JIWYNET:

A project to implement the control of the mechatronic system JIWy over a computer network

Laurence Smith
IOO Project

Supervisors: Dr.ir. J.F. Broenink
ir. D.S. Jovanovic
ir. G.H. Hilderink

August 2002
Control Laboratory
Electrical Engineering Department
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands
Summary

JIWY is currently a mechatronic system in the Control Laboratory at the University of Twente. It consists of two motors that control the apparatus used to hold a webcam. The two motors are controlled by the x and y positions of a joystick. The control loop is run in real-time with RTLinux, a real-time operating system. RTLinux effectively runs Linux as its lowest priority to generate real-time characteristics (i.e. meeting hard deadlines, being completely reliable, etc). The webcam, which is attached to the apparatus, streams its captured video data into the third-party CamStream software that runs in the user mode of RTLinux’s lowest priority. The CamStream software uses the Video4Linux API along with Qt, a multi-platform C++ GUI toolkit to display the video stream.

The idea behind the JIWYNET project is to implement the control of the JIWY system from a remote PC, over an IP network. The remote PC would have access to the video stream from the webcam and the controls of JIWY’s actuators, thus enabling an interactive demonstration of the JIWY system via an Internet connection.

The JIWYNET application consists of two sets of code. The client software is written in JAVA, while the server side software is the altered C++, CamStream. By writing the client side software in JAVA, the user should be able to download the code at any remote PC and then run the application on the JVM layer. The network connection effectively uncouples the Java and C++ programs.

The server side also runs a controller implemented in 20SIM. The final part of the JIWYNET project was to write kernel and user applications for the RTLinux server, enabling the real-time controller to react to the remote clients camera position requests. This report outlines the design efforts put forth to stream live video over an IP network for the JIWYNET project. It also documents the aspects of design regarding the real-time control and optimization of the JIWYNET project.
# Table of Contents

Summary ................................................................................................................................. i
Table of Contents ................................................................................................................. ii
List of Figures .......................................................................................................................... iii

1.0 Introduction ....................................................................................................................... 1
  1.1 The Basic Ideas Behind the JIWYNET Project .............................................................. 1
  1.2 Computer Networks and the Internet ........................................................................... 1
  1.3 Protocol Layers ............................................................................................................. 1
  1.4 Overview of the Transport Layer in the Internet ......................................................... 2
    1.4.1 Connection Oriented Transport: TCP ................................................................. 3
    1.4.2 Connectionless Transport: UDP ........................................................................ 3
  1.5 Making the Decision for JIWYNET: TCP or UDP ...................................................... 3

2.0 JIWYNET Application Overview ................................................................................. 5
  2.1 Programming Language Decisions ............................................................................ 5
  2.2 CamStream Orientation and Program Flow ............................................................... 5
  2.3 JIWYNET Classes and CamStream Alterations ......................................................... 7
  2.4 RTLinux and JIWYNET Control ............................................................................... 8

3.0 Socket and Multimedia Programming ........................................................................... 10
  3.1 Socket Programming .................................................................................................. 10
    3.1.1 UDP Socket Programming in C++ (Unix Environment) ..................................... 10
    3.1.2 UDP Socket Programming in JAVA .................................................................. 11
  3.2 Streaming Live Multimedia Over an IP Network ...................................................... 12
  3.3 Packet Stamping and Dealing with Lost Packets ....................................................... 12

4.0 Discussion ....................................................................................................................... 14
  4.1 Packet Size, Transmission Rate and Network Traffic .............................................. 14
  4.2 JIWYNET Results .................................................................................................... 17
  4.3 Video compression .................................................................................................. 19

5.0 Conclusions and Recommendations ........................................................................... 20
  5.1 Conclusions ............................................................................................................ 20
  5.2 Recommendations .................................................................................................. 20

References ............................................................................................................................ 22

Appendix A - JIWYNET System Overview ........................................................................ 23
Appendix B - RGB 32 Image Format and Network Byte Order ........................................ 24
Appendix C - Video4Linux API ......................................................................................... 25
Appendix D - Qt SocketNotifiers .................................................................................... 27
List of Figures

