DEVELOPMENT ISSUES FOR A NETWORKED, OBJECT ORIENTED GAMING ARCHITECTURE (NOOGA) TEACHING TOOL

Owen L. Astrachan, David Bernstein, Andrew English, Benjamin Koh

Abstract *— We describe the outcome and experience of trying to develop an architecture and framework for a Networked Object Oriented Gaming Architecture (NOOGA). The aim of this project was to create an easily extensible framework that facilitates teaching students about object oriented design, design patterns, and software engineering in an interesting context. Our original goal was to develop a game serve to serve multi-player games such as battleship, dots, and Boggle. We planned to implement these three games, but to design an extensible server that would permit new games to be added to the server framework. The NOOGA framework would be studied and used from both a client and server standpoint. Students would first develop clients that would connect to the server to play a specific game and would, in more advanced courses, add new games to the server. This paper will focus on the development process and design decisions made during development. We initially decided to implement the client/server model using Java's Remote Method Invocation (RMI). Later, we developed a protocol for communicating between the server and clients using low-level sockets to facilitate client implementation in C and C++. We describe the success in developing an extensible server, but describe failures in ensuring robustness in the face of faulty clients. Since the NOOGA server was intended to facilitate student development of networked clients, our original goals were not fully realized.*

Keywords — Networking, computer science education, games, Java

INTRODUCTION

As part of a project to introduce design patterns [1] into the undergraduate curriculum [2], [3], [4] three undergraduates worked during the summer of 2001 to develop a Networked Object Oriented Gaming Architecture (NOOGA). NOOGA was intended to serve as a framework for use in several courses including our second course for majors (data structures), a required course in software design, and an elective course in networking. Previous work [3] was successful in developing a framework that served as the foundation for several assignments.

The genesis for NOOGA was a series of successful assignments used to introduce the Model-View-Controller [5] (MVC) architecture to students in the software design course. These assignments required students to develop a non-networked architecture and framework for implementing simple board games. In these projects, named OOGA (Object Oriented Gaming Architecture) each team of students had to develop a suite of four games using MVC with simple but extensive and expressive GUIs as a required component of the games. Games ranged from single-player games like hangman to multi-player games like dots. In the former, the player is playing a kind of solitaire game whereas in the latter several players are interacting by passing the mouse back-and-forth or are playing against the computer.

This work was supported by NSF grant CAREER 9702550 and NSF CRCD 0080078, Department of Computer Science, Duke University, Durham, NC 27705, ola@cs.duke.edu

Successful student projects required the development of a framework with reusable components so that, in theory, new games could be added by consulting the student-developed OOGA documentation. Student feedback for this project has traditionally been overwhelmingly positive. By designing several games, students could see the benefit of a well-documented and well-designed architecture. Using games, graphics, and GUIs (the 3G's of student satisfaction) was an important part of the success of the assignments from a student perspective, but student evaluations indicate that OOGA was instrumental in understanding MVC and design patterns in general. Because many groups first developed two games, then designed the architecture and re-implemented the games using the new architecture, students also realized the benefits of agile methodologies [6], [7] and refactoring [8].

Students were motivated by the creation of near (to them) commercial-quality games that they could play themselves as well as show off to their friends. Also, since students were free to pick which games they implemented and what features were incorporated in each game, they were responsible for project specifications and requirements to a larger extent than in previous assignments in the course.

In essence, the existing OOGA project challenges students to write clients for a suite of games. As part of the work we report on in this paper, we were to write a NOOGA server so that in the future students would be able to add networked games to the OOGA framework. The high-level requirements for the NOOGA server were to (1) wait for clients to connect to play a specific game; (2) allocate a thread to handle the game when a sufficient number of clients had connected; (3) be robust in the face of poorly-designed clients and interrupted network connections.

In this paper we discuss the design and implementation of NOOGA at levels ranging from specification to implementation. We report on the success and failure of this project from our points-of-view as developers as well as from a pedagogical perspective.

DEVELOPMENT ISSUES

The first major decision we had to make was the target language for implementing NOOGA. We chose Java because of its extensive API, because of our familiarity with the language, and because it was a viable choice for use in our curriculum. We anticipated that deploying a Java-based server at other schools would be straightforward if Java was used, e.g., compared to C++ or C#. The choice of language was relatively easy to make.

