DISTRIBUTED NETWORK COMMUNICATION FOR AN OLFACTORY ROBOT

NSF Summer Undergraduate Fellowship in Sensor Technologies
Jiong Shen (EECS) - University of California, Berkeley
Advisor: Professor Dan Lee

ABSTRACT

The olfactory robot combines a nose sensor with robotics to provide automated information gathering without human labor. To achieve this goal, however, the information the robot gathers must be transferred to a computer to be analyzed. One way to transfer the data is through a wireless network that provides mobility as well as high speed. For this project, a strong-arm processor machine, Netwinder, is used to send information from the olfactory robot. Then, the sending and receiving C++ object-oriented codes were designed to use UDP multicast send and receive with RTP-specific packets to form a distributed network. Multicast network provides an easy way to distribute information from one sender to multiple receivers by using a group IP address. Then to further improve the speed of the data transfers, shared memory is implemented to control the sending and receiving flow so that no bottlenecks occur in between. Moreover, the shared memory provides a way for the receiver computer to distribute its information among its internal processes. The result achieved was the continuous streaming of the JPEG image frames from the source to any computers listening on the multicast channel. The receiver could receive and pipe the images to multiple processes through the shared memory, creating a large two-level distributed network.

1. INTRODUCTION

With the recent blooming of the wireless Internet, many different robots and sensors are being adapted to the wireless world, making their domains of movements larger and their applications wider. One sensory robot that can adapt to the wireless Internet is the olfactory robot.

An old prototype of the olfactory robot would have cables attached behind it leading to a computer to which it could send information. However, the cables limited where the robot could go and the distance it could travel from the computer. Using the wireless Internet, the olfactory robot will be able to move more freely, following smells and transmitting real-time image, audio, and odor information back to the controller.

The purpose of this project is to make the olfactory robot transmit real-time JPEG image streams over a wireless network. However, there are several obstacles. First, a

piece of code is required that takes in the image data from the camera and compresses it into JPEG images. Second, a protocol and network set is needed to construct the packets for real-time transmission. Finally, a working operating system to transmit the data must be small enough to fit onboard the robot.

Luckily, both the first and last problems were solved by two seniors at University of Pennsylvania for their senior project under the direction of Professor Dan Lee. That project developed the code for packing the JPEG image and installed a stripped-down version of Linux on a Netwinder mini strong-arm processed computer to be used onboard the olfactory robot. The work to be done for this project therefore is the development of network protocol and transmission code. The codes for UDP and some shared memory are generally available from many sources. The main task is to piece together the available codes, adapt them as needed for this project, and write the RTP protocol handler for send and receive.

Several key decisions must be made before proceeding with this work. They include types of network, network protocols, and memory management.

Most available networks are broadcast networks, where one user broadcasts a message to the network and it gets bounced around until it reaches the receiver. For fast real-time data transmission and distributed networking, a multicast network is used. It relieves the burden on the sender's Internet traffic by sending only one copy of the data to a common address instead of sending it specifically to each individual receiver. This makes it possible for many receivers to get the information at the same time instead of there being a delay because the sender has to send the packets to different receivers.

There are several types of network protocol as well. TCP (transmission control protocol) is the type most commonly used for the Internet. However, TCP is not good for real-time data because for every packet sent an acknowledgment packet is returned, wasting bandwidth because old data that are lost or broken don't need to be resent. Therefore, UDP (user datagram protocol) is used. UDP is a simple one-way, non-secure protocol usually used to send real-time data. Because it does not have acknowledgment packets, it does not clog up the network with useless information and resends.

UDP itself is still not sufficient to achieve real-time information transmission, however. A protocol called RTP (real-time protocol) is built on top of UDP specifically for sending real-time data. RTP is a fairly new type of protocol used mostly for streaming real-time video/audio over the Internet. It attaches its own header to the data packets to specify time and other information. In this project, JPEG video stream was the only type of data being sent. Therefore, JPEG-specific headers are also added while the packets are made.

With so many frames of images captured per second, only one per second can be sent over the Internet because of the constraint on the network. Therefore, a bottleneck will develop between the time frames are captured and when they are sent. Shared memory is needed to solve this problem. When the image is captured, it is packed into

several RTP packets and stored into a shared memory. Then when it is time to send the next image, the sending code reads the shared memory, takes the latest image, and sends it. Old images are ignored, and the shared memory does not waste bandwidth sending old pictures. The process is the same on the receiving side, so that the viewer always has the most recent picture. With shared memory, the image frames can also be distributed inside the receiver computer. After the image is read in from the network, it is stored into another shared memory section. Multiple processes that need this image can access the shared memory at the same time and process it in their own ways.

