Lab4 Experiment Guide: Communication and Camera

Objectives

The purpose of this lab is understanding socket programming, implement a remote video streaming system using webcam on the Bone, and design a remote keyboard commander for beaglebone on PC.
Finally, you will use this part for teleoperation of the robot. You will see the webcam (which is attached on the robot) on the PC, and teleoperate arm.

- Learning basic socket programming
- Writing a code for remote commander
- Test streaming Webcam screen

Problem statement

Problem 4A.

Test server and client example using datagram socket(UDP).

Problem 4B.

Implement a remote keyboard commander on PC to teleoperate a robot manipulator using WiFi and datagram socket.

Problem 4C.

Test WebCam streaming via FFmpeg and Implement video functionality composed of

- Camera on beaglebone (capture from webcam and send video to network)
- Viewer on PC (Receive video from network and display video to user)

Backgrounds

1. Stream Socket (TCP Protocol)

https://en.wikipedia.org/wiki/Stream_socket

In computer operating systems, a stream socket is a type of interprocess communications socket or network socket which provides a connection-oriented, sequenced, and unique flow of data without record boundaries, with well-defined mechanisms for creating and destroying connections and for detecting errors.

What is Socket?

http://beej.us/guide/bgnet/html/multi/index.html

You hear talk of "sockets" all the time, and perhaps you are wondering just what they are exactly. Well,they're this: a way to speak to other programs using standard Unix file descriptors.

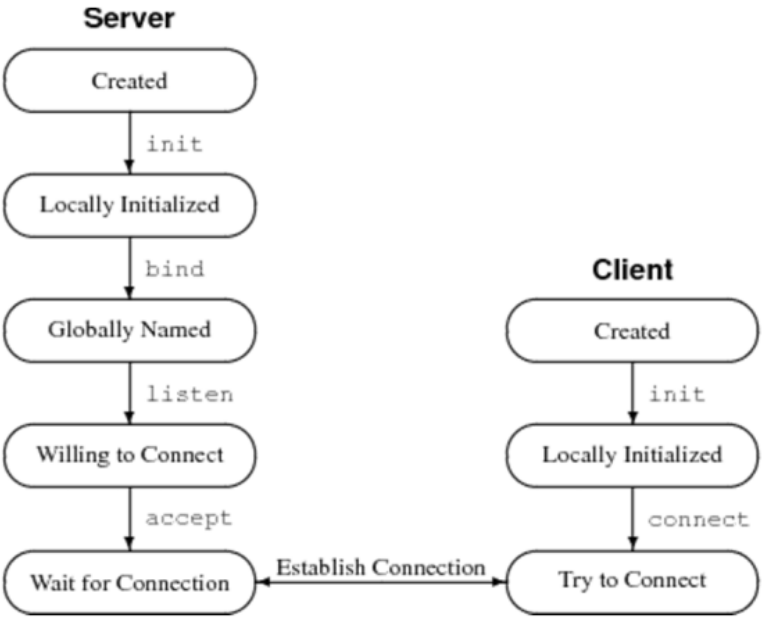
What?

Ok—you may have heard some Unix hacker state, "Jeez, everything in Unix is a file!" What that person may have been talking about is the fact that when Unix programs do any sort of I/O, they do it by reading or writing to a file descriptor. A file descriptor is simply an integer associated with an open file. But (and here's the catch), that file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in Unix is a file! So when you want to communicate with another program over the Internet you're gonna do it through a file descriptor, you'd better believe it.

"Where do I get this file descriptor for network communication, Mr. Smarty-Pants?" is probably the last question on your mind right now, but I'm going to answer it anyway: You make a call to the socket() system routine. It returns the socket descriptor, and you communicate through it using the specialized send() and recv() (man send, man recv) socket calls.

"But, hey!" you might be exclaiming right about now. "If it's a file descriptor, why in the name of Neptune can't I just use the normal read() and write() calls to communicate through the socket?" The short answer is, "You can!" The longer answer is, "You can, but send() and recv() offer much greater control over your data transmission."

Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order "1, 2", they will arrive in the order "1, 2" at the opposite end. They will also be error-free. I'm so certain, in fact, they will be error-free, that I'm just going to put my fingers in my ears and chant la la la la if anyone tries to claim otherwise.