Client/Server Communication

Early in the development process we had to choose a method for implementing the actual communication between the server and clients. Our two choices, initially, were:

1. Use Java's Remote Method Invocation (RMI) functionality.
2. Open up a socket between the server and each client and pass plain text between them using some sort of protocol we would need to develop.

Using Java's RMI was attractive from a programming perspective. Both server and client publish a list of functions that each allows Remote Objects to make calls on. As Sun's RMI tutorial says "*The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM. RMI provides for remote communication between programs written in the Java programming language.*"[9]

For example, among the list of functions the server publishes is `public void JoinGame(String GameType)`. When a client is run it uses a simple RMI command to get a reference to the server and then calls `myServer.JoinGame("Battleship")`. The advantage of RMI is that it is extremely easy and simple to use. For example, in some games there is a lot of information about the board that needs to be passed back in forth between the server and clients. Using RMI, we could simply publish the definition of a `ComplicatedBoard` object, and then pass a `ComplicatedBoard` object between the client and server. One disadvantage of RMI, however, is that it does not support communication between a server and a client that are written in different languages. Another is that configuring RMI to work independently of CLASSPATH variables is complex and often leads to errors that are difficult to debug. This makes it more difficult to develop a server that can be deployed in any setting at other institutions.

The other communication option was to open a socket between the server and each client and push ASCII commands that follow some specified protocol between the clients and the server. For example, when the server wanted to let a Dots client know it is time to move the server could write `"move"` to the client whose turn it is to move. Then, when the player on the client side had made a move, the client would report back information on where the player moved. For example, the client could write: `"dot 1 connected to dot 2."` This is different from RMI because instead of calling methods directly, client and server communicate simply by writing information down a socket, and reading from the socket according to a specific protocol. The big advantage of this type of communication is that the server and client could be written in different languages. We could write the NOOGA server in Java, but students could write the client in C, C++, Java, or any language with a socket library.

One downside to using sockets is that it becomes unwieldy to pass a large amount of information back and forth. Also, developing a general protocol is potentially complicated. RMI doesn't have this problem. When using RMI, all we need to do is have the server or client invoke a function call on the other side, and pass as a parameter any type of complicated object. For example, the server could call `myClients[1].doTurn(ComplicatedObject)`. With the definition of `ComplicatedObject` published on the web for the world to see, it would then be up to the client to use this object in any way it saw fit.

We decided to begin the project using RMI as our method of client-server communication. This seemed like the easiest choice, and since we were unsure of how difficult the project was going to be, we determined it was better to end up with something functional, but with fewer features. The ability to write clients and servers in different programming languages wasn't worth the ease of use we would have had to give up. With this in mind, we set about actually developing the server, and some sample clients to test out the server.

NOOGA Architecture

In this section we describe the NOOGA architecture for both server and clients and describe design decisions in implementing the architecture.

Server Architecture

We developed a general `NOOGAServer` class that prospective clients could register with to begin to play a game. A `NOOGAServer` object waits for clients to connect to it. After two clients had connected to play the same game, the server would wait 10 more seconds for more players, and then start a game in a separate thread to be played by all connected clients. More technically, the `NOOGAServer` would create a `Runnable` game-specific `GameHandler` to facilitate communication between the server side and the client side. After a game-handler had been spun off, the `NOOGAServer` would no longer have any knowledge of that specific game. The `NOOGAServer`'s role is to wait for clients to connect to it and to start a `GameHandler`. We started off by implementing two games: Dots and Boggle.

Cient/Server Communication using RMI

When two or more Dots players have connected to the `NOOGAServer`, it creates a `DotsGameHandler` to handle the game. Similarly, the server creates a `BoggleGameHandler` when clients have connected to play Boggle. Both of these classes inherit common functionality from the superclass `NOOGAGameHandler`. Each handler handles all communication between server and client, and dictates the flow of the game. For example, in dots the flow is:

1. `DotsGameHandler` tells all clients that a game is beginning, and how big the board size (e.g., via RMI call.)
2. `DotsGameHandler` informs [first] client to move, client handles player response.
3. Client tells server (e.g., via RMI) what move it is making.
4. After checking to see that the move is valid, `DotsGameHandler` informs all connected clients of player one's move.
5. `DotsGameHandler` informs next client it is their turn to move.
6. Repeat steps 2 through 4 until the game is over.
7. `DotsGameHandler` informs all clients that the game is over, and delivers final scores to all the clients.