Although this project involves only JPEG image streaming, adding other kinds of data should not be very difficult; the receiving and packaging would need to be modified. The code developed for this project was left in a form that will make it easy to add new data.

2. DISTRIBUTED NETWORK

2.1 Multicast Network

Multicast network code for UDP transmission was largely completed by Professor Dan Lee before this project started. The idea is to send one packet to a destined IP address, where everyone who wants that packet can receive it. As shown in Figures 1 and 2, broadcast network sends a copy of data to each individual computer where as the multicast network only sends one copy to an IP and everyone can listen on that IP.

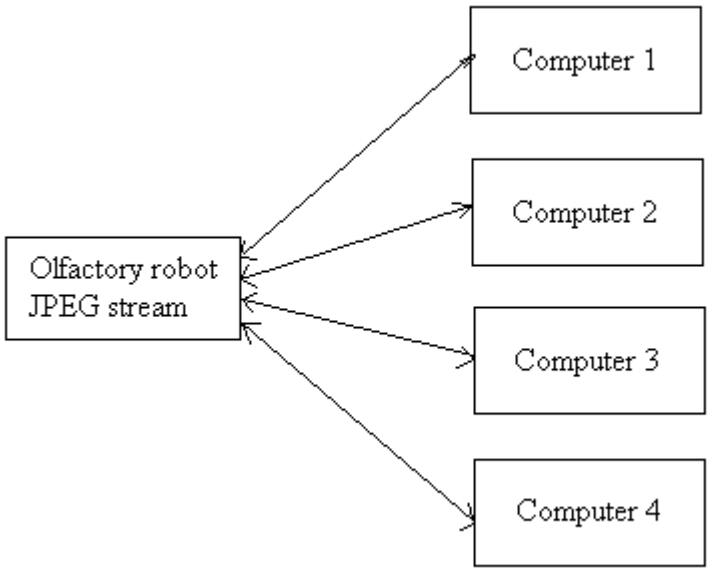

Figure 1: TCP broadcast transmission model.

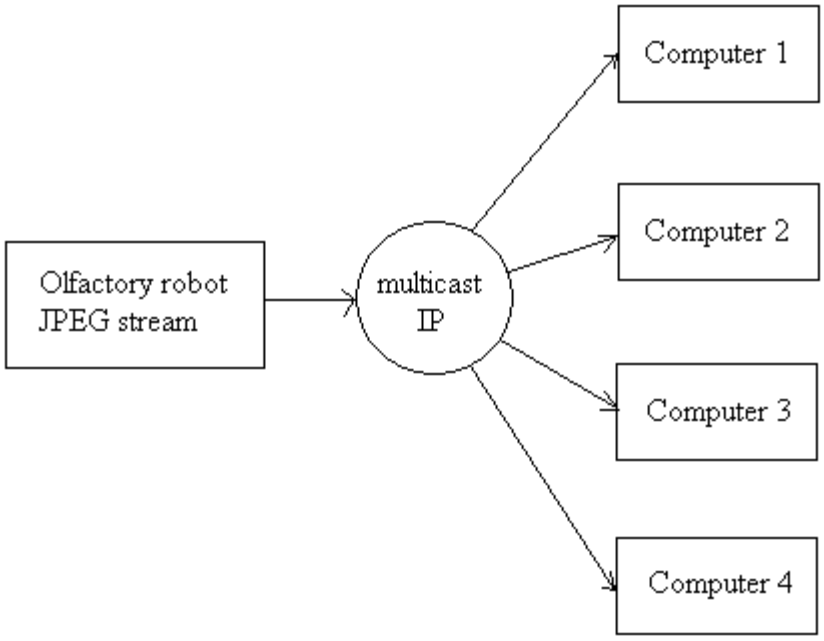

Figure 2: UDP multicast transmission model.

There are four different types of Internet IP addresses. They are characterized into four classes. Class A is used for a network that has many hosts on a single network. Class C is used for ones that have fewer hosts on the network. Class B is for the ones in the middle. Finally, the address used for this project is the multicast address, Class D. Class D has a restriction on the first four bits, which makes the first byte of the IP address having a value in between 224 to 240. Typical address used is 225.0.0.1.

