WURDE Robotics Middleware Programming Guide

Frederick Heckel

February 2, 2007

Contents

Chapter 1

Introduction

WURDE is Washington University's Robotics Development Environment. The WURDE robotics framework is designed to provide the simplest possible API for robotics programming. WURDE is capable of using different communication adaptors for transporting data between different processes; these adaptors are entirely hidden from the user. We believe that the roboticist should be able to request access to a device on a robot by simply declaring an object of its type. The programmer should not be required to know anything about the underlying communication protocol, or even whether the particular device is connected to the same computer.

1.1 Quick Start

1.1.1 Installation

The current version requires an installation of CMU IPC, Xerces-C, and libxml-dom-perl. Additional libraries used are OpenCV, dc1394, and FLTK. The configure script will search for all of these packages except for libxml-dom-perl and FLTK. DC1394 must be version 2.0.0pre7, OpenCV must be version 0.9.7(or compatible), and FLTK must be version 1.1 (or compatible).

- Unpack wurde and enter the directory
- Run *./configure*, using the appropriate options. *−−prefix=/usr/local* will install WURDE into /usr/local.
- Run *make*. WURDE should now build
- Run *make install*. WURDE is now installed.

Other options are available for the configure script: the must important are *−−with-docs*,*−−with-vision*, *−−with-firewire*, *−−with-apps*, and *−−with-ptu*. Using the PTU module requires additional files from Directed Perception; see modules/ptu/README for more information.

1.1.2 Simple Program

Using the framework is very simple; include the header files for all interfaces you wish to use, and also "CommsManager.H". Here's a very simple example which creates a server supplying the "Foo" interface.

```
1    #include <CommsManager.H>
2    #include <Foo.H>
3    #include <Logger.H>
4    #include <math.h>
```

```
5
6     int main (int argc, char *argv[]){
7             CommsManager myManager("FooServer");
8             Logger myLogger("FooServer");
9             Foo myFoo("Bar");
10            int i=0;
11
12            myManager.parseOptions(argc,argv);
13
14            myFoo.setQueueMode(true);
15
16            myManager.registerSupplier(&myFoo);
17            myManager.registerSupplier(&myLogger);
18            myManager.setSleep(0.2);
19
20            myFoo.info.maxRange.setValue(8.0);
21            myFoo.info.angleMin.setValue(0);
22            myFoo.info.angleMax.setValue(M_PI);
23            myFoo.publishInfo();
24
25            myFoo.data.ranges.push_back(1);
26            while(myManager.runUpdate()==STATE_RUN){
27                    myFoo.data.ranges[0]=i;
28                    myFoo.publishData();
29                    i++;
30            }
31
32            myManager.cleanUp();
33
34            return 0;
35    }
```

Lines 1-4 include the needed header files. CommsManager.H provides an abstraction around the available communication adaptors; it takes care of all adaptor initialization and manages the data channels of the interfaces used. Foo.H is our simple dummy interface, which looks a lot like a range finder sensor. Logger.H provides a logging service which utilizes syslog and network logging through the communication adaptors. math.h is needed in this example for the value of M_PI.

Lines 7-10 show the declaration of the objects which will be used. There are only three (non-trivial) objects. CommsManager is called with a string naming the process, in this case, "SomeServer". This name must be unique among the modules running at any given time. Logger is also called with "SomeServer"; while this is not required, it is recommended so that logging messages are tagged with a single process name. The Foo object is given "FooServer", which is the identifier which consumer interfaces will use to identify data from this supplier.

Terminology: A *server* in this document will refer to some process which sends data to be received by another process. A *client* will refer to some process which receives data from another process. Note that a process will often be both a client and a server.

Supplier will refer to a specific instance of an interface object which sends data to other processes. *Consumer* will refer to a specific instance of an interface object which receives data from another process.

Module will refer to any self-contained program which is involved in robot operation.

Line 12 uses the CommsManager to parse command line options defined by the framework. This should be included so that the "-r" switch can be used for all managed objects (see the section on the MCP for more information about this switch).

Line 14 sets an option in the myFoo object, turning on its data queue mode so that incoming requests will be queued up rather than overwriting old requests.

Lines 16-17 connect our objects to the communication channel, and registers it with the CommsManager, which takes care of updating it in the future. At the same time, they are each registered as suppliers of data; information which is placed in the "data" object of myFoo will now be sent out over the interface. For the info channel, "requests" will contain incoming data, while "info" will contain outgoing data. For the logger object, this states that logged strings will be sent out over the network.

Lines 20-23 set information about this Foo supplier, and line 24 marks the info as ready to be sent over the communication adaptor. Note that the data members of the info structure are not set directly– they must be set using accessor functions defined as part of the Writable template class.

Line 26 initializes the data provided by this Foo supplier; data.ranges is an STL vector of double values, so we use the standard accessors for the STL vector class. Lines 27 through 31 are the event loop of the program. At each step, the single range value in myFoo's data.ranges vector is updated with a new value, this new data is timestamped, and then myManager is called to take over. The CommsManager will first send out new data and information, then receive new requests, and finally sleep for a period before returning. Also at this time, the CommsManager will send a heartbeat event to the MCP, if it is being used. When the CommsManager receives a request to exit the program, the while loop will terminate.

Line 33 is the final line of the program; myManager.cleanUp() takes care of the steps needed to gracefully shut down.

Now we need a client to go with this server.

```
1   #include <CommsManager.H>
2   #include <Foo.H>
3   #include <Logger.H>
4
5   int main (int argc, char *argv[]){
6     CommsManager myManager("FooClient");
7     Logger myLogger("FooClient");
8     Foo myFoo("Bar");
9
10    myManager.parseOptions(argc,argv);
11
12    myFoo.setQueueMode(true);
13
14    myManager.registerSupplier(&myLogger);
15    myManager.registerConsumer(&myFoo);
16
17    while(myManager.runUpdate()==STATE_RUN){
18      while(myFoo.newInfo()){
```

```
19          myFoo.getNextInfo();
20          cout <<"Info: MaxRange "<< myFoo.info.maxRange.getValue()<<
21               " angleMin " << myFoo.info.angleMin.getValue()<<
22               " angleMax " << myFoo.info.angleMax.getValue()<< endl;
23        }
24
25        while(myFoo.newData()){
26          myFoo.getNextData();
27          if(myFoo.data.ranges.size()>0){
28              cout <<"Timestamp: "<<myFoo.data.timestamp.getSeconds()
29                   << " Val " << 0 << ": "
30                   << (double) myFoo.data.ranges[0] << endl;
31          }
32        }
33      }
34
35      myManager.cleanUp();
36
37      return 0;
38    }
```

The first thing to notice about the client program is that the first fifteen lines are very similar to the server code. The process name is now "FooClient" instead of "FooServer". The myFoo object in this program is meant to connect to the Foo supplier from the server program, so we call it with the same name. Also, instead of registering the myFoo object as a supplier, it is now a consumer. Line 12 causes the myFoo consumer to queue up received data events, instead of overwriting old events, so that we receive and process every one.

Lines 17-34are the main loop of the client. At each loop, the data is first updated by calling the manager's update function (this could also occur at the end; it merely is placed at the front in this example to make sure that if there is data available, we have it before the first run of the loop). The loop checks to see if there is new data by comparing the timestamps, then reads each value from the range data member. It also accesses the info structure to display the attributes of the object it has connected to.

Note the loops at lines 18 and 25. These will work whether the consumer is in queue mode or normal mode, and prevent processing if there is no new data or information. When in queue mode, it will process all new events.

1.1.3 Compiling WURDE Programs

First off, be sure that LD_LIBRARY_PATH contains the lib directory of the WURDE install. Also insure that the package config directory where wurde.pc is installed is present in PKG_CONFIG_PATH environment variable.