Figure 1.1 - The Internet Protocol Stack ...............................................................2
Figure 1.2 - Popular Internet Applications and their Underlying Protocols ............2
Figure 2.1 - Relevant CamStream Classes ..............................................................6
Figure 2.2 - Program Flow of Unaltered CamStream ............................................7
Figure 2.3 - JIWYNET Client Software’s State Chart.............................................8
Figure 2.4 - Application Architecture for JIWYNET Control .................................9
Figure 4.1 - Network Bottleneck Scenario .............................................................14
Figure 4.2 - Effect of UDP Packet Transmission Rate on UDP Packet Loss ..........15
Figure 4.3 - Effect of UDP Packet Size on UDP Packet Loss (Heterogeneous Case) ....16
Figure 4.4 - Effect of UDP Packet Size on UDP Packet Loss (Homogeneous Case) ....16
Figure 4.5 - Effect of Packet Size on the Time spent receiving Packets (JIWYNET) ...18
Figure 4.6 - Effect of Packet Size on Packet Loss (JIWYNET) .............................18
Figure A - JIWYNET System Overview .................................................................23
1.0 Introduction

1.1 The Basic Ideas Behind the JIWYNET Project

The underlying idea behind this project is to implement the control of a real-time mechatronic system from a remote PC, over a network. Successfully completing this task is not easy as real-time systems need to both meet hard deadlines and be completely reliable. As discussed later, the protocols that deal with packets of information travelling over the Internet are not completely reliable. Furthermore, unpredictable network traffic can cause delays in packet transfer, and even the loss of packets.

In the case of JIWYNET, the remote PC has access to the video stream from JIWY’s webcam and the controls of its actuators, thus enabling an interactive demonstration of the JIWY system via an Internet connection. JIWYNET has also provided the initial framework for further exploration within this field. This report discusses the design solution regarding the networking software that conducts the transfer of the live media for the JIWYNET project. It deals with important issues such as which transport layer protocol to use along with other networking details vital to the successful completion of the JIWYNET project. It also delves into the workings of CamStream, a software tool that enables the use of webcams among other video devices in Linux. Furthermore, the implementation of remote control is also dealt with.

1.2 Computer Networks and the Internet

The Internet is a public, worldwide computer network. It consists of a complex combination of hardware and software that allows for inter-computer communication. The hardware consists of items such as servers, personal computers and routers while the software components are applications such as web browsers or electronic mail servers. Furthermore, computer networks consist of Local Area Networks (LAN) and Wide Area Networks (WAN). To complicate things further, different networks have various transport rates and available bandwidths, which in turn introduces network bottlenecks. In order for information to be sent effectively throughout such a complex web of network devices, protocols have been developed to formalize the procedure.

1.3 Protocol Layers

Protocols define the format and the order of messages exchanged between two or more communication entities, as well as the actions taken on the transmission and/or receipt of a message or other event. The Internet is recognized as a five-layer protocol stack. The protocols are layered as shown in Figure 1.1 [6].
At the lowest level, the physical layer is responsible for moving individual bits between links through the transmission medium (i.e. twisted-pair copper wire, single-mode fiber optics). Above the physical layer lies the link layer. It provides the services that transfer information between the individual nodes in the network. Nodes are network components such as routers. The network layer makes the decision of which nodes information must pass through while traveling between hosts. The protocol for this layer is the familiar Internet Protocol (IP). Next is the transport layer. It consists of two protocols: User Datagram Protocol (UDP), and Transmission Control Protocol (TCP). These protocols provide the necessary services to transport application layer information between the client and server sides of an application. These protocols are discussed in further detail in the next section. Finally, at the top is the application layer. It is responsible for supporting the actual network application. Examples include: FTP for file transfer programs, HTTP for web browsers, and SMTP for electronic mail. This report discusses both the proprietary application layer protocol developed for the JIWYNET application and the transport layer protocol used, namely, the UDP protocol.

### 1.4 Overview of the Transport Layer in the Internet

The Transport Layer consists of the TCP and UDP protocols. When designing a network application, the software developer must decide which of these two protocols to use. The decision should be based on the nature of the application and what kind of data transfer is necessary. Some popular Internet applications and their underlying transport protocols are found in Figure 1.2 [6].

<table>
<thead>
<tr>
<th>Application</th>
<th>Application Layer Protocol</th>
<th>Transport Layer Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronic Mail</td>
<td>SMTP</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>HTTP</td>
<td>TCP</td>
</tr>
<tr>
<td>File Transfer</td>
<td>FTP</td>
<td>TCP</td>
</tr>
<tr>
<td>Streaming Media</td>
<td>Proprietary</td>
<td>UDP</td>
</tr>
<tr>
<td>Routing Protocol</td>
<td>RIP</td>
<td>UDP</td>
</tr>
<tr>
<td>Network Management</td>
<td>SNMP</td>
<td>UDP</td>
</tr>
</tbody>
</table>

**Figure 1.2 - Popular Internet Applications and their Underlying Protocols**
1.4.1 Connection Oriented Transport: TCP

TCP is a connection oriented transport protocol. That is, before one application process can begin to send data to another, a ‘handshake’ must occur between the two processes. The ‘handshake’ is a method whereby the two sides basically send information to each other to establish the parameters of their subsequent data transfers.