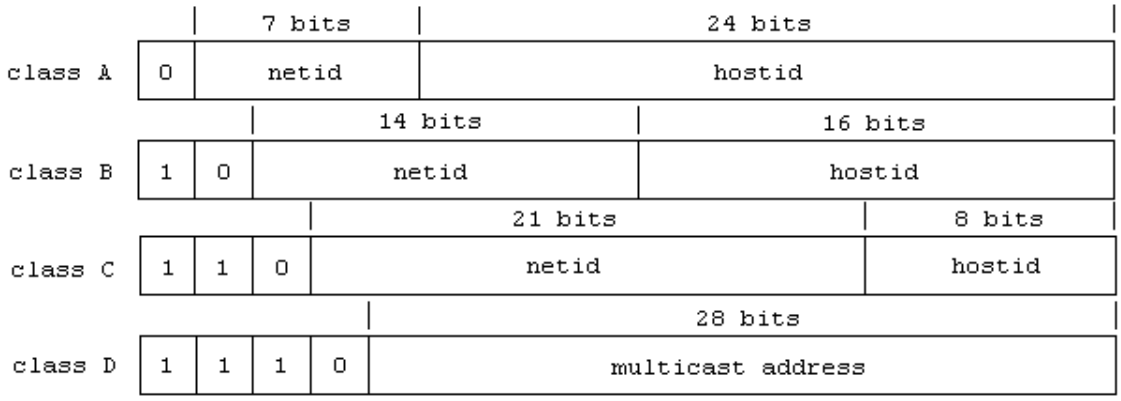

Figure 3: Internet Address Formats

2.2 Protocol

Network protocol makes a big difference in the speed and the reliability of data transmission. As mentioned earlier, two major network protocols are TCP and UDP (see Figures 1 and 2).

TCP is a secure network protocol with an acknowledgment system that ensures data being sent. It starts with a connection request; once the request is acknowledged, it sends the first packet. On the receiving side, when a packet is received TCP sends an acknowledgment packet back. When the sender receives acknowledgment of a set of packets, TCP sends a new set of packets. This process continues until the last packet is sent and acknowledged. However, for real-time streaming not every packet is needed. If the data is lost, it is by definition old and useless. Moreover, multicast for TCP is different. Each computer is distributing information to several computers at once and those computers distribute the information further. Therefore, using TCP will form a multilevel acknowledgment and make the network even slower.

By contrast, when UDP protocol is used with a multicast network, data is sent only to the predetermined IP address. When everything is done, the sender closes the connection. The receiver simply opens the connection to listen and get the packets being sent. Of course, many packets will be lost because a wireless network is very unreliable, but that is fine because they should not be resent. Therefore, UDP is clearly the better choice for this project.

2.2.1 UDP

User datagram protocol is a non-secure one-way transfer protocol. As explained earlier, UDP is good for the kind of streaming this project needs. To set up a multicast UDP connection, the address has to be read in from the command line. The type of address used was discussed above in multicast. Once the address is obtained, a socket is created. The socket is then set to the correct address with the appropriate options applied according to specific usage. Then connect is called on that socket. After connection is established, write will send any data over the network using the socket created. UDP has a default buffer size of 1408 bytes per packet, so data has to be broken down to smaller sizes. However, auto splitting of data cannot be used because each packet must have its own headers. Therefore, the raw data is split beforehand to match the buffer size of UDP.

2.2.2 RTP

RTP, real-time protocol, is built on top of UDP. As mentioned above, it is a specific protocol that ensures the real-time transfer of data by adding its special header to the front of the data packets. The header usually contains timing information, sequence number, type of data, and other flags. Each type of data has its own specific of the header and is also attached to the end of the RTP header. For JPEG, there is a JPEG specific header. The headers are packed into the raw data packets before sending and extracted

after being received. Then various checks are done to ensure the receipt of correct full packets.

2.2.2.1 RTP Header

The RTP header contains 16 bytes, as shown in Figure 3. The first 2 bits indicate the version of the RTP protocol used, which has the constant value of 2 for this project. P is padding; X is extension; CC is CSRC count. P, X, CC, and CSRC are not used for this project and so are left as zero. M is the flag for last packets. PT is the payload type, meaning the type of data this RTP packet is sending. For the purposes of this project, PT is set to the JPEG RTP type and fixed. Next is a 16-bit sequence number which can be used to check the continuation of the incoming packets. The next three 4-byte numbers are used for time stamping and some source information, but the SSRC and CSRC are also set to zero for the purposes of this project.