For the compilation flags, use *pkg-config −−cflags wurde*. For linking flags, call *pkg-config −−libs wurde*.

I recommend using Makefile and Makefile.common from the role/src/robot/modules/sickLaser directory for the build process. Just set the ROOT_DIR to the appropriate directory so it can find WURDE. Alternatively, Bill's Makefile system can be used– see the mcp directory for an example. I will clean this up in the future.

1.1.4 Running WURDE Programs

Before running a WURDE program, you should must be sure to start up any required servers for the communication adaptor. In the case of the CMU IPC adaptor, you must start the "central" server. It has a number of options, but the most important option is "-c". When run as "central -c", data connections will

be made directly between modules rather than sending data through the central server. Note that when run in this mode, you will not be able to monitor messages passing through the system.

1.1.5 Advanced Example

The next example will demonstrate how to use write for WURDE using the MCP, and covers requests in more detail. It is a slightly simplified version of the blindTeleop program included with the WURDE distribution, omitting only the sound and image controllers that are part of that program for the sake of remaining concise. The joystick code is also skipped.

What is the MCP? MCP stands for Master Control Program. In short, the MCP performs the task of module management, making it unnecessary to hard-code consumer/supplier pairs, isolating the programmer from systems-level details. It also assists at run-time, by making it unecessary to start up each individual module by hand. This is minor when only two or three modules are in use, but when particular robot applications require a chain of 5 or more modules, it becomes extremely helpful.

```
1   #include <iostream>
2   #include <GameController.H>
3   #include <CommsManager.H>
4   #include <Egomotion.H>
5   #include <ObstacleAvoider.H>
6   #include <PTUnit.H>
7
8   void configureJoystick(std::string config, GameController &joystick);
9
10  using namespace RobotObjects;
11  using namespace std;
12
13  int main(int argc, char *argv[]) {
14          GameController joystick;
15          CommsManager myManager("BlindTeleop");
16
17          myManager.parseOptions(argc,argv);
18
```

The first part of the program is very similar to the initial code from the Foo examples earlier. The only additions relate to the joystick.

```
19          Egomotion myEgo(STRAT\_ASSIGNED,"EgoClient");
20          ObstacleAvoider myAvoider(STRAT\_ASSIGNED,"AvoiderClient");
21          PTUnit myPTUnit(STRAT\_ASSIGNED, "PTUClient");
```

The object declaration lines are quite different, however. Now, instead of the name for the information stream, we have the value STRAT_ASSIGNED followed by an alias. This marks these objects as consumers, and tells the CommsManager that supplier assignments will come from the MCP.

STRAT_ASSIGNED is one of three possibilities, and the only one currently recommended for use. The other *connection strategies* are STRAT_NORMAL and STRAT_AUTO. STRAT_NORMAL makes use of the old hard-coded method of connecting consumers to suppliers, but also requests the MCP to start up the required module. Since this still requires hard-coding the supplier name (therefore requiring recompiles with system reconfiguration), it is not recommended. STRAT_AUTO is still in development, but in the future will ask the MCP to make a decision about what modules to start up.

The connection strategy used most often throughout the WURDE modules is STRAT_ASSIGNED. This strategy allows the user to provide a configuration file which specifies the connections, or *mappings* between consumers and suppliers. This file will be explained after the code.

```
22
23          myManager.registerConsumer(&myEgo);
24          myManager.registerConsumer(&myAvoider);
25          myManager.registerConsumer(&myPTUnit);
26
27          bool running=true,brakeon=true;
28          configureJoystick("joystick-config.xml",joystick);
29
30          Pose curr;
31          Point goal;
```

Despite the addition of the connection strategy parameter, the registration process remains the *same*. Here we're registering an Egomotion consumer for odometry data, an ObstacleAvoider consumer to move the robot, and a PTUnit consumer to gain access to the robot's pan-tilt unit.

```
32          while (running) {
33                 myManager.runUpdate();
34
35                 /* receive new messages */
36                 if(myEgo.newData()){
37                        myEgo.getNextData();
38                        curr=myEgo.data.location;
39                 }
40
41                 if(myAvoider.newInfo()){
42                        myAvoider.getNextInfo();
43                        if(myAvoider.info.brake.getValue()){
44                               brakeon=true;
45                        }else{
46                               brakeon=false;
47                        }
48                 }
49
50                 goal.x(0);
51                 goal.y(0);
52
53                 if(joystick.axis("translate")/32767.0 < -0.15){
54                        goal.x(joystick.axis("translate")/32767.0*2.0*-1);
55
56                        if(fabs(joystick.axis("rotate")/32767.0) > 0.1){
57                               goal.y(joystick.axis("rotate")/32767.0 * 1.5*-1);
58                        }
59                        myAvoider.requests.brake.setValue(false);
60                        myAvoider.moveToRelativePoint(goal);
61                 }else{
62                        if(fabs(joystick.axis("rotate")/32767.0) > 0.5){
63
64                               myAvoider.requests.brake.setValue(false);
65                               if(joystick.axis("rotate") > 0){
66                                      myAvoider.rotateToRelativeAngle(-M_PI/2);
67                                      goal.y(-3.14);
68                               }else{
```

```
69                                      myAvoider.rotateToRelativeAngle(M_PI/2);
70                                      goal.y(3.14);
71                              }
72                      }else{
73                              myAvoider.stop();
74                      }
75              }
76
77              if(fabs(joystick.axis("quicktilt"))/32767 > 0.05){
78                      myPTUnit.requests.tilt.setValue(joystick.axis("quicktilt")/32767.0*0.2);
79              }else{
80                      myPTUnit.requests.tilt.setValue(0);
81              }
82
83              if(fabs(joystick.axis("quickpan"))/32767 > 0.05){
84                      int segment=joystick.axis("quickpan")/32767.0 * 6.0;
85                      myPTUnit.requests.pan.setValue(segment*(32767.0/6.0)/32767*M_PI/4.0*-1);
86              }else{
87                      myPTUnit.requests.pan.setValue(0);
88              }
89
90              myPTUnit.publishRequest();
91
92              if(joystick.button("resetPTU")){
93                      myPTUnit.requests.pan.setValue(0);
94                      myPTUnit.requests.tilt.setValue(0);
95                      myPTUnit.publishRequest();
96              }
97
98              if(joystick.button("stop")){
99                      myAvoider.stop();
100             }
101
102             if(joystick.button("quit")){
103                     myAvoider.stop();
104                     running=false;
105             }
106
107             printf("\015");
108             printf("Location: (%lf,%lf,%lf)\t\tRelative Goal: (%lf,%lf)",curr.x(),curr.y(),curr.t
109
110             fflush(stdout);
111         }
112
113         myManager.cleanUp();
114
115         return 0;
116 }
117
```

There are several important points to notice in the remainder of the program. First is the use of requests; this is exemplified by the PTU requests on lines 92-96. Note that making requests through a consumer looks

identical to publishing info by the supplier. The Obstacle Avoider looks a little different in places. Note that we can call functions like *myAvoider.stop*() on line 73– the ObstacleAvoider is one of a small number of special classes which wrap around a RobotObject (in this case, an ObstacleAvoiderTransport object) to provide a nicer API. In the future, all objects will have accessor functions to read and write data from the data/info/request structures, but for now, there are only a small number of objects which have additional logic to make programming simpler. It is important to know whether you're using one of these objects when creating your mappings for the MCP.

Again, note that programming with the MCP is only slightly different from usual; the rest of the code remains the same. There is additional code in this program to deal with the joystick, but nothing different or unusual about the use of WURDE.

