Concurrent Online Multiplayer Browser Chess

Dennis Ideler, Graham Sharp
Computer Science Department
Brock University
{di07ty, gs07kw}@brocku.ca

***Abstract*—In this paper we present *Online Chess Park*, an application that allows multiple players to play simultaneous chess games online. The introduction covers the problem description and applicability of networking. The design and implementation section documents how it works and how it is used. We finish with the results and our conclusions.**

I. Introduction

Online Chess Park was created as a final project for our Introduction to Computer Networking course (COSC 3P01). Students had the choice of a written paper or a coding project. We chose a coding project to gain more practical, real world, experience.

Guidelines for the coding project focused on developing an application or suite that accomplishes some networking-related task. The project need not focus soley on network protocols and socket programming. In the real world, the integration of software systems with communication abilities often benefits most. Concurrent multiplayer networked chess is a good real world example of a problem that computer networking can be applied to.

Many different platforms were considered for building the application. Some of these include *node.js, Ajax-Push Engine (APE), GameJS, HTML5 WebSocket,* and *Processing.js*. We also considered writing the full application in Java (i.e. Java server and Swing client) and then have the applet run from a webpage. Eventually we settled with a Java server which we communicate with using HTTP and AJAX from a webpage. Being browser based, it goes without saying that we made heavy use of HTML, CSS, and JavaScript.

II. Design and Implementation

Users interact with the client-facing side of the application via their browser. The back end of the client-side communicates with the server-side. The server is passive and responds when its services are needed. The different services are explained in the succeeding sections by pseudocode along with some commentary. Diagrams and screenshots are included to visualize the data. Note that some of the screenshots may not represent the final product.

The client-side consists of everything that is user-facing. This includes the user interface and the networking code it uses to communicate with the server.

For the user interface we had two goals in mind: (1) a minimalist design; and (2) a theme. We used a park theme because casual chess is often played in outdoor parks, so we wanted to recreate that experience online.

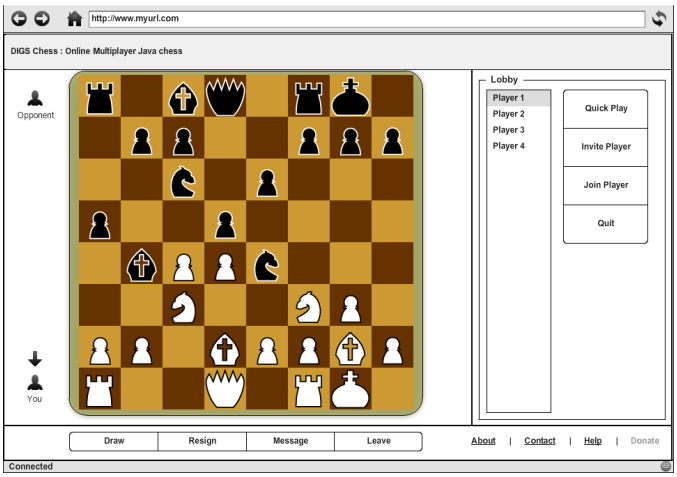

Fig. 1. Mockup of our first design.

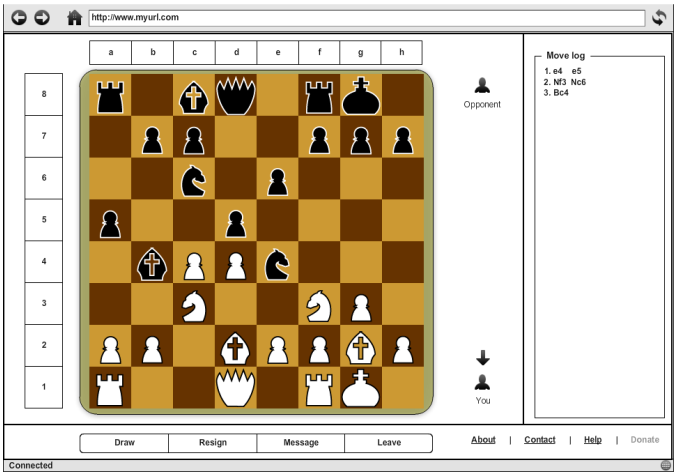

Fig. 2. Early mockup of the game interface.

