htm", &ref) == B_OK)
        be_roster->Launch(&ref);
```

We can see again the relative path to the file. That actually will be pointing to the absolute path *"/help/help.htm"* or by other hand the file we want if we run in the Terminal. So we must build ourselves the absolute path of the reference who is pointing the Help. We could look to the **Team** information but what we did is retrieving such information from the BApplication Object. We should know or remember that there is a pointer named be_app that points to our BApplications object that can accessed everywhere inside our Application:

```
        app_info ai;
        be_app->GetAppInfo(&ai);_
        BEntry entry(&ai.ref);
```

By this way we can get an app_info structure that contains a field named **ref** that points to our application as we see in the code. The next step will be appending the relative path were the main html file is located:

```
BPath path;
entry.GetPath(&path);
path.GetParent(&path);
path.Append("help/help.htm");
```

From the **ref** we get the BEntry object, then the BPath, then we clean the path (delete the Binary because we want where is located not the binary itself) with path.GetParent. And finally we can append the file.

```
char *help = new char[strlen(path.Path())+1];
sprintf(help, path.Path());
be_roster->Launch("text/html",1, &help);
```

As last step we create the string and call to the roster to launch the application set by default for the mime-type text/html.

This case we increased the number of lines and the simplicity to solve the problem. However we took a look to open sourced applications. Concretely SampleStudio (the best sound editing application for BeOS from the company Xentronix) solved this by this way.

The audio hook:

This is the most critical part of BeNet. You must have general idea at least of the media_roster to understand this part properly.

Media kit in Zeta must be understood like nodes that are connected with wires. Then the media is transferred through the nodes to be played, converted and stored or anything you can do with media data.

You can think of a media node (represented by the media_node structure) as a component in a home theater system you might have at home. It has inputs for audio and video (possibly multiple inputs for each), and outputs to pass that audio and video along to other components in the system. To use the component, you have to connect wires from the outputs of some other components into the component's inputs, and the outputs into the inputs of other components.

BeNet's sound engine looks for an Audio input in the system and then connects it with a SoundConsummer. So it connects the output of the Audio input(MIC) with the input of the Sound consumer.

This is analogous to choosing an audio output from your new DVD player and matching it to an audio input jack on your stereo receiver.

In the soundConsumer node you have to specify 2 hooks functions. The notify and the data function. The first one is just to know the internal changes in the SoundConsumer node, the second is the important one. It sends audio buffers retrieved by the MIC. At this point we will say that the buffers waiting to allocate them were just 2 KB, when the SoundConsummer were sending us Buffers of 32KB, and this crashed the application.

3.3.3. Network and UDP connections:

UDP sockets:

As we said in the compilation stage just for making it compiling a lot of changes needed to be done. The calls to sockets were changed to be able to compile with the new BONE system. However it was impossible to place a UDP communication between 2 machines. The next step, was looking around Internet for simple examples of UDP sockets and test them under Zeta.

A simple time server and client from the Network classes in the university was tested. It only worked as we expected in local. It means only placing connections with the localhost, the client tries to connect to a server that is listening in the same host as him. This example consists in a server that is listening to clients who asks the time. The client asks for the time and waits a reply from the server

Such example worked in Linux as well; we looked for an other source example. An easier one. Just a server waiting for strings and it prints it to the screen. It worked not only in a local stage. Both code examples were POSIX standard. So it meant that such code was not Zeta or BeOS code.

As we had a simple and clean solution working under Zeta for UDP sockets, we replaced the BeNet setting creation and socket calls. Because of this, and other needed changes, we could say that the BeNet connection object is 100% replaced.

Initializing the local sockets:

```
//Address: Local RTCP (port 2001)
addressLocalRtcpUdp.sin_family = AF_INET;
addressLocalRtcpUdp.sin_port = htons(PORT_RTCP_UDP);
//memset(addressLocalRtcpUdp.sin_zero, 0, sizeof(addressLocalRtcpUdp.sin_zero));
//addressLocalRtcpUdp.sin_addr.s_addr = GetLocalAddr();
addressLocalRtcpUdp.sin_addr.s_addr = htonl(INADDR_ANY);
```

Initializing the local sockets:

```
//Address: Remote RTCP (port 2001)
addressRemoteRtcpUdp.sin_family = AF_INET;
addressRemoteRtcpUdp.sin_port = htons(PORT_RTCP_UDP);
//memset(addressRemoteRtcpUdp.sin_zero, 0, sizeof(addressRemoteRtcpUdp.sin_zero)); //out
//addressRemoteRtcpUdp.sin_addr.s_addr = *(long *)hostRemote->h_addr;
memcpy((char *) &addressRemoteRtcpUdp.sin_addr.s_addr, hostRemote->h_addr_list[0], hostRemote-
>h_length);
```

Sending:

```
//if (sendto(socketSendRtpUdp, data, RTPHEADERLEN + RTPPLLEN_CODEC, 0, (struct sockaddr
*)&addressRemoteRtpUdp, sizeof(addressRemoteRtpUdp)) < 0)

if (sendto(socketSendRtpUdp, data, RTPHEADERLEN + size, 0, (struct sockaddr
*)&addressRemoteRtpUdp, sizeof(addressRemoteRtpUdp)) < 0)
```

Receiving:

```
//recvfrom(socketRecvRtpUdp, pchBuffer, nMaxLen, 0, (struct sockaddr *)&lRemoteAddress,
&lenAddr);  // Oli: Im not sure
recvfrom(socketRecvRtpUdp, pchBuffer, nMaxLen, 0, NULL, NULL);
```

Due to all this changes done this method in the connection object got unused.

