Cooperative Robotic Simulator: Environment Control Panel

Aaron Chavez
Kansas State University
mchav@ksu.edu

1. Overview

The Cooperative Robotic Simulator (RoboSim) is a project designed to emulate real-life scenarios involving teams of robotic agents. The goal for RoboSim is to create a robustly designed artificial environment, in which various robot control algorithms on various robots can be tested. The program is broken up into five primary sections, as well as a few additional support libraries.

At the center is the environment module, which serves as the model for the world that the robots see and manipulate. Tied to the environment is the communications package, a method set that handles the various forms of message passing between agents. The robot module represents the actual robots, and contains data describing each robot's physical properties. It also serves as a hardware emulator, taking a real robot's control code and matching the API line-for-line. The viewer module utilizes Java3D to dynamically display the environment on-screen. Currently a special add-on to the viewer is supported, which allows for "seeing" objects' infrared emissions. The environment-building module is an autonomous tool for creating environment files, which are worlds that are loaded into the environment.

The portion of RoboSim for which I am responsible is the control panel, a GUI that provides an anonymous user a simple means for manipulating the environment. From the control panel, it is possible to load environment files, start and stop a simulation, and change numerous properties in the environment. These properties include network options, factors in the simulation, and communication settings.

2. Design

(*Please refer to Figure 1 for a class diagram of the controlpanel package.)*

The control panel is tied directly to the environment model, as well as the CRSViewer. The relationship between a control panel and an environment is "many-to-one". An environment may have an arbitrary number of control panels, but a control panel has a single environment. However, an environment does possess at most one master control panel, which is the only control panel capable of actually modifying the environment. The sole purpose of the other control panel(s) would be to facilitate the creation of viewers, etc.

This interaction between environment and control panel is accomplished through basic Java networking. An environment, upon creation, establishes a server on a designated port. The control panel operator is expected to have knowledge of the port on which to search for the server. Upon making a connection, the environment creates a new thread, with the purpose of listening for requests from one particular control panel, and handling said requests. The additional thread does not adversely affect performance, even in the case of multiple control panels, because it is only in a state of waiting while performing a read on its input port. This is a blocking read that allows the thread to sleep.

The runnable class in this thread, a RequestHandler, receives EnvironmentRequest objects from the control panel. From the EnvironmentRequest, it gathers the name of the method to invoke in the environment, and the parameters to use. The name is gathered by matching a lowercase string in the request to the lowercase form of the corresponding environment method name. So, supporting a method with this protocol is as simple as adding an additional check in the RequestHandler for string equality. So if the string equals "getmillisperstep", the function getMillisPerStep() is invoked.

On completion, the RequestHandler then issues an EnvironmentMessage back to the control panel, which in turn has a separate thread specifically for environment messages. At this point, some synchronization occurs. The environment has but one type of message to receive from a control panel: request for a method invocation. The control panel, however, must accommodate two types of data sent from the environment: success/return values from the methods it requests, and status updates. Status updates are generic strings containing changes in the environment state, errors, warnings, etc., sent at the environment's discretion. To this end, the ControlPanelClient, the class responsible for message handling on the control panel side, runs in one thread, but the main control panel thread calls the method that requests environment functions be performed (callEnvironmentMethod()). The ControlPanelClient's line of execution is as follows:

1. Initial state – sleeping on a blocking read, waiting for a message from the environment.
2. When a message is received, it is either a status update or a response from an environment method call. The EnvironmentMessage object has a field that denotes which type it is.
   a. If it is a status update, it is sent to the control panel to display.
   b. If it is a response, then the control panel is currently waiting. The return value is placed in a local variable called currentResponse, for the control panel thread to retrieve. The ControlPanelClient acquires the responseLock, sets currentResponse equal to the current response, then notifies the waiting control panel thread. The client is notified that the value has been read by the control panel, after currentResponse has been set to null. If, for some reason, the client receives a second response before the first has been read by the control panel, it waits on the responseLock until notification from the control panel.

It is worth noting that the synchronization employed in the ControlPanelClient is not technically necessary. The control panel runs as a single thread, and only this thread can call environment method requests. Then, it is deterministic that a second method will be requested only after the first is resolved. However, this implementation is more robust, and does not depend on the invariance of the remote environment's behavior, which may change as the scope of the project grows.

While the interface between the environment and the control panel has been developed to a reasonable, network-capable level, the viewer and control panel do not have such a relationship. It is intended that a control panel's 3D viewer will reside on the same machine and display screen as the control panel. Currently, the control panel creates a 3D viewer and allows it to function independently, establishing its own connection with the environment. The interface, as it exists, is not sufficient to handle data exchange between the viewer and control panel, which will be necessary for object selection (discussed later).

3. Future

The control panel's current incarnation handles all of the tasks of managing a simulation that are absolutely necessary, and in a very streamlined fashion. However, there is certainly a potential for growth, especially a few valuable capabilities that should be implemented in the near future.

Eventually, one feature that should exist in the control panel is a function for viewing object properties. A user could click on an object (a robot, a target, etc.) in the viewer, and the control panel could display a window with vital information about that object. The process would be three-fold, with the first step being the viewer's responsibility. It must determine what object was selected, by utilizing a process called "picking". Then, some data would be relayed to the control panel, identifying the object. The control panel could query the environment for additional information on the selected item, and, finally, display it in a window.

