1 Introduction

We chose to implement Dots! as a client-server application/applet in Java. Dots! is a game played on a grid of dots. Players fill in lines between dots and whenever a player completes the four sides of a box, they score a point. The player with the most points wins. Figure 1 is a screen shot of a game in progress. The system is designed to reuse classes and objects between the two applications: client-side and server-side. Communication between the two applications is handled by passing Java Objects over the network. The server supports up to 250 simultaneous players and 100 games. Welcome to Dots!

2 Design

2.1 Overall Design

Dots! uses a standard client-server architecture. Where possible, the two programs share the same classes. On occasion, the server needs more functionality, or different functionality, in a particular class. In these cases, the server versions are derived from the classes used by the client.

2.1.1 Client Design

The Client class is both an applet and an application. Figure 2 is a basic class diagram for the client classes. It presents the user with the user interface to connect to the server, create or join a game, and then play the game. The major dialogs have their own classes:

- ConnectionStatusDialog - This dialog is presented while the client is waiting for communication from the server when that communication may take a significant amount of time.

- GameListDialog - After the user connects to the server, the Client displays the list of games that have been created but not yet started in this dialog. When the user is not playing a game, this dialog is the main interaction window for the client. It also allows the user to create a new game, join a game, or quit playing and logout. Figure 3 is a screen shot of the GameListDialog with two possible games for the player to join.
Figure 1: Screen Shot of a Dots! Game in Progress

Player 3: Score : 2
Player 1: Score : 1
It's your turn!!! Click on a gray line!!!
Figure 2: Class Diagram for Client-Side Classes
CreateGameDialog - When the user wants to create a game, the client provides this dialog to get the name, number of players, and board size of their game.

WaitingForPlayersDialog - After creating a game, or selecting a game to join, the client waits for the game to start. This dialog displays the number of people currently waiting for the game to start. For the game’s creator, it also provides a Start Game button that allows them to start the game as soon as at least one other person has joined.

Client also handles the initial connection to the server. When the connection is completed, Client creates a new Player instance to handle all further communication with the server. After a game is started, the Client creates a new Game instance and displays it. Game in turn creates a BoardCanvas which contains the BoardLines and BoardSquares needed to display the game and handle interaction with the user.

The Player class handles interaction with the server and the Game. Its superclass, Opponent, provides the minimum of data and functionality needed to uniquely identify the player to the server and within a game (in the current incarnation, that data is simply the player’s name). Player also contains one instance each of MessageSender and MessageReceiver. As suggested by their names, these classes handle the mechanics of sending messages to and receiving messages from the server. Player listens to the MessageReceiver and uses an internal thread to decode and dispatch Messages it receives to the appropriate listeners.

Game tracks the overall state of the game, and provides a user interface panel for interaction. As GameEventMessages are received, Game updates the display as needed. If the game is ended prematurely by the server, Game initiates a graceful shutdown of the interaction.
2.1.2 Server Design

The Server class handles the overall startup, monitoring, and shutdown of the Dots! server application. It provides a very basic interface to start, stop, and quit the application, as well as monitor for server state (running or stopped), the number of players connected, and the number of games created (either in progress or waiting to start). Figure 4 is a screen shot of the server keeping track of this information. Figure 5 is a basic class diagram for the server classes. When the Start Server button is pushed, a new DotsConnectServer is started. The DotsConnectServer listens for clients to connect and then handles the login process. Once a client successfully logs in, DotsConnectServer creates a new ServerPlayer and adds it to the PlayerTracker.

The PlayerTracker implements the Singleton pattern. It keeps a thread-safe list of all players logged into the server. Each player must have a unique name, so the PlayerTracker is consulted during the login process to ensure there are no duplicate players. When the server shuts down, the PlayerTracker is used to shut down the connection to all players logged into the server.

After a client has logged in, all messages sent to the server are received by the ServerPlayer instance. It directly handles all SetupMessages from the client. When a client sends a CreateGameMessage, ServerPlayer checks with the GameMonitor to make sure the game’s name is unique. If it is, the game is accepted and ServerPlayer creates a new ServerGame instance and adds it to the GameMonitor.