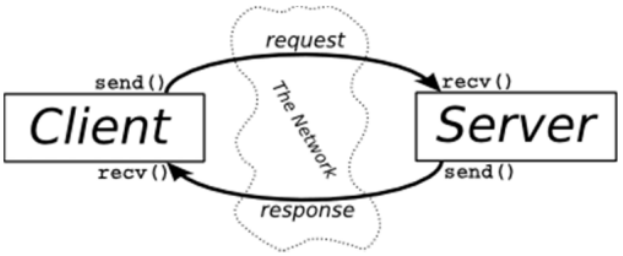


System calls for socket.

Client-Server background

It's a client-server world, baby. Just about everything on the network deals with client processes talking to server processes and vice-versa. Take telnet, for instance. When you connect to a remote host on port 23 with telnet (the client), a program on that host (called telnetd, the server) springs to life. It handles the incoming telnet connection, sets you up with a login prompt, etc.

The exchange of information between client and server is summarized as follows:



Client-server example:
See Section 6.1 & 6.2 in
https://beej.us/guide/bgnet/html/split/client-server-background.html .

2. Datagram Sockets

Datagram socket [http://en.wikipedia.org/wiki/Datagram_socket]
A datagram socket is a type of connectionless network socket, which is the point for sending or receiving of packet delivery services. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may arrive in any order and might not arrive at the receiving computer.
UDP broadcasts sends are always enabled on a datagram socket. In order to receive broadcast packets, a datagram socket should be bound to the wildcard address. Broadcast packets may also be received when a datagram socket is bound to a more specific address.

Two Types of Internet Sockets
http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html#twotypes
Datagram sockets also use IP for routing, but they don't use TCP; they use the "User Datagram Protocol", or "UDP" (see RFC 768.)
Why are they connectionless? Well, basically, it's because you don't have to maintain an open connection as you do with stream sockets. You just build a packet, slap an IP header on it with destination information, and send it out. No connection needed. They are generally used either when a TCP stack is unavailable or when a few dropped packets here and there don't mean the end of the Universe. Sample applications: tftp (trivial file transfer protocol, a little brother to FTP), dhcpcd (a DHCP client), multiplayer games, streaming audio, video conferencing, etc.

"Wait a minute! tftp and dhcpcd are used to transfer binary applications from one host to another! Data can't be lost if you expect the application to work when it arrives! What kind of dark magic is this?"

Well, my human friend, tftp and similar programs have their own protocol on top of UDP. For example, the tftp protocol says that for each packet that gets sent, the recipient has to send back a packet that says, "I got it!" (an "ACK" packet.) If the sender of the original packet gets no reply in, say, five seconds, he'll re-transmit the packet until he finally gets an ACK. This acknowledgment procedure is very important when implementing reliable SOCK_DGRAM applications.

For unreliable applications like games, audio, or video, you just ignore the dropped packets, or perhaps try to cleverly compensate for them. (Quake players will know the manifestation this effect by the technical term: accursed lag . The word "accursed", in this case, represents any extremely profane utterance.)

Why would you use an unreliable underlying protocol? Two reasons: speed and speed. It's way faster to fire-and-forget than it is to keep track of what has arrived safely and make sure it's in order and all that. If you're sending chat messages, TCP is great; if you're sending 40 positional updates per second of the players in the world, maybe it doesn't matter so much if one or two get dropped, and UDP is a good choice.

What's the difference between sockets (stream) vs sockets (datagrams)? Why use one over the other? What's the difference between streams and datagrams in network programming? - Stack Overflow

A stream socket is like a phone call -- one side places the call, the other answers, you say hello to each other (SYN/ACK in TCP), and then you exchange information. Once you are done, you say goodbye (FIN/ACK in TCP). If one side doesn't hear a goodbye, they will

usually call the other back since this is an unexpected event; usually the client will reconnect to the server. There is a guarantee that data will not arrive in a different order than you sent it, and there is a reasonable guarantee that data will not be damaged.

A datagram socket is like passing a note in class. Consider the case where you are not directly next to the person you are passing the note to; the note will travel from person to person. It may not reach its destination, and it may be modified by the time it gets there. If you pass two notes to the same person, they may arrive in an order you didn't intend, since the route the notes take through the classroom may not be the same, one person might not pass a note as fast as another, etc.

So you use a stream socket when having information in order and intact is important. File transfer protocols are a good example here. You don't want to download some file with its contents randomly shuffled around and damaged! You'd use a datagram socket when order is less important than timely delivery (think VoIP or game protocols), when you don't want the higher overhead of a stream (this is why DNS is primarily a datagram protocol, so that servers can respond to many, many requests at once very quickly), or when you don't care too much if the data ever reaches its destination.