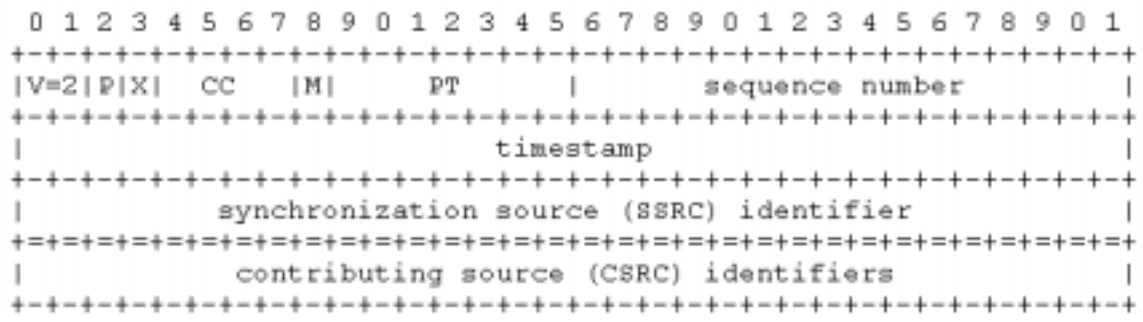

Figure 4: RTP packet header.

2.2.2.2 JPEG Header

A JPEG-specific RTP packet has its own small header as well, 8 bytes long (see Figure 4). The first byte, used for type-specific information, is zero for this project. Then, there are 3 bytes of frame offset, used to index where the data in the packet is located relative to the whole JPEG file. Usually, JPEG data is larger than one RTP packet, so it is cut into smaller chunks, each with an offset that gives the chunk's position. The next four bytes are type, quality, width, and height of the JPEG image. Type and quality are zeroed out for this project. Width and height are specified by the camera properties.

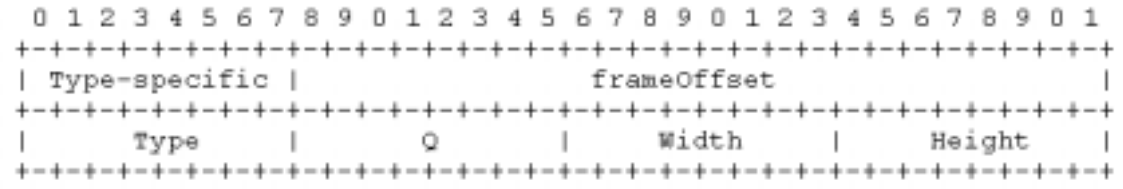

Figure 5: JPEG packet header for RTP.

3. DISTRIBUTED SYSTEM

3.1 Shared Memory

Shared memory, a way of saving memory and speeding up communication and processing, is an important part of operating systems. It is used in this project to reduce the bottleneck between the network and the program that processes the information. It also helps the receiving computer distribute the data to many processes at once. Figure 6 shows how the shared memory works on the sender side. JPEG packets are written into the shared memory, and UDP takes the packets and sends them. Figure 7 shows the shared memory for the receiving side. Together the two figures show how distributed computing is achieved through out the whole system.

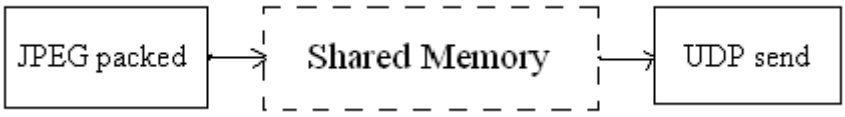

Figure 6: Sender shared memory.

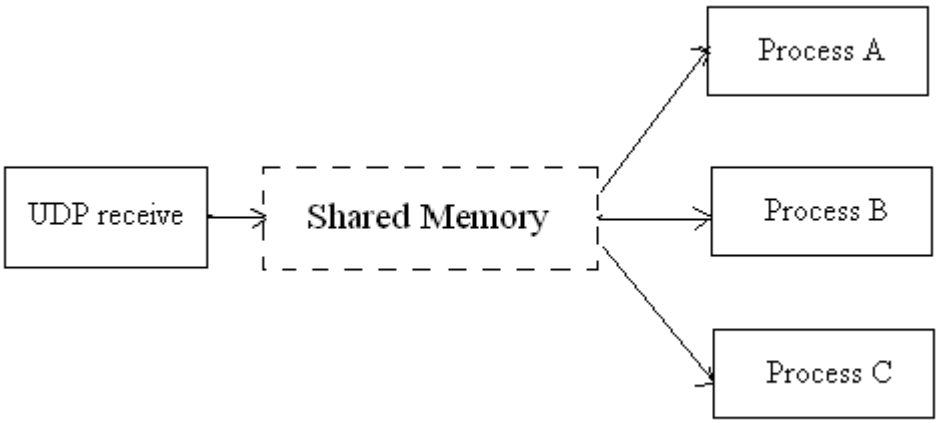

Figure 7: Receiver shared memory.