The GameMonitor, like the PlayerTracker, implements the Singleton pattern. GameMonitor tracks all games that exist on the server, similar to PlayerTracker. It keeps separate lists for games that have already started and games that have been created but have not yet started. GameMonitor is also responsible for notifying clients not currently playing a game when a game is no longer available to join, as well as removing a player from their games if they log out of the...
Figure 5: Class Diagram for Server-Side Classes
The `ServerGame` class inherits from `Game`. While it performs the same function on the server (tracking the state of the game), most of its inherited functions are overridden. `ServerGame` keeps all of the `ServerPlayer`s in the game up to date with the moves of the other players. It also keeps track of the current player. If a `ServerPlayer` leaves the game unexpectedly, `ServerGame` notifies the other players. If only one player is left in the game, `ServerGame` will force the game to end. When a game is over, no matter how it ended (forcibly or naturally), `ServerGame` removes itself from the `GameMonitor` and the `ServerPlayer`s who were playing.

2.2 Event Listeners

Communication between different classes is handled primarily through messages and event listening. All messages are in the package `edu.ucsc.dots.message` and are subclasses of the abstract superclass `Message`. There are two main classes of messages within the package: `SetupMessage` and `GameEventMessage`. `SetupMessage` is the superclass for any messages that relate to the connection of the user and the initial creation of a game. `GameEventMessage` is the superclass for all messages relating to the playing of a game, specifically the communication of moves between players. `Player` listens to a `MessageReceiver` for all messages which it then dispatches to the appropriate listener list.

All classes responding to `SetupMessages` must implement the `SetupMessageListener` interface. These classes are `Client` and `ServerPlayer`. Basically, the `Client` listens to the `Player` for all `SetupMessages`. `ServerPlayer` also listens for these messages but since `ServerPlayer` is a subclass of `Player`, it essentially listens to itself. The `Client` listens for all messages that indicate a game has been created or joined by the `Player`. This communication is sent with a `CreateGameMessage` created by the `Client` and received by the `ServerPlayer`, which in turn sends back a `GameCreateResponseMessage`, which tells the `Client` whether or not the game was successfully created. When a user first signs in, they are presented with a list of all currently available games. The `ServerPlayer` gets this list from the `GameMonitor`, and the `Client` is responsible for then creating a dialog to display it to the user. This communication is handled with instances of the `GameListMessageData`, `GameListMessage`, and `GameListRequestMessage` classes. When a game is started, the `ServerGame` tells the `GameMonitor` this and the `Client` is responsible for removing that game from any currently open “Available Game List” that a user is viewing. If a new game is created, the `Client` is also responsible for adding that game to the list. The `ServerPlayer` is responsible for telling the `GameMonitor` and `PlayerTracker` to remove a player from all lists if that player has logged out. The `ServerPlayer` is also responsible for telling the `Game` class when the game has begun and ended, with the messages `StartGameMessage` and `EndGameMessage`. The `ServerPlayer` also lets a player who is waiting for a game to begin know when another player has joined that game. These “administrative” duties are handled by the `SetupMessageListeners`.

All classes responding to `GameEventMessages` must implement the `GameMessageListener` interface. These classes are `BoardCanvas`, `Game`, and `ServerGame`. The `GameEventMessages` are essentially used to track the state of the game (the moves made by the players). They also determine which player is next in line for making a move. Finally, they handle when one of the players in a game quits before the game has completed. Typically, a message
is generated in BoardCanvas and propagated up to the Game, which sends it to the Player. Player then sends the message to the ServerGame which interprets the message, creates a message in response if needed, and sends the message back down the chain: ServerGame to Player to Game to BoardCanvas, which displays any changes to the user. The three main subclasses of GameEventMessage are MoveMadeMessage, RemoteMoveMadeMessage, and SetTurnMessage.

There are also ScoreUpdateMessages which are sent from the BoardCanvas, who keeps track of the scores, to the Game, which displays the scores in JLabels. These messages are not sent over the network.

2.3 Random Player Design

The client can also choose to play against the computer, a random player. The client specifies the size of board to play and the random player has a list of possible line moves. The AI of the random player is straightforward. The RandomBoardCanvas checks for any possibility to score, if it finds one, the line to complete the square is chosen as the random player move. If there is no possibility to score, an available line is randomly selected. When the Player or the random player makes a move, the line selected is removed from the list of possible moves.