```
long CConnection::GetLocalAddr()
```

Even not used, there was a bug testing the return code of a *gethostname* call. After that, this method could be used in the future.

Setting the sockets options:

In the connection object there are two methods that set the receiving and sending sockets. One method is for setting the RTP sockets (receiving and sending) `CConnection::RtpUdpMode(MODE mode)` and the other `CConnection::RtcpUdpMode(MODE mode)` is for the RTCP protocol. Both make the same but with different sockets and are called by the Network layer.

The sending sockets must be set with *setsockopt* but the receiving must be *binded* too
These calls were not correct and were replaced because an integer must be passed with the function *setsockopt*:

```
void CConnection::RtpUdpMode(MODE mode)
{
int n = 1;  // OliverESP fix
int error;  // OliverESP debugging

[…]

if ( (error = setsockopt(socketSendRtpUdp, SOL_SOCKET, SO_NONBLOCK, &n, sizeof(n))) != B_OK )

[…]

if ( (error = setsockopt(socketRecvRtpUdp, SOL_SOCKET, SO_NONBLOCK, &n, sizeof(n))) != B_OK ){

[…]

if ((bind(socketRecvRtpUdp, (struct sockaddr *)&addressLocalRtpUdp,
sizeof(addressLocalRtpUdp))) < 0)

[…]

}
```

Setting the structures:

When the connection object is created the local ***sockaddr_in*** structures are set. When is time to set the remote structures? We only know this information when we want to connect a remote host, or when we are receiving a connection.

Attempting the connection:

The user writes a address in the text box, and clicks the Connect button

```
CNetwork::Instance()->ConnectFromUI(inet_addr(GetIP()));
```

The interface as we can see calls the Network object

```
status_t ConnectFromUI(ulong lRemoteAddress);
```

Finally this function calls the connection object to set the address

```
protocol->connection->SetRemoteAddress(host)
```

where:

```
void SetRemoteAddress(ulong RemoteAddress);
```

The first thing we notice is that BeNet is passing ulong types for the connection address. In the interface the string of the address is translated into a ulong type with inet_addr(GetIP()). With this we can only use IP addresses, not hosts names. We can go further and in the SetRemoteAddress the connection object gets the hostent structure with gethostbyaddr

```
struct hostent* hostRemote = gethostbyaddr((const char *)&RemoteAddress, 4, AF_INET);
```

Where *RemoteAddress* is a ulong type. The main problem is that gethostbyaddr not always give us a satisfactory result it depends in the DNS servers in the network. The same result was got under Linux with source examples of this function. All this was changed, from the user interface to the connection object. We used the function gethostbyname that requires a String, it can be a ip address or a host address, giving BeNet the possibility of connecting hostnames. All the headers and the user interface was changed.

```
status_t CNetwork::ConnectFromUI(char* host)
status_t CConnection::SetRemoteAddress(char* host)
```

Receiving a connection:

Always when you are receiving something form the UDP socket, you get the sockaddr structure of the receiving part.

```
int rv = recvfrom(socketRecvRtcpUdp, pchBuffer, nMaxLen, 0, (struct sockaddr *)&lRemoteAddress, &lenAddr);
```

So we have to be sure that we do not lost this information, and as soon the the user accepts de connection, set the information in the socket.

As before BeNet used the ulong number to store the addresses, before only 1 method was used to set remote addresses. Now when we are going to connect we are passing a String (with ip or host) and when we are going to receive a connection, we have the sockaddr structure. Different ways, so a new method was created with this intention:

```
void CConnection::SetRemoteConnectingAddress(sockaddr_in& RemoteAddress)
```

ISo in the thread is waiting the RTCP data (Control data) we have:

```
sockaddr_in lRemoteAddress;
[...]
switch(protocol->RecvRTCPPacket(lRemoteAddress))
[...]
protocol->connection->SetRemoteConnectingAddress(lRemoteAddress);
```

Always when we are receiving a packet,

```
int     RecvRtcpUdp(char* pchBuffer, int nMaxLen, sockaddr_in& lRemoteAddress);
int     RecvRTCPPacket(sockaddr_in& RemoteAddress);
```

We are giving by reference the *sockaddr_in*. So each time we receive we know who is doing it. We only have to call *SetRemoteConnectingAddress* when we are receiving a connection request to set definitively the structures we are going to communicate.

3.3.4. Retrieving audio:

First of all we should read the "*The audio hook"* explained some pages before in the chapter "Avoiding potential crashes".

As we commented to record sound we do:

- Ask to the system to get an audio input device.

```
m_roster->GetAudioInput(&m_audioInputNode)
```

- Create our instantiation of your Bufferconsumer derived class which will get all the audio data. Now we have the 2 media nodes.

```
m_recordNode = new SoundConsumer("Sound Recorder");
m_roster->RegisterNode(m_recordNode);
```

- Now we need connect the 2 nodes. As the flow of the media goes from the input to the BufferConsumer, we have to find an output in the audioInput and to connect it to the input on the Bufferconsumer.

```
m_roster->GetFreeOutputsFor(input, &m_audioOutput, 1, &count, B_MEDIA_RAW_AUDIO);
m_roster->GetFreeInputsFor(m_recordNode->Node(),&m_recInput,
1,&count,B_MEDIA_RAW_AUDIO);
```

- Once we set the time sources for the nodes to get synchronized

```
BTimeSource * tsobj = m_roster->MakeTimeSourceFor(input);
m_roster->SetTimeSourceFor(m_recordNode->Node().node, tsobj->Node().node);
```

- Now we have to set the hooks functions. These are the functions which are going to receive the media data.

```
m_recordNode->SetHooks(DataEncode, NotifyDataEncode, this);
```

- Then place the connection (plug the wire) between the two nodes.