Figure 1 shows our very first design. The main difference from our current design is that our initial design tried doing everything on one page. For example, this was our only interface and the board and lobby were on the same page. We split the application into three user interfaces:

1) Login: Users have to login to the application ("enter the park") via this page. See Figure 3.
2) Lobby: This page (also called the "park") is where users can find other players to compete with. See Figure 4.

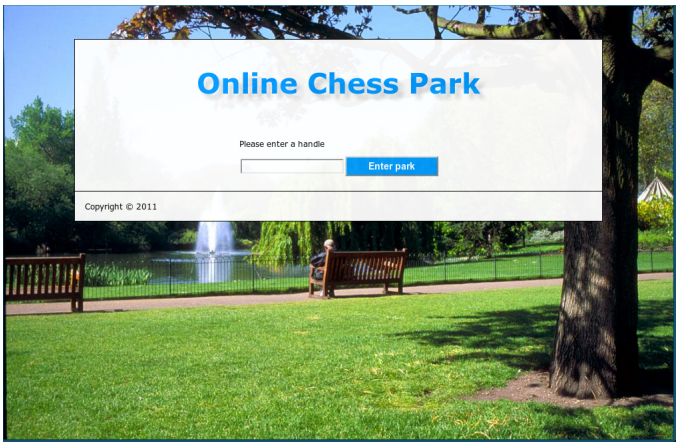

Fig. 3. Login UI.

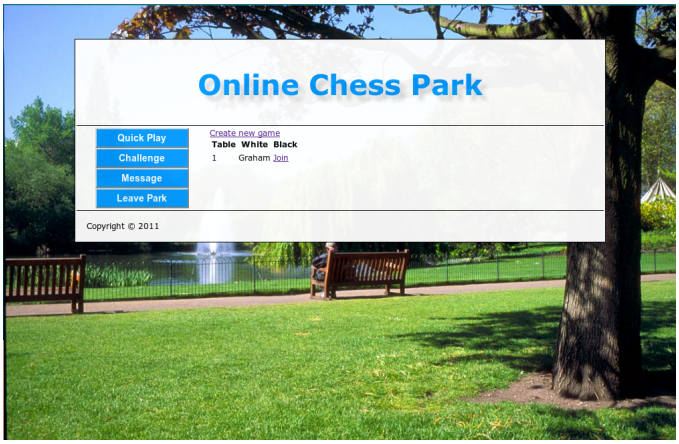

Fig. 4. Lobby UI.

3) Game: This is the chess board interface. The game is played from this page. Two versions of the board exist. One where unicode is used to render the chess pieces and another where transparent PNG images are used for the pieces (for browsers that do not render the chess font). An HTML table is used for the chess board in both versions. See Figure 5.

We decided to separate the board and the lobby for simplicity reasons and to prevent the user interface from becoming too cluttered. We also decided to add a login screen that allows users to choose a nickname with which they can enter the game lobby, rather than have an automatic ID assigned to them.

Unlike the client program that requests a service and only runs when needed, the server program provides services and should thus run all the time as it does not know when it will receive client requests (i.e. when its services will be needed). The server program processes client requests by performing the tasks requested by clients. These tasks include checking move validity, serving pages, creating chess tables, and more. It's possible for multiple client programs to share the services of the server program. For example, multiple clients can request the same web page to read in their browser at the same time. Multiple chess games can be played by different players simultaneously on the server [1]. The server keeps track of all the players and games and plays along with a game on an internal server-side copy so it can easily check board states.

Fig. 5. Game UI.

Overall Server Structure

The server implementation is divided between four Java Source Packages:

1) chess
2) chess.game
3) chess.lobby
4) eventserver

chess package: The core of the server. Creates and runs the server. Contains the classes: Server, LoginHandler, LobbyHandler, FileHandler, ExitHandler, and ClientInfo.

chess.game package: A subpackage of the chess package. Classes in this package focus on building and running a networked chess game board between two players. Classes include: ChessGame, GameEvent, GameHandler, ListenHandler, ListenThread, ListenerHandler, MoveHandler, and NewTableHandler.

chess.lobby package: Another subpackage of the chess package. Classes in this package focus on lobby communication. Classes include LobbyEvent, LobbyListenHandler, LobbyListenThread, and LobbyListenerHandler.

eventserver package: This package concentrates on communicating events between the application components. It uses a behavioral software design pattern, to be specific, a modified version of the *observer pattern* [1]. The observer pattern uses a publisher/subscriber design which allows a number of observer objects to see an event. When the publisher wants the subscribers to handle an event, it notifies each subscriber in the subscriber list. This pattern is often used for event-handling systems.