TCP is also a reliable data transfer service. Information sent with TCP is not only received at the other end of the application, but it is also received in the correct order. This makes TCP a useful protocol for networking applications that require the complete transfer of information between two computers. All of the data transfer security does not come without a cost however. The TCP protocol introduces overhead in each packet of information it sends along with an acknowledgement and retransmission routine it must follow in order to guaranty delivery. It also provides a flow control service that eliminates the possibility of the sender process sending too much data too quickly, which would result in an overflow at the receiver’s end. Thus TCP sends data slower than UDP.

1.4.2 Connectionless Transport: UDP

UDP is a much simpler protocol than TCP. It does not provide retransmission of lost packets or any sort of transmission control. UDP simply takes messages from the application layer, attaches information about the destination of the packet of information, and then passes the resulting segment to the network layer where it is sent. No connection is necessary with UDP, thus it is a means of connectionless transportation.

Since UDP is a simple protocol it has the ability to send data much quicker than the TCP protocol. As in the case of the TCP protocol where there was cost associated with reliability, there is a cost with UDP and speed. When using the UDP protocol there is no guaranty that a packet will make it to its destination. In fact, if the packet does make it to its destination, there isn’t even a guaranty that it has arrived in the correct order.

1.5 Making the Decision for JIWYNET: TCP or UDP

The basic idea behind JIWYNET is to send media across a network. This premise introduces the issue of speed. In order to send and receive streaming media across the network the connection must be quick. Also, large bandwidth must be utilized when sending media. Although TCP guaranties delivery of information in the correct order, it has several issues that make it unsuitable for media delivery, especially in the case of live media.
TCP introduces latency by making an initial handshake connection. UDP does not use any formal preliminaries; it just sends the information. Thus UDP lacks any initial delay.

For information to be sent over a network it must be divided into small packets that can be sent individually. In the case of TCP, an overhead of 20 bytes is introduced into each packet. For UDP, only 8 bytes of overhead are necessary per packet.

TCP has a congestion control mechanism that throttles the amount of information being sent. This helps reduce lost packets but it keeps information from being sent as quick as possible. UDP on the other hand, sends information as fast as the local application can generate it. The only constraints are the capabilities of the local machine and the access bandwidth to the network.

For the JIWYNET application, ideally video would be displayed on the remote hosts monitor as it happens. Sending data as quickly as possible is the main concern. Thus the UDP protocol was chosen to implement the application.
2.0 JIWYNET Application Overview

2.1 Programming Language Decisions

The JIWYNET application has its server side program written in C++, while its client side is written in JAVA. Justification for the two programming languages is the following. Existing CamStream software, written in C++, already exists for the Philips webcam and Linux system that JIWY is running on. Thus it makes sense to alter the existing C++ code for the server application purposes. For the client however, software that runs on a variety of platforms is desired.

JAVA runs on the JAVA Virtual Machine (JVM). The JVM is a software layer on top of the local computer’s existing hardware platform and operating system. It is capable of running compiled JAVA programs. JAVA code does not have to be rewritten for different operating systems or hardware platforms. Thus JAVA was the choice of the client software.

The uncoupling of the two programming languages is achieved by converting the information to bytes in a specific format when sending the information over the network. More information on this topic is found in part 3 of this report.

2.2 CamStream Orientation and Program Flow

CamStream is a collection of tools for webcams and other video devices. It is written in C++ for an UNIX environment. It uses the Qt Framework for its GUI [13]. To communicate with the webcam or video device the CamStream software uses the Video4Linux API (refer to Appendix C for more on the Video4Linux API) [7].

CamStream is composed of many classes. Figure 2.1 lists some of the important classes relative to the JIWYNET project and how they relate. Note that only the significant member functions and variables are listed.

The CCamPanel and CImagePanelRGB classes are responsible for displaying the image captured by the webcam or video device. The timing between function calls is implemented by slot/signal technique using QSocketNotifiers (refer to Appendix D for more on QSocketNotifiers).