```
m_roster->Connect(m_audioOutput.source, m_recInput.destination, &fmt, &m_audioOutput,
&m_recInput);
```

- Now we have to start the all the nodes. Start the time source

```
m_roster->StartNode(tsobj->Node(), BTimeSource::RealTime());Start
```

- The Audio input

```
m_roster->StartNode(m_audioInputNode,     m_recordNode->TimeSource()->RealTimeFor(then,
0));
```

- The soundConsumer

```
m_roster->StartNode(m_recordNode->Node(), then);
```

Once we have done all this and our microphone is correctly set in the Zeta media preferences we will start to receive audio buffers in the hook function we have set:

```
Void Recorder::DataEncode(void* cookie, bigtime_t timestamp, void* data, size_t size, const
media_raw_audio_format& format)
```

Here we can see that we receive more information moreover the buffer.
The cookie is internal data of the SoundConsumer, we could see in the variable timestamp that we are receiving information about when this hook function is invoked.

Obviously size and date are the size and the pointer of the buffer. The important information that we are receiving here is the media format.

Here we have the media_raw_audio Structure:

Here in **frame_rate** we can see the sample rate, the number of channels (MONO or STEREO) in **channel_count**, **format** describes how is specified each sample in the buffer, **byte_order**e indicates the endianness of the data (either B_MEDIA_BIG_ENDIAN or B_MEDIA_LITTLE_ENDIAN). And finally **buffer_size** is the size of this buffer.

Here are the descriptions of the format:

- **B_AUDIO_FLOAT**. Each sample is four bytes; 0 is the middle, -1.0 is the bottom, 1.0 is the top.
- **B_AUDIO_INT**. Each sample is four bytes; 0 is the middle, 0x80000001 is the bottom, 0x7FFFFFFF is the top.
- **B_AUDIO_SHORT**. Each sample is two bytes; 0 is the middle, -32767 is the bottom, 32767 is the top.
- **B_AUDIO_UCHAR**. Each sample is one byte; 128 is the midpoint, 1 is the bottom, 255 is the top.
- **B_AUDIO_CHAR**. Each sample is one byte; 0 is the midpoint

The question is: who decides these parameters?

Nowhere in the source code is decided how the format should be. It is decided by the audio input node, so the answer of the question is unfortunately: the Device.

So on the soundcard is installed on our system depend the type of the format we are going to receive.

In this table we can se some of the hardware we tested.

Device name	Frame rate	Channel_count	Format	Byte_order	Buffer_size
Maestro	44100	2	B_AUDIO_SHORT	**B_MEDIA_LITTLE_ENDIAN**	32768
ct5880	48000	2	B_AUDIO_SHORT	**B_MEDIA_LITTLE_ENDIAN**	4096
Auvia	48000	2	B_AUDIO_SHORT	**B_MEDIA_LITTLE_ENDIAN**	2048
SB live	48000	2	B_AUDIO_SHORT	**B_MEDIA_LITTLE_ENDIAN**	1024

We can see the big difference between the buffers received by a Maestro soundcard and a SB live.

This difference of size is translated in how much time of voice represents with each buffer. So we could calculate:

$$\#samples\_in\_buffer = \frac{\#bytes}{B\_AUDIO\_SHORT * Channel\_count}$$

$$\#samples = \frac{1Hz}{1sample} \cdot \frac{1\text{sec}}{44100Hz} = \#\text{sec}$$

bytes	milliseconds
32768	185,7596372
4096	23,21995465
2048	11,60997732
1024	5,804988662

We can see that for the SB live each buffer is just 5 milliseconds of speech. The standards recommends more time. And we tested with so small packets and the sound quality is not audible.

In the beginning of this chapter it is said that the version 2 of BeNet could run without crashing. Here we have the reason. The application was designed to get 2KB buffers. So

it can run in Auvia soundcards and any other soundcard with 2KB in its hardware specifications.

Assuming we are not going to change the protocol. We cannot send information to the other peer about how large is the buffer we are recording and sending. Also notice that there could be confusion with the different sizes of the compressions of the codec we are using (g721).

Also notice if we set in the code a buffer of 32KB (which seems to be the maximum) there could be another soundcard sending us 64KB which will provoke a segmentation fault problem and an unexpected data received by the.

The best solution, and the solution yellowTAB is working on is to fix the format we are receiving the audio buffers. This can be accomplished by setting another node between the audio input node and the soundconsumer. In the Media_kit documentation it can be known as a mixer node which works as a codec node. These nodes are used to convert encoded media to a RAW format which can be viewed in a media player. The same happens when you have a Wav file and you want to encode it to MP3 format, you need such nodes to make the conversions.

This problem is also in openh323 they could fix the data that we are receiving but the way they are using, delays too much the audio feedback as we explained in the introduction of the documentation.

In yellowTAB we are working in the solution of the mixer node.

While this is on the way, the solution is:

- Building a buffer enough big to have an audible audio quality
- Fix the size before the packet is out from the network

In other words, in the hook function

We will receive the buffers with size as small as it is wanted, and we will store until we have 32KB of RAW data which it means around 200 milliseconds of speech.

So we will send **always** 200 milliseconds of speech compressed in different rates.

How do we know the compression rate of the other peer?

We know this information looking at the size of the buffer we received from the network, and with the following code:

```
nbr_skip = 32768 / size / 8;
```

We know the NBR rate.

So it is possible to have 2 persons talking with different compression rates.

How we do now

```
Void Recorder::DataEncode(void* cookie, bigtime_t timestamp, void* data, size_t size, const
media_raw_audio_format& format)
{
```

- Building the buffer.

```
memcpy((void*)&dataR.data[pointer],data,size);
pointer += size;
```

- Only when we have the 200 milliseconds of speech:

```
if (pointer == RTPPLLEN_RAW) {
        pointer = 0;
        size = RTPPLLEN_RAW;
        data = dataR.data;
```

- Ask for the compression

```
int32 nbr_skip = Preferences::Instance()->Compression();
```

- Check with the filter if the captured audio is not just containing silence, if it happens we will not send the data.

