HW3 – Reliable Inter-process Communication

Estimated time: 20-28 hours

Objectives

- Gain experience with creating reliable communication from underlying unreliable communications
- Master one typical architectural for reliable communications
- Become more familiar with unit testing techniques

Overview

During this assignment, you will implement one process (a player) for a distributed game, focusing on the software components that support reliable communications. In other words, you will be implementing middleware components that provide a convenient platform for the inter-process communication that is specific to our distributed application. This middleware will include the classes described below or at least the functionality represented by those classes. It may include other middleware functionality or supporting classes, as you see fit. Figures 1 -3 illustrate the beginnings of one possible design. Figures 1-2 illustrate the classes and relations from a logical structure perspective, where as Figure 3 illustrates the classes from a software component perspective. You are responsible for completing and refining the design, and are feel to change whatever you'd like as long as your middleware communicates with others and follows the communication protocols.

Communicator

This class is an abstraction for communicating with other processes via UDP-based messages. It should handle the basic send and receive operations for *Message* objects, or more precisely Envelop objects, which are wrappers for *Messages* with sender or receiver *End Points*). Each process in the system will have one *Communicator* through which it sends and received all messages.

Listener

A *Listener* object is responsible for grabbing messages (and more specifically, envelops containing messages) received by a *Communicator* (at least the primary communicator) and placing them into a *MessageQueue* for later processing by a *Doer* object or its *Conversation Execution Strategy* objects. Note that the *Listener* object does not interpret or process the message beyond checking the message number and conversion id. If the message number is the same as the conversion id, then the incoming envelop

is supposed to start a new conversation, so it places it in the *Request MessageQueue.* Otherwise, the message is a following message for an supposedly existing conversation. The listener should look to see if there is a *Conversation MessageQueue* for that conversation id, and if there is one, place the envelop in that queue. If there isn't a *Conversion Message Queue*, then that message is an erroneous message and should be ignored.

The *Listener* object is an active object, running on this own thread. It should try to get messages from the *Communicator* as soon as they are available so internal UDP buffers are not overrun.

Doer

A *Doer* is an active object that takes messages out of *Request MessageQueue* and starts the appropriate *Conversation Execution Strategy* for handling the processing of the request and the rest of the conversation (e.g., subsequent communications) as defined by the communication protocols. A *Conversation Execution Strategy* could be base class or interface in an implementation of the GoF (Gang-of-Four) Strategy Design Pattern. Specializations of this base class are specific to an agent and part of the Application Layer logic for the agent. By using a Strategy Design Pattern, the Doer doesn't actually have to know the logic for following the protocols. Initialization code for an agent can create specializations of the Conversation Execution Strategy and given them to Doer to user. The Doer simply needs to choose the right concrete strategy object based on message type. In other words, the Doers should contain as subparts one or more concrete *Conversation Execution Strategy* objects (one for each message type it has to handle). The classes for those concrete strategy objects should be defined and created in application-layer components, not the Common library.

BackgroundThread

This is a base class for *Listener* and *Doer*. It is a wrapper for whatever underlying framework (e.g., .NET or JDK) provides for threading. As a minimum, it should allow the agent to stop and start the *BackgroundThread.* It can also encapsulate all the attributes and any other behaviors that are common to the *Listener* and *Doers*.

MessageQueue

A *MessageQueue* object is a queue of messages that have been received by the *Listener*, but not yet processed by a *Doer* or *Conversation Execution Strategy*. Since a process's *Listener, Doer*, and *Conversion Execution Strategies* will be running as separate threads, a *MessageQueue* object must guarantee correct queue behavior in the present of concurrent access. One *MessageQueue* will be the *Request Message Queue*, and others will be *Conversation Message Queues*. There will be need to be a way for the Listener to know or discover the *Request Message Queue* and create new *Conversation Message*

Queue. The *Doer* will need to know the *Request Message*, and *Conversation Execution Strategies* will need to know or discover their respective *Conversation Message Queues*

Message Classes and Share Object Classes (Provided by Instructor)

These are classes that implement messages use in all of the communication protocol. The instructor will provide source for this classes in C# and Java. You may choose to implement your own message and share object classes, as long as you adhere to the communication protocols.

Envelope

An *Envelope* is a simple wrapper for a message that adds a *Public End Point*. For incoming messages, the *Public End Point* is the sender's end point. For outgoing messages, the *Public End Point* it the target receiver.

Using this communications subsystem and the communication protocol definitions, you will begin to design and implement the necessary resource managers and application components for your agents.

Instructions

Your development process needs to include the following activities:

- Design, implement, and test communication components
- Design, implement, and test the resource managers (does not need to be completed)
- Design, implement, and test the resource users (does not need to be completed)

For HW3, you need to thoroughly test at least your *Communicator*, *Listener*, and *MessageQueue* classes. (See the Basic Grading Criteria.) However, it is recommended that you to test other major components, as well. (See the Advanced Grading Criteria.)

Submission Instructions

Zip up your document and your entire solution into an archive file called CS5200_hw3_<*fullname*>.zip, where *fullname* is your first and last names. Then, submit the zip file to the Canvas system.

Grading Criteria

Basic Criteria (worth up to 90 points)	**Max Points**
A quality implementation of the classes that make up the communication subsystem (i.e., Middleware)	30
Thorough unit testing of the Communicator, Listener,	30

MessageQueue classes.	
Some initial work on the implementation of some of the Conversion Execution Strategies. Try to have approximately 30% of the anticipated strategies completed. You don't need to test these at this point.	30
Advanced Requirements (Worth up to 15 points)	**Max Points**
Thorough unit testing of at least two more classes, e.g. Doer, other supporting classes, etc.	15
A feature that allows the server's IP Address and Port to be specified on the command line.	5