CS162 Fall 2011 - Project 3
Cloud Computing Go Server
(Specification Version 1.3)

Instructors: Anthony D. Joseph and Ion Stoica

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).[1] 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) [2] 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”[3] 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

[3] http://en.wikipedia.org/wiki/Salt_(cryptography)

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. A more detailed description of how to use these scripts follows under “Using Our EC2 Scripts”.

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.

Using our EC2 scripts

To run our scripts, you must be SSH-ed into `stella.cs.berkeley.edu`.

Follow these steps to set up your account to run EC2 instances:

1. SSH into `stella.cs.berkeley.edu` using your class login account.
2. Run `/var/tmp/cs162/bin/ec2_monitor --init`. This command will create an SSH key pair for you and add a `.pem` file to your home directory. You should only need to run `--init` once per class login account, but if you accidentally delete your `.pem` file, you can run `--init` again to restore it.

Once you have taken these steps, you are now ready to manage EC2 instances using your account. Each member of your group can register and manage instances independently. All management will be done using the `/var/tmp/cs162/bin/ec2_monitor` script.[4]

The `--help` command will show you how to use the script. Options are:

```
 -h, --help                  Show this help message and exit.
```

[4] You may want to edit your `$PATH` in `.bash_profile` to include `/var/tmp/cs162/bin/`, but only do this if you are sure of what you are doing.

```
 -i, --init                    Initialize your class EC2 account. Run once.

 -l, --list                    List instances that you have access to.

 -a, --launch                  Launch a new instance.

 -s STOP, --stop=STOP          Stop a running instance.

 -S, --stop-all                Stop all running instances.

 -r RESUME, --resume=RESUME    Resume a stopped instance.

 -R, --resume-all              Resume all stopped instances.

 -t TERMINATE, --              Terminate a running or stopped instance.
terminate=TERMINATE

 -T, --terminate-all           Terminate all of your instances.
```

The `--list` option will only show instances that you have launched and not yet terminated. Any stopped instances appearing in this list may be resumed using the `--resume=` option with the Instance ID of the instance you wish to resume. Options for stopping and terminating instances follow a similar pattern.

To manage a particular running instance, obtain the instance's hostname by using the `--list` option. You can then SSH into that instance by running the following command:

```
ssh -i <keyfile> ec2-user@<hostname>
```

where `<keyfile>` is the filename of the `.pem` file created when you ran `--init`, and `<hostname>` is the instance's hostname.

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.

Here is a a short example sequence of commands to `ec2_monitor`, in which we list instances, resume a stopped instance, and login to it.

```
stella [501] ~ # /var/tmp/cs162/bin/ec2_monitor -l
Instance ID   State          Uptime   Hostname
i-9d840efe    stopped        45:26
```

In this example, instance `i-9d840efe` is stopped, so we first resume it.

```
stella [508] ~ # /var/tmp/cs162/bin/ec2_monitor -r i-9d840efe
Attempting to resume instance i-9d840efe...
Done
```

We allow a minute or two after seeing this output for Amazon Web Services to actually get the instance back up. We then verify that it is running.

```
stella [510] ~ # /var/tmp/cs162/bin/ec2_monitor -l
Instance ID   State          Uptime   Hostname
i-9d840efe    running        00:00    ec2-107-22-0-156.compute-
1.amazonaws.com
```

Its hostname is `ec2-107-22-0-156.compute-1.amazonaws.com`. We proceed to login to it.

```
stella [511] ~ # ssh -i cs162-kl-default.pem ec2-user@ec2-107-22-0-
156.compute-1.amazonaws.com
```

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 teminated 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. Clients should only be able to change their own passwords.

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. The `MessageDigest` methods you will be using require byte arrays as input; `String.getBytes()` should be useful for this. Strings should be converted to bytes using UTF-16 encoding, the same encoding that Java uses internally.

Here is a complete example of how we expect you to hash and salt passwords. Suppose our password, in plaintext, is "`plainTextPassword`":

1. We convert it to byte[] using the UTF-16 character set and convert to hex, resulting in the following hash: 01faa5e19c568a4a322c8a1ee53d747c64e9e960f68bad74b5235425fe799029.
2. We append the salt, convert to byte[] again using UTF-16, compute the SHA-256 hash, and convert to hex, resulting in the following hash: b1b216d0522b7f5f841b1f5cbd8e5d86ba808a81f5e37826f28bf12dc53fe5ee

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: SQLiteJDBC[5]. 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. You should expect this file to be in the GameServer's current working directory. This means that you should be opening database connections as follows:

```
Connection conn = DriverManager.getConnection("jdbc:sqlite:cs162-
project3.db");
```

The GameServer is expected to check for the existence of the database file. If it exists and the tables exist, it should load state from the database. Otherwise, it should create this file, along with the tables specified here.

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 passwordHash not null`

games

[5] http://www.zentus.com/sqlitejdbc/

- `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`
- `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 (`clientId`, `gameId`, `moveId`, `stoneId`) 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.

In general, all state should be written synchronously to the database before it is externalized to clients. In other words, commit the results of a `getMove` before sending out `makeMove` and/or `gameOver` messages.

- `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 start a one minute timer when the first player succeeds in rejoining the server and doing a waitForGame. If the second player does not succesfully rejoin and waitForGame 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.

However, if the connected player disconnects before the second player does a waitForGame and before a minute has passed, the game remains partially played. The one minute timer will be reset and restarted the next time a player in the partially played game does a waitForGame. Players are forced to complete any of their partially played games before playing in another game. Once a partially played game has resumed, 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. `listGames` should only return the games that are currently active with both players connected, so that observers can only observe games which have both players present.

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.

Note that there is no specific name field in the database schema for storing games - you should set each game's name (in its `GameInfo` object) to be its `gameId`, as automatically assigned by the database, in string form. This will allow each game to have a unique name the server can identify it with.

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

Opcode	Parameters	Reply	Description
`register`	ClientInfo `player`, String `passwordHash`	`STATUS_OK` -or- `ERROR_REJECTED`	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, or if `register` is sent at any point after a connect, such as during a game.
`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. `ERROR_REJECTED` is returned if the player sends

			a ClientInfo that does not match the one they connected with.

`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`
		-or-
		`ERROR_UNCONNECTED`
		-or-
		`ERROR_REJECTED`
			For players. Signals that the player wants to play in the next game created.
			
			`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 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.
listGames		STATUS_OK, [GameInfo g1, GameInfo g2, ...] -or- ERROR_UNCONNECTED -or- ERROR_REJECTED	For observers. Lists the games in progress that the observer can watch. ERROR_REJECTED is returned if a Player sends this message.
join	GameInfo game	STATUS_OK, BoardInfo board, ClientInfo blackPlayer, ClientInfo whitePlayer -or- ERROR_INVALID_GAME -or- ERROR_UNCONNECTED -or- ERROR_REJECTED	For observers. Tells the server that the observer wants to join the given game. ERROR_REJECTED is returned if a Player sends this