```
if(nbr_skip != 0)
{
if((non_null_trame = encodeur.decisionToEncode(data, size, nbr_skip, m_recVolume)) == true)
        {
```

- Compress/Encode it from the built buffer and leaving directly to the RTP structure

```
int size_of_buf = size / 4 / (nbr_skip*2);
char *buffer = rtpdata.data;

encodeur.encodeBuffer(data, size, buffer, nbr_skip);
data = (void*)buffer;
size = size_of_buf;
```

- Full the RTP information.

```
rtpdata.timestamp = CNetwork::Instance()->GetTimeStamp();
rtpdata.size = size;
```

- Send the packet as NET_ID_CODEC

```
if (CNetwork::Instance()->IsConnectionReady())
{
        if (nbr_skip != 0 ) // there is compression
        {
        if(send_data(find_thread(NET_THREAD_SEND_RTP), NET_ID_CODEC, &rtpdata, sizeof(TRtpData)
- RTPPLLEN_MAX + size /* sizeof(TRtpData)*/) != B_NO_ERROR)
                        printf("FATAL ERROR BLEU SCREEN\n");


        }
        else
        {
        if(send_data(find_thread(NET_THREAD_SEND_RTP), NET_ID_RAW, &rtpdata, sizeof(TRtpData))
!= B_NO_ERROR)
                                printf("FATAL ERROR BLEU SCREEN\n");
        }
}
```

Features of the sound

We solved the big problem of the different sizes that our buffers are sent to us. But notice that such buffers as they depend on the device we could have a soundcard send us in MONO, or with a different format. We have experimented already different frequencies. Almost all could be fixed with coding. But the mixer seems to be the safer and the most elegant solution.

3.4. *Improvements*

In most of the bugs commented in the last chapter, the fact of fixing them meant also an improvement. It means that moreover fixing the problem, it avoids other future problems. May they were fixed just making things easier that sometimes means also making things faster.

Looking at the code and *commented code* is possible to find out the intention of the previous developer. Some things were wanted to be included but were left for the next release. Such discovered things were also implemented, or in some cases just, finished to implement.

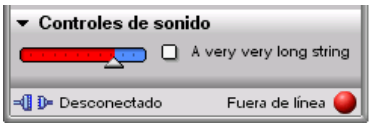
3.4.1. GUI and user friendliness

The graphical user interface had some factors that made it confusing:

Control as much as possible the strings:

Zeta gives the possibility in appearance preferences to change the font and the size of the system. The Locale kit gives us the possibility to change the language of our applications without rewriting them; I will comment how this is done later.

We can notice that our application should sometimes resize their controls to accommodate the different sized string or the size in a new language. Here we have an example how it can be done:



In the sound control we can see a checkbox item to disable the recording of sound. It is called "Demp" in Norwegian and "Silencio" in Spanish. If the string is larger it could overlap the slider object. So we have to place correctly this checkbox item making visible the whole string and then resize the slider to not to cover the checkbox.

```
m_pMuteCheckBox = new BCheckBox(BRect(…), "…", _T("Mute"), new BMessage(…),…);
m_pMuteCheckBox->GetPreferredSize( &w , &h);
m_pMuteCheckBox->ResizeToPreferred();
m_pMuteCheckBox->MoveTo(BPoint(AFrame.right - w - 5, 17) );
AddChild(m_pMuteCheckBox);
```

Where ***w*** is the width of the control string 5 just the separation from the String to the right side and 17 the coordinate y of the position of the control.

We create the control. Then we ask it which is it best size. As we are using the locale kit we don't know at compilation time how big it could be. We resize it and after moving it we add it to the view.

Disable controls when should not be used:

When we ask to connect to a host, the controls to connect or to type the IP or selecting the bookmarks stay enabled, giving to the user the possibility to start a new connection process. We should know that there are 3 controls to start the connection process: The pictured button in the toolbar, in the status bar and the *BButton* in the *ConnectView*. The first we did is disabling such components when the user asks to start a connection.

```
Output::Instance()->Network("Attempting direct Connection\n");
CNetwork::Instance()->SetView(this);
SetEnabled(false);
err = CNetwork::Instance()->ConnectFromUI(GetIP());
```

We also provided a new method to inform the Network which object need to send the message if a connection is finished.

Moreover is very important to provide a way to enable again such components when the connection is finished.

- Usually the network realizes that the connection is finished and it sends to the *connectView* us a message:

  ```
  BMessenger(m_connectview).SendMessage(new BMessage(MSG_CONNECTION_FINISH));
  ```

  And we reply to the main window by:

  ```
  case MSG_CONNECTION_FINISH:
  BMessenger(this->Window()).SendMessage(new BMessage(MSG_NOT_CONNECTED));
  SetEnabled(true);
  ```

- But could happen that we realize in the *ConnectView* due to the connection even cannot be placed. Here as before we inform the window which contains us.

  ```
  BMessenger(this->Window()).SendMessage(new BMessage(MSG_NOT_CONNECTED));
  SetEnabled(true);
  ```

The window when receives the message behaves like:

```
case MSG_NOT_CONNECTED:                ToggleConection(false);
                                       EnableActions(true);
                                       break;
```

It tells the buttons in the Status bar and in the toolbar to change their icons, and then to be enabled again.

```
BeNetWindow::ToggleConection(bool tog)
{
      if (tog) {
            m_pStatusBar->Connected();
            m_pToolBar->Connect();
      } else {
            m_pStatusBar->Disconnected();
            m_pToolBar->Disconnect();
      }
}
void BeNetWindow::EnableActions(bool tog)
{
            m_pStatusBar->SetEnable(tog);
            m_pToolBar->SetEnable(tog);
}
```

The methods in the classes *ToolbarView* and *StatusBar* to enable or disable its components (*BeNetButton's*) were not provided, so they were added:

```
void ToolBarView::SetEnable(bool enabled)
{
      _m_pRemote->SetEnabled(enabled);
      _m_pServer->SetEnabled(enabled);
```

```
}

void StatusBar::SetEnable(bool enabled)
{
	_m_pRemoteButton->SetEnabled(enabled);
	_m_pServerButton->SetEnabled(enabled);
}
```

There is a file called *def.h* which contains all the BMessage definitions. To implement all this, new messages needed to be created.