The CVideoDevice class does a majority of the image processing. It keeps track of where and when image data is stored in memory along with the format of the data. Its Init() function uses the Video4Linux API to set up most of the variables. The MCapture() and Msync() functions are used to communication with the webcam or video device. Mcapture() tells the webcam to start capturing data while the Msync() function forces the program to wait for the capturing process of the current image to finish.
The flow of the unaltered CamStream application is described in Figure 2.2. First, the Notify() signal is connected to the UpdateImage() function. Then most of the video device and image variables are set in the init() routine. The StartCapture() function starts the repetitive image capturing and displaying schedule. Whenever the activated() signal is received, the LoadImage() function is entered, followed by the ReadImage() function. CamStream works with two images at a time. While the webcam is busy capturing a frame, the software displays the most recent image in memory. After the image has been shown, the Msync() function is called to finish the capture of the current image. Upon completion, the Mcapture() function is called, starting the capture of the next image. The Notify() function is then executed, commencing the image display algorithm again, and thus repeating the whole loop.
2.3 JIWYNET Classes and CamStream Alterations

To send the image data gathered by the CamStream software, a new class, NetworkDevice, was created. It is responsible for dividing images into stamped packets and sending them to the client. In the altered CamStream software, a NetworkDevice object is created when the VideoDevice object calls its init() function. Then, immediately after the Mcapture() function is called in the loop above, the sendUDP() function of the NetworkDevice object is called to send the most recently acquired image.

To receive the UDP packets sent over the network by the server application, three JAVA classes were designed and created:

**JNETsocket** - This class is responsible for opening a socket and listening for UDP packets. It converts the first two bytes of each packet into a packet id number with which it uses to organize the packets it receives into the correct order. It stores the data it receives into an array of ints.
**JNETviewer** - This class’s responsibilities include decrypting the image data from RGB 32 format (more on RGB 32 in Appendix B) and into a displayable image. It also displays the GUI with the incoming images.

**JNETclient** - This class runs the main loop, commencing the acquisition of an image from the network, and then triggering the image display routine. It also implements double buffering.

The client program executes as a relatively simple three state process as shown in Figure 2.3. The main control state keeps track of what image data is available and what image has been displayed. This is necessary for the double buffering that makes the displayed video appear smoother.

![Figure 2.3 - JIWYNET Client Software’s State Chart](image)

### 2.4 RTLinux and JIWYNET Control

JIWY’s control loop is run in real-time with RTLinux, a real-time operating system. RTLinux effectively runs Linux in its lowest priority to generate real-time characteristics (i.e. meeting hard deadlines, being completely reliable, etc). Real-time code is written as a kernel module that is managed by the real-time kernel, while regular code is run in Linux as a user space application.

Programs running in kernel mode have restrictions. For instance, they cannot open files. For this reason, a FIFO buffer is available to allow kernel mode applications to communicate with user mode applications. For the JIWYNET project, a FIFO buffer was implemented to allow JIWY’s kernel mode controller to communicate with the user mode application that receives UDP packets of control commands. This setup is shown in Figure 2.4.
At the client side, the user can use the mouse to click on the image. The corresponding position of the click relative to the image is sent to the control loop, and the camera is moved so that the chosen position is the center point of the displayed image.
3.0 Socket and Multimedia Programming

3.1 Socket Programming

Network applications consist of a pair of programs: a server side program and a client side program. In order for the processes running in these programs to communicate with each other they must read and write to sockets. Sockets are the doorways to the network that programs must send and receive information through.

3.1.1 UDP Socket Programming in C++ (Unix Environment)

A large part of programming network applications is keeping track of addresses. To work with addresses in C++ the structure sockaddr has been created. It contains address information for different kinds of sockets:

```
struct sockaddr {
    unsigned short     sa_family;     // Address family
    char                sa_data[14];   // 14 bytes for protocol address
};
```

For the JIWYNET application, UDP sockets were used [3]. Thus the sockaddr_in structure was implemented (the ‘in’ stands for Internet). The sockaddr_in structure is:

```
struct sockaddr_in {
    short int           sin_family;   // Address family
    unsigned short  sin_port;     // Port number
    struct in_addr      sin_addr;     // Internet address
    unsigned char       sin_zero[8];  // So same size as struct sockaddr
};
```

For UDP sockets or TCP sockets for that matter, sin_family is set to AF_INET. The sin_port and sin_addr are the port number and IP addresses, respectively, and must be in network byte order. The sin_zero[8] variable is simply set to all zeros. Its purpose is to make the sockaddr_in structure the same size as the sockaddr structure. Note that a pointer to a sockaddr_in structure can be cast to a pointer of sockaddr structure type. This is important as some of the socket programming functions expect a sockaddr structure type.

To communicate with sockets, a file descriptor to a socket must be declared. This is done with the socket() function. To use this function both the sys/types.h and sys/socket.h header files must be included.

```
int socket( int domain, int type, int protocol )
```
For socket programming, the domain is AF_INET, and the protocol can be set to 0. The type can be either SOCK_STREAM or SOCK_DGRAM. Using SOCK_STREAM would declare a TCP socket. For JIWYNET, SOCK_DGRAM is used as it leads to a UDP socket. The socket() function returns an int that is the file descriptor for the specific socket.