Next are the MCP files. The first important file is the mcp-config file.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <!DOCTYPE mcp-config SYSTEM "mcp-config.dtd">
4  <mcp-config>
5
6    <module name="VectorMover">
7      <binary>vectorMover</binary>
8    </module>
9
10   <module name="laserAvoider">
11     <binary>laserAvoider</binary>
12   </module>
13
14   <module name="HollowayLMS">
15     <binary>sickServer</binary>
16     <options>-n HollowayLMS</options>
17   </module>
18
19   <module name="safePTUserver">
20     <binary>safePTUserver</binary>
21   </module>
22
23   <module name="HollowayRFlex">
24     <binary>rFlexServerII</binary>
25     <options>-n HollowayRFlex</options>
26   </module>
27
28 </mcp-config>
```

This file merely lists the names and binary locations of the various modules which the MCP is allowed to use. We have five modules available which we will use. The next important file is the mcp-mappings file.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <!DOCTYPE mcp-mappings SYSTEM "mcp-mappings.dtd">
4  <mcp-mappings>
5
6          <connect module="VectorMover">
7                  <input interface="SynchroDrive" source="HollowayRFlexDrive" object="DriveClient" />
8                  <input interface="Egomotion" source="HollowayRFlexMotion" object="EgoClient" />
```

```
9           </connect>
10
11
12          <connect module="laserAvoider">
13                  <input interface="RangeFinder" source="HollowayLMS" object="LaserClient"/>
14                  <input interface="Egomotion" source="HollowayRFlexMotion" object="EgoClient"/>
15                  <input interface="NavigatorTransport" source="VectorMoverNavigator" object="NavClien
16          </connect>
17
18          <connect module="BlindTeleop">
19                  <input interface="ObstacleAvoiderTransport" source="SimpleAvoider" object="AvoiderCl
20                  <input interface="Egomotion" source="HollowayRFlexMotion" object="EgoClient" />
21                  <input interface="PTUnit" source="SafePTUServer" object="PTUClient"/>
22          </connect>
23
24  </mcp-mappings>
```

This file lists three modules: our blindTeleop program, the obstacle avoider it uses, and the vector mover required by the obstacle avoider. Within each "connect" element, there is an element for each consumer which needs to be connected to a supplier in another module. The "interface" attribute specifies the interface type. The "source" attribute specifies the name of the stream providing this interface, and the "object" attribute specifies the local alias of the consumer in the module. In the case of the VectorMover, the SynchroDrive RobotObject called "DriveClient" is connected to the supplier with the name "HollowayRFlexDrive" which provides access to the motors on our ATRV Jr.

Chapter 2

Architecture

The WURDE architecture uses four layers of abstraction: Communications, Interfaces, Applications, and Systems. Each layer provides a buffer from the other layers in order to simplify the process of development.

The Communications layer consists of one or more *communications adaptors*. These adaptors take care of the basic process of moving data using a particular protocol. The primary adaptor in WURDE at the moment uses CMU IPC (Carnegie-Mellon University Inter Process Communication), but an adaptor can be developed for any transport mechanism which is compatible with Linux/C++. Communications adaptors do have to interact with the interface layer to generate any protocol specific code which is required; note that this is the method of the CMU IPC adaptor, but may use templates or another method for other adaptors.

The Interfaces layer provides definitions for different possible robot capabilities. The WURDE interface layer allows two different messages to be defined per interface; these are defined as standard C++ classes. The first message is the *data* message. Data messages move one direction only; to a *supplier* of the capability, they are write-only, while to a *consumer* of the capability, they are read-only. The second type is the *info* message, which is bi-directional. This allows consumers to send requests to suppliers. The info message type carries some additional overhead, so the data type is intended to be the primary message for high traffic capabilities such as RangeFinders. Interfaces are created using XML definition files, from which WURDE will generate C++ code for each available interface.

The Applications layer provides a consistent API for working with WURDE. Each interface has a similar API for setting standard options, and sending/receiving data. In addition, there are several useful classes and a small collection of libraries for accessing different types of hardware. The API requires no knowledge of the underlying communication adaptor; all of this is handled behind the scenes. Furthermore, the programmer need not specify the exact connections to be made between different modules, only the types of data required. WURDE connects modules automatically using the MCP at the systems level.

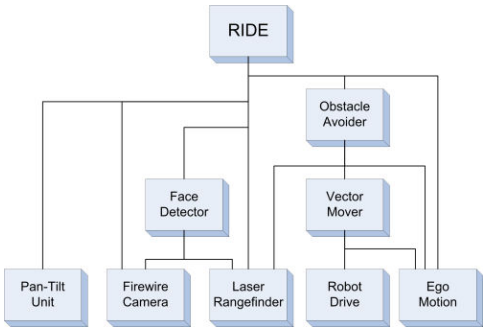

Figure 2.1: Module startup with MCP.

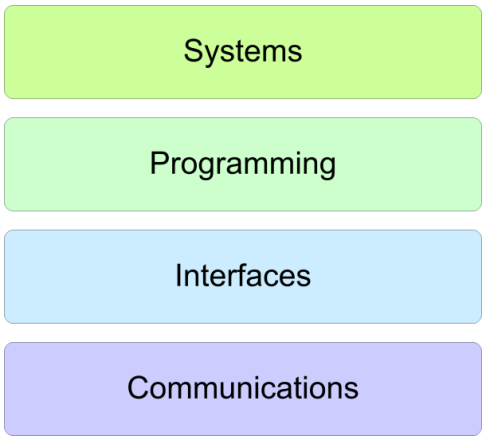

Figure 2.2: Module startup with MCP.

The Systems layer

Chapter 3

Interfaces

As a general definition, interfaces provide access to robot functionality. An interface could allow a program to send commands to the motor, to get range finder readings, to get images, or more abstract things. There could be a blobfinder interface, for example. More specifically, interfaces provide a transparent way to get information from another process.

Every interface has three major parts: the specification, the high level API, and the communication adaptor. The specification takes the form of an XML file describing the data which the interface contains. The high level API is the C++ object which a programmer actually uses, and the communication adaptor is code which connects the object to an underlying communication protocol. An interface may have (and is expected to have, in the future) multiple communication adaptors which it may use.

This section is meant primarily as a guide to building new interfaces. First the built-in types that WURDE knows about will be described, then the syntax for the interface specification file, and finally a list of the interfaces currently included as part of WURDE. Communications adaptors will be covered in a later chapter.

3.1 Built-in Types

WURDE provides a number of built-in types specifically for use with robots. All types should use standard metric units(seconds, meters, and radians) as appropriate. There is currently no mechanism to enforce this, so just be careful!

> **Note:** All new interfaces or types defined for use with WURDE should use only standard metric units.

3.2 Interface Configuration

There are several common options which can be set for interfaces. Usually these will be available for all interfaces, but some may be affected by the choice of communication adaptor. When in doubt, check the documentation.

3.2.1 Auto-Tagging

Functions: *setAutoTag(bool val), bool getAutoTag()*

Auto-tagging sets the "source" value of info and data events before they are published. In some cases, it may be necessary to mimic other modules, so it can be turned off. The default is on.

3.2.2 Auto-Timestamp

Functions: *setAutoTimestamp(bool val), bool getAutoTimestamp()*

Auto-timestamp sets the "timestamp" value of info and data events before they are published. There may be cases when it is necessary to set the timestamp to an earlier or later value, so it can be turned off. The default is on.

