Comp 533 - Assignment 1: Distributed Non-Blocking Halloween Simulation Project

Date Assigned: Thu Jan 12 2017

Completion Date: Thu Feb 2, 2017, 11:55pm

The goal of the assignment is allow you to learn and use Java's non-blocking mechanisms to write a "realistic" distributed program. You will take an existing non-distributed program - a Halloween simulation - and create a distributed version of it. It should be easy to interface with this simulation. The assignment will expose you to threads, sharing of state between threads, and NIO sockets.

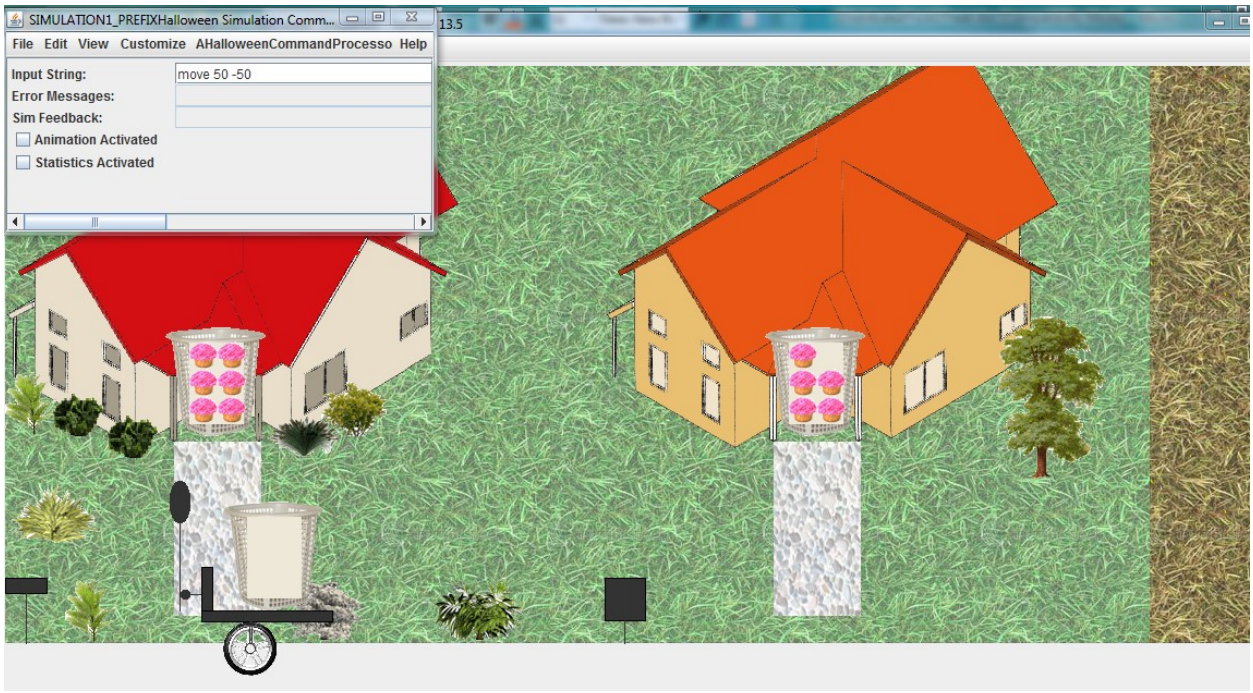
Non Distributed Simulation

The non-distributed simulation is a project I gave to my Fall Comp 401 (http://www.cs.unc.edu/~dewan/comp114/f10/) class. You will work with the code of one of the students in the class – Beau Anderson – who created a particularly nice simulation. The simulation created two windows, a command window, and a graphics window. The command window is used to manipulate the objects shown in the graphics window.

