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
- Become familiar with logging techniques and a logging tool

Overview

During this assignment, you will begin to implement your agents for the distributed game, focusing on the software components that support reliable communications. To minimize your overall development costs, you will implement a reusable, extensible communications subsystem with the following classes of objects. Also, see Figures 1 -3. 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.

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 *Conversation Execution Strategy* on handling the processing of the request and the rest of the conversation (e.g., subsequent communications) as defined by the communication protocols. Doer should contain as subparts one or more *Conversation Execution Strategies*, but concrete instances of those strategies should be defined and created in application-layer components, not the Common library. The concrete strategy (specializations of *Conversation Execution Strategies* define a process's behavior for handling a conversation.

BackgroundThread

This is a base class for *Listener* and *Doer*. It encapsulates all the attributes and behaviors that are common to both classes of objects.

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

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 port them to any other language of your choice. You may also choice to implement your own message classes, as long as you adhere to the communication protocols.

Envelop

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

Common Resource Classes

These are classes that implement common data structures for shared resources, e.g. *Tick*, *WhiningTwine*, *Excuse*, etc. The instructor will provide implementations for these classes. You can specialize or wrap them to adapt them to your own application-layer logic.

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.

Figures 4-6 illustrate the sequence of messages (in three parts) of a single "GetResource" conversation.

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 85 points)	**Max Points**
A quality implementation of the classes that make up the communication subsystem.	30
Thorough unit testing of the Communicator, Listener, MessageQueue classes.	25
A quality design and initial implementation of your resource managers and resource users. Note, it is not necessary to have all of your application-level functionality complete, but you should have at least 30% working and demonstrable.	30

Advanced Requirements (Worth up to 15 points)	**Max Points**
Thorough unit testing of at least two more classes, e.g. Doer, supporting protocol handlers, resource classes, etc.	15
A feature that allows configuration parameters to be specified on the command line	15