3.2.3 Queue Mode

Functions: *setQueueMode(bool val), bool getQueueMode(), DataStruct getNextData(), InfoStruct getNextInfo(), InfoStruct getNextRequest()*

Queue mode places each new event into a queue rather than overwriting the old data or info. While it is okay for many interfaces to overwrite the old data each time, in other cases it may be important to catch every event. Turning queue mode guarantees that all events will be caught. The *get* functions return the next event on the queue, and also place copy this event into the data/info/request structs as appropriate.

3.2.4 RobotObject Types

Time

The Time object provides time values with microsecond precision. The primary purpose of the Time object is to make time comparisons simple; comparison operators (==, !=, <, >, <=, >=) are provided for Time, time_t, and timeval struct variables. See the API documentation for a full description.

Writable

The Writable template class provides access control for the information/requests channel, which allows processes to lock control over entire devices, or just specific attributes of devices (IE, resolution of a camera). Devices can be locked so that only suppliers, only consumers, or a only specific process may modify the values. See the API documentation for a full description.

Point and Point3D

The Point types have simple coordinate values. Point has x and y, while Point3D also includes a z coordinate. Both types include a single double-value weight. All coordinates are stored as doubles.

- *Point()*: Zero-value constructor. Creates a point at (0,0) with weight value 0.
- *Point(x,y)*: Standard constructor. Creates a point at (x,y) with weight value 0.
- *Point(x,y,w)*: Weight constructor. Creates a point at (x,y) with weight value w.
- *Point3D()*: Zero-value constructor. Creates a point at (0,0,0) with weight value 0.
- *Point3D(x,y,z)*: Standard constructor. Creates a point at (x,y,z) with weight value 0.
- *Point3D(x,y,z,w)*: Weight constructor. Creates a point at (x,y,z) with weight value w.
- *value()*: Returns the weight value of the point.
- *value(double)*: Sets the weight value of the point.
- *x()*: Returns the x coordinate of the point.
- *x(double)*: Sets the x coordinate of the point.

- *y()*: Returns the y coordinate of the point.
- *y(double)*: Sets the y coordinate of the point.
- *z()*: (Point3D only) Returns the z coordinate of the point.
- *z(double)*: (Point3D only) Sets the z coordinate of the point.

The Point3D class does not currently inherit from the Point class. This is under consideration.

Pose and Pose3D

The *Pose* types include angle values in addition to coordinates. Pose has just one, *theta*, while Pose3D has three: *theta*, *phi*, and *psi*. Like the Point classes, the Pose classes also have a single double-value weight (this may be changed in the future to be more generic). The value(), x(), y(), and z() methods from the Point classes apply as appropriate to each of these objects.

- *Pose()* Zero contructor. Creates a pose at (0,0) with theta 0, weight 0.
- *Pose(x,y)* Zero theta constructor. Creates a pose at (x,y) with theta 0, weight 0.
- *Pose(x,y,t)* Standard constructor. Creates a pose at (x,y) with theta t, weight 0.
- *Pose(x,y,t,w)* Weight constructor. Creates a pose at (x,y) with theta t, weight w.
- *Pose3D()* Zero contructor. Creates a pose at (0,0,0) with orientation (0,0,0), weight 0.
- *Pose3D(x,y,z)* Zero orientation constructor. Creates a pose at (x,y,z) with orientation (0,0,0), weight 0.
- *Pose3D(x,y,z,t,p,s)* Standard constructor. Creates a pose at (x,y,z) with orientation (t,p,s), weight 0.
- *Pose3D(x,y,z,t,p,s,w)* Weight constructor. Creates a pose at (x,y,z) with orientation (t,p,s), weight w.
- *theta()* Returns the theta value of the orientation.
- *theta(double)* Sets the theta value of the orientation.
- *phi()* Returns the phi value of the orientation.
- *phi(double)* Sets the phi value of the orientation.
- *psi()* Returns the psi value of the orientation.
- *psi(double)* Sets the psi value of the orientation.

The Pose classes currently do not inherit from the Point classes, and the Pose3D class does not currently inherit from the Pose class. This is under consideration.

3.3 The Interface Definition File

Interface definition files are XML files which, when placed in the *src/robot/interfaces/definitions* directory, will generate new RobotObject types which are capable of transferring the specified data. The file begins with:

```
<?xml version=''1.0''?>
<!DOCTYPE interface SYSTEM ``include/interface.dtd''>
```

The interface definition file includes a mandatory *interface* entity, a mandatory *author* section, a mandatory *description* section, optional *type* sections, and one each optional *info* and *publish* sections. The file must end in an *interface* closing tag.

3.3.1 Interface Entity

The interface entity has two attributes, *name* and *version*. Example:

```
<interface name=''Foo'' version=''1.0''>
```

Never create multiple interfaces with the same name (even if the version is different).

3.3.2 Author/Description Sections

These sections are self-explanatory. The description section should include a CVS Id tag. Example:

```
<author>Frederick Heckel</author>
<description>
        Source: $Id: ProgrammingGuide.tex 10 2006-11-15 15:22:43Z fwph $

An example interface which kinda looks like a RangeFinder merged with an Image provider. It's
whacky and doesn't make sense.
</description>
```

Type/Info/Publish Sections

type sections can be used to create data structures which will be included in the *info* and/or *publish* sections. Most standard types can be used here, though enumerations are not supported. Supported types are: Byte, Integer, Long, String, Float, Double, Bool, Pose, Pose3D, CVector, PVector, Point, Point3D, RunState, ProcessLocale, ProcessType, StatusMode,ConnectionStrategy. Bytes are implmented as chars, though this will probably be changed to unsigned char soon. Arrays are supported by using the *array_size* attribute. Example:

```
<type name="Pixel">
         <data type="Byte" name="r"/>
         <data type="Byte" name="g"/>
         <data type="Byte" name="b"/>
</type>

<info>
         <data name="max_range" type="Float"/>
         <data name="uniform" type="Boolean"/>
         <data name="range_count" type="Integer"/>
         <data name="half_angle" type="Float"/>
         <data name="sensor_positions" type="Pose3D" array_size="range_count"/>
         <data name="x_offset" type="Float"/>
         <data name="y_offset" type="Float"/>
         <data name="z_offset" type="Float"/>
         <data name="theta_offset" type="Float"/>
         <data name="theta_separation" type="Float"/>
         <data name="phi_offset" type="Float"/>
         <data name="psi_offset" type="Float"/>
         <data name="x_separation" type="Float"/>
         <data name="y_separation" type="Float"/>
         <data name="z_separation" type="Float"/>
         <data name="phi_separation" type="Float"/>
         <data name="psi_separation" type="Float"/>
</info>
```

```
<publish>
        <data type="Integer" name="size" />
        <data type="Pixel" name="image" array_size="size"/>
</publish>
```

The presence of an "array_size" attribute causes the data member indicated to be removed from the data structure if it is present, as all arrays are implemented with the STL vector class. This silly syntax will be fixed in the future.

3.4 Included Interfaces

3.4.1 RangeFinder

The RangeFinder interface includes an *info* structure specifying the location and parameters of the sensors, and a *publish* structure which provides the actual range values.