To expand on the VoIP/game case, such protocols include their own data-ordering mechanism. But if one packet is damaged or lost, you don't want to wait on the stream protocol (usually TCP) to issue a re-send request -- you need to recover quickly. TCP can take up to some number of minutes to recover, and for real- time protocols like gaming or VoIP even three seconds may be unacceptable! Using a datagram protocol like UDP allows the software to recover from such an event extremely quickly, by simply ignoring the lost data or re-requesting it sooner than TCP would.

VoIP is a good candidate for simply ignoring the lost data -- one party would just hear a short gap, similar to what happens when talking to someone on a cell phone when they have poor reception. Gaming protocols are often a little more complex, but the actions taken will usually be to either ignore the missing data (if subsequently-received data supercedes the data that was lost), re-request the missing data, or request a complete state update to ensure that the client's state is in sync with the server's.

- **System calls for datagram socket**

  Datagram socket calls:
  sendto()
  recvfrom()

  See Section 5.8 in https://beej.us/guide/bgnet/html/split/system-calls-or-bust.html#sendtorecv.

- **Datagram Socket Programming example**

  See Section 6.3 lsitener.c and talker.c in
  https://beej.us/guide/bgnet/html/split/client-server-background.html#datagram.

3. Test Capture WebCam on Bone

Video for Linux

[“Beaglebone Images, Video and OpenCV, http://derekmolloy.ie/beaglebone-images-video-and-opencv/]

Using v4l2 (Video for Linux version 2), you can capture images and get camera informations. Video4Linux or V4L is a video capture application programming interface for Linux, supporting many USB webcams, TV tuners, and other devices. Video4Linux is closely integrated with the Linux kernel.

V4L2 Commands

Getting help

```
# v4l2-ctl --help
```

3Show driver info: -D, --info | show driver info [VIDIOC_QUERYCAP]

```
# v4l2-ctl -D
```

Video capture info

```
# v4l2-ctl --help-vidcap
```

List supported video formats

```
# v4l2-ctl --list-formats
```

Notice that C110 supports two formats: YUYV and MJPG.

Check current video format

```
# v4l2-ctl -V
```

Set video format

```
-v, --set-fmt-video=width=<w>,height=<h>,pixelformat=<f>
```

set the video capture format [VIDEOC_S_FMT] pixelformat is either the format index as reported by --list-formats, or the fourcc value as a string

To YUYV:

```
# v4l2-ctl --set-fmt-video=width=640,height=480,pixelformat=0
```

To MJPG:

```
# v4l2-ctl --set-fmt-video=width=640,height=480,pixelformat=1
```

Capture.c

https://github.com/derekmolloy/boneCV

Download to capture.c. This is an simple example to capture from WebCam using V4L2. You will use this program to streaming video with some modifications.

FFmpeg

FFmpeg is the leading multimedia framework, able to decode, encode, transcode, mux, demux, stream, filter and play pretty much anything that humans and machines have created. It supports the most obscure ancient formats up to the cutting edge. No matter if they were designed by some standards committee, the community or a corporation. It is also highly portable: FFmpeg compiles, runs, and passes our testing infrastructure FATE across Linux, Mac OS X, Microsoft Windows, the BSDs, Solaris, etc. under a wide variety of build environments, machine architectures, and configurations.

Specially, we will use ffmpeg for streaming service. At the Bone, FFmpeg takes WebCam screen as input via *capture.c* file, and stream to the UDP network. At the PC, receive data from UDP network and show the WebCam video via FFplay(included in the FFmpeg).

Preparation

List of components

- Beagle Bone
- Linux PC
- Ethernet cable
- WebCam (Logitech C270 in the lab)

Lab Procedures

- See the documentation “EE405 Lab5 Programming Guide.”

Final Report

Discussion for the following questions should be included in the report

1. Compare TCP and UDP. Explain with examples which method is suitable for which situation.
2. We will stream the WebCam video via UDP network. Why UDP is prefer method for streaming service? What if TCP network is chosen?
3. Choose your own discussion topic related to Lab 4 and provide an answer to it.

References

[1] Beej's Guide to Network Programming, https://beej.us/guide/bgnet/html/split/index.html

[2] Capture program for Beaglebone, https://github.com/derekmolloy/boneCV