Project 5: Remote Control Of Pocket PCs

In this lab you will integrate your video and audio work from the previous two PPC labs, and you will make it possible to control one PPC from another. Each PPC will function, simultaneously, in a client and a server role. Specifically, from a client PPC, you'll be able to talk to a server PPC and say:

- Please start/stop streaming me video of what your camera sees
- Please start/stop streaming me audio from your microphone

It should be possible in your system for a PPC to stream video to multiple PPCs simultaneously, and for a PPC to display video coming in from multiple PPCs simultaneously. You do not need to support simultaneous audio playback from multiple PPCs, but you should be able to stream audio to multiple PPCs simultaneously.

You want to think ahead as you do this project. In the next two labs, you'll be extending remote control to the motes, and integrating mote audio and other sensor readings into your system.

At least for this lab, the distributed system does not have naming or discovery services.

We will loan each group an additional PPC.

Before you start

You will be using the following components and features of the code base:

- Messages and MessageQueues.
- Audio component (sound input and output)
- Video component (video input and output)
- Communication component (UDP and Bluetooth connectivity)

Getting and installing the class code base

As before, you'll need to update your code base:

```
cd ~/real-time/code/ppc
svn update
```

Adding your project and building

Similar to the last project, you will want to add a new project, "client-server", to your workspace. It should be dependent on all of the component libraries. There are a fairly large number of include paths, library paths, and libraries that you will need to incorporate. You should copy these from example_component.

As with the last project, you are creating a simple hello-world application, and you'll want to add #includes for all the components. Note that you will be using "comm2".

As before, all communication in this project is via UDP packets sent to port 5199. The communication component takes care of this. The address you'll supply for messages is the IP address of the destination PPC.

How will this work?

The user interface of this program is relatively straightforward. You'll be using a menu to initiate the following actions:

- Start/stop audio from PPC – If you're currently listening to audio from some PPC, this will stop it. If not, it will open a dialog box asking you which PPC to start streaming audio from. You'll start playing that stream as it comes in.
- Start video from PPC – This will open a dialog box asking you which PPC to start streaming video from. You'll start playing that stream as it comes in.
- Stop video from PPC – This will open a dialog box listing current video streams and let you choose one to stop.
- Restart PPC – This will show a dialog box with remote PPCs and let you send am message to one that will restart the instance of your application running on it.
- Heartbeat PPC – This will show a dialog box with remote PPCs and let you send a heartbeat message to it.

The main part of the interface will show video from source PPCs, either tiled onto the display or with some mechanism to switch from one stream to the next.

Details of protocol

You're already familiar with the audio streaming protocol from the walkie-talkie lab. The video streaming protocol is exactly the same. However, note that a video frame may not fit into a UDP packet (64K-packet headers). Also note that even if you send a <64K UDP packet, it will be fragmented across multiple 802.11 packets (~1500 bytes). This means you have two basic options:

- You can downsample the video frame so that its corresponding PPCPictureOrVideoFragmentMessage fits into a UDP packet (or smaller). This is straightforward to do.
- You can fragment the frame across multiple PPCPictureOrVideoFragmentMessages. Note that the structure already has fields to support this. This is a bit more difficult to do, but notice that the video component will happily draw a fragment appropriately.

You should note that the menu-selected operations described above correspond to the PPC control messages listed in messages.h:

- Audio and video samples are to be sent using PPCAudioMEssage and PPCPictureOrVideoFragmentMessage. These simply stream and are played on arrival.

- Start/stop audio+video are done with PPCControlMessages. These are request/response. You send a request. You get a reply. Then the stream starts or stops.
- Heartbeats are HeartBeatMessages. You send a request heartbeat (see below) and expect a replay heartbeat within a window of time.
- Restarts are RestartMessages. You send the request. There is no reply.

A PPCControlMessage is a ControlMessage which is a Message. Here are the particularly relevant fields that you haven't seen before:

- response_deadline – the time by which you expect a response to be returned
- reqresp – zero if the message is a request, nonzero if it is a reply
- optype – the type of operation. For example, PPC_CONTROL_START_AUDIO. See message_types.h for more
- success – set to nonzero in a response if the request was successfully carried out. Set to zero if not.
- target – If you're requesting that a stream be initiated, this is the destination. Note that this means you from PPC A you can tell PPC B to stream to PPC C.
- until – If you're requesting a stream to be initiated, this tells the streaming system when it can stop, even if it never gets an explicit request to stop the stream.

UDP is unreliable. Your message may get dropped or it may arrive out of order (we use the 802.11 and UDP checksums so it is highly unlike that the packet arrives corrupted). To deal with the first issue, our protocol makes use of two elements:

- The requests are idempotent. If two or more identical requests are received, the effect is as if only a single request was received.
- The responses have deadlines. If we send a request and do not receive a response before the deadline, we will send the request again.

You may ignore the second issue. If we send start+stop+start+stop and it is received as stop+start+stop+start, we are allowed to rely on the user to send an addition stop, for example. You are welcome to be a bit more clever if you like. For example, you could make use of the seqno field of Message to determine when Messages arrive out of order and simply ignore the ones that do. You could also use the deadline field of Message to discard Messages that arrive too late.

Implementation approaches

You can implement this project using a variety of approaches. A simple approach would be to make your code entirely event driven. If you think about it, everything happens here in response to an event, the existence of which is conveyed to you as a windows message. However, there is one kind of event that you do not get explicit notification for, and that is the passage of time. You can get notification of the passage of an interval of time in essentially one of two ways:

- You can use the Windows SetTimer(). SetTimer lets you say "After t seconds have passed, send a WM_TIMER message to this window (or call that procedure)".

- You can modify the message loop in WinMain. Instead of calling GetMessage, you can call PeekMessage. PeekMessage does not block. Your message loop can thus note the passage of time even when there are no messages to be processed.

Why do you need to know about the passage of time? Because you will need to resend requests for which responses have not arrived yet. Conceptually, from your GUI, you'll queue up a request to be sent. The idea is that we want to resend the request, at intervals appropriate to its response deadline, until we get a response. The passage of time initiates the resend.

Another way to implement this is to have a separate thread whose sole responsibility is to resend outstanding request messages until their corresponding response messages arrive. If you have a separate thread, it can simply Sleep until the next time it needs to resend.

Note that you will also need to match requests with incoming responses. You can also match requests with subsequent requests. For example, if the user starts video, and then stops it before the video response has arrived, you could just eliminate the request from the queue.

Once again, do not get stuck on the GUI elements. We'd be happy to help you with dialog boxes, menus, etc.

For the ambitious (Extra Credit)

Add a simple discovery service. The discovery protocol is that PPCs periodically send PPCAnnounce messages to the broadcast address (255.255.255.255). All other PPCs on the subnet get these messages. You can use the announcements to build up a dynamic list of available PPCs.

Integrate the Bluetooth mote audio streaming from the previous lab.