Quick Guide to Aibo Networking

15-491: CMRoboBits

1 Introduction

Communication between the robots is accomplished by sending Aperios messages between them. These messages pass through a proxy running on a Linux workstation and are forwarded to the other robot. This forwarding by proxy is transparent on both ends; the messages are indestinguishable from other messages arriving from other objects such as Motion.

Sending or receiving data requires changes on many levels starting from behaviors, moving down to an Aperios level in MainObject, and over connections between objects and the proxies via the TCPGateway object before ascending through the same layers.

2 Passing information to MainObject from behaviors

Getting your messages or data to MainObject is as simple as adding a function call to the MainObject class. This class is defined in agent/Main/MainObject.h and agent/Main/MainObject.cc.

The only unusual part is retrieving a copy of the MainObject that is in use by the program. It turns out that the way MainObject is defined only allows a single instance of the class to exist. You can retrieve a copy of this instance by calling a static function which will return a reference to the single instance. This looks like:

- MainObject::GetObject().someMethodName(arguements)

The above calls someMethodName and passes it the arguements that you specify. You will write your own method that takes the arguements pass to it from behaviors and sends that data out as an Aperios message.

3 Sending Aperios Messages from MainObject

The function MainObject::SendStatusAndRoleMsgs() is an example of a simple function that sends an Aperios message. It is unusual because it sends queued data instead of data that is passed as an arguement, but otherwise it calls the

OPEN-R functions SetData and NotifyObservers like any other function would to send a message.

Let's examine a stripped down function that only sends a single number as its message. It would look something like this:

- void MainObject::SendDouble(double val) {
- subject[sbjDoubleOutput]->SetData(&val, sizeof(val));
- subject[sbjDoubleOutput]->NotifyObservers();
- }

Recall that messages in Aperios pass between a Subject (or sender) and an Observer (the receiver). Every Aperios object (which is what MainObject is) has an array of subjects and observers specific to that object. Aperios maintains message queues and routes messages between them.

In the above example, we access a Subject (OSubject is another class - the subject array contains pointers to these objects). We call OSubject::SetData, which copies data into the queue of messages going out over the channel connected to that subject. Immediately after this, we call NotifyObservers. This tells Aperios to actually send the data to the waiting observer (OObserver is another class. The "Level 2 Reference" that Sony provides with their OPEN-R SDK contains a list of classes and their methods).

4 Receiving Ready Messages in MainObject

There is one more addition we need to make to MainObject before setting up our communication channels. Part of what Aperios does is queue messages and arrange for flow control between the objects. Generally it is bad for subjects to "spam" their observers with data if the observer is not ready for new data. To facilitate this, observers send a special message to their subjects indicating that they are ready to receive data. Subjects can then forward more data to the observers after receiving this signal.

As an aside, subjects are free to ignore ready messages, which is what we tend to do. Waiting for a ready before each send increases latency. The flip side of this is that extra messages are dropped by the OS without warning you that they were dropped. Aperios maintains a small queue for each observer which can hold up to 16 (or possibly 32 - check the reference for a definitive answer) messages. (By default it is not set to the max value. The queueing takes place on the observer end and observers can call the SetBufCtrlParam function to increase the queue size) We'll cover how to do this when modifying MainObject to receive Aperios messages later.

Regardless of whether we pay attention to ready messages, we need to define a callback function in MainObject to receive them. This callback is a good place to send more data from if you do use ready messages to sync communications.

The method MainObject::NetOutput(const OReadyEvent &event) is an example of a callback that does nothing. Consult the reference if you want to access the fields of the OReadyEvent message (which has details about which observer is ready, since the same callback can service multiple ready messages).

You may add your own ready callback method or just recycle NetOutput (as it does nothing other than return).

5 Defining Subjects and Observers

We've been talking about subjects and observers but we haven't described how to actually define them. They are defined in a file called stub.cfg. Every Aperios object has a file like this. The file for MainObject is located in agent/Main. The file for the Motion object is located in agent/Motion. There is also a separate file for WLOut, etc. Every Aperios object should have a stub.cfg.