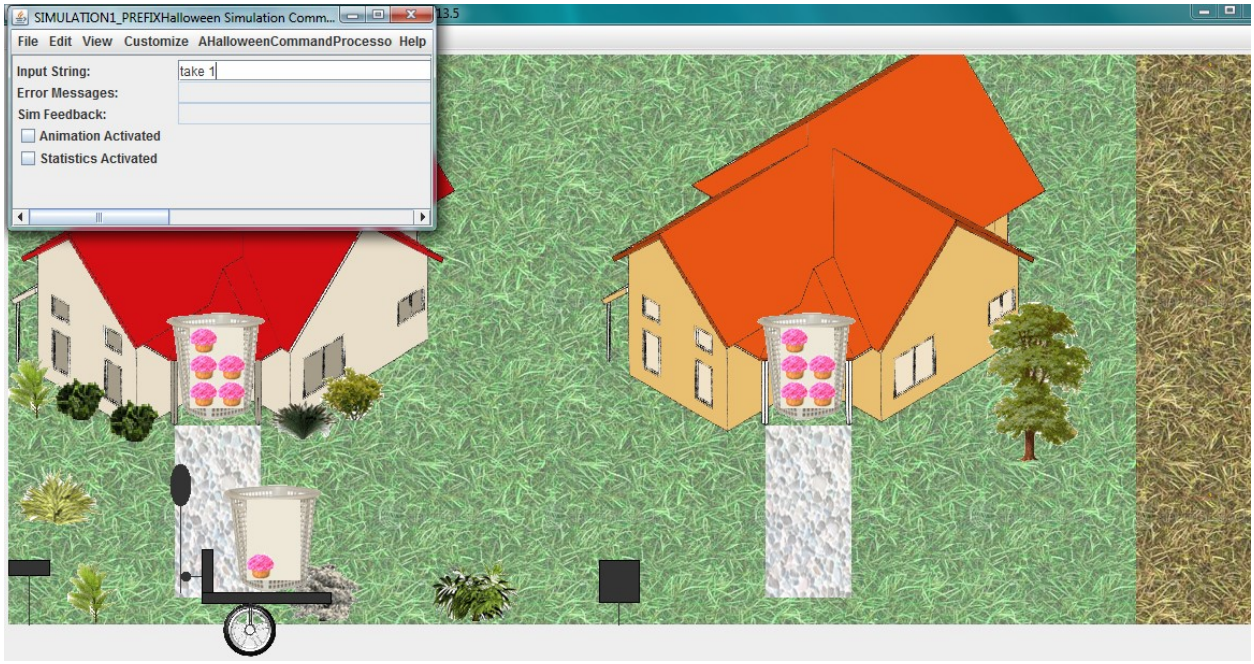
As shown in Figure 1, each graphics window consists of (a) one or more houses, each with a path and a candy container; and (b) a movable avatar with a candy container. The move command can be used to move the avatar in both the x and y directions, as shown in Figure 1.

If the feet of the avatar are in the path of a house, then the take and give commands can be used to transfer candies between the house and avatar containers. In addition, commands are provided to add and remove a house in the simulation, and undo or redo previous commands. The following is the syntax of the commands:

<Command> → <Move Command> | <Add Command> | <Remove Command> | <Take Command> | <Give Command> | <Undo Command> | <Redo Command>
<Move Command> → move <number> <number>
<Take Command> → take <number>
<Give Command> → give <number>
<Add Command> → addHouse
<Remove Command> → removeHouse
<Undo Command> → undo
<Redo Command> → redo



(a) Avatar moves into path of leftmost house



(b) Avatar takes one candy

Figure 1Beau's Non-Distributed Simulation

You can ignore most implementation details of the non-distributed simulation. All you need to know is how to trap (using an observer) a command entered by the user and execute such a command programmatically. I have created an Eclipse project, Coupled

Halloween Simulations, which shows how these steps are done. The program TwoCoupledHalloweenSimulations in the project couples two simulations in the same process – your task will be to couple simulations in (possibly an arbitrary number of) processes – this means you must run a different program for each process. The project references both Beau's code and a user-interface library I wrote, ObjectEditor, which is used by Beau's code. All three pieces of code are available from the course home page. The Eclipse project must be uncompressed. ObjectEditor and Beau' code are also compressed, but can be used in this form as they are referenced as external libraries in the Eclipse project. You need to change the paths in the project in order to use them correctly.

The coupled simulations use public methods in the following interface:

```
public interface HalloweenCommandProcessor{ … }
```

A relevant method in this interface is:

```
void processCommand(String newInputString);
```

This method asks the simulation to invoke the command, `newInputString`, on the local simulation. This method must be invoked by a simulation process in response to commands executed in other simulation processes.