3.1.1 Circular Queue

Inside shared memory, the data is structured in a pseudo circular queue fashion. As illustrated in Figure 8, the first byte of the queue stores the section number for the most recent sets of packets stored in the shared memory. Then there are a set number of sections, each with four bytes for length and about nine kilobytes for data. The memory is filled from the top down until all sections are taken, then it starts to fill from the top down again. The first byte is changed after a section of data is filled to ensure that the latest packets are available at all times.

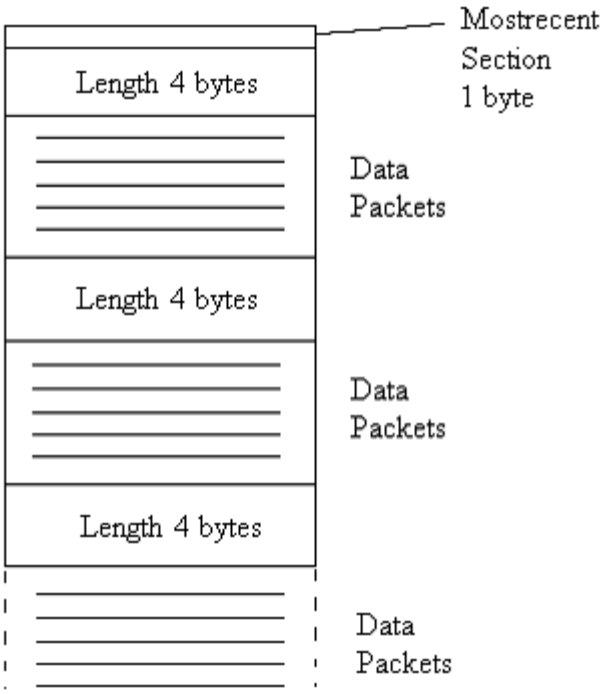

Figure 8: Circular queue data structure.

Results

The JPEG compression code produces JPEG images at a speed of ten per second. Therefore, by adjusting the speed of UDP send, we can simulate the situation of the code under different reliability of the network. Here is a plot made from several tests.