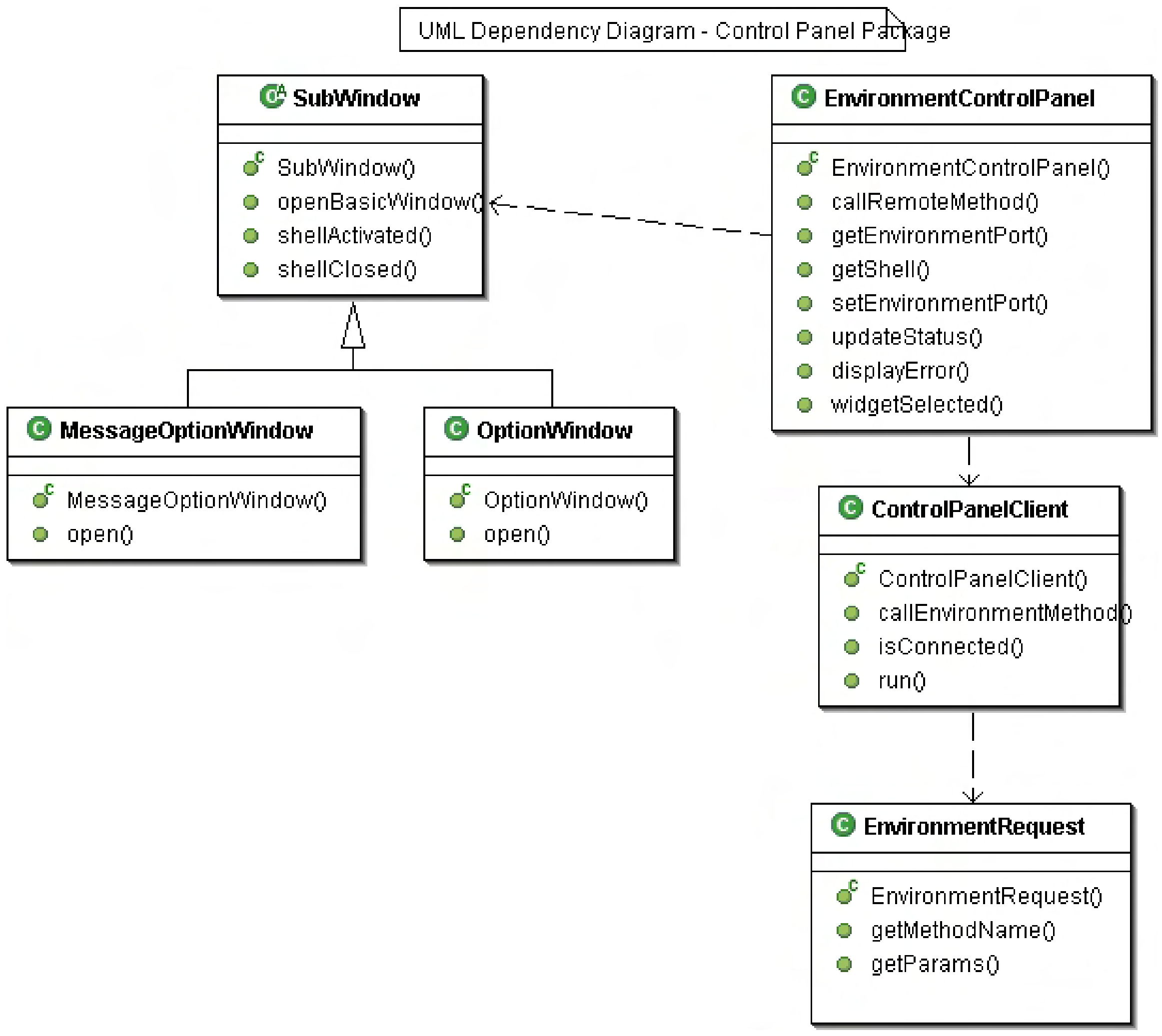
In order for this to be feasible, the viewer must have some means of communicating with its control panel owner (when applicable). Presuming that the control panel and viewer will co-exist on one machine, a constructor for a 3D viewer that includes a reference to the control panel would suffice. The CRSViewer could call a public method in the control panel after discerning the picked item. For a viewer on another computer, a networking interface similar to the control panel-environment setup would be needed.

The control panel has the capacity to control its environment completely, but is dependent on the methods that the environment provides for it. Extending the panel's degree of control will be a matter of extending the environment model's ability to control itself. Still, one possible improvement to communication between the two would be to allow multiple method requests or responses in a single communication. This could improve performance, which may become an issue with an "overworked" environment.

At present, all control panels are created with full functionality. It will be worthwhile to pursue the development of different types of control panels. Most will have limited potential to manipulate the environment, based on what is deemed appropriate for a given user. A secondary control panel might not be able to adjust environment settings, but rather only retrieve information from the environment. A robot control panel might be designed to handle and follow a specific robot. Hiding certain options from certain users will enhance security and stable performance.

These suggestions represent what is in store for the environment control panel. As RoboSim and its various aspects mature, the control panel will need to harness its power and make it available to inexperienced and informed users alike.

Figure 1


UML Dependency Diagram - Control Panel Package
SubWindow
SubWindow()
openBasicWindow()
shellActivated()
shellClosed()
EnvironmentControlPanel
EnvironmentControlPanel()
callRemoteMethod()
getEnvironmentPort()
getShell()
setEnvironmentPort()
updateStatus()
displayError()
widgetSelected()
MessageOptionWindow
MessageOptionWindow()
open()
OptionWindow
OptionWindow()
open()
ControlPanelClient
ControlPanelClient()
callEnvironmentMethod()
isConnected()
run()
EnvironmentRequest
EnvironmentRequest()
getMethodName()
getParams()