For TCP socket programming, after opening the socket, functions such as bind(), listen() and accept() must be called in order for the successful transfer of information. With UDP however, the only other command needed is the sendto() function:

```c
int sendto( int sockfd, const void *data, int len, unsigned int flags, const struct sockaddr *to, int tolen)
```

The parameters of the function define the socket file descriptor, the data to be sent, the length of the data to be sent, the address the packet is to be sent to and the address’s length.

Other functions useful for UDP socket programming are the close() function and the recvfrom() functions. The close() function simply terminates the file descriptor to the socket. The recvfrom() function is used on the receiving end of socket communication.

### 3.1.2 UDP Socket Programming in JAVA

Compared to socket programming in C++, socket programming in JAVA is simpler as fewer details need to be accounted for. There are two main classes for receiving datagram packets in JAVA [6]. They are: java.net.DatagramSocket and java.net.DatagramPacket. The DatagramSocket class represents the socket used to send datagram packets, while the DatagramPacket class represents the actual packets that are sent.

To open a socket that enables the application to listen for UDP packets the following command is used:

```java
DatagramSocket client_socket = new DatagramSocket();
```

Since no arguments are passed to the constructor, the socket is not bound to any particular socket or IP address. A DatagramPacket must be constructed with information about where and how much data to store, in order for the information received at the socket to be processed. This is accomplished with:

```java
DatagramPacket receive_packet = new DatagramPacket( byte[] data, int length);
```
where data is the array that information from the incoming UDP packet is stored in, and length is the length in bytes of the amount of data to be stored. To actually receive the data, the command used is:

```java
client_socket.receive( receive_packet );
```

This command simply places the data in from the DatagramPacket into the specified byte array. As in the C++ case, after the information has been successfully transferred the socket must be closed. This is accomplished with the close() command.

### 3.2 Streaming Live Multimedia Over an IP Network

The Internets network layer protocol, IP, is a best-effort service. That is it makes no guarantees about the end-to-end delays experienced by the packets of information it transports; it only makes its best effort to deliver them as quickly as possible. Thus, two main concerns for the successful streaming of live media over an IP network are: time considerations and tolerance to data loss.

Networked media applications are highly sensitive to delays. Packets of information that incur a sender-to-receiver time delay of more than a couple hundred milliseconds become useless. For example, receiving data about a previous video frame for a game or movie is of no use to the client application. Conversely, multimedia is tolerant to lost data. A small percentage of lost data may cause occasional glitches in video or audio playback; however, it can often be partially or fully concealed.

### 3.3 Packet Stamping and Dealing with Lost Packets

To meet time constraints, the JIWYNET project was kept simple. Thus for the initial application, uncompressed images were sent over the network. In this case, data formatting was kept to a minimum (refer to Appendix B for information on the RGB 32 image format). Also, to organize the packets into the appropriate order and thus images at the receiving end, each packet was stamped with a 2-byte integer before it was sent.

The first two bytes of each packet were stamped according to the packets place in an overall image. Because UDP transmission does not guarantee packets arrive in the correct order, this simple stamping routine allows the client software to arrange the packets it receives. It also allows for a simple way of parsing the images at the client end. In pseudo code, ‘when the packet with a certain stamp arrives, move on to the next image.’ Problems may arise with this algorithm if the last packet in a set is lost somewhere over the network. A frame or two may be lost, however, in this case the application will keep running. By stamping the packets of each image repeatedly
with the same identification numbers, eventually the last packet of an image will arrive at the client end, triggering the next image.

It is inevitable that some of the transmitted UDP packets will be lost; using packet stamping helps to conceal missing image data. By leaving data from the previous image in the image buffer, any areas in an image that are not updated for a new frame due to lost packets, simply repeat the last image. Since most images are similar to the last, packet loss will have a minimal effect on the video stream.
4.0 Discussion

4.1 Packet Size, Transmission Rate and Network Traffic

TCP-based applications such as SMTP and HTTP currently account for approximately 80 percent of the Internet’s bandwidth. This is partly due to both the number of TCP applications, and that TCP implements a sliding window flow control mechanism. When TCP detects packet loss, not only does it retransmit packets, it also reduces its output rate in order to limit further network congestion. That is, it reduces its window size. When there is no congestion, TCP increases its window size and output rate. Conversely, there is no flow control involved with UDP. Since TCP and UDP must share Internet bandwidth, their performances are affected by the others presence.

UDP packet loss happens frequently at network nodes similar to the ones represented in Figure 4.1 where the LAN and WAN meet. This is because the LAN has a set bandwidth of 10 Mbps and the WAN only has a bandwidth of 1.5 Mbps. When a large quantity of data is flowing from the LAN to the WAN the buffer will fill quickly and any additional packets are lost.