Stub.cfg is read by a preprocessor which creates C++ code that serves as the glue between objects. This is how the subject array gets filled in and where that array index sbjDoubleOutput came from.

We need to create a new subject. Let's add a line like this:

- Service : "MainObject.DoubleOutput.DOUBLE.S", null, NetOutput()

We also need to increment to NumOfSubject field because we just added a subject. The .S on the end of the string means that we are defining a subject. The MainObject, obviously, identifies which object we are adding the subject to. This must match the ObjectName. The second field, DoubleOutput, is the name of the subject. This defines what the index into the subject or observer arrays is called. In our case, this constant will be sbjDoubleOutput. Next comes the data type. This is just a string and has no official meaning. It must match the data type expected by the observer. Also, since messages are retrieved as pointers to a buffer, you need to track what you sent and how large it was on your own, so having this field is a good reminder telling you what is actually in the message. Finally, we tell the preprocessor to use NetOutput() as the callback when our observer tells us that it is ready.

Now we're going to add an observer. Here's the tricky bit: MainObject runs on every robot. So every copy of MainObject needs to have an Observer to receive the messages. So while we will not connect the subject running on dog A with the observer on dog A, we will connect A's subject to B's observer. It just happens that both functions are in MainObject and both are defined in the same stub.cfg file. But the same robot will not be sending messages to itself. (We'll take care of this arrangement in the next section, just bear with us for now)

Adding a subject is just like adding an observer. You need to specify the object name, the observer name, the data type, and .O because it's an observer instead of a subject. Finally, you need to specify a callback to receive the ONotify messages that are sent when messages arrive. ONotify messages actually

contain data, so they are different from ready messages in this regard. Let's define our subject:

- Service : "MainObject.DoubleInput.DOUBLE.O", null, DoubleInput()

Note that you never list arguements with function names in stub.cfg. It is not a C++ file - the ()'s just tell the preprocessor that the bit in front of them is the name of a function in the object.

Do not forget to increment the number of observers.

6 Connecting Subjects to Observers

Aperios is designed so that different objects can be easily mixed and matched. As a part of this philosophy, the connections between subjects and objects are not static. They are defined at runtime by a text file called connect.cfg. This file, unlike stub.cfg (which is only used at compile time), is needed on the memory stick. The stickit program will copy an image of the connect.cfg file in agent/config/open-r/mw/conf to the memory stick. Specifically, to the MS/open-r/mw/conf directory. The case of the path on the memory stick may vary - the memstick's FAT filesystems are not case sensitive.

Again, we will fill in both the subject and observer halves of things while we're editing this file. And again the same robot will not be talking to itself. It will instead connect to an object running on the robot called TCPGateway which will act as a proxy for the other robot. Behind the scenes, TCPGateway will forward the Aperios message via TCP/IP to *another* TCPGateway object running under Linux which will in turn forward it to a third TCPGateway object (this one on the second robot). This final TCPGateway sends the message to MainObject on the second robot.

This is a fairly roundabout way of doing things. You gain in flexibility because robots have the ability to send messages to objects running on the same robot, another robot, or a Linux workstation (using the remote processing API) using the exact same code. This is a huge win for developement because you could, for example, develop and debug a Vision object offboard (camera images are Aperios messages, after all...) and then move it to the robot once it has been tested.

And of course there is a flipside. The latency involved in copying the data between buffers, sending it between multiple objects on multiple hosts is fairly high. Round trip times of *seconds* have been seen under poor network conditions (lots of other traffic from third party hosts). If you need low latency, look up how to send TCP or UDP directly between robots. This is how we get our debugging information and video for Chokechain - the robot listens for incoming TCP connections from the Chokechain program and forwards data once Chokechain connects. This allows us to view full framerate RLE images on the workstation. We couldn't do this using messages, although more recent versions of the SDK have reduced latency some so perhaps it's possible now.