Another relevant method in the interface is:

```
void addPropertyChangeListener(PropertyChangeListener newListener);
```

It can be used to listen to trap commands entered by a user by registering an instance of predefined Java type java.beans.PropertyChangeListener. The simulation notifies each registered instance of the new command by calling the:

```
void propertyChange(PropertyChangeEvent evt)
```

method in the instance. The new command is announced as the new value of the property name, "InputString". New values of this property are needed to send user commands to remote processes.

Two important methods in this interface that the coupled simulations project does not show are:

```
boolean isConnectedToSimulation();
void setConnectedToSimulation(boolean connectedToSimulation);
```

Normally, the command processor changes the simulation and then notifies observers. To support atomic broadcast, you will need to disable local processing, and send the command only to the observers. By calling the setter with the false value, you can disable the local processing. This feature is needed in a later part of the assignments.

An instance of the command processor can be created using the static method BeauAndersonFinalProject.createSimulation() – its use is illustrated in the coupled simulation project.

Beau created the project using an older version of the ObjectEditor library: oeall17.jar. The latest version in oeall22.jar, but it gives warnings, which oeall17.jar does not. These can be disabled using the following call:

Tracer.showWarnings(false);

Both versions are on the web site. oeall17 does not let you input the same string command twice, while oeall22 does. Oeall22 is needed for the traceable functionality, so use it unless it gives you problems. Let me know about the problems.

Distributed Simulation

You should create a distributed version of this simulation that involves an arbitrary number of simulation processes. The simulations created by these processes are coupled similarly to the way the two simulations are coupled in the demo project – after a user submits a command to one simulation, the command should be executed by each of the simulations in some order chosen by you. You should use the Java NIO library for communication and all I/O including the connection call should be non-blocking.

I used the following tutorial to understand it. http://rox-xmlrpc.sourceforge.net/niotut/. You can download and modify the example program or write it from scratch. One feature the library does not show is the keyFor() method in a SocketChannel(), which given a SocketChannel and Selector, returns the SelectionKey instance, on which you can execute interestOps() operations.

You can assume a simulation process does not join a session dynamically, that is, joins a session after a user of some existing process in the session has entered a command. More simply, your implementation need not support latecomers (whose displays are consistent with those who joiner earlier).

You must consider two important issues in your implementation:

(a) How do simulation processes know about and communicate with other distributes processes? You can implement a server that knows about all of the client processes and communicates the information. You are free to implement something more sophisticated, such as peer-to-peer (P2P) communication (direct communication between client processes), with the server process simply keeping the addresses of the clients. However, you will have to use a server for the atomic broadcast component of this project.

(b) How do the interest ops registered with the NIO selector change? Make sure you carefully consider the transitions before you implement them.

You can ignore but are free to address failures, access control, and race conditions.

Downloading and Installation Instructions

First get oeall22 and install it in some location.

Next get the zip beau_project.zip , uncompress it, and create a new Java project from the uncompressed version. In Eclipse this means Project>New Java Project> , uncheck default location and then browse to the location where the project is downloaded. Right click the project, Properties>Build Path. Go the Projects tab and remove any referenced projects. Then go to the Libraries tab and remove all libraries other than the JRE. In this tab, do Add External Jar and reference oeall22. Run main.BeauAndersonFinalProject. Type move 100 -100 in the command window and see if the avatar moved.

Now get CoupledTrickOrTreat.zip and go through the same process as above. However, this time, in the Project tab, add Beau's project to the build path. As before you need oeall22. Run coupledsims.TwoCoupledHalloweenSimulations. Two command windows and simulations should show up. Typing in one window should change avatars in both windows. This project tells you how to take intercept a command from one simulation and inject it in another simulation in the same address space. Your task in this project is to inject the commands in a remote simulation.

Finally clone the following Git repository https://github.com/pdewan/GIPC.git. In Eclipse do File>Import>Git>Projects from Git>Clone URL and paste the link above and follow the instructions to add the project. Now go through the process of removing any projects and libraries in the build path and adding oeall22.
The niotut package contains the code form an NIO tutorial (http://rox-xmlrpc.sourceforge.net/niotut/) Run niotut.NIOServer. Next run niotut.NIOClient. You should see "Hello World" in the client console. You can also run niotut.NIOClientGoogle to get a google page echoed back.