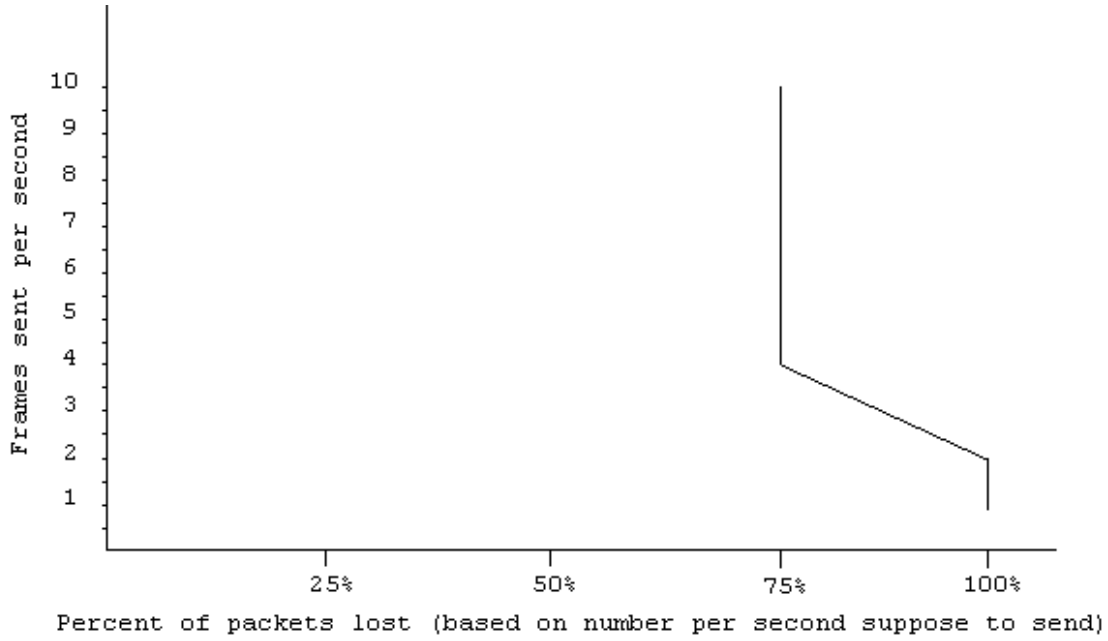

Figure 9: Speed vs. Percent of sent data

More tests have been conducted to test the reliability change when more than one process on the receiving side read from the shared memory. When tested with one frame per second, up to four processes read from the shared memory without any errors. When tested with two frames per second, everything was correct as well. However, when sending at four frames per second with two processes reading, six packets were sent totally but one process only read four while everything was read by the other process.

4. DISCUSSION AND CONCLUSIONS

From the results gathered, the performance of the UDP send dropped as the speed of sending increases. It is mostly due to the time spent reading from the output of the JPEG image code and packing and writing of the data packets. However, shown on the graph, the packets are still sent at 75 percent when its sending at the same speed as the capturing of the images, meaning the code still functioned when the limits are pushed.

There are many things that can be done after this project. This project tries to combine a wireless network with robotics to increase the mobility and usage of robots. After the project successfully works out, other data can be attached on top and infinite amount of possibilities are open to research. Like many others have done, wireless robots and sensors already start to appear everywhere in the industry and research labs. Human lifestyles will be changed by them soon.

5. RECOMMENDATIONS

For future development of this project, audio and nose sensor information would be added to the RTP packet lists and be processed on receiver side. A new packing and

unpacking function must be made for the new data and parsing has to be done before sending and after receiving. However, this will not be difficult because RTP packing is very modular and is done with object oriented C++ classes. Therefore, the new data will inherent the parent class and add its own small header. Furthermore, more performance checks need to be done and any kind of bottlenecks and deadlock must be reduced because real time is the key of this project and any delay can cause serious problems. Due to the timing of this project, extensive tests have not been performed.

An extensive example of how to implement audio streams will be provided in Appendices A.

6. ACKNOWLEDGMENTS

I would like to thank Professor Dan Lee for his assistance and support on this project, without which my work this summer could not have been accomplished. I would also like to thank the National Science Foundation for supporting this project through the REU program.

7. REFERENCES

1. W. Richard Stevens, *UNIX Network Programming*, Prentice Hall, Englewood Cliffs, N.J., 1990.

2. "RTP: About RTP and the Audio-Video Transport Working Group" Available from: http://www.cs.columbia.edu/~hgs/rtp/ Accessed: 29 May 2002.

3. "How to Write a Makefile" Available from: http://cs.gmu.edu/~tmaddox/cs211/sample_code/makefiles/make.html Accessed: 15 July 2002.

APPENDIX A

Implementing Audio Stream

First of all, code for the audio stream has to be written. Then the code to write audio packets to shared memory has to be written. The format of this write to shared memory code is similar to the JPEG one, with only a few exceptions.

Inside rtp.h (code for packet headers), a new class AUDIO needs to be made to inherent from RTP class and has its own header information specified as standard in a pack and unpack function. Then within write to shared memory, the incoming data has to be split and made into AUDIO class packets just like JPEG data is broken into JPEG packets.

```
while there is enough for a full size packet {
  //make new packet with the next part of data
  audio = new AUDIO(whatever parameters);
  //pack the data and put it in shared memory
  audio->pack(address the packed date to be copied into);
  seq++;  //increment seq number
}
//make new packet with the rest of the data
audio = new AUDIO(whatever parameters for the rest of data);
//pack the data and put it in shared memory
audio->pack(address the packed date to be copied into);
seq++;  //increment seq
```

The reading from shared memory code is the same because it sends whatever is in the shared memory.

On the receiver side, the writing to shared memory is the same, because it just dumps the packets into shared memory on the receiving side. The reading of the shared memory will be specific for audio. It will read in the most recent packet and check the payload type, if it is used for audio, it will be extracted.

```
switch (p->datatype()) {  //get data type
case RTP_PT_AUDIO:  //predefined value of audio rtp payload
  if (p->checkm()) {  //check if last packet
```

This switch statement checks for the type of data and only extracts it if it is the right type.

Other than those mentioned changes, no changes for the current setup are needed. Therefore, It is really easy to implement extra data streams.