3 Sequences

3.1 Overall Server Sequence

When the user clicks the Start Server button in the server application, the program creates a new instance of the DotsConnectServer, a Thread subclass, and tells it to start. One of the parameters is a ThreadShutdownMonitor; instances of this class are used throughout the Dots! classes to enforce graceful shutdown of the numerous threads running in the program. The other parameter is simply the port number that clients should connect to.

The DotsConnectServer’s execution is a simple loop. While it is not time to quit, it listens for new clients to connect. As soon as a client does connect, the DotsConnectServer creates a new instance of the internal PlayerConnectionThread class. As suggested by its name, this inner class is also derived from Thread. This separate thread is needed because although a connection has been made to another program, the connection protocol requires client programs to send information about the player before the client is accepted. This process could take a (relatively) significant amount of time, especially if there is network lag in the connection. A new thread to handle the actual connection allows the DotsConnectServer to go back to listening to the port for other clients as soon as possible.

The PlayerConnectionThread first attempts to read a CreatePlayerMessage object from the newly connected client. Assuming the message is sent within a time window defined by the DotsConnectServer, the player name passed in the message is compared with the names of players already logged into the system. The name, socket, and I/O streams for the client are used to create a new ServerPlayer. The server then attempts to add the newly created player to the PlayerTracker’s list of players. Player names must be unique, so if the name already exists, the PlayerTracker sends a DuplicatePlayerException. This exception is caught and the client is sent a PlayerRejectedMessage along with the reason for the rejection. The connection is then closed. Any errors in communication during the login process
will cause the server to close the connection with the client. Once a player is added to the PlayerTracker, all messages sent to the server by the client are handled by the ServerPlayer.

When a client gracefully exits, it sends a LogoutMessage to the server first. The ServerPlayer immediately closes the connection to the client, and then removes the player from any game they might have been involved in. Technically, a player is only allowed to participate in one game at a time, but the server checks them all just to be safe. If the player was involved in a game, the other players in that game are sent a message notifying them of the event. A variety of messages could be sent depending on the state of the game. If the game hasn’t started, and the player was the game’s creator, a GameCanceledMessage is sent. If there is only one player left in the game (after it has started), the server sends an EndGameMessage because there is nobody left to play against. If there are still two or more players in the game, they are sent a PlayerQuitGameMessage and allowed to continue playing. Finally, after the player being logged out is removed from the game, the ServerPlayer removes itself from the PlayerTracker.

An ungraceful shutdown by the client is detected by the PlayerTracker's internal PlayerCleanupThread class. This thread wakes up once an hour and checks each player for the time of the last message received from that client. If more than one hour has passed since the last message, the ServerPlayer is sent a LogoutMessage programmatically. This triggers the same shutdown process described above.

When the user stops the server gracefully, the DotsConnectServer is stopped via a call to the ThreadShutdownMonitor it shares with Server. Then the PlayerTracker is told to shutdown all clients which it does by simply closing the connection to each client. The GameMonitor then clears the started and waiting game lists. Unless the user has also chosen to quit the server application, a new DotsConnectServer can be created and the server restarted.

### 3.2 Creating/Joining a New Game

When the client chooses to create a game in the game list window, the user must specify the number of rows and columns of the board, the game name and the maximum number of players in the game. If the game is accepted by the server, a dialog appears displaying how many players are in the game. The ServerPlayer that receives the requests adds the game to the waiting list of GameMonitor. A ServerGame is created to keep track of all the players in the game and to handle interactions between them. Figure 6 shows the sequence diagram for creating a game.

When a client joins a game from the game list window, the ServerPlayer that receives the request adds itself to the specified game. The ServerGame is tested from the ServerPlayer to find if the maximum number of players limit of the game is reached. When the maximum players limit is reached, the ServerPlayer sends a message across the network to the Client to start the game. If the maximum number of players limit has not been reached, the game continues to wait until the limit is reached or until the game’s creator explicitly starts the game. When the game starts, a Game is created with the players in the game added to the game player list. The ServerPlayer removes this game from the waitingGames list in the GameMonitor. Figure 7 shows the sequence diagram for joining a game.