We succeeded in testing and deploying a server that implemented the framework described above and wrote specific handlers for Dots and Boggle. Also, we wrote clients to play these

games, because testing a server without clients is relatively difficult.

Bulletproofing the Server

It became immediately apparent that we needed to bulletproof the server as many problems arose from faulty clients.

Server Should Maintain All Game State

In Dots, our initial design was to have a `DotsBoard` object that encapsulated the state about what moves had been made. Both the server and the clients would have access to the `DotsBoard.java` file that defines this class. Originally, we had the `DotsGameHandler` create an empty `DotsBoard` object. Then, the handler would pass this object to a client when it was the client's turn. The client would then modify this object to indicate where it had moved, and send the modified object back via RMI when its user had completed its move. Next, the handler would, without any error checking, broadcast this board to all the clients so they knew what the new board looked like. As far as implementation on the server side was concerned, this was very easy. The `DotsGameHandler` didn't have to know anything about the board or how the game was progressing (except it did need to know when the game had ended). Rather, the `DotsGameHandler` originally just passed this `DotsBoard` object between the clients. As we were developing clients, however, it became clear very quickly that even a good-intentioned client could easily do undesirable things such as make illegal moves, or make no move at all. A malicious client could easily claim that it had just taken all the moves and won the game after the first turn. Clearly, this was unacceptable. We decided to keep the real copy of a `DotsBoard` object on the server. Instead of asking for the entire board back after a client had taken a turn, we asked instead for only the specific move the client wished to make. The `DotsGameHandler` checked that move to make sure it was indeed a valid move. If it was, the move would be added to the official board kept by the `DotsGameHandler`, and the move would be broadcast to all clients so they could update their displays. If the move were invalid, the `DotsGameHandler` would ask the same client to choose another move. In this way, we resolved the problem of allowing clients to unfairly change the board in ways inconsistent with the rules of the game. Of course in retrospect this problem and solution seem obvious, and perhaps they would have been to more experienced developers.

Disconnecting Idle Players

While we were testing the system by playing each other using clients on computers situated in different rooms, one of the clients went into an infinite loop while it was that client's turn. The other players in the game waited impatiently for a few minutes before disconnecting from the game, and yelled across the hall to ask "what happened?" Clearly, this is undesirable behavior. The best way to handle an unresponsive player would be to kick that player out of the game, and allow everybody else connected to continue competing. So, we re-wrote the handler code to do just that. We installed a timer. If a player hadn't moved in a fixed amount of time, say 30 seconds, they would get a warning. If after more time had elapsed, say another 30 seconds, that player would be kicked out of the game and all players would be notified of this action. In this way, a client who had inadvertently gone into an infinite loop, or a player whose machine had disconnected would not be able to render the game unplayable for all players.

Client-Identity Verification

We next turned to client verification and authentication. During game play, for example, the `DotsGameHandler` waited for a remote client to invoke the `DotsGameHandler.turnOver(mov` function via RMI. The handler trusted that the client had just made its move, and proceeded accordingly. However, there was no way for the handler to know which client had actually invoked the turnover function. For example, suppose that client two is a malicious client. Whenever it is client one's turn, client two could still call `DotsGameHandler.turnover(moveInfo)` passing in a poor move. The handler would assume that client one had just made a move! In our initial design and implementation a malicious client could wreak havoc. A more likely scenario would be that a poorly coded client could wreak havoc, but this is still clearly unacceptable. To solve this problem, we had the server side issue each client a unique integer ID. Whenever a client needs to send information to the game handler, the client also has to pass in its unique ID. Since the server only tells each client its specific unique ID, there is no simple way for one client to spoof another client. Now, the function's signature is `turnover(Object moveInfo, int uniqueID)` Thus if it is player one's turn, the handler will only allow a move to be made if the uniqueID passed in is that of player one.

Some minor problems are apparent in this solution, however. If a client fails to record its uniqueID when it is initially given, the client cannot make a move, and will eventually be disconnected for taking too long. Also, if a client calls turnover() with the wrong ID, there is no way for the handler to know which client just made the error. In the case of a malicious client this means that the handler doesn't know which client to remove from the game. In the case of a poorly coded benign client the server doesn't know which client to inform that they used an incorrect unique ID. The latter problem could be solved by broadcasting to all clients that somebody had used an incorrect ID, though that solution still leaves much to be desired.