```
const uint32 MSG_CONNECTION_FINISH = 'cnfh';  //OliverESP
const uint32 MSG_NOT_CONNECTED     = 'ncOn';  //OliverESP
```

Error feedback to the user:

To accomplish this in most cases is just adding an Alert box when we got a returned code error.

A class called *WaitWindow* was around the source files of BeNet without using it. It was reused to show information to the user while it is connecting.

```
WaitWindow* waitWindow = new WaitWindow;
waitWindow->Show();
waitWindow->SetText(GetIP());
```

If the user closes the Remote conferencing window, there is no information about the ending of the communication. An alert box was added to ask confirmation to the user about finishing the conversation.

3.4.2. Multithreading Performance:

Avoiding blocks

It was usual to see in the code a thread the looks itself to ask itself if he has data in his internals buffers (sent by himself or another thread).

The *has_data()* (Kernel kit) function usually developers use it to avoid getting blocked in a receive function or sending function. If we call *receive_data()* and we have no data in our buffer we will get blocked, till someone send us some data. Something similar will happen if we send data with the call *send_data()* to a thread that has unread data in his buffer, this case we will get blocked until the target thread reads the data.

You should notice that the call to has_data() will not alwyas save to get blocked. For example between a has_data call and send another thread could send data to the same target before us, then we get blocked. Explained better

- We call *has_data*() and the thread buffer is free, so we decide that sending him data will not block us.
- A third thread makes the same and sends the data, filling the target thread buffer.
- We took the decision but the 3rd thread sent before us and calling *send_data()* will block us.

This works for receive_data(), but notice that there's a race condition between the has_data() and send_data() calls. Another thread could send a message to the target in the interim.

- In BeNet we see the case of *receive_data()*. We always need to know our thread_id:

  ```
  if (has_data(find_thread(NET_THREAD_WAIT_CONNECTION))) {
  ```

  We can see here that he looks for his thread_id passing his own name with *find_thread()* function. BeBook offers a faster way; we can retrieve the id of the calling thread by this way:

  ```
  if (has_data(find_thread(NULL))) {
  ```

- With *has_data()* we prevent a possible block. So this should be used if you want to some other processing while you are waiting your data. But when BeNet uses this, it just sleeps:

```
if (has_data(find_thread(NULL)))
{
	switch(receive_data(NULL, &rtpDataBuffer, sizeof(TRtpData)))  {
	}
[…]
}
else
	snooze(5000);
```

Looking for the same

Almost all these threads keep communication with others. They use the same function *find_thread* to look for others. The problem is that some of them are in an iteration and asks to the system multiple times for the identifier of the **same** thread *(thread_id)*. It could be done at the beginning of the code of the thread and reuse this identifier. The problem of this is that may the thread could die by some reason and than we have no control about this.

3.4.3. Data Transference through the stack and variable encoding compression

The codec we are using enables to set a NBR ratio. By default comes 2 presets in the Audio preferences "USE CODEC" and "MODEM QUALITY" these are ratios of 4 and 6.

This change was needed to solve the Audio problem explained in the chapter 3.3

Now there is a slider in the Audio preferences where you can change this ratio from 1 to 32. Some of the values are not mathematically possible and is not possible to select them. This compression rate what actually does is skipping samples for the audio buffer:

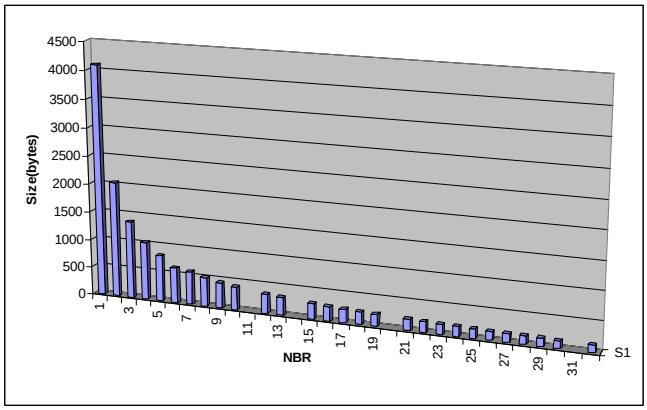
```
for(unsigned int i = 0; i < size / 2; i += no_of_trame_to_skip*2){
		sample_short = data_short [i];
		code = g721_encoder(sample_short, AUDIO_ENCODING_LINEAR, &state);
		resid = pack_output(code, enc_bits, buffer);
	}
```

NBR / freq	44100 kHz	48000 kHz
2	14700	16000
3	11025	12000
1	22050	24000
4	8820	9600

We can see how the index of the iteration is increased with the NBR rate and also multiplying by 2 it skips the stereo part of the buffer. By some way we could say that we down-sample the buffer, and really it is, we are not sending really 44,1 kHz or 48 kHz a depends the NBR rate, we are actually working with lower frequency sample rates. We can see in this table the supposed frequencies. Although we will not assume that we are down-sampling correctly only with this iteration.

5	7350	**8000**
6	6300	6857
7	5513	6000
8	4900	5333
9	4410	4800
10	4009	4364

The size of the packets in relation with this ratio can be seen in the diagram. BeNet was only possible to send 3 types of data: the 2 presets we have said (USE CODEC and MODEM QUALITY) and there was the possibility of sending non encoded audio (possible but takes 320 KB/s in a full duplex conversation). Adding this variable rate of audio compression makes the audio buckets have around 30 different sizes.



When an audio bucket is traced, it is send to the network sending RTP thread. Then the bucket goes to the protocol layer and finally to the connection layer. As the bucket audio in BeNet was fixed to 3 possible sizes, all the traffic the layers, the interfaces between them are prepared to these 3 fix sizes, RAW, CODEC or MODEM.

To identify the type of the bucket around the layers is there are the following definitions:

```
#define NET_ID_RAW          001
#define NET_ID_CODEC        002
#define NET_ID_MODEM        003
```

Actually now the only important thing is the real size of the bucket. It is stored in the *TRtpData* type:

```
struct TRtpData
{
	size_t size;
	int timestamp;
	char data[RTPPLLEN_MAX]; //OliverESP: change the order for spliting
};
```

So now the interfaces just send the pointer to this structure and the type of the bucket to keep compatibility in the protocol layer.

```
status_t CProtocol::SendRTPPacket(TRtpData& payload, int packetType)
void CConnection::SendRtpUdp(char *data, int packetType,int size)
```

An important improvement did here is that before the **whole** *TRtpData* was sent, even if only 1KB is in the data field (note that *RTPPLLEN_MAX* is set to 32 KB). So we moved the field to the end of the structure to be able just to send the first 2 fields and the necessary from the 3rd and not the whole, splitting ourselves the structure.

So now we can send buckets with multiple different sizes. Not passing information about packet type but the real size of it.

3.4.4. Buffering

As we explained to accomplish all the different inputs from the soundcards, we had to make wait till we have a considerable buffer.

When we had such buffer then we encode it, and we sent it. BeNet uses a function in the *Encode* object to create a buffer to accommodate the encoded buffer:

```
char *buffer = encodeur.createBuffer(size, size_of_buf, nbr_skip);
```

Where

```
char* Encode::createBuffer(size_t size, int &size_of_buf, int no_of_trame_to_skip)
{
	size_of_buf = size / 4 / (no_of_trame_to_skip*2);
	return new char[size_of_buf];
}
```

And then we have to copy the buffer to our *TRtpData* structure again.

The target is to create ourselves the buffer placing the pointer directly to the *TRtpData* structure. By this way we save 1 *memcpy*, save the dynamic creation of the buffer.
The call was replaced to:

```
size_of_buf = size / 4 / (nbr_skip*2);
char *buffer = rtpdata.data;
```

```
struct TRtpData
{
	size_t size;
	int timestamp;
	char data[RTPPLLEN_MAX];
};

struct TRtpHeader
{
	unsigned int v:2;
	unsigned int p:1;
	unsigned int x:1;
	unsigned int cc:4;
	unsigned int m:1;
	unsigned int pt:7;
	unsigned short seq;
	unsigned int ts;
	unsigned int ssrc;
	unsigned int csrc[CC];
};
```

3.4.5. Localization of the Application:

Make your application extensible to multiple translations in Zeta is easy thanks to the Locale kit and the Locale Server. And the most important is that any person could add another translation without modifications in the sources, just with a simple plain text file like this:

```
# Spanish Translation by Oliver Ruiz Dorantes
# 31 July 2004

#Window
"Whisper"                           "Whisper"
"Oscilloscope"                      "Osciloscopio"
"Sound Controls"                    "Controles de sonido"
"Connection"                        "Conexión"

#Conection
"Connect"                           "Conectar"
"Using IP Address"                  "Utilizando dirección IP"
"Protocol"                          "Protocolo"

#Status bar
"Connected"                         "Conectado"
"Disconnected"                      "Desconectado"
"Offline"                           "Fuera de línea"
"Online"                            "En línea"
```

On the left side there are the labels, and in the right side the translations. Notice that the labels are in English, we could say as the standard language. We elected this because when if our system is set to a language that the translation is not available, by default it shows the label. So it is good to have defined the label to a default language.
As we can see in the text file making new translations is just replacing the string in the right side.

The file must be called as the name of the application dot and the acronym of the language and country, for example this file was called "*Whisper.esES*" Spanish from Spain. The translation to the Portuguese talked in Brazil, should be called "*Whisper.ptBR*". Such file must be placed in a directory called Language/Dictionaries inside the folder of your application.

How the application uses these files?

The first thing is including the header from the *locale_kit*:

```
#include <locale/Locale.h>
```

We will see how to create a localized *MenuItem*. The normal way is:

```
BMenuItem* about = new BMenuItem("About Whisper", ... , ... );
```

This will create a *MenuItem* with this label "About Whisper". What we want in the application is:

- Ask the *locale_server* in what language our system is set.
- Look in the applications folder if we have the file language.
- Ask for the translated string.

We want a translated version of the *MenuItem* "About Whisper", just we need a translated String, and those 3 steps are reduced in:

```
BMenuItem* about = new BMenuItem(_T("About Whisper"), ... , ... );
```

You can notice *_T(x)* modifier, it is defined in *Locale.h*:

```
[...]
extern BLocale be_locale;
#define _T(str)____be_locale.GetString(str)
[...]
```

A closer look to de *BLocale* class can be taken in the headers of the Locale kit:

`/boot/beos/etc/develop/headers/be/locale`

4. The future

4.1. The Java server vs. a Muscle server

BeNet had integrated the interoperability to connect with a Java server. This java server receives registrations from the users and keeps in formations about them. It keeps the name, the surname, email as key, and the IP of the registering users in an internal database.
Then when a user wants to connect with another user, if it is registered in the server, just remembering his email (and if he registered in the server), the java server provides us his IP address. Also there is a feature who let us have the name, surname and email stored in a file, so we have like a buddy list of all our contacts and their mail addresses. We can see in the picture its behaviour.

This part in BeNet needed also to be fixed, because there were some problems if the name or surname of a person contains a space. Also updating the contacts list when a new contact was added, it usually crashed the application.

This solution makes us not remembering the IP address of a user but the email. It doesn't solve us too much. This is the information in the BeNet help about this Java server:

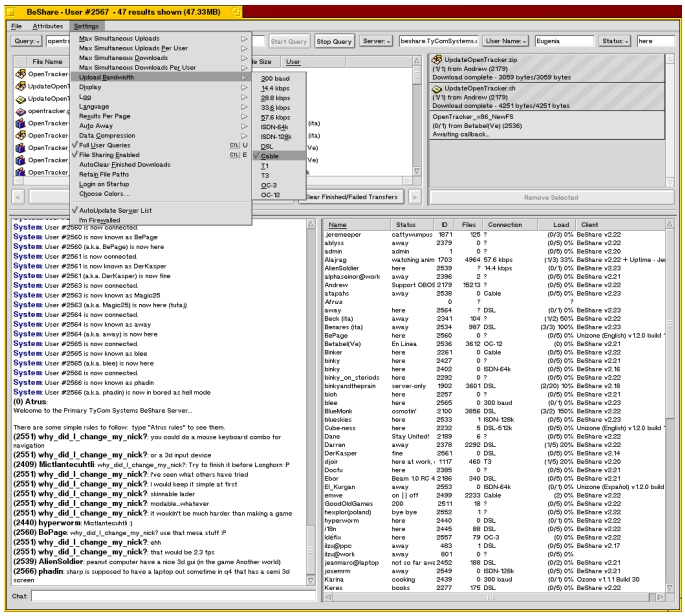
```
Installing the Java IP server

If you want to install the BeNet Java Server you must follow these steps :
You must get JRE (Java Runtime Environment) from Sun microsystems
From the command prompt of JRE, type:

java ServerIPAddress

The default command (as above) is similar to :