### 3.3 Playing the Game

When a move is made, the move is communicated between remote users by a series of messages. First, the BoardCanvas detects a user’s mouse click. The BoardCanvas then checks the
Figure 6: Creating the Game Sequence Diagram

Figure 7: Joining the Game Sequence Diagram
legality of this click by iterating through the list of BoardLines. If the click was within the area of a BoardLine which has not been taken, the click is valid and that BoardLine is set to be taken. Then, the BoardCanvas generates a new MoveMadeMessage. This message is sent to all the listeners of the BoardCanvas, namely the Game. The Game then propagates the move by sending a MoveMadeMessage to all players. This is necessary because the BoardCanvas has no notion of which player is the local player, only a list of all existing players. The Game, however knows which player it is associated with, and thus adds this information to the message. The Player now propagates the MoveMadeMessage to the network and the message is received by the corresponding ServerPlayer. ServerPlayer then propagates the message on to its listeners, namely the ServerGame. The ServerGame receives this message and translates it into a RemoteMoveMadeMessage. This message is then broadcast to every Player (actually, ServerPlayer which re-broadcasts the message to Player). When a Player receives this message, they propagate the message to the Game, which then calls addRemoteMove() in BoardCanvas. This class then finds the BoardLine that is associated with the move and sets that line to have been taken. Then all BoardSquares are iterated through to see if any were completed. If so, they are set to be completed by the Opponent who made the last move (this information is contained within the RemoteMoveMadeMessage. The BoardCanvas also calculates the scores by cycling through all BoardSquares and if any were newly taken, it updates the score for the Player by which they are taken. Scores are stored in a hash table indexed by the Player with the value being the actual score.

The ServerGame is also responsible for determining for which player should make the next move. This happens after the ServerGame generates a RemoteMoveMadeMessage. The ServerGame has a list of all players. The first Player is removed from front of the list and sent to the back. Two new SetTurnMessages are generated: one with a false parameter and one with true. The “true” message is sent to the Player who is first in the list, and the “false” message is sent to all other Players. Every Player then receives their appropriate message, and propagates this message to the Game which in turn tells the BoardCanvas whether or not it is the local turn. If it is not the local turn, the BoardCanvas ignores all user mouse clicks. Figure 8 shows the sequence diagram for both making a move and setting a turn.

3.4 Ending the Game

The RemoteMoveMadeMessage handler in Game checks for weather or not the game is over on the client side. If there are no more possible moves, the game is over and the endGame() in Game is invoked to tells the BoardCanvas the game is over. The appropriate end of game message is displayed in a dialog with the game winners calculated from the BoardCanvas. A setup message is sent to display the game list window. Figure 9 shows the sequence diagram for the end game sequence on the client side. On the server side, ServerGame listens to MoveMadeMessages and when there are no more moves, the game is removed from the GameMonitor list and from all the ServerPlayers. Figure 10 shows the sequence diagram for the end game sequence on the server side.
Figure 8: Playing the Game Sequence Diagram
4 Performance

The original limit to the number of games that could be running on the server was one thousand twenty four. As it turns out, this was an extremely optimistic number. To test the performance of the server, a special client application was written to connect with a number of players, create games, and simulate the playing of those games. There is a random amount of delay built in (from one to ten seconds) between receiving a message from the server and sending one back. This represents a simple attempt to mimic the time a real human would take to think about their next move.

The tests were run with the Dots! server running on a dual 700 MHz processor Pentium III. The operating system on the machine is RedHat Linux 7.2, and the JRE was version 1.4. Four stress clients were started on different computers running a variety of operating systems and JREs (version 1.3.1 or 1.4). In the following discussion, all processor usage numbers will be an average of the two processors as reported by the `top` UNIX utility; they are only an approximation of the actual CPU usage.

Before any players had connected to the server, the CPUs were 98% idle. This is consistent with the CPU usage before the server had even started. In the first test, each client was told to create twenty five games, with a random number of players for each game (at least two, but no more than four). They attached a total of two hundred fifty five players to the server, and simulated one hundred games being played by these players. During the entire time the CPU usage did not exceed 50%, and after the games had all started, the usage settled down to about 30%. The second test had each client attempt to create fifty games. After just over three hundred players had
connected and one hundred twenty four games created, the server was unable to create any more threads and was shutdown. This exceeds the one thousand threads per process limit that is built into the Linux kernel.