The following scenario was set up and tested by Sawashima, Hori and Sunahara [9], to quantify the effects of TCP network traffic on UDP packet loss.

![Network Bottleneck Scenario](image)

Figure 4.1 - Network Bottleneck Scenario

The correlations explored were the effect of TCP traffic on UDP packet loss, namely, the effect of UDP transmission rate on UDP packet loss and the effect of UDP packet size on UDP packet loss. In both cases two scenarios were examined: homogeneous and heterogeneous network connections. In the homogeneous case all the network delays were uniform, namely, 1ms on the LAN sides and 50ms on the WAN. For the
heterogeneous case, network delay increased by 2ms for each additional connection. For example, the delays on the LAN side were 1ms, 3ms, 5ms, etc.

Figure 4.2 shows the effect of UDP Transmission Rate on UDP Packet Loss. The packet size is constant at 80 bytes, and the UDP connection shares the network with 12 TCP connections. From Figure 4.2 it is clear that packet loss does not vary much with the transmission rate in the heterogeneous case, however, in the homogeneous case, packet loss raises steeply at lower transmission rates. The TCP window sliding mechanism explains this effect. That is, at a lower transmission rate, the UDP connection allows the TCP connections to increase their share of the network bandwidth. In the homogeneous case, the TCP connections have synchronized window size corrections. Thus the buffer at the network node is filled at a higher frequency and the UDP packet loss rate increases.

![Figure 4.2 – Effect of UDP Packet Transmission Rate on UDP Packet Loss](image)

Figures 4.3 and 4.4 show results of streaming data at a constant rate of 64 Kbps with 6, 9, and 12 TCP connections respectively. The figures show both the homogeneous and heterogeneous cases. The packet loss rate is defined as the ratio of lost packets to sent packets.
As above, packet loss rate seems to have substantial increases only in the homogeneous case when the TCP connections are synchronized. The synchronization of the TCP connections allows for many TCP packets to arrive at the
network node around the same time. This causes the buffer to fill rapidly on a periodic basis and leads to the loss of successive UDP packets in a periodic manner.

Figure 4.4 explicitly shows that the smaller packet sizes do not seem to be affected by the TCP synchronization. This too is explained with the characteristics of TCP congestion control. To be transferred at 64 Kbps, UDP packets of 80 bytes are sent at a rate of 100 packets/sec. On the other hand, UDP packets of 320 bytes are sent at a rate of 25 packet/sec to meet the 64 Kbps target. Therefore, more packets are sent when the packet size is 80 bytes. Since the network node congestion is based on the number of packets in the FIFO buffer, the TCP congestion control mechanism keeps the window size relatively smaller when the experiment is run with smaller UDP packet sizes. The smaller TCP window sizes have less affect on the packet loss.

The above results conclude that small packets sizes at high transmission rates lead to the most efficient transfer of UDP packets. These results are based on the fact that there is network traffic. And the results are most prominent when the traffic consists of synchronized TCP connections. It should also be noted however, that with smaller packet sizes the number of packet headers increases. This increases the bandwidth use of the application sending the UDP packets.

4.2 JIWYNET Results

Two tests were executed to determine the most efficient packet size for the JIWYNET project. First, a comparison of packet size to the time it took to receive 600 images worth of packets. Then a comparison of packet size to the percentage of packets lost during transmission. The results are shown in figures 4.5 and 4.6 respectively.

For both cases, the tests were run on the local network with limited network traffic, that is, practically undisturbed conditions. From the first test, it was observed that a packet size of 258 bytes lead to the least amount of processing time at the client side. Although this result is very similar to that of packet sizes of 514 and 1026 bytes, it is not close to the results shown at lower packet sizes. By lowering the packet size, more overhead has to be introduced to the transmission as each packet has a 2 byte packet id, along with the 8 byte overhead introduced by the underlying UDP protocol.

The results shown in Figure 4.6 show that the rate of packet loss increases with the size of the packets. The packet loss rate is very small, however, and in fact undetectable in the final video stream.