Your task is to combine the CoupledTrickOrTreat project with the code in the NIO Tutorial. I would first change the NIO tutorial code to create a collaborative program that echoes HelloWorld from one client to all of the other clients. Next I would change the program to interact with Beau's simulations.

Key Components of Basic Implementation

Each simulation process will create a selector thread – the thread that makes the select() call on a Selector instance. In addition, a simulation process should create a remote-operations thread to process commands received from other simulation processes. Use an instance of Java BlockingQueue to communicate between the selector thread and the remote-operations thread. Use the setName() operation on a Thread object to give these two threads names that indicate their roles. There are at least two other predefined threads involved in each simulation process – the main thread, which invokes the main method, and the AWT thread (whose name begins with AWT-EventQueue), which processes GUI actions – processing GUI input and painting the GUI.

The main thread in each simulation process will make a connect() call. In addition, it will enqueue a request for the selection thread to process the connection completion event, and then invoke the wakeup() operation on a Selector . The woken up selector thread will

consume this event by invoking a finishConnect() operation (on a SocketChannel) and an appropriate interestOps() operation (on a SelectionKey).

A command input in one process will send the command to other simulation processes.

The AWT thread in the sending process will enqueue a write request for the selector thread, and invoke interestOps() and wakeup(). The woken up selector thread will invoke the write() operation on the SocketChannel representing the server, and an appropriate interestOps() operation.

The selector thread in the server process will execute the read() operation and communicate the read data to a broadcast thread. The broadcast thread will enqueue write requests on the SocketChannel objects representing the clients to which the data has to be sent.

If you are not using a server, then the difference is that the sending process will send the data, not to the server, but the other clients directly.

In either case, the selector thread in the receiving simulation process will execute the read() operation to receive the data and enqueue the received data for the remote-operations threads in an instance of Java BlockingQueue, which will be dequeued by the remote-operations thread. The remote-operations thread will process a dequeued command by executing processCommand() on HalloweenCommandProcessor. The remote-operations thread should block if the queue is empty; thus it should invoke the take() queue operation. However, the selector thread should not block if the queue is full (why?); thus it should invoke the add()method rather than the put() method.

Submission Instructions

Create a YouTube video demonstrating the working of your program and submit your code on Sakai (the assignment will be created before the due date). The video can be submitted as a private or public link in piazza. If Piazza and YouTube make you uncomfortable, you can create a shared folder or use email. I used the following software to make videos: http://sourceforge.net/projects/camstudio/files/legacy/Camstudio2-0.exe/download.

Your demo should have the following components, which try to ensure you compose a correct solution and, more important, understand both the code you borrowed and the code you added.

Single-Step Demo

You demo should involve at least three simulation processes.

Use breakpoints and the Debug window in your programming environment to demonstrate thread communication in your client.

In particular, show that before any input:

- The main thread in the client invokes connect() on a SocketChannel..
- The selection thread in the client invokes finishConnect().

Next show that when a new command is executed in a client simulation:

- The AWT thread in the inputting simulation executes the propertyChange() method.
- The selection thread in the inputting simulation executes write() on a SocketChannel().
- The selection thread in a receiving simulation executes read() on a SocketChannel.
- The remote-operations thread in a receiving simulation executes the processCommand() method on a simulation.

You can put breakpoints in only one receiving process.

Tracing using Traceables

You should also trace, in the consoles, the key steps in your implementation described above. To do so, use the traceable classes in the GIPC trace.port.nio package in the GIPC project (which were built before and have more functionality, for the purposes of this assignment, than the Java logger package). This package has several abstract and concrete info traceable classes.

Each concrete traceable class corresponds to a type of operation you need to trace. For example, the traceable class SocketChannelConnectInitiated corresponds to the connect() call on a socket channel. A concrete traceable class has the method newCase(). This method should be invoked *after* the associated operation is executed to trace the operation. The first argument of newCase() should be assigned the Java object that executes the traced operation. The remaining arguments of newCase() should be assigned the target and parameters of the operation. The following code illustrates how operations are traced:

```
// operation to be traced
socketChannel.connect(socketAddress);
// trace statement
SocketChannelConnectInitiated.newCase(this,
                                      socketChannel, socketAddress);
```

Here, the operation to be traced is connect(), which is invoked by the current instance (this) on the target, socketChannel, with parameter socketAddress. The newCase() static method executed on the corresponding concrete traceable class, SocketChannelConnectInitiated, takes as arguments the invoker (this), the target (socketChannel), and the parameters ({socketAddress}). It is invoked after the associated operation finishes execution.