[1]The same player can also be in multiple concurrent game sessions.

Server Creation

The server uses several exchange handlers for dealing with different services. Handlers are attached to the server by HttpContext objects. An HttpContext instance simply represents a mapping from a Uniform Resource Identifier (URI) path to an exchange handler on this HttpServer. Once created, all requests received by the server for a known path will be handled by calling the associated handler object.

The server instance contains three hash tables. The first two hash tables both contain ClientInfo objects, so they are both used to look up clients. The difference is that one looks up by user handle and the other looks up by player ID. If a non-existing player is looked up, a new ClientInfo object is created for that player and inserted into both client lookup tables. The last hash table contains ChessGame objects. It is used for looking up existing chess games or creating and storing chess games.

The server instance also has an EventServer which it uses as a lobby server. It is set to contain up to 100 lobby events. There are three lobby events: (1) NEW, (2) FILL, and (3) CLOSE. A NEW lobby event means that a new table (chess board) wants to be created. A FILL lobby event means a player wants to join an open table. A CLOSE lobby event means a table that was open has now been filled (i.e. two players are sitting at the table) and should be marked as closed.

Some of the server functionality includes extracting a value from a cookie, loading a file into a byte array, creating a chess table, joining a chess table, finding tables, finding clients, redirecting a client to another page, and of course sending a response to the client's request.

Algorithm 1 START-SERVER()

```
create a Java HttpServer object
have it listen on some port {we used 22222}
attach exchange handlers
start the server {starts in a new background thread}
```

Algorithm 1 is a high level description of how the server is started. This list shows the mapping between the exchange handlers and URI paths.

- LoginHandler ← /chess/login
- LobbyHandler ← /chess/lobby
- LobbyListenerHandler ← /chess/lobbylistener
- LobbyListenHandler ← /chess/lobbylisten
- NewTableHandler ← /chess/newtable
- GameHandler ← /chess/game
- ListenerHandler ← /chess/listener
- ListenHandler ← /chess/listen
- MoveHandler ← /chess/move
- ExitHandler ← /chess/stop
- FileHandler ← /chess/ *various paths*

Handling User Login

As mentioned above, the LoginHandler handles requests received by the server for the login path */chess/login*. This is the path that the client must visit to login. However, the client can simply visit the domain or root address (or any page that cannot be found) and the RedirectHandler will redirect the client to the login page if they do not have a handle yet.

Algorithm 2 HANDLE-LOGIN($HttpExchange$)

```
extract the body of the request from HttpExchange
if request is a GET request then
    response ← server-generated HTML login page
    send response back to client
else if request is a POST request then
    handle ← extract from POST data
    if handle is valid then
        set cookie that contains handle
        redirect client to /chess/lobby
    else
        send error message back to the client
    end if
end if
```

The high level algorithm is shown in Algorithm 2. The first client request will be a GET request. That request is handled by creating a HTML page that will be used to login. It contains a form that uses the POST method submit the data. The data submitted is the handle the client wishes to use. Once the page is built, the response headers are set and the page is sent back in the response to the client. The client loads the page, enters their handle, and sends back the data in a POST request. When the LoginHandler notices this request it validates the contents to make sure it is an acceptable handle. If it is valid, a cookie is set with the given handle and the client is redirected to the home page (i.e. the lobby). Otherwise the client is notified of their error and has to try again.

Handling Lobby Events

This is where it starts to become more complex. The LobbyHandler event handles requests received for the */chess/lobby* path. The client reaches this path after they have gone through the login process.

Algorithm 3 HANDLE-LOBBY($HttpExchange$)

```
extract handle from cookie that came with HttpExchange
if no handle exists then
    redirect client to the login page
else
    respond to the client with the HTML lobby page
end if
```

The algorithm is explained from a high level in Algorithm 4. The handle is checked as a cautionary method because a client that is not logged in should not be able to go to the lobby. The server responds with the lobby page which is displayed on the client-side.