Info structure:

- *max_range*: Floating point value. The maximum reliable range of the sensors in this bank.
- *half_angle*: Floating point value. Gives the half angle of the sensors in this bank.
- *uniform*: Boolean value. If true, then the offset and separation values can be used to locate the individual sensors. Otherwise, the sensor_positions array should be used.
- *sensor_positions*: Array of Pose3D values. Gives the location, relative to the center of the robot, of each sensor in the bank. The zero Z value should be the bottom of the robot– in most cases, this should be where the wheels touch the floor.
- *x_offset*: Floating point value. What do these mean? I think we need to rework these position specifiers, they seem much clunkier than necessary. Doesn't polar make a lot more sense here? most sensors can be specified as r,theta,h– very few will be pointing in strange directions like directly up. Much of the time, we'll only be using r and theta anyway, as part of fused sensor data. We can even have a special interface to deal with weird sensors, if we actually need them, but I really think this is overkill. Also, quite frankly, I think we should just use a straight array of each pose value. The offsets/separations are nice for the person who writes the rangefinder server, but terrible, awful, and completely miserable for the consumer. We should cater to the consumers.
- *y_offset*
- *z_offset*
- *theta_offset*
- *phi_offset*
- *psi_offset*
- *theta_separation*
- *phi_offset*
- *psi_offset*
- *x_separation*
- *y_separation*

- *z_separation*
- *phi_separation*
- *psi_separation*

Publish Structure

- *ranges*: Array of doubles. Gives the range value reading for each sensor.

3.4.2 BlobFinder

3.4.3 Bumper

The *info* section of the Bumper interface is very similar to the RangeFinder info section, except that it does not include *max_range* and *half_angle* values. All other position values are identical.

The *publish* section contains an array of boolean values for each individual sensor, plus boolean a value which specifies whether any are active.

- *bumped*: Boolean value. Specifies whether any sensors are active.
- *bumpers*: Array (STL vector) of boolean values. Gives the bumped value for each sensor.

3.4.4 Egomotion

3.4.5 FaceDetector

3.4.6 Foo

3.4.7 Heartbeat

The heartbeat interface provides a facility for sending a simple keep-alive message. This message is sent automatically by the communication manager, and enables the MCP to track module status.

Each keepalive message contains a single data field: TTL. This defines how long the MCP should wait for the next message. The read-write channel of this interface provides information about the process, including its managed state and locale (see the MCP documentation), possibly a list of interfaces provided and their stream names, and the run state of the process. The MCP and other controllers will use this interface to control the state of the module.

In general, it is rare to use this interface directly.

3.4.8 ImageDisplay

Empty

3.4.9 ImageTransport/ImageSource

Empty

3.4.10 LoggerTransport/Logger

The LoggerTransport interface is extended to create the Logger class, which is the core of the logging system. See section on logging.

3.4.11 McpRequest

The McpRequest interface is used for requesting modules and stream information from the MCP, which maintains a directory of available binaries and information sources. This interface will rarely be useful outside of the CommsManager.

3.4.12 NavigatorTransport/Navigator

3.4.13 ObstacleAvoiderTransport/ObstacleAvoider

3.4.14 Power

3.4.15 PTUnit

3.4.16 RangeFinder

3.4.17 SimpleControl

This interface currently only provides control over the MCP.

3.4.18 SoundController

3.4.19 SynchroDrive

SynchroDrive is a control interface for robot motion, and does not publish any data. The info fields are as follows (fields marked optional may have no effect, depending on the implementation):

- *brake*: Boolean value. When set true, prevents the robot from moving through either a hardware brake mechanism, or software logic. It is recommended that this should not be locked to "false" as it should always be possible to turn the brake on.
- *trans*: Floating point value. This sets the translational velocity of the robot. Negative values should be interpreted as backwards motion on platforms that allow it.
- *angular*: Floating point value. This sets the angular velocity of the robot. Positive values should be interpreted as counter-clockwise (increasing angle in SI units), negative values as clockwise.
- *transLimit*: Floating point value. Sets a limit on the translational velocity of the robot. This is treated as an absolute value, and limits both forwards and backwards motion.
- *angularLimit*: Floating point value. Sets a limit on the angular velocity of the robot. This is is treated as an absolute value, and limits both clockwise and counter-clockwise motion.
- *transAccel*: (Optional) Floating point value. Sets a limit on the translational acceleration of the robot.
- *angularAccel*: (Optional) Floating point value. Sets a limit on the angular acceleration of the robot.
- *transDecel*: (Optional) Floating point value. Sets a limit on the translational deceleration of the robot.
- *angularDecel*: (Optional) Floating point value. Sets a limit on the angular deceleration of the robot.

3.4.20 VisionControl

3.4.21 WallBuilder

Chapter 4

The Logging System

WURDE includes a basic logging system which makes it possible to direct robot log messages to a number of locations. Currently supported are output to the syslog facility and through the communication adaptor via the LoggerTransport object. In the future, logging to file as well as formatted data logging will be supported, but these are not complete yet.

When syslog output is enabled, the logging system sends to the "local0" facility. These messages can be captured in a single file by adding a line to /etc/syslog.conf:

```
local0.*                                /var/log/robot.log
```

By default, the logger will also direct the syslog output to stderr. When the first logger object is created, a global logger pointer is set, and the global logging functions (g_debug, g_info, etc.) can be used. If no logger object has been created, these global functions will send the log messages to stderr. Stream versions of each of these are also available, and significantly more convenient. These are g_logdebug, g_loginfo, etc. They can be used exactly like cout/cerr, though it is recommended to always use "endl" to end a message, rather than a newline character.

See the Doxygen documentation for more complete documentation of all Logger functions.

Chapter 5

The Communication Manager

The communication manager(CommsManager) handles all the work necessary to send and receive new data. It manages the various communication adaptors, and can control the rate at which the program loop will run. When the MCP or another software manager is used, the CommsManager will send out the heartbeat, and can be used to configure modules automatically from a configuration file.

5.1 Configuration

5.1.1 Command Line Options

The *parseOptions* methods of the CommsManager is used to parse standard and user-supplied configuration options. Standard options include:

- *-r* : Registration mode. This switch starts the module in registration mode, causing it to start up, send information to the MCP, and immediately shut down.
- *-o* : Namespace override. For communication adaptors with strict naming services (which do not allow two modules of the same type to connect), this flag is used to inform the communication adaptor to override the naming service and connect if there is already another module of the same name connected. This may result in the original module being disconnected. It has no effect in the CMUIPC adaptor.
- *-n* : Module name override. This switch overrides the given name of the module as supplied to the CommsManager at initialization. This is important of multiple modules of the same type will be initialized. This option takes a string argument. Programs must be written to properly take advantage of this.
- *-g* : Configuration file specification. The default role-config file is role-config.xml in the working directory. This flag allows use of a different configuration file.

5.1.2 The Configuration File

The WURDE configuration file can be used to specify global variables for configuration files, data directories, logging options and module-specific options.

- *data-directory*: Specifies the location for storing data to be used and output by different modules.
- *config-directory*: Specifies the location for configuration files used by modules. For the MCP, passing a path on the command line, or specifying an absolute path otherwise will override the config directory.

- *bin-directory*: Specifies the location of module binaries (primarily for use by the MCP for finding internal modules). Setting an absolute path in the mcp-config file will override this directory on a module basis.
- *log*: Specifies the directory for storing log files and the message priority to start logging. Possible values for the level attribute: debug (default), info, notice, warn, error, fatal.
- *mcp*: mcp-config specifies the name of the mcp configuration file (in the config-directory), and mcp-mappings specifies the name of the mcp-mappings file.
- *option*: The name attribute gives an option name, and the value attribute the value of that option. Options can be found with the RoleConfiguration::getOption method.
- *module*: The name attribute gives the (CommsManager) name of a module. Can contain multiple option elements.

5.1.3 Environment Variables

Environment variables will override the values set in the role-config file. The following variables can be set:

- WURDE_CONFIG_DIR
- WURDE_DATA_DIR
- WURDE_BIN_DIR
- WURDE_MCP_CONFIG
- WURDE_MCP_MAPPINGS
- WURDE_LOG_LEVEL
- WURDE_LOG_DIRECTORY

5.1.4 Using the RoleConfiguration Object

The global RoleConfiguration object, g_globalConfiguration, can be used to obtain values of options specified in the configuration file.