java -DTIMEOUT=60000 -DPORT=3030 -DCONNECTION=256 ServerIPAddress

You can costomize the command for your own needs :
```

DTIMEOUT	This set the time that the server will wait when connected to an idle connection without response
DPORT	This tell the server the port to wait on
DCONNECTION	This tell BeNet Java Server the number of connections to accept in a simultaneous way

Downloading the JRE package in the website of Sun Microsystems there is no clue about a class or a java file called `ServerIPAddress`. There was no way obtaining such server and looking its behaviour. This part could not be tested and fixed. It uses TCP sockets to make the connections between BeNet and the Java server. We believe that the same problems we would have with it correcting the sockets. So there was no point correcting this part if we didn't have a way to test it.

Moreover this is not a classical Be solution. Not long ago yellowTAB achieved the Java Virtual Machine for Zeta. It also don't give solutions to another bigger problems: Just connection to IP´s, what happens if the user we want to connect with is inside a LAN?, NAT, Proxies... etc, etc.

We looked at the BeShare software and its server MUSCLE: BeShare is a file sharing program, much like Napster or Kaaza, but tuned to take advantage of BeOS's unique features. We could describe this software as a file sharing application plus an IRC chat platform. Users use BeShare to connect the MUSCLE server, then they could see all the connected users to this MUSCLE server, chat with them in a public Chat room or in private. Not only this but watching the files they are sharing make queries from all the connected users and download them.



BeShare users experienced that they can get connected to the Muscle server almost everywhere, keeping file sharing capabilities and chat, inside LAN's with proxies present etc.

Most of the BeOS users use this application to share their files. This is one advantage that such Java server will not have. This reason and assuming that with Muscle we could solve the barriers of the proxies and the IP´s inside LAN's, make us at this moment disable the Java server thinking seriously about replacing it with a MUSCLE server.

A Be solution.

4.2. Interoperability

This feature we want to improve is already commented in the last point. About the Network is wanted to add support for SIP.

SIP has in a short time become popular with manufacturers and most of today's softphones support this protocol. SIP is based on well-known Internet protocols such as HTTP, and is defined as an Application Layer protocol, where placing and terminating VoIP calls, event notifications, or multimedia sessions are typical applications. Besides HTTP, SIP also uses some other IETF protocols. DNS (Domain Name System) and URLs are used to handle naming. Nodes use the SDP (Session Description Protocol) to exchange capabilities, and Multipurpose Internet Mail Extensions (MIME), allow for application integration. This makes it easy to integrate SIP with other Internet technologies and more importantly for yellowTAB, with Zeta applications. When MacOS and Windows were developed, hardly anyone knew the Internet even existed. But BeOS had the advantage of evolving while the Internet was in heavy growth mode, and Be was paying attention. Of course, the MIME standard was never designed to support an entire operating system, so Be had to add a lot of extra functionality to the basic idea. MIME handled documents well, but was never designed to handle the application files them selves. As an illustration we will use the application CDPlayer, it has the following MIME type registration: application/x-vnd.Be-CDP.

Work through NAT. Most users with a DSL connection will have a router configured to make use of NAT. This can cause major issues with the UDP-based SIP standard. NAT uses a single public address to support multiple internal machines. The SIP standard does not support NAT address translation, and this means that the softphones will use

incorrect IPs in the packet headers. Most SIP providers use a workaround involving STUN servers to avoid this problem.
The most important would be integrating ohphone application, bringing Whisper the openh323 protocol. With the h323 gateways and gatekeepers will solve a lot of problems with network interoperability and the native video already working for Zeta.

Inside yellowTAB there have been suggestions on integrating VoIP (and Whisper) with the BeOS IM Kit (an Instant Messaging kit natively developed for BeOS) with all it means in the contact list, to have a Voice mailbox, and interoperability with another open Standards and VoIP servers. To accomplish this also new codecs should be added.

Currently almost all IM clients bring some videoconference support: MSN Messenger, Yahoo! and AOL IM. Here we have some other existing projects:

http://www.skype.com – probably the most downloaded and popular VoIP application. It exists on several platforms, and is know for being heavily criticised for violating every VoIP standard.

http://www.voipster.com – similar to Skype, but follows the existing standards.

http://www.xten.com – probably the most commonly provided softphone from VoIP providers. They are also these days finishing their video conference product: http://www.xten.com/index.php?menu=products&smenu=eyebeam

http://www.apple.com/ichat - Apple user's first choice. Having a proper Internet connection and Apple's iSight firewire webcam you have one of the market's best video conference solutions.

4.3. The Audio engine

4.3.1. Threads priority

We will have to ensure the correct transfer of data between the threads in the application.
By default the treads that are dealing with media data it is assigned a real time priority (refer to the kernel kit for more information).

Summarizing BeOS / Zeta approach the Real Time Operating System definition with the priority of its threads.

Thread Priorities

In a multi-threaded environment, the CPUs must divide their attention between the candidate threads, executing a few instructions from this thread, then a few from that thread, and so on. But the division of attention isn't always equal: You can assign a higher or lower *priority* to a thread and so declare it to be more or less important than other threads.

You assign a thread's priority (an integer) as the third argument to spawn_thread(). There are two categories of priorities: "time-sharing" and "real-time."

- **Time-sharing (*values from 1 to 99)***. A time-sharing thread is executed only if there are no real-time threads in the ready queue. In the absence of real-time threads, a time-sharing thread is elected to run once every "scheduler quantum" (currently, every three milliseconds). The higher the time-sharing thread's priority value, the greater the chance that it will be the next thread to run.
- **Real-time (*100 and greater)***. A real-time thread is executed as soon as it's ready. If more than one real-time thread is ready at the same time, the thread with the highest priority is executed first. The thread is allowed to run without being pre-empted (except by a real-time thread with a higher priority) until it blocks, snoozes, is suspended, or otherwise gives up its plea for attention.

The Kernel Kit defines seven priority constants (see Thread Priority Values for the list). Although you can use other, "in-between" value as the priority argument to spawn_thread(), it's suggested that you stick with these.
Furthermore, you can call the suggest_thread_priority() function to let the Kernel Kit determine a good priority for your thread. This function takes information about the thread's scheduling and CPU needs, and returns a reasonable priority value to use when spawning the thread.

Moreover the media threads in BeNet the other important threads are the network/RTP threads. They are as much important as them, because the information the media threads are generating is going processed by the network threads.

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

We even should ensure that after the recorder produces a buffer and it ask to the network for sending it, the next thread to get the CPU to send it. We could ensure it by setting the Network thread a real time priority some points **lower** than the recording thread. It cannot be **higher** because we need to produce the sound before sending it.

Setting a higher priority, the BeOS scheduler will assign the CPU first to the thread with higher priority, in this case the RTP thread which is waiting to receive sound data form the recorder thread (which still had not the CPU to produce it). We have to keep in mind this possible deadlock.

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

Increasing the priority of some threads is a fact the can help us to improve the quality of the conversation and performance.

4.3.2. Double buffering

An essential part when we are managing with media buffers that need to be played is implementing a double buffer system. This technical is based to ensure that always we have media data to play having a continuous reproduction.

How can we understand that in a VoIP application that when the packets are send as soon as possible and the other peer plays it as soon as they are received?

How can we have a second buffer ready do be played when we are playing speech if supposedly the other peer is speaking it?

The solution is delaying us more, much more, two times more. The other peer will start reproducing the buffers when it has 2, and while he is reproducing the first, he has already the second, and in normal conditions when it is going to be reproduced the second buffer, should arrive the third. By this way we always have a buffer stored, while we are playing the previous.

The problem: we delayed us two times. We already had a delay of 200 milliseconds and two times this will mean 400 milliseconds. Solution: with the mixer we can control everything from the audio format we are receiving the packets. Instead of 32KB of RAW audio we could set it at 16KB having as 100 milliseconds of standard delay and implementing the double buffering it will become again 200 milliseconds, so we achieved double buffering with the same delay.

4.3.3. Overlapping

If just a packet of the sequence arrives a little bit delayed, it produces that its reproduction is also delayed. Then we could imagine if the next packet arrives on the expected time. It could happen that the packet on time is ready to be played when the previous packet is still being reproducing.

A solution could be keep a queue, but this is assuming that we have could have an important delay, it not VoIP philosophy. Keeping such queue will make us keep this delay for the whole conversation and possibly increasing it. How we could know when its time to skip a frame to recover the synchronization?

4.3.4. Discarding Packets

In bad network quality condition as we don't have a QoS (quality of service) guaranteed we could have the following case:

- We receive a buffer and we play it.
- We receive another buffer but we realize with the timestamp or with the sequence number that it should be played before the one we played.

In this case we must just discard the package and not playing it because its information is now not useful. Playing it just will introduce confusion in the conversation.

4.3.5. Statistics

An important thing to have if we want to improve the application is to have full reports about the behavior of the connection. Also it could be interesting to the user.

Providing this information will give us the chance to controls our delays, modify our compression rates. By some way adjust to the capabilities of your network, scaling with ease to bandwidth needs.

Times

Here there are some times taken from a Laptop Compaq Armada E500 with a Pentium III at 800 MHz with 384 MB of RAM and Zeta RC3:

- The hook function is called around each 200 ms
- The hook function takes around 800 μseconds
- around 1200 μseconds when it has a build buffer
- around 2700 μseconds when it encodes the buffer (max latency of the hook function)
- Since the buffer is sent by the hook till it is sent through the network in most cases it can take less than1 millisecond, but we experienced delays till 12 milliseconds. Adding real time priority to the thread this delay could be fixed. Anyway the average is 1.5 milliseconds, and these times were taken when the sys was occupied by several applications. Those 12 milliseconds supposedly were cases that another application obtained the CPU and not our thread.