The lobby page contains a JS function for handling messages. The messages it can handle are:

- displaying errors
- displaying warnings
- adding a table to the lobby
- removing a table from the lobby

The page also contains an embedded page which acts as a listener. To create the listener, the LobbyListenerHandler is invoked from the /chess/lobbylistener path.

Algorithm 4 LOBBY-LISTENER($HttpExchange$)

```
extract handle from cookie that came with HttpExchange
if no handle exists then
    redirect client to the login page
else
    look up the ClientInfo associated with the handle
    set the lobby observer in the ClientInfo to null
    respond to the client with the HTML lobby-listener page
end if
```

The response from LobbyListenerHandler is the lobby listener (which is embedded in the lobby page). When the listener first loads, it sends a GET request using AJAX to the server with a message to reset the lobby (to get a clean lobby state). If the response is error free, then a callback function starts the listener. The response is also sent to the message handler in the lobby page. Once the listener is started, it continues to run by calling itself (if the responses are error free). As the listener is running, it makes GET requests to LobbyListenHandler.

The LobbyListenHandler essentially creates a runs LobbyListenThread which it passes the request and client info to. This thread polls for lobby events and handles them.

Subscriber is an object that waits for an event to take place. The subscriber does little polling when checking for more events but it has an instant response to events. This is a good balance between frequent polling which can bog down the server, and long intervals between polls which are slow and cannot respond instantly when an event is received.

Handling Gameplay

The chess board uses the same model as the lobby, so it will not be explained in as much detail. A listener is embedded on the page and a thread occassionally polls for new events Polling is done occasionally and is in a sleep state most of the time where it instantly wakes up on an event. The game equivalent of the LobbyEvent class is the GameEvent class. The three events are (1) WHITE_JOIN, (2) BLACK_JOIN, and (3) MOVE.

A new chess table can be created via the lobby by clicking on a link to /chess/newgame. The associated client request is handled by NewTableHandler as outlined in Algorithm 6.

When NewTableHandler redirects the user to /chess/game then GameHandler takes over as it is the exchange handler for that path. GameHandler creates the game by combining two pre-existing HTML files (game-head.html & game-body.html). The unique table ID from the chess game created in NewTableHandler and the client's handle are also inserted in the generated game page. The client sees the generated

Algorithm 5 LOBBY-LISTEN-THREAD($HttpExchange$, $ClientInfo$)

```
if client's lobby observer is not observing then
    have it start observing for lobby events
    {now list all available tables}
    xmlData
    for all existing games do
        if game has an open spot then
            append game info to xmlData
        end if
    end for
    send xmlData as response to client
else
    if client's lobby observer does not have any more events then
        create new server instance
        create new subscriber
        add subscriber to the lobby event server
        let the subscriber wait 10 sec for an event
        remove the subscriber from the lobby event server
    end if
    {now we build an XML response that holds the events}
    xmlData
    if client's lobby observer is observing then
        for all events in lobby observer do
            event ← get next lobby event from client's lobby observer
            if event == CLOSE or FILL then
                append 'remove table' instruction and table ID to xmlData
            else if event == NEW then
                create a new chess game using the table ID from event
                append 'add table' instruction and other game info to xmlData
            end if
        end for
    end if
    send xmlData as a response to the client
end if
```

Algorithm 6 HANDLE-NEW-TABLE($HttpExchange$)

```
extract handle from cookie that came with HttpExchange
if no handle exists then
    redirect client to the login page
else
    look up the client info associated with the handle
    create new chess game using the client info
    redirect the client to their table-specific game page
end if
```

Algorithm 7 HANDLE-GAME($HttpExchange$)

```
extract handle from cookie that came with HttpExchange
if no handle exists then
    redirect client to the login page
else
    look up the client info associated with the handle
    get table ID from the GET query
    find or create the game, using client info and table ID
    if the client is not in this game already then
        try to join the game
    end if
    build the HTML game page using a pre-existing head and
    body for that page
    append table ID and client handle to the page
    send the constructed game page as a response to the client
end if
```

game page when they create or join a table. The game can begin when there are two players at a table. Most of the JavaScript used here is externally stored in the game.js file. When board.html is fully loaded, it calls a function in game.js to draw the pieces on the board.