Since we're doing both the sending and receiving here, we'll add two connections:

- MainObject.DoubleOutput.DOUBLE.S TCPGateway.DoubleOut.DOUBLE.O
- TCPGateway.DoubleIn.DOUBLE.S MainObject.DoubleInput.DOUBLE.O

Notice that we assume TCPGateway has a subject called DoubleIn and an observer called DoubleOut. Which is odd because TCPGateway is a precompiled system object - we can't edit its stub.cfg. TCPGateway is special. It loads a list of its subjects and observers from a separate file on the memory stick. We'll describe that next. In the meantime, notice that TCPGateway.DoubleOut is an *observer*. Remember: MainObject sends data to TCPGateway which fowards it via TCP. Which is why TCPGatway needs to listen for messages from MainObject along the output channel for DoubleOutput. The same logic explains why TCPGateway.DoubleIn is a subject - it must forward incoming data to MainObject's observer.

The final step in this section is to create TCPGateway's subjects and observers. By doing this we will also define the TCP/IP endpoint that the Linux proxy will connect to by specifying a port number (in the TCP/IP sense. E.g. port 80 is what most web servers listen on). We do this in robotgw.cfg. This file may be found in: agent/config/open-r/mw/conf/ and also resides on the memory stick in the corresponding location. Add the lines:

- TCPGateway.DoubleOut.DOUBLE.O 2001
- TCPGateway.DoubleIn.DOUBLE.S 2002

The port numbers don't particularly matter. They must be unique and should be above 1024. An oddity here is that DoubleOut is declared as an observer here. This is because robotgw.cfg has more in common with connect.cfg than stub.cfg. The first line says: "Forwad everything that comes into TCPGateway over DoubleOut over the TCP stream connected to port 2001. And the second line says to forward data coming in over port 2002 out to whatever observer is listening to TCPGateway.DoubleIn.S.

7 Remember to Copy the Config Files

Remember to do a stickit -a or a stickit -c so that the relevant config files are actually copied to the stick. You'll need to do this again after you add code for an observer to MainObject, but we're saying it now because it's easy to forget that the config files need to be copied over onto the stick. We in fact recommend mounting the memory stick and examining the versions on there afterwards to ensure that they did in fact get copied; stickit only copies come of the config files. Others, such as camera.cfg and run.cfg are not copied. Make sure connect.cfg and robotgw.cfg are on the list of files that stickit does transfer.

8 Add an Observer to MainObject

Before we describe how to setup the Linux proxy, we'll describe how to receive the data in MainObject. And example function exists in the form of MainObject::GameMgrInput. However, we'll go through and create our own to show the general case.

- void MainObject::DoubleInput(const ONotifyEvent &event) {
- double buffer;
- // Aperios can deliver multiple messages in
- // a single notify.
- for(int i=0; i¡event.NumOfData(); i++) {
- buffer = *((double *)event.Data(i));
- // Do something with data, such as copy it to
- // a class variable so behaviors can retrieve it
- // later on.
- }
- observer[obsDoubleInput]->AssertReady();
- }

Notice that we handle more than one message in the same notify event. In practice, a second message is almost never present. But to be correct, you must perform this check. To get data back out of the notify event, we call Data(int) which returns data for the message number that we specify (virtually always message 0, since you almost never get more than one message and the first one has index 0). This returns a pointer. You must cast this pointer to whatever data type that you sent (in our case, it is a pointer to a double). Since it's an atomic type, we just dereference it and copy it into a separate buffer. The other examples in MainObject illustrate more complicated behavior and rely on data types defined in agent/headers/Wavelan.h. Finally, we call the OObserver::AssertReady method. This sends a message back to the subject connected to this particular observer to alert is that we received its data and have finishedd processing it. This call is what triggers the ready message. (We are not sure what behavior results if you do not assert ready. It is possible that Aperios will discard future messages once the queue fills up because you never say that it is safe to empty it. Or it might automatically discard messages delivered in a Notify. Check the Sony documentation or just always assert ready)

9 Setting up the Linux Proxy

The final step is to configure the copy of TCPGateway that will be running under Linux. All of the files needed are in dogs/util/InterRobotComm/RP/host/MS/OPEN-R/MW/CONF. They do not get copied to the memory stick; the odd directory structure mirrors a memory stick on the computer's harddrive, but no actual memory stick is involved.