It is important to note that these results are based only on a local network with limited network traffic. They may change depending on the network traffic and the distance of travel between source and destination.
Figure 4.5 – Effect of Packet Size on the Time spent receiving Packets (JIWYNET)

Figure 4.6 – Effect of Packet Size on Packet Loss (JIWYNET)
4.3 Video compression

From the previous results, a packet size of 258 bytes allows images sent at a frame rate of 10 frames/sec to be received and displayed at approximately 10 frames/sec. The measured rate was actually 9.5 frames/sec. At this rate the bandwidth used by the JIWYNET application is 7.7 Mbps. This is a relatively large bandwidth and is due to the fact that the images that are being sent are not compressed. They are more than 100KB each, ie, 176x144 pixels at 32 bits per pixel or 101376 Bytes/Image. This is all right when dealing with a local network with small amounts of network traffic; however, networking applications that transfer multimedia over a network must deal with the realities of network traffic and time delays. By using a simple compression with a 10:1 ratio, the bandwidth would be reduced by a factor of 10 to a very manageable 0.77 Mbps. In that case, the possibility of sending larger images would be achievable.
5.0 Conclusions and Recommendations

5.1 Conclusions

To date, the JIWYNET project has been successful. It uses the UDP transport layer protocol to transfer a live image stream captured by a webcam. The image server is written in C++ and runs in the user mode of the host RTLinux PC. The client side is written in JAVA. It enables a remote user to view the live video stream and send commands to the server about the camera’s position. The commands are received in the user mode of the server application, and then passed via a RT_FIFO [10] to the kernel mode. JIWY’s controller then processes the commands in order to move the camera appropriately.

When designing a networked multimedia application, many bottlenecks must be examined in order for the quickest and most reliable system to be developed. Through tests, it was found that a packet size of 258 bytes leads to the fastest transfer rate for the JIWYNET system on the local network. Also, this packet size has a loss rate of less than 0.1% which makes it relatively reliable. These tests were only conducted on the local network however, and different results should be gathered about how the system reacts to other network connections.

Nonetheless, JIWYNET provides the framework for further work with remotely controlled systems via computer networks.

5.2 Recommendations

The JIWYNET application has been coded under time restraints and most address and port information is hard-coded. In order to use the JIWYNET system easily from a variety of PCs, the user must be enabled to change the address information. An efficient way to implement this would be through the use of a GUI.

Compression was not implemented. It would be able to reduce the amount of information sent over the network. Not only would this speed up transfer times, allowing for more frames a second and a larger image, but it would also decrease the bandwidth utilized by the application.

The JIWYNET system is composed of many smaller pieces. Combining them to make a single server/client pair would allow for much easier usage. Also, GUIs could be implemented to further usability.

CamStream offers many tools that are not needed in the JIWYNET application. Stripping the CamStream application, removing all of its unused code would simplify the JIWYNET system and speed it up. There is no need to produce the images at the server side of the application.
Using the CSP/CT [4] ideas with the JIWYNET project could have many advantages. For instance, using channels, a plug and play type design could be implemented. Thus the desired hardware could be used at the client end to control the camera (ie, mouse, joystick, keyboard). Also, the networking capabilities could be carried over to other similar control systems, easing further research.

So far the JIWYNET system has not been tested sufficiently. Reactions to network traffic, processor time slicing, and RTLinux priority control have not been fully researched. Running tests to investigate JIWYNET’s reactions should be done to reliably predict future performance.

The control commands issued by the client side are sent to the server via the unreliable UDP protocol. In the future, implementing a TCP connection may be of use to make the commands completely reliable. Also, by creating a TCP connection while initialising the application, the client’s address could be passed to the server. This would relieve some of the hard coding and help make the system truly dynamic with respect to the user using a connection from any remote client.
References


   http://www.ecst.csuchico.edu/~beej/guide/net/html/

   http://www.rt.el.utwente.nl/javapp/

   http://www.embedded.com/story/OEG20010529S0127


   http://alpha.dyndns.org/ov511/v4l_api.html


   http://www.isoc.org/isoc/whatis/conferences/inet/97/proceedings/F3/F3_1.HTM


    http://java.sun.com/j2se/1.3/docs/api/

    http://doc.trolltech.com/3.0/index.html

    http://www.smcc.demon.nl/camstream/


Appendix A - JIWYNET System Overview

The local or server side of the JIWYNET project includes the JIWY mechatronic system, the local PC running RTLinux, a joystick, a keyboard and the local monitor. There is a FIFO buffer between RTLinux and Linux that allows them to communicate. On the remote or client side the JVM allows the application to run regardless of the hardware platform and underlying operating system.
Appendix B - RGB 32 Image Format and Network Byte Order

The format of the images mapped to memory by the digital camera is RGB_32. That is, 32 bits of data per pixel. Each set of 32 bits contains 3 bytes that conceal the red, green and blue color information of the pixel. Then the remaining byte describes the luminance of the particular pixel with the alpha parameter. For example each pixel stored in memory has the configuration: 0xARARRGBBB.