The board uses a function called *handleMessage()* (see Algorithm 8) to handle any type of message it receives from the server. It is very similar to the handleMessage() function from the lobby page.

Algorithm 8 HANDLE-MESSAGE($data$)

```
if data contains an error then
    display error
else if data contains a warning then
    display warning
else if data contains an event list then
    draw the updated board
    update the user icons for black and white players
    if a player has entered the game then
        alert the players
    end if
else if data contains a move then
    extract move origin
    extract move destination
    MOVE-PIECE(origin, destination)
    redraw the pieces on the board
end if
```

The MovePiece() function calls the MoveHandler. It checks if both players are at the table. If someone is missing, the client is asked to wait. MoveHandler tries to move a piece by actually moving it in a server-side copy of the game. This server-side copy lives in a ChessGame object. If the move was illegal, then MoveHandler receives an empty move array and responds to the client with an error. Otherwise the move is successful and it is made.

Just like the lobby page, the game page has an embedded listener which it uses to communicate with the server page

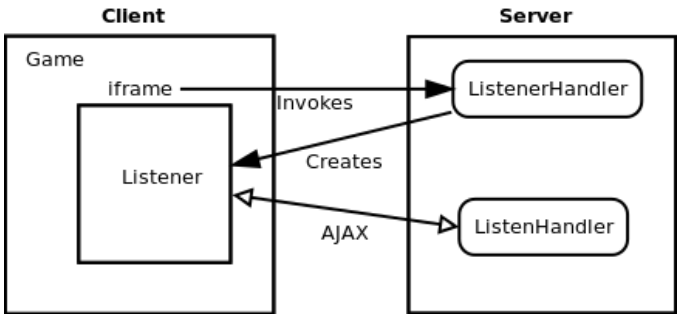

Fig. 6. Overview of the game listener.

loads. It continually makes client requests to the server. These are HTTP GET requests using AJAX. When it receives a response, it sends it to the handleMessage() function in the board page.

Handling Server Termination

There are two ways to terminate the server program. The first is from the server-side. Upon creation of the server, a message dialogue box is created after the server thread is started. This box is visible during the entire uptime of the server and it gives the server administrator the option of shutting down the server program.

It is also possible to terminate from the client-side. To do this, a client must navigate to the /chess/stop path and the ExitHandler will shut down the server program. This of course would be disabled if the application ever went into production. Currently it is enabled to give users extra control when testing remotely.

III. Conclusion

Online Chess Park used Java on the server-side and HTML/CSS/JS on the client-side. We believe this combination worked well for several reasons. The platform for the client-side allowed us to quickly develop the user interface. We could see changes as soon as they were made without recompiling and could make live changes using debugging tools that are built into modern browsers (e.g. *Firebug and Chrome's webkit*). Another benefit is that our code is easily cross- platform, and cross-browser for the most part, without much additional work on our side. Making an application for the web also allows us to have a greater audience because our application can be easily accesible by anyone on a network (more or less). Virtually every computer has networking capabilities and a browser now so they are ready to run the application by default.

For the server-side platform, Java offers certain benefits as well. It allows us to have communication between multiple threads simultaneously, which would not be possible with another typical scripting language such as PHP. This allows us to have efficient yet very quick polling, a major benefit for server programs. The main server can also be completely stored in memory rather than making use of a database. A scripting language would lose all its global data once the script

has finished running, which is why they make excessive use of databases. Another benefit is that Java uses a virtual machine which makes the server cross-platform. This allows it to be easily distributable for anyone who wants to run their own server.

We were able to have concurrent chess games be played on the server by different players and by the same player (i.e. the same player can be in multiple games simultaneously) thus we believe we have successfully accomplished our goal of creating a concurrent online multiplayer browser chess application.

Future Work

Online Chess Park was made as a proof of concept and many improvements can still be made. For instance, the current state of the application does not have garbage collection implemented for removing inactive or non-responsive players. If a player left a game in progress or completely left the application, then a lot of associated objects will still be referenced. This can turn into a significant memory leak if it happens often. We only tested the application on Chrome (mostly) and FireFox. We did not focus on scalability because this application may never go into production. The closest it may come is hosting it on the department's BrockBots server, but only after some improvements have been made.

REFERENCES

[1] http://en.wikipedia.org/wiki/Observer_pattern