At this point, the server was fairly robust. However, there is a big difference between "fairly robust" and "completely bulletproof". The server does crash for reasons that we have yet to fully explain.

Adding Socket Communication

Although the server was not completely error-free, we decided to move toward replacing RMI with communication via sockets. We wanted to add socket connectivity in such a way that students using the NOOGA framework could either use RMI to have clients communicate to the server, or they could choose to communicate with sockets. That is, we wanted to have the same server support both RMI and simple socket communication at the same time.

At this point in development, we had the server side communicating with the client side via RMI. We did not want to write a completely new implementation on the server side to communicate with the the client side using sockets. Instead we abstracted the socket functionality away from the server by creating a dummy client which would talk to directly to a game handler, and a dummy `NOOGAServer`, called `NOOGASocketConnector`, which would forward all requests to the `NOOGAServer`. The `NOOGASocketConnector` was written in Java, and thus could communicate with the `NOOGAServer` via RMI.

The scenario we envisioned worked as follows. First, the `NOOGAServer` would be started. Now, with the server waiting for players to connect, we would run the `NOOGASocketConnector`. This socket server uses RMI to get a reference to the `NOOGAServer`, and then listens on a socket for clients to connect. When a client asks to play a game, by writing `"Join Boggle"` to the socket, the `NOOGASocketConnector` handles this request and informs the `NOOGAServer` via RMI that some player, say Joe, wishes to play a game. By implementing socket connectivity in this way, we did not have to modify `NOOGAServer` at all. When a game should begin the `NOOGASocketConnector` essentially makes pseudo-clients that will handle all the RMI communication with the game-specific handler. The pseudo-client is an intermediary that translates between RMI on the server side and socket strings on the client side. For example, suppose Joe guesses the word `"wash"` while playing Boggle. Joe's client would send `"Word_Guessed: wash"` down the socket. Then the pseudo-client would parse the string to determine that the user wishes to guess `"wash"`. The pseudo-client then sends this guess to the `BoggleGameHandler` via RMI. The handler checks this word and sends a response via RMI to the client. However, this client is the pseudo-client which takes the information from the handler via RMI, parses it into a string (e.g., `"wash valid 2."`, then writes this string out to the socket. Finally, the real client receives this string over the socket and processes it. Presumably this information would be displayed to the user somehow, but the server isn't aware of how the real client uses this information. By abstracting the socket connectivity away from the RMI connectivity already in place, we didn't have to re- write any of the code in either the handlers or the `NOOGAServer`. If we were not able to get socket connectivity to work, it would have been very easy to scrap it without having to redo functionality we had done previously. This prevented us from having to essentially develop two versions of all handlers: one for RMI, and one for socket connectivity. The downside to implementing socket connectivity this way is increased complexity in certain areas. We had to develop a whole new layer of communication, which we deemed middle-ware. This often leads to confusing documentation and explanations and increased complexity in determining which objects are communicating and where breakdowns occur. So, both `NOOGAServer` and `NOOGASocketConnector` have to be running in order to allow socket connectivity. If one of these two is not started, socket connectivity won't work; if either one of these goes down connectivity will not be supported. As discussed earlier, there are certain disadvantages to using socket connectivity at all. The protocol for communication has to be very well defined. The client must know exactly what to expect and when, and likewise must know exactly how to parse the information it gets.

Adding XML Connectivity

We had an idea, however, that some of the complexity associated with this socket communication could be removed using the Extensible Markup Language (XML). XML is a language specifically designed for structuring data and then sending it over the web. We wanted to leverage XML so that when we sent data back and forth over the socket, we would send it in XML format rather than simply using plain text. For example, before we employed XML, we might send the following data over the socket to indicate a Dots move:

```
move: 1 2
```

This would indicate that a move had been made connecting dots one and two. This is a bit cryptic to anyone who doesn't know the protocol being employed. However, with XML, the client would send something equivalent to the following over the socket:

```
<?xml version="1.0"?>
<move>
<first_point>1</first_point>
<second_point>2</second_point>
</move>
```

The above text also indicates a move connecting dots one and two. There are advantages and disadvantages to employing XML for all socket communication. The major advantage is that in many cases it was clearer what was going on. Still, both the client and server sides had to know what to expect when, but it many ways it was easier to handle the data once it had been received. Since XML technology appears to be gaining popularity, it would be good to introduce students to it so using XML has some pedagogical validity. The downside to using XML instead of a plain-text protocol for socket communication is the complexity needed to actually form the commands. There are XML libraries to help, but using them is more complicated than simply writing text down a socket. At times, it seemed very much like overkill.

Analysis of XML

Although we think it is overkill to use XML to support the games currently implemented in our server, someone may want to add a new game to NOOGA that requires passing complex data between server and client. In complex board games, for example, using XML would make the communication clearer without adding too much unneeded complexity. Once we had XML functionality working, we decided it would be beneficial to maintain the plain-text functionality for the games. Upon connecting via a socket to the `NOOGASocketConnector`, the client must specify whether it wishes to use the plain-text protocol or the XML protocol. From that point on, all communication is dictated by that decision. If XML is elected, the server creates an XML pseudo-client that will talk to the real client in XML. If plain-text is selected, a different pseudo-client is created that

talks to the real client. It was very satisfying when we were first able to connect three clients to the server each using a different protocol: one player connected via RMI, another connected via a socket using the XML protocol, and the third client used a socket with the plain-text protocol.

TESTING NOOGA: ADDING A NEW GAME

With all this functionality in our `NOOGAServer`, we wanted to give it a real architectural test: how hard would it be to add a new game? Ultimately, this is what future students would have to do. We decided to add Battleship to our NOOGA suite and began by first adding RMI functionality only. We decided exactly how the client and server would communicate, defining the protocol for what information would be sent when. We divided the implementation so that part of the team wrote the server while the remainder of the team wrote the client. On the server side, we had to write a new handler `BattleshipGameHandler`. when two Battleship clients have connected, the `BattleshipGameHandler` takes care of coordinating the game play. We implemented this handler without too much complication. On the client side, as well, there was not too much extra complication with writing the actual GUI front end. At times we ran into some problems coordinating the battleship GUI/client with the actual user. In addition to writing the GUI front end for Battleship, we had to write a helper class `RMIBattleship` which is code the user doesn't see. This is a small class with very little code, but as the architecture stands now it is necessary.

This test, however, was very contrived. All three of us had intricate knowledge of the NOOGA architecture after working on it every day for some period of time. Thus, when we added a new game, it was not a true test of how easy it will be for students to add a new game. Before deploying the NOOGA architecture for use, it would be very beneficial to have an outsider try to add a new game.

CONCLUSIONS

The biggest roadblock to the NOOGA server right now is that the server is not bulletproof. If hundreds of students were to start writing clients and connect to the server, it is not clear that the server would be able to stay up and running. Many of these clients, while in development, would certainly contain bugs. Given what is known about the server, it would be unrealistic to assume that the server would be able to stay up and running with all sorts of buggy clients trying to connect. Before a central NOOGAServer is released for widespread use, it needs far more extensive stress-testing. If a student developing a client connects to the server, he will expect any flaw to be a result of his own coding, not the server's! If an error occurs on the server side, and students don't have access to the server, then it will be very frustrating. However, the server is reasonably reliable. The best current use for the software written would be to have each team of students run their own server. That is, suppose the current NOOGA code was released to the class with the requirement to add two new games to both the server and client side. Each team of students could run their own server, and connect to it using the clients they wrote. That way, error-prone clients that the team was not responsible for could not bring down the server. Also, if an error occurs on the server, the team of students will be able to restart the server.

Our goal to implement a server accessible to students everywhere was only partially realized. Students can use the server, but not every student can use the same server. We gained a real appreciation for some aspects of real-world software development as part of this project.

REFERENCES

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[2] Landon Cox Owen Astrachan, Geoffrey Berry and Garrett Mitchener, "Design patterns: An essential component of cs curricula," in *The Papers of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*. February 1998, pp. 153–160, ACM Press.

[3] W. Garret Mitchener and Amin Vahdat, "A chat room assignment for teaching network security," in *The Papers of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, 2001.

[4] Owen Astrachan, "Oo overkill: When simple is better than not," in *The Papers of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, 2001, pp. 302–306.

[5] G.E. Krasner and S.T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *Journal of Object Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[6] "Manifesto for agile software development," `http://agilealliance.org/`.

[7] Martin Fowler, "Put your process on a diet," *Software Development*, December 2000, `http://www.martinfowler.com/articles/newMethodology.html`.

[8] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[9] Ann Wollrath and Jim Waldo, "Rmi: Remote method invocation," `http://java.sun.com/docs/books/tutorial/rmi/`.