Edit the files HOSTGW.CFG and CONNECT.CFG. HOSTGW is similar to robotgw on the memory stick with a few additions. It contains entries for *all* of the robots that will be running. It also contains the IP addresses of the robots as well as port numbers.

Comment out everything (using a # symbol) that does not apply to your robots' IP addresses. Some versions of the Linux proxy do not work if extra IP addresses are present because it tries (and fails) to connect to the extra robots instead of forwarding messages. Be sure to leave !ROBOT_PROXY entries for your robots.

You will add a pair of entries to HOSTGW for each robot that you are using. An example for two robots is below.

- TCPGateway.DoubleOutA.DOUBLE.S 2001 128.2.x.y
- TCPGateway.DoubleInA.DOUBLE.O 2002 128.2.x.y
- TCPGateway.DoubleOutB.DOUBLE.S 2001 128.2.x'.y'
- TCPGateway.DoubleInB.DOUBLE.O 2002 128.2.x'.y'

Notice that the port numbers match up with the ones that we used in robotgw.cfg. Again, the .S and .O classifications are non-intuitive. How come output is being associated with a subject? This is because we are forming a connection from (128.2.x.y, 2001) to a subject. Everything that comes in over the TCP connection gets translated into an Aperios message and fowarded via the subject to the left of the IP and port number. The same arguement applies to the observers. Everything that comes in to the observer is fowarded over the connection going to (128.2.x.y, 2002). Presumably the robot with IP 128.2.x.y is listening for connections on port 2002 and will do something with the messages that are fowarded.

The final bit of configuration takes place in CONNECT.CFG. Entries must be added to tie the subject on one robot to the observer on another. This looks like:

- TCPGateway.DoubleOutA.DOUBLE.S TCPGateway.DoubleInB.DOUBLE.O
- TCPGateway.DoubleOutB.DOUBLE.S TCPGateway.DoubleInA.DOUBLE.O

TCPGateway has an important limitation. You cannot send one copy of a message from the robot to multiple observers. You must call SetData with multiple subjects on the robot and have multiple channels. This is why we break out

communication between the robots into a series of entries such as FromAToB, FromAToC, etc. The same data gets sent each time. The proxy does not do multicast reliably. (In practice, it somewhat works, but odd race conditions exist with ready messages/etc which lead to robots silently becoming unable to send. They still hear their peers, but they will not send outgoing messages. Don't do it. It's extremely painful to figure out what is going on.)

10 Running the Proxy

10.1 Building RP-OPEN-R libraries (do this once)

The RP-OPEN-R objects do not come precompiled in the current version of the SDK. To correct this, run the script:
/usr/local/OPEN_R_SDK/RP_OPEN_R/bin/setup-rp-openr This should build the necessary objects. If it fails, it is most likely due to an incorrect version of GCC. We've had to either force old versions of the SDK to use GCC-2.95 or force new versions to alias GCC-3.3 as GCC-3.2. Lots of link errors has been the common indicator of compiler version mismatches.

10.2 Running the Proxy (the common case)

To actually run the proxy, follow these steps.

- Boot all of the robots and allow them to stand.
- Change to dogs/util/InterRobotComm/RP/host
- Run "start-rp-openr"
- You will see messages that the proxy has connected to your robots (or error messages saying that it cannot connect). Check your IP addresses and make sure that you can ping the robots to ensure that they are on the network.
- Once you are finished, stop the proxy by pressing CTRL-C.
- Run the command "rp-openr-ipcrm" This final step cleans up any resources that the proxy left around when it was terminated. (The proxy uses SYS-V IPC for communication and does not always clean up when terminated)