One way to overcome this limitation in the server would be to increase the number of threads allowed in any one process. In Linux, this is accomplished by increasing the number of threads allowed in a process and recompiling a variety of libraries and possible the kernel. However, not all operating systems can be adjusted in this manner (at least, not by the end user). A better solution would reduce the number of threads needed by the server. Rather than creating separate threads for input and output with each client, a single combined thread could easily handle the communication for a single client. To take this idea further, one thread could handle the communication needs for multiple clients. Although extensive performance tuning would need to be done, it is likely that one thread could service the communication needs between ten and twenty five clients. Combined with a larger number of threads per process, it is conceivable that the test computer could service the one thousand twenty four game maximum original hoped for.

5 Extensions

5.1 More Colors/Players

One extension would be to allow the user to choose their own color, rather than have the system assign a color to that user. When a user creates a game, they would have to choose their color.
as part of dialog. The \texttt{Player} class would then need to store each user’s player. When another user joins a game that is waiting to be played, the color that the game creator chose, and the colors of any other users who have joined the game would have to be removed from the list of available colors. Adding this dialog would not be difficult, but multiple classes would need to store the color, as opposed to just the \texttt{BoardCanvas} class as is in the current implementation.

Currently, the maximum number of players per game is a constant set to four, but this number could be increased. The game could be equipped to handle any number of players. The only change that would need to be made is that the four possible colors are currently hard-coded. Instead, the possible colors should be dynamically created according to the maximum number of players. In theory, there could be no limit on the number of players in a game. One way to address the color assignment for this possibility is to create a color by sending the \texttt{Color} constructor three randomly selected \texttt{float}s. It would only need to be checked that the same floats were never sent twice. This method would be easy to implement. It is possible, however, that two randomly generated colors could be difficult for the user to distinguish between. This possibility would need to be weighed against the desire for a higher maximum number of players.

5.2 \textbf{Smarter Random Player}

We would like to add more intelligence to the random player, like not adding a third line to a square, thereby avoiding a chance for the opponent to score. The random player can also learn to play smarter using any standard machine learning algorithm. We can also use mini-max trees to look ahead several moves and to choose the best move.

5.3 \textbf{Extra Turn}

Dots! can be played with the additional rule that when a box is completed, the user who completed the box gets an additional turn. In order to implement this, additional information would need to be added to the \texttt{MoveMadeMessage}. This message would also need to contain whether or not a \texttt{BoardSquare} is completed. If a \texttt{BoardSquare} has been completed, the \texttt{ServerGame} would simply not execute the code that sets the turn to the next player. If the list of \texttt{Players} is never altered, the \texttt{Player} who just made a move will get a “true” \texttt{SetTurnMessage} twice in a row, thus allowing for an extra turn. Whether or not this rule is in place can have implications on a user’s strategy of game play.

5.4 \textbf{Text Messaging}

A desirable feature of any game to be played on-line is the ability to “talk” with your opponents. This capability would not be difficult to add to our game. The components for the text field and scrollable text panel are already available with Java. The only additional implementation would be a new set of messages that sends the text between users and displays it to the screen. Messages are already sent between users that contain moves that have been made. It would not be difficult to create new messages that contain text.

5.5 \textbf{Player Statistics}

With a few additions to the \texttt{Player} class, and some refactoring of the functionality, the Dots! client and server could be extended to allow persistence of player information between logins.
On both the client and the server side the communication functionality that currently resides in Player and ServerPlayer should be moved into a new class hierarchy (a Connection superclass with ClientConnection and ServerConnection subclasses). The Opponent class data would still be used as the unique key to identify an individual player and access their history and status. Additional fields would need to be added to the Player class to store the new information and send it to other clients for display.

The client side changes would be mostly limited to additional user interface dialogs and capability, and support for the additional messages needed to pass Player statistics back and forth. Among the changes to the existing interface, support for entering a password in the login dialog, and display of the player statistics in the “Waiting for game” dialog would be desirable.

On the server side, players that are not currently logged in would have their persistent data stored in a database (either a simple one implemented by the server, or a full scale database accessed through Sun’s JDBC API). Using a database to store the player data would be likely to slow the process of logging in. However, the scalability and reliability provided by the database is enough of an incentive to justify its use.

6 Conclusions

Dots! is clearly a fun game. We implemented Dots! in Java with a classical client-server architecture. A key aspect of this design is the reusable classes with the server-side classes inheriting from and expanding upon the client-side classes. The applications are massively threaded. While the design is not currently scalable in a distributed sense, the modifications needed to change this would be easily handled by the program design. Our project makes extensive use of objects, inheritance, threads, and event listening to implement Dots!