In order to decrypt this information and produce an image at the client side, a JAVA ColorModel was utilized. The lines of code below create the ColorModel for the standard RGB_32 image format.

```java
import java.awt.image.ColorModel;

static final int BITS_PER_PIXEL = 32;
    static final int ALPHA_MASK = 0xFF000000;
    static final int RED_MASK   = 0x00FF0000;
    static final int GREEN_MASK = 0x0000FF00;
    static final int BLUE_MASK  = 0x000000FF;

    ColorModel cm = new DirectColorModel(BITS_PER_PIXEL, RED_MASK,
                                           GREEN_MASK, BLUE_MASK, ALPHA_MASK);
```

Most computers systems are byte addressable. Furthermore, computer architectures usually store integers in sets of four bytes. This allows for two ways to store integers in memory: Big Endian and Little Endian. Suppose the address of some integer is A. In the Big Endian format the highest order byte is stored at A and the lowest order byte is stored at address A+3. In the Little Endian format address A stores the least significant byte and the most significant byte is at address A+3.

Computer networks work in Big Endian byte order while PC computers usually work in Little Endian byte order. Thus in order to correctly decrypt the image information, a color model with the following masks was utilized:

```java
    static final int ALPHA_MASK   = 0x000000FF;
    static final int RED_MASK     = 0x0000FF00;
    static final int GREEN_MASK   = 0x00FF0000;
    static final int BLUE_MASK    = 0xFF000000;
```

This is simply the reverse of the standard RGB_32 image format.
Appendix C - Video4Linux API

The Video4Linux API allows the CamStream software to access the webcam. The main C++ command used in the program is:

```c
ioctl( int file_decsription, int request, char *argp )
```

Using this command, the file descriptor would be to the webcam, the request would be the data or action that is wanted from the webcam, and argp is reference to an address in memory.

A list of important ioctl calls used by the CamStream software:

**VIDIOCGCAP** is an ioctl call used to obtain the capability information of a video device. The *struct video_capability* object is passed to the ioctl, and the following information is returned:

- **name[32]**  Name for this interface
- **type**  Type of interface
- **channels**  Number of radio/ tv channels if appropriate
- **audios**  Number of audio devices if appropriate
- **maxwidth**  Maximum capture width in pixels
- **maxheight**  Maximum capture height in pixels
- **minwidth**  Minimum capture width in pixels
- **minheight**  Minimum capture height in pixels

**VIDIOCGWIN** is an ioctl call that returns relevant information about window caption area parameters. The *struct video_window* object is passed to the, and the following information is returned:

- **x**  The X co-ordinate specified in X windows format.
- **y**  The Y co-ordinate specified in X windows format.
- **width**  The width of the image capture.
- **height**  The height of the image capture.
- **chromakey**  A host order RGB32 value for the chroma key.
- **flags**  Additional capture flags.
- **clips**  A list of clipping rectangles.
- **clipcount**  The number of clipping rectangles.

**VIDIOCGMBUF** is an ioctl call that determines the depth of the image frames stored in memory. The *struct video_mbuf* object is passed to the ioctl returning the size and number of frames to be stored in memory.
VIDIOCMCAPTURE is an ioctl call that instructs the video device to start capturing. The struct video_mmap object is passed to the ioctl with the following information:

frame  Frame number
format Type of image Palette
width  Image width
height Image height

VIDIOCSYNC is an ioctl call that waits for the current frame to be captured.
VIDIOCGBPCT is an ioctl call that gets the properties of the video devices’ images. The struct video_picture object is passed to the ioctl, and the following information is returned:

brightness Picture brightness
hue Picture hue (colour only)
colour Picture colour (colour only)
contrast Picture contrast
whiteness The whiteness (greyscale only)
depth The capture depth (may need to match the frame buffer depth)
palette Reports the palette that should be used for this image
Appendix D – Qt SocketNotifiers

Qt provides a QSocketNotifier class that makes it possible to write asynchronous socket-based code. This technique is used extensively within CamStream’s event-driven GUI and video capture framework.

QSocketNotifiers allow for the monitoring of sockets such as TCP or UPD sockets. They connect an activation signal to a slot. Thus a slot is a response event that corresponds to an activation event. There are three types of socket notifiers, namely: read, write and exception.

The code snippet below is an example from CamStream’s CvideoDevice Class.

```cpp
pImageSocket = new QSocketNotifier(CamFD, QSocketNotifier::Read, this);
connect(pImageSocket, SIGNAL(activated(int)), this, SLOT(LoadImage()));
```

The first line creates a QSocketNotifier that will listen to the CamFD file descriptor. The CamFD file descriptor is the socket to the webcam. The second line connects the QSocketNotifier, pImageSocket, to an activation signal and a slot response. Namely, when the activated() signal is processed, call the LoadImage() function.