- *getDataDirectory*:
- *getConfigDirectory*:
- *getBinDirectory*:
- *getMCPConfigFile*:
- *getMCPMappingsFile*:
- *getLogDirectory*:
- *getModuleName*:
- *getLogLevel*:
- *haveOption(string name)*: Returns true if the specified option has a value.
- *getOption(string name)*: Returns the value of the specified option if set, or "empty" otherwise.

- *haveModuleOption(string module, string name)*: Returns true if the specified option is set for the specified module.
- *getModuleOption(string module, string name)*: Returns the value of the specified option if set for this module, or "empty" otherwise.

For methods to set the module options, see the Doxygen documentation. Note that setting the options manually via these methods is not recommended– the configuration file or the environment variables should be used, as changes will have no effect after initialization.

5.2 Program State

5.2.1 The Heartbeat

In addition to being a keepalive signal, the Heartbeat interface is used to change the state of the program. The MCP can order the CommsManager to put the program into several states: Idle, Reset, Restart, Quit, and Run. It is up to the main function of the program to fully interpret these states, but the CommsManager will not send or receive data in any of these modes except Run and Idle. Idle data is limited to the Heartbeat and MCPRequest channels.

Once the appropriate action for a state is taken, the setRun() method of the manager can be called to return the program to its normal state.

setHeartbeatOnly(double) sets the heartbeat TTL to be the specified number of seconds.

setNoHeartbeat() turns off the heartbeat function.

5.2.2 Execution Rate Control

The communication manager can be used to control the rate of execution in a loose manner. It does not account for the execution time of the program loop, instead sleeping for a set period of time. By default, this is 0.1 seconds, providing a maximum execution rate of 10 hz.

setSleep(double) sets the sleep time to be the specified number of seconds. Fractional numbers of seconds are, naturally, allowed. The TTL of the heartbeat is set to twice this value.

setSleepOnly(double) sets the sleep time to be the specified number of seconds, but does not change the heartbeat TTL time.

setNoSleep() turns off execution rate control in the communication manager.

setPreSleep() causes the communication manager to sleep *before* updating the objects.

setPostSleep() causes the communication manager to sleep *after* updating the objects. This is the default.

5.3 Triggers

Triggers can be used to activate program execution only when new data has arrived for a given object. First Trigger objects must be created, and each object which should activate the trigger must be registered with it. All objects in the trigger must receive new data before it will be activated (treat the contents of a single trigger as tied together with AND), but any one of multiple triggers registered with the CommsManager can cause program execution (multiple triggers are OR'd together).

An object will activate a trigger if either new data or new info has arrived. This behavior may be modifiable in the future.

The CommsManager does not keep track of which trigger caused activation. This behavior may be modified in the future.

Chapter 6

The Master Control Program

At the most basic level, the MCP is a daemon which gathers information about available binary modules which can be used by any WURDE module. Basic information (binary information, and possibly other optional information) is provided through a configuration file which is read at startup. Once the MCP is aware of the locations of module binaries, it initializes each binary to gather information. It can then track the state of modules, answer requests for modules to be started up, and provide the names of modules which provide specific stream types.

6.1 MCP Configuration

The MCP config file provides basic information about binary modules available to the system. Five pieces of information can be specified.

Terminology:

The *locale* of a module specifies whether this module exists on the local system. By default, this is set to *internal*, indicating that the module exists on the same system as the MCP. The MCP cannot automatically start *external* modules.

The *managed state* of a module specifies how the MCP should track the module. *Managed* modules are tracked normally, and the MCP will attempt to restart modules if they die. *Unmanaged* modules are ignored, though can still be started and stopped if internal. *Critical* modules cause the MCP to send an idle command to all other running modules if the module goes into a zombie state, until the module is successfully restarted.

The *name* of the module is required, Other information is optional. The *binary* location of the module should be specified for all *internal* modules.

The MCP will look for a config file named "mcp-config.xml"; an alternate file can be specifie with the "-c" flag. See the appendix for an example of an MCP configuration file.

6.2 The MCP mapping file

The mapping file is an optional configuration file which provides a number of *mappings* for the MCP. Each binary module may have a single "connect" section. The connect section contains a number of "input" entities, each of which specifies an interface type t, the name of a supplier s, and (optionally) the name of a consumer n.

What this means: A consumer of type t from the specified module will be given the stream name s if a request from this module is made. If an object name is specified, the consumer of type t with global name n is connected to this stream. A mapping which specifies an object name will not be used to fill other generic requests; requests that do not specify an "assigned" connection strategy will be handled in a more general manner.

The MCP will look for a mapping file named "mcp-mapping.xml"; an alternate file can be specified with the "-m" flag. See the appendix for a specific example of a mapping file.

6.3 Automatic Stream Selection

There is one constructor for all RobotObjects which make use of MCP stream selection. The standard constructor does *not* currently make a request to the MCP to start up a particular module, so you should use the mcp-mapping file and the special constructor any time you wish to use module auto-loading.

Type(ConnectionStrategy, string) is the stream selection/auto-loading constructor. Connection Strategy has three possible values: STRAT_AUTO, STRAT_ASSIGNED, and STRAT_NORMAL. If STRAT_NORMAL is used, the provided string is treated as the stream name, and MCP will attempt to load a module that provides a stream by this name (global name will be set automatically). If STRAT_AUTO is used, the mcp-mapping file will be used to locate a stream which can provide this interface to this module, and this object will be given the global name specified by *string*. If STRAT_ASSIGNED is used, the mcp-mapping file will be used to locate the exact assignment for the global name provided (and *string* will be used as this object's global name).

STRAT_NORMAL, is, ironically enough, untested at this time.

Chapter 7

Official WURDE Modules

All WURDE modules that consume data (some configurations of the Vision Module, the SynchroMover, the Laser Avoider, and others) require the MCP to be active, and reasonable mappings available.

All modules have a set of common command line options:

- -r This sends module Capability information (consumers and suppliers) to the MCP (or other receiver. It also prints all available capabilities in the module to standard out.
- -n Module name override. Specifies a name different from the one hard-coded into the module; this is important for some modules which use this setting to choose a configuration section from a config file.
- -h Displays a help message.

To make use of these options in your program, be sure to call parseOptions(argc,argv) after instantiating the CommsManager. This routine will also parse user-specified arguments when they are passed as a third string to the parseOptions call. Simple (switch) options should be specified as a single character, and options which take an parameter should be specified as a single character followed by a colon (example: "c:ab" would specify a "-c" option which takes an argument, and two simple switches a and b). These options can then be accessed through the global WURDEConfiguration object, g_globalConfiguration. To add additional information to the "-h" output, use the CommsManager setHelpString(string) call **before** calling parseOptions.

7.1 Utils

Utils provides a pair of test programs, FooServer and FooClient, and two useful utilities for controlling running modules, turnOff and mcpController. FooServer and FooClient can be used to test that the middleware is operating correctly with the MCP. turnOff takes zero or one arguments; if no arguments are given, all non-MCP modules are sent a quit message. If an argument is given, the specified module (CommsManager name) is sent a quit message.

7.2 VisionModule

The WURDE Vision Module is designed as a single-binary vision system capable of loading different "operators" based on an XML configuration file.

VisionModule command line options:

- -c Specifies the camera-config file to use
- -v Specifies the vision-config file to use

- -n Changes the name of the VisionModule, and also specifies which module layout to use.

7.2.1 Camera Configuration

The camera configuration file contains one or more "camera" entities. Each entity specifies the name and the type of the camera as attributes. Supported types are "firewire" and "playback". In addition, *option* entities allow the setting of several different options for the camera. An option entity has two attributes, *name* and *value*.