By default, the newCase() statement performs no logging. The following code in a client shows how to enable logging of info traceable classes, turn it on for selected info classes, and then control what is printed in each trace.

```
      Tracer.showWarnings(false); // do not show oeall22 and other
warnings
      Tracer.showInfo(true); // log info traceables
      // but only when the newCase() method invoker is one of the
following classes
      Tracer.setKeywordPrintStatus(TrickOrTreatNioClient.class, true);
      Tracer.setKeywordPrintStatus(AnNIOSimulationOutCoupler.class,
true);
      Tracer.setKeywordPrintStatus(AnNIOSimulationInCoupler.class,
true);
      // Show the current thread in each log item
      Tracer.setDisplayThreadName(true);
      // show the name of the traceable class in each log item
      TraceableInfo.setPrintTraceable(true);
      // show the current time in each log item
      TraceableInfo.setPrintTime(true);
```

The showInfo() call provides a coarse-grained mechanism to toggle between displaying and not displaying the trace; you do not have to change any other code to do so.

This example assumes that each client has separate objects for running the selector thread, trapping remote simulation events, and receiving remote events. All three objects invoke newCase() operations; hence their classes are listed in the `setKeywordPrintStatus()` calls. I would recommend that you also create separate classes for these three functions.

During your demo, you should have tracing on and briefly show the traces for the sending client, a receiving client, and the server so that the LA can verify that the relevant calls have been traced.

Creating Inconsistencies

So far, we have assumed that a simulation executes a command before sending it to others. This can cause inconsistencies if two simulations execute commands concurrently, that is, each simulation executes its local command before executing the remote command. Such inconsistencies can occur only when the two concurrent commands do not commute. Use breakpoints and/or delays to create such an inconsistency in the demo. You can look at my demo, if necessary, to do this part of your demo.

Atomic Broadcast

It is possible to remove such inconsistencies by supporting atomic broadcast, which ensures that all processes execute the same sequence (rather than set) of operations. A simple approach to implement atomic broadcast is to postpone execution of a local operation until it is echoed back from a common server. This means that both the client and server must change to support such broadcast. Define runtime parameters to allow

this change to occur dynamically. Use the setConnectedToSimulation() in the client to postpone local processing. Use the client and server parameters to demonstrate how atomic broadcast leads to consistency.

Timings

Programmatic Input: The demo above assumes you provide input interactively. Extend your your implementation to allow input commands in a sending simulation to be provided programmatically, from the main thread.

In receiving simulations, you have used the processCommand() method of the simulation command processor to programmatically change its state. As you have noticed, this method does not fire events, which is useful when you are replaying a command broadcast by another site. The method setInputString() has the same behavior except that it also fires (vetoable) events. Use this method to enter a series of commands in a a sending simulations.

Your simulations communicate a byte stream rather than a sequence of independent chunks. As a result, multiple commands can be received in a single bytebuffer read, which must be separated into individual commands before they can be delivered to the command processor. At the sending site, add a special marker character (such as ",") between commands to allow them to easily split at the receiving site. See my demo video to better understand the need for such marking.

Local Execution: Extend your implementation to allow pure local execution of commands. In this mode, your client behaves like the original simulation you extended – it does not transmit any message in response to a command. Define a local-execution parameter that can be changed at runtime (using the console or a GUI) that determines if input is sent to remote processes or not.

Timings at input site: After adding the above two features (local execution and programmatic input), the next task is to use them to time the execution of a series of input commands at the site issuing these commands under various conditions. Time the execution of a series of 500 input commands under the following conditions:

1. Local execution: the simulation executes the commands locally without sending any messages to remote sites.
2. Basic Distributed 3-User: Same as above except that commands are broadcast using non-atomic broadcast to two other simulations.
3. Atomic Distributed 3-User: Same as above except that the broadcast is atomic.

Ideally, you would like to visually see the effects of these commands on the screen. If you are executing the move command, you can give it a small increment (e.g. 1) and provide separate commands to move in the x and y direction.

Your video should show the results of the three performance experiments.