Introduction

Project 3 introduces a number of new concepts that are presented as additions to your Project 2 Go Server.

First, a major part of the project is learning how to effectively use a public cloud computing service, in this case the Amazon Elastic Compute Cloud (EC2). With luck, your socket-based Project 2 code will work unmodified when running the clients and server on different EC2 machines, but this - effectively using and running code on EC2 - is a stated requirement for Project 3. The difficulty here stems from learning to create and manage EC2 virtual machine instances, using the command line, and working with remote servers with tools like bash, ssh, screen, vim (or emacs, if you are so inclined), and the like. Please read the EC2 sections of this document carefully, especially our notes on EC2 billing and how to avoid incurring large charges.

Another major concept introduced in Project 3 is the use of a database and SQL. This takes the form of a database maintained by the GameServer, which represents a consistent view of the global state of the system. Having state durably and consistently stored enables fault tolerance in the system, something especially important in a distributed cloud environment. The GameServer is able to recover the state of ongoing games and connected clients using the database, even in the case of GameServer failures. Clients can reconnect to the GameServer, set their state accordingly, and resume playing or observing games. The challenges here are manifold. There is some overhead to learning to use the Java Database Connectivity (JDBC) API and writing SQL to query and modify the database. The rest is understanding how to correctly use transactions to implement a recovery protocol.

The final component of Project 3 is secure authentication. This will involve a “hash and salt” method to avoid sending or storing passwords in plaintext. We will be using SHA-256 for hashing, which is part of the SHA-2 family of cryptographic hash functions. The salient aspects here are adhering to the hashing and salting scheme specified, and understanding the efficacy of this scheme.

---

1 http://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud
2 http://www.oracle.com/technetwork/java/javase/jdbc/index.html
Overall Requirements

Unless otherwise specified, all requirements from Project 1 and Project 2 regarding Go rules and timeout behavior carry over to Project 3. Some protocols have been updated to reflect new requirements for secure authentication and support for database logging.

More detail to follow, but at a high level:

- Your system must work across multiple computers and wide-area networks.
- The server may fail at any time and players must be able to resume their game from the same state at a later time when the server has restarted. Recovery is not required for observers.
- Clients can now register with the GameServer with a password and subsequently log in to the server using that password. Clients should only ever need to register once with the system, but can change their password any number of times.
- Client names should be unique in the system. There should be only one client with a given name registered at a given time.
- Passwords should never be stored or sent in plaintext.
- All clients must be authenticated before they can play in or observe games. Authentication is now part of the connect message.
- The GameServer now maintains a database containing tables for clients, games, and moves. The database will be used for authentication and recovery, and will only be updated by the GameServer.

EC2

Amazon EC2 is a cloud service that lets users start and stop virtual servers on demand, termed “instances”. These instances are based off a virtual machine image that can be specified, termed an Amazon Machine Image (AMI). You are given full and complete access to your instance (meaning root access), but since it is a virtualized platform, you might still be sharing physical resources with other EC2 users. We are using “small” EC2 instances, which means that the level of hardware multiplexing is pretty high, but enough for the minimal requirements of our Go server and clients.

CS162 has been granted a single “master” EC2 account, from which we are spinning off “subaccounts” for each student. To make this work, we’ve written a number of scripts that reside on stella.cs.berkeley.edu, which allow you to start, stop, and otherwise manage your own instances through this master EC2 account. These scripts will also provide you with SSH keys that will allow you to log into instances that you start. You must use this SSH key to login to your EC2 instances. Password SSH access is not enabled. We will release more information on using these scripts shortly - stay tuned for an announcement.

All of the CS162 instances that are started will be based off of a slightly customized AMI, which has important packages like svn, javac, vim, and screen installed. Some familiarity with command line tools is necessary to interact with your machines. You need to be able to run Java programs on the command line. You are also given root access through sudo on your
own instances, so you can install additional packages and otherwise configure your instance however you wish.

Email cs162 if you have any suggestions for other packages that would be nice to have included in the base AMI. Student subaccounts cannot roll their own AMIs, but we are definitely open to suggestions on how the base one can be improved.

**Operational notes**

Another word on EC2 terminology. Instances can be launched, started, stopped, and terminated. Here is a summary of what these operations mean:

- **Launch**: start a brand new instance based on an AMI
- **Stop**: pause a running instance, shutting it down but saving local changes
- **Start**: unpause a previously stopped instance, recovering the saved state
- **Terminate**: stop an instance and throw away local changes. Have to launch a new AMI.

This means that if you copy your code over, work on it a bit, stop it when you’re done, then start it back up again later, your code will still be there on the instance. If you terminate the instance, this completely wipes the instance, and you cannot recover any changes you made. Terminating an old instance and launching a fresh instance based on the base AMI is a good way of resetting everything, if you have really messed up the instance somehow.

Also note that it can take a minute or two to do these operations. Give the instance a minute or two to boot before deciding you can’t SSH in.

**EC2 billing**

Using EC2 costs money. We have been allocated enough EC2 credit that this should not be a problem, but we are depending on all of you to be responsible with your usage. This basically boils down to not starting a large number of instances and then leaving them on overnight when you’re not using them, but let us break it down a little further

Instances are charged at an **hourly rate** while they are started. The minimum charge per instance is also a single hour. Instances incur negligible cost when they are stopped, and zero cost after they have been terminated. This means a few simple rules can keep our costs in line and everyone happy:

- Do not start a large number of instances. 2-5 is okay, 20 is not. We will not be testing with 20 instances, and it is rather unnecessary.
- Do not leave instances on when you are not using them. Stop them if you care about state, or just terminate and blow them away.
- Make sure your instances are stopped or terminated before logging off. It takes some time.
- Do not start and stop instances frequently in a short timespan. Every time you start an instance, it charges a minimum of 1 hour of usage, so this behavior can become expensive quickly.

We are also considering providing accounting scripts to help you monitor your own usage, or
configuring instances to automatically stop themselves after a few hours to prevent any “I forgot and left for the weekend” type situations. However, strict enforcement and budgeting should not be necessary as long as everyone is careful about their usage.

**Hello World Assignment**

To get you started with EC2, you will be running a simple Hello World program on an EC2 instance. You must have this assignment completed by **11/17 at 5:00pm** along with your initial design doc. You will be submitting a screenshot of your HelloWorld output as HelloWorld.jpg with your design doc using the command submit proj3-initial-design. The idea behind this assignment is to make sure you and your group have the basics down early and will be able to run your GameServer and client code on EC2 when you are ready to test it.

We have provided code for a simple Hello World program in edu/berkeley/cs/cs162/HelloWorld.java. Your job will be to move HelloWorld onto your EC2 instance, run it, take a screenshot of your shell with the HelloWorld output, and save it as HelloWorld.jpg. You will know your HelloWorld has successfully run if it prints Hello World! to the console. Please refer to the “Using EC2 Effectively” section at the end of this document for more information about how to run programs on EC2.

**Secure Authentication**

In this project, you will modify your GameServer to enforce secure user authentication. The GameServer will have a database of valid clients and passwords, specified in the section “Database Schema”. To establish a connection, the server and client must adhere to the following protocol:

1. As part of the connect message, the client sends the server a SHA-256 hash of their password.
2. Upon receiving this, the server concatenates with it a salt, and computes a SHA-256 hash of the resulting string.
3. If this matches the value stored in the database, the server authenticates the client. Otherwise, the client is denied.

We define the salt to be the string “cs162project3istasty” without quotes. The server should use this value of the salt for all clients’ passwords.

Once authenticated and connected, clients may also choose to change their password at any time by sending the server a changePassword message with a hash of their proposed new password. Once a client’s password has been changed, that client should no longer be able to authenticate using its previous password; subsequent connect messages must include the hash of the new password in order to be authenticated properly.

New clients should also be able to register themselves with the server, using the register message. The register message includes their name and type (a ClientInfo parameter),
and a SHA-256 hash of their proposed password. Client names should be unique in the system. The server should then update the clients table in the database as necessary, and the client should subsequently be able to authenticate using a connect message, and change their password with the changePassword message if they so desire.

It will be helpful to refer to the java.security.MessageDigest API for details on how to compute hashes. For details on the exact parameters and expected responses for the messages used in the authentication protocol, see sections “Client-to-server messages” and “Server-to-client messages”.

Database Schema

For this project, we will be using SQLite, a small, free, embeddable database with wrappers provided by many languages. Java has a unified database interface (JDBC) that is interoperable with SQLite: SQLiteJDBC4. We are standardizing on the most recent version, v056.

Your SQLite database file should be named “cs162-project3.db” without quotes. This is necessary to standardize for testing purposes.

The GameServer will store client data, game data, and game moves in the database. The tables should adhere to the schema we’re specifying here. Bold headers are the name of the table, and each bullet point is a column in the table.

clients

- int clientId primary key
- text name unique not null
- int type not null
- text password not null

games

- int gameId primary key
- int blackPlayer foreign key (clients.clientId) not null
- int whitePlayer foreign key (clients.clientId) not null
- int boardSize not null
- real blackScore
- real whiteScore
- int winner foreign key (clients.clientId)
- int moveNum not null
- int reason

moves

- int moveId primary key
- int clientId foreign key (clients.clientId) not null
- int gameId foreign key (games.gameId) not null

4 http://www.zentus.com/sqlitejdbc/
- int moveType not null
- int x
- int y
- int moveNum not null

**captured_stones**
- int stoneId primary key
- int moveId foreign key (moves.moveId)
- int x
- int y

The various primary key id fields (moveId, gameId, moveId) will auto-increment as new rows are added, and are used internally to specify relationships between rows. They are not exposed externally to clients. We have also specified foreign key relationships between clients, moves, and games, as well as some additional constraints (not null, unique). This schema is not to be modified; if you have a good reason why something should be changed or a question about what something is for, talk to a member of the teaching staff.

Note that there are relationships across rows in some of these tables. For instance, games.moveNum needs to be incremented each time a new row is added to the moves table that references that game. captured_stones rows should also be added together with the moves row that caused them. These atomic operations have to be implemented with database transactions to maintain consistency; simple application-level locking is not sufficient in the presence of failures.

- games.blackScore and games.whiteScore are set only when the game is finished.
- games.blackScore and games.whiteScore should be set according to Rules.java in the case of GAME_OK. For other cases (forfeit), the winner's score should be set to 1, and the loser's score to 0.
- games.reason is null while the game has been started, and not ended. When the game finishes, it's set to the same reason code sent out by gameOver, as defined in MessageProtocol.java when the game finishes.
- games.winner is null while the game is in progress, and is set when the game ends. There are no ties in Go.
- moves.moveNum is set to 1 for the first move in a game. Each subsequent move in the game increases its moveNum by 1.
- games.moveNum is set to the highest move.moveNum of moves related to that game. It is set to 0 initially.
- moves.x and moves.y should be set to -1 for moves that are not a MOVE_STONE.

**Fault-tolerance and Recovery**

Because the GameServer stores all game state in the database, it is possible to resume playing a game even after GameServer failure. This is done by loading all the state from the database upon startup. Here, we are specifically targeting the situation of GameServer failure and restart. Client timeouts and failures are still handled by the GameServer as in Project 2: forfeiture, closing sockets, and cleaning up state.
Clients now have to attempt to retry connecting to the GameServer if they experience a timeout or socket close. Since they have to reestablish both of their paired sockets and re-authenticate, clients should essentially reset their local state and attempt to 3-way handshake and re-authenticate via a connect message.

Immediately upon recovery, the GameServer will construct a list of partially played games. The GameServer must not accept connections until this list has been constructed. The GameServer persists this list of partially played games indefinitely, waiting for both Players to rejoin and resume playing. If only a single Player manages to reconnect, the GameServer should allow one minute from the time the first player succeeds in reconnecting for the other Player to reconnect as well. If the second player does not reconnect within this one minute, it is treated as a forfeit, and the first Player is the winner. The game is now over and the GameServer may treat it as a completed game. Players are forced to complete any of their partially played games before playing in another game. They must either play the game to completion or forfeit if they wish to move on to another game immediately.

Observer state is not saved by the GameServer, so their behavior on reconnect is no different from behavior on the initial connection. They are forced to again listGames and rejoin any games they wish to observe.

**Client-Server Protocol**

We are using the same protocol as in Project 2. Unless otherwise stated, all serialization and deserialization is performed exactly the same way as defined in the latest version of the Project 2 specification.

The list of messages and their parameters has been updated to account for the additional requirements in this project. All messages are synchronous except disconnect, which does not expect a reply.

**Client-to-server messages**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Parameters</th>
<th>Reply</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>ClientInfo player, String passwordHash</td>
<td>STATUS_OK or ERROR_REJECTED</td>
<td>Registers the client with the game server. passwordHash is the SHA-256 hash of their proposed password. ERROR_REJECTED is returned if a client with the same name is already registered.</td>
</tr>
<tr>
<td>Function</td>
<td>Parameters</td>
<td>Status Codes</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| changePassword    | ClientInfo player, String newPasswordHash                                    | STATUS_OK or ERROR_UNCONNECTED                                                                     | Notifies the server that the client would like to change its password. newPasswordHash is the SHA-256 hash of their proposed new password. ERROR_UNCONNECTED is returned if the player has not yet sent a connect message.  
| connect           | ClientInfo player, String passwordHash                                      | STATUS_OK or ERROR_REJECTED or ERROR_BAD_AUTH                                                     | Connects to the game server. The given password must be the correct password for the player for the connection to be established. ERROR_REJECTED is returned if an already connected client tries to connect again. ERROR_BAD_AUTH is returned if a client attempts to connect with an invalid password or if the client is not registered. |
| disconnect        | Asynchronous. No expected reply.                                            |                                                                                                   | Disconnects from the game server. If a Player calls this, they forfeit the game they are playing. If an Observer calls this, they leave all games they are observing. After this, the server must close down the Client’s sockets.                                                                                                                                  |
| waitForGame       | STATUS_OK or STATUS_RESUME, GameInfo game, BoardInfo board, ClientInfo blackPlayer, ClientInfo whitePlayer |                                                                                                   | For players. Signals that the player wants to play in the next game created.  
<p>|                   | -or-                                                                         |                                                                                                   |<br />
|                   |                                                                               |                                                                                                   | STATUS_RESUME is returned if the player has an unfinished game that they still need to complete. All necessary information needed to continue the game.                                                                                                                                  |</p>
<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_UNCONNECTED -or- ERROR_REJECTED</td>
<td>game is also sent back in the response. ERROR_UNCONNECTED is returned if the player has not yet sent a connect message. ERROR_REJECTED is returned if an Observer sends this message, or a Player calls this when already waiting or when in a game.</td>
<td></td>
</tr>
<tr>
<td>listGames</td>
<td>STATUS_OK, [GameInfo g1, GameInfo g2, ...] -or- ERROR_UNCONNECTED -or- ERROR_REJECTED</td>
<td>For observers. Lists the games in progress that the observer can watch. ERROR_REJECTED is returned if a Player sends this message.</td>
</tr>
<tr>
<td>join</td>
<td>GameInfo game</td>
<td>STATUS_OK, BoardInfo board, ClientInfo blackPlayer, ClientInfo whitePlayer -or- ERROR_INVALID_GAME -or- ERROR_UNCONNECTED -or- ERROR_REJECTED</td>
</tr>
<tr>
<td>leave</td>
<td>GameInfo game</td>
<td>STATUS_OK -or- ERROR_INVALID_GAME -or- ERROR_UNCONNECTED -or- ERROR_REJECTED</td>
</tr>
</tbody>
</table>
### Server-to-client messages

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Parameters</th>
<th>Reply</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gameStart</td>
<td>GameInfo game, BoardInfo board, ClientInfo blackPlayer, ClientInfo whitePlayer</td>
<td>STATUS_OK</td>
<td>Tells two players that they are playing against each other in a new game.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>blackPlayer is assigned as the black player, and moves first.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>whitePlayer is the white player.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>board is the initial board. The server picks the size, and clients should be able to handle any size between 3 and 19.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Observers may also receive this message, if they join after a game is created, but before it's started.</td>
</tr>
<tr>
<td>gameOver</td>
<td>GameInfo game, double blackScore, double whiteScore, ClientInfo winner, byte reason</td>
<td>STATUS_OK</td>
<td>Broadcasts that a game is over.</td>
</tr>
<tr>
<td></td>
<td>Extra error parameters: ClientInfo player, String errorMsg</td>
<td></td>
<td>winner is the winner of the game.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reason is either GAME_OK, or if the game ended because of a player error. The possible values for reason are specified in MessageProtocol.java.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If reason is an error (that is, not GAME_OK), the optional extra parameters indicate the player response for the error, along with a human-readable string</td>
</tr>
<tr>
<td>Function</td>
<td>Arguments</td>
<td>Status Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------------</td>
<td>--------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>makeMove</td>
<td>GameInfo game, ClientInfo player, byte moveType, Location loc, [Location capturedStone1, Location capturedStone2, ...]</td>
<td>STATUS_OK</td>
<td>Broadcasts that player placed a stone at loc. There is a moveType parameter (MOVE_STONE, MOVE_PASS), and a list of Locations of stones captured by the move. This is followed by a getMove to the player whose turn it is next.</td>
</tr>
<tr>
<td>getMove</td>
<td></td>
<td>STATUS_OK, byte moveType, Location loc</td>
<td>The server sends this to a specific player to request a move. Players respond with a moveType and a location. Specific timeouts need to be enforced for players to reply to getMove. Please refer to the latest version of the Project 2 specification.</td>
</tr>
</tbody>
</table>

**Design Document**

Your design document should include, at a minimum, the following non-exhaustive list of items:

- A description of how and when you save state in the GameServer’s database.
- A diagram of the three tables in the GameServer’s database and the relationships between them.
- A description of the changes you will be making to your GameServer, with emphasis on dealing with the database, implementing authentication, and implementing fault tolerance.
- A description of the changes you will be making to your clients, again with emphasis on the changes you will need to ensure correct behavior in the instance of GameServer failure.
- A state diagram depicting the behavior of a new player registering with the GameServer and authenticating. Transitions between states should be the messages and responses specified in the protocol section. Include initial and termination states.
- A description of why the hashing and salting scheme specified is a relatively secure way of sending and storing passwords.
- An testing plan that covers the essential classes and different aspects of system
behavior. This means both unit tests and integration tests.

## Hints

### Using EC2 Effectively

- When working remotely, at some point you will probably want to start a session, run some commands, log out, and resume the same session again later. **screen** is a useful tool for this purpose, and we recommend that you learn how to use it for this project. Here is a very simple tutorial on how to use **screen** and which commands are most important to know: [http://www.mattcutts.com/blog/a-quick-tutorial-on-screen/](http://www.mattcutts.com/blog/a-quick-tutorial-on-screen/)
- Stay tuned for an EC2 tutorial!