It is planned to include camera calibration matrices in this file as well in the future. The "-c" flag specifies the camera configuration file. By default, this module looks for a file called "camera-config.xml." An example camera configuration file can be found in the appendix.

Firewire Camera

Firewire (IEEE 1394 IIDC) cameras are supported using the dc1394 library. Currently exactly version 2.0.0 pre7 of dc1394 is required; as of this writing, 2.0 has not been released, and the API not finalized. Once 2.0 has been released, we will update the library.

Firewire cameras require the *id* option, which specifies the ID of the firewire camera. The complete list of options:

- id A 64-bit number specifying the camera ID.
- brightness An integer specifying the brightness value for the camera.
- exposure An integer specifying the exposure value for the camera.
- sharpness An integer specifying the sharpness value for the camera.
- hue An integer specifying the hue value for the camera.
- saturation An integer specifying the saturation value for the camera.
- gamma An integer specifying the gamma value for the camera.
- shutter An integer specifying the shutter value for the camera.
- gain An integer specifying the gain value for the camera.
- iris An integer specifying the iris value for the camera.
- focus An integer specifying the focus value for the camera.
- zoom An integer specifying the zoom value for the camera.
- autoiris A string "true" or "false" specifying whether to use the autoiris feature of the camera.

It is highly recommended to set default values for all of these options, especially when using two cameras as a stereo pair.

Playback Camera

The playback camera is a special camera device emulator. It returns previously returned frames, making it simple to run operators on logged data. The playback camera requires the imageDirectory option. The complete list of options:

- imageDirectory Specifies the directory containing the logged images.

- imageSuffix Specifies the suffix of the images. Default is ".ppm". All image types supported by OpenCV can be used.
- imagePrefix Specifies an image prefix (ie, "FaceTrainingSet") that is part of all image filenames.
- imageDevice Specifies an additional string that is part of all image filenames. When used with data logged by the StereoDataCollector, this will be "main" or "stereo".

Filenames are expected to be in the form
imagePrefix-[integer frame number]-*imageDevice*-[floating point timestamp in seconds].*imageSuffix*

7.2.2 Module/Operator Configuration

The vision module configuration file contains one or more "module" sections which describe the configuration for instances of the vision module. When the module starts up, it will load the configuration which matches its name as passed in using the "-n" flag. The "-v" flag specifies the configuration file for the vision module. By default, this module looks for a file called "vision-config.xml." The configuration file also includes "plugin" entities providing paths to shared libraries which should be loaded at runtime. These plugin libraries contain the operator classes which will be used by the vision module.

Each module section contains one or more camera entities and one or more operator entities. Each operator entity as two attributes: "name" and "type". Name specifies a unique name for this instance of the operator, and type specifies which operator should be instantiated. An operator entity also has one or two *camera* entities and one or two *option* entities. The operator camera entity has a single "name" attribute to specify which camera is to be loaded. The option entity has two attributes, "name" and "value". Interpretation of these attributes will be left as an exercise to the reader.

Finally, there are plugin entities. Each plugin entity has a single attribute: "path". This specifies the full path to an operator dynamic library.

An example vision module configuration file can be found in the appendix.

7.2.3 Operators/Plugins

Currently available plugins are ImageWriter, ImagePublisher, and StereoDataCollector.

ImageWriter

Writes images to disk. Can write images at a speed of about 25 fps for a single camera on an ATA 100 system, at 320x240 color ppms. Currently only 320x240 RGB images are supported.

Operator options:

- imageDir Specifies where to write images. By default, the WURDE data directory is used.
- imagePrefix Specifies a prefix for the images. By default, uses the operator name.

ImagePublisher

Publishes uncompressed images over the communications adaptor. Recommended for use only to transmit images over the network for the time being, as operators should be used locally when possible. Currently only supports 320x240 images.

Operator options:

- format One of yuv24, yuv422, gray, or rgb. Default is gray.

StereoDataCollector

This operator simultaneously logs data from the camera, a RangeFinder source, and an Egomotion source. Timestamps of the original data are maintained, but each set matching data (data that is available at the same time) is tagged with an additional frameindex. All of this information is contained in the filename, which is as follows:

imagePrefix-[integer frame number]-*device*-[floating point timestamp in seconds].*suffix*

Device is one of: main, stereo, laser, odometry. "main" is the first camera listed for the operator, and "stereo" is the second camera listed for the operator. The images are saved as .ppms. Rangefinder output is saved to a .txt file which contains only a list of numbers representing a single reading (no header information, just data). Odometry data is a list of numbers of the form: x, y, theta, translational velocity, angular velocity (no commas).

Options for the StereoDataCollector:

- imageDir Where to save the data. Default is the WURDE data directory
- imagePrefix A string prefix for all data files. Default is the name of the operator.

7.3 SynchroMover

SynchroMover is an implementation of the VectorMover/VectorMoverTransport interface. VectorMover should be used as the consumer object (it merely extends NavigatorTransport to make construction of the commands simpler). SynchroMover does *not* queue commands; when it receives a new goal point, it overrides its current goal point. See the VectorMover class and the VectorMoverTransport interface.

SynchroMover requires the RFlexServer, PlayerServer, or other sources of Egomotion (such as a localization algorithm) and RobotDrive capabilities. It is designed for Synchro/Differential Drive robots, and will not work with Ackermann drive systems or legged robots.

7.4 PTU Server

The PTU server is designed to control a directed perception pan-tilt unit. The PTUnit interface can be used to control this module.

Note: Because our implementation of the ptu module uses code from Directed Perception, we cannot distribute the full module. If you own one of these units, a number of source files must be added to build the module; see the README file in modules/ptu for more information.

7.5 SICKServer

This module provides access to SICK PLS and LMS lasers. This module makes use of elements from the CARMEN toolkit from CMU. The SICK Server requires a configuration file to specify the serial device, the type of laser used, whether to log data to a file, and whether the laser is inverted (mounted upside down). Mounting points which are not parallel to the ground are not currently modeled by the system. See the RangeFinder interface.

The laser config file has a root entity *laser-config* with one or more *device* entities. Device entities have three required attributes: "port", "name" and "type". Port specifies the serial port the rangefinder is attached to. Name specifies a unique name for this laser, and type is one of PLS or LMS, to specify the type of laser. There are two more optional attributes: "invert" and "log". If the laser is mounted upside down, invert should be set to true. If log is set to true, readings from the laser will be sent to a log file; each line of this file takes the form of

[timestamp] : [readings]

The output file will be in WURDE_DATA_DIRECTORY/LaserLog.txt. If capturing other data simultaneously with laser data, we recommend using the StereoDataCollector module (if capturing images) or writing a new module to capture the needed data.

This module has two important command line options:

- -c Specifies an alternate config file (default is WURDE_CONFIG_DIRECTORY/laser-config.xml).
- -n Specifies an alternate name for the module. This is used to choose the device from the laser config file, and should always be specified.

7.6 RFlexServer

This module provides access to RFlex-based robot controllers. iRobot B21r and ATRV Jr. robots are supported at present. The RFlexServer module requires a configuration file. This file specifies the RFlex serial device, the type of robot being controlled, and whether the module should be allowed to reset the brake.

Insert Example Here.

7.7 LaserAvoider

This module implements a laser rangefinder-based obstacle avoidance algorithm. It requires the VectorMover, a source of Egomotion, and a source of RangeFinder (should be laser). See the ObstacleAvoider class and the ObstacleAvoiderTransport interface.

7.8 Eye

This module displays images published by the ImagePublisher vision operator or other ImageSource capability provider. See the ImageSource class and the ImageTransport interface.

7.9 BlindTeleop

This module allows tele-operation using a standard USB joystick. It does not receive images (use Eye for image display).

Chapter 8

Communication Adaptors

8.1 The CMUIPC Communication Adaptor

The main communication adaptor is based on the Carnegie Mellon University IPC system developed by Reid Simmons.

8.2 Writing a New Adaptor

The short version, until I write the long version: Look at the COMObject header file. See all the virtuals? You need to implement them. The adaptor is expected to copy the data and info into the native structures from whatever structure the protocol uses for transport.

Chapter 9

Troubleshooting

9.1 CMU IPC Related

How do I connect to a central server running on a different machine? Set the environment variable CENTRALHOST to the correct hostname:

```
export CENTRALHOST=tom.cse.wustl.edu
```

Chapter 10

Appendix

10.1 Example Configuration Files

There are examples of each of these files included with the WURDE install. Some of these examples are more complex than those provided as part of the package.

10.1.1 Example Combined WURDE/MCP Config file

```
<?xml version="1.0" encoding="UTF-8" ?>

<robot>
<wurde-config xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://wurde.sf.net/wurde-config"
        xs:schemaLocation="http://wurde.sf.net/wurde-config wurde.xsd">

        <data-directory location="/home/fwph/data"/>

        <config-directory location="/home/fwph/config"/>

        <bin-directory location="/usr/local/bin/"/>

        <log directory="/home/fwph/logs/" level="info"/>

        <mcp mcp-config="local-config.xml">
              <comment>The config files should be in the config directory.</comment>
        </mcp>

        <option name="example" value="15">
               <comment>Option entities should always be commented!</comment>
        </option>
</wurde-config>

<mcp-config xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://wurde.sf.net/mcp-config"
        xs:schemaLocation="http://wurde.sf.net/mcp-config mcp.xsd">

        <module name="TomRFlex" binary="rFlexServer"/>
```

```
        <module name="TomPLS" binary="sickLaser"/>
        <module name="SynchroMover" binary="synchroMover">
                <connect consumer="DriveClient" supplier="TomRFlexDrive"
                                                interface="RobotDrive"/>
                <connect consumer="EgoClient" supplier="TomRFlexMotion"
                                                interface="Egomotion"/>
        </module>
        <module name="laserAvoider" binary="laserAvoider">
                <connect consumer="LaserClient" supplier="TomPLS"
                                                interface="RangeFinder"/>
                <connect consumer="EgoClient" supplier="TomRFlexMotion"
                                                interface="Egomotion"/>
                <connect consumer="VectorMoverClient" supplier="SynchroMover"
                                                interface="VectorMoverTransport"/>
        </module>
        <module name="BlindTeleop">
                <connect consumer="EgoClient" supplier="TomRFlexMotion"
                                                interface="Egomotion"/>
                <connect consumer="AvoiderClient" supplier="SimpleAvoider"
                                                interface="ObstacleAvoiderTransport"/>
        </module>
        <module name="FooClient">
                <connect consumer="BarClient" supplier="Bar" interface="Foo" />
        </module>
        <module name="FooServer" binary="fooServer"/>

        <module name="StereoVisionModule" binary="visionModule">
                <arg value="-n StereoVisionModule"/>
                <arg value="-v local-vision-config.xml"/>
                <connect consumer="DataEgoMotionClient" supplier="TomRFlexMotion"
                   interface="Egomotion"/>
                <connect consumer="DataRangeFinderClient" supplier="TomPLS"
                   interface="RangeFinder"/>
        </module>

        <module name="Eye">
                <connect consumer="EyeImageClient" supplier="LeftImagePublisher"
                   interface="ImageTransport"/>
        </module>
</mcp-config>

</robot>
```

10.1.2 Example Camera Config file

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE camera-config SYSTEM "camera-config.dtd">
<camera-config>

        <camera name="rightCamera" type="firewire">
       <option name="id" value="0x8004602000301fc"/>
```

```
       <option name="brightness" value="1024"/>
       <option name="exposure" value="128"/>
       <option name="sharpness" value="0"/>
       <option name="hue" value="128"/>
       <option name="saturation" value="128"/>
       <option name="gamma" value="128"/>
       <option name="shutter" value="2048"/>
       <option name="gain" value="0"/>
       <option name="iris" value="2048"/>
       <option name="focus" value="256"/>
       <option name="zoom" value="0"/>
       <option name="autoiris" value="true"/>
</camera>

        <camera name="leftCamera" type="firewire">
       <option name="id" value="0x80046020003007f"/>
       <option name="brightness" value="1024"/>
       <option name="exposure" value="128"/>
       <option name="sharpness" value="0"/>
       <option name="hue" value="128"/>
       <option name="saturation" value="128"/>
       <option name="gamma" value="128"/>
       <option name="shutter" value="2048"/>
       <option name="gain" value="0"/>
       <option name="iris" value="2048"/>
       <option name="focus" value="256"/>
       <option name="zoom" value="0"/>
       <option name="autoiris" value="true"/>
</camera>

<camera name="DesktopPlayback" type="playback">
       <option name="imageDirectory" value="/mnt/usb/faceSet"/>
       <option name="imageSuffix" value=".ppm"/>
       <option name="imagePrefix" value="faceTest"/>
       <option name="imageDevice" value="main"/>
</camera>

</camera-config>
```

10.1.3 Example Vision Config file

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE vision-config SYSTEM "vision-config.dtd">
<vision-config>

  <module name="LeftVisionModule">
       <comment>Module for the left camera only</comment>
       <camera name="leftCamera" />

      <operator name="LeftWriter" type="ImageWriter" camera="leftCamera"/>
  </module>
```

```
  <module name="RightVisionModule">
       <comment>Module for the right camera only</comment>
       <camera name="rightCamera" />

      <operator name="RightWriter" type="ImageWriter" camera="rightCamera"/>
  </module>

  <module name="DesktopVisionModule">
      <comment>Module for testing</comment>
      <camera name="DesktopPlayback" />
      <operator name="DesktopWriter" type="ImageWriter">
            <camera name="DesktopPlayback" />
      </operator>
  </module>

  <plugin path="/home/fwph/code/wurde-install/lib/libImageWriter.so"/>
  <plugin path="/home/fwph/code/wurde-install/lib/libImagePublisher.so"/>
  <plugin path="/home/fwph/code/wurde-install/lib/libDataCollector.so"/>
  <plugin path="/home/fwph/code/wurde-install/lib/libStereoDataCollector.so"/>

</vision-config>
```

10.1.4 Example Laser Config file

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE laser-config SYSTEM "laser-config.dtd">

<laser-config>
     <device port="/dev/ttyR1" name="TomPLS" type="PLS"/>
</laser-config>
```

10.1.5 Example RFlex Config File

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE rflex-config SYSTEM "rflex-config.dtd">

<rflex-config>
     <device port="/dev/ttyR0" name="TomRFlex" type="B21r"/>
</rflex-config>
```

10.1.6 Example Joystick Config File

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE joystick-config SYSTEM "joystick-config.dtd">
<joystick-config>

      <joystick name="Saitek P990 Dual Analog Pad">
            <axis id="0" name="rotate"/>
```

```
        <axis id="1" name="translate"/>
    <axis id="2" name="quicktilt"/>
    <axis id="3" name="quickpan"/>
    <axis id="4" name="pan"/>
    <axis id="5" name="tilt"/>

    <button id="0" name="sound1"/>
    <button id="1" name="sound2"/>
    <button id="2" name="sound3"/>
    <button id="3" name="sound4"/>
    <button id="8" name="sound5"/>
    <button id="9" name="sound6"/>

    <button id="4" name="stop"/>
    <button id="5" name="quit"/>
     </joystick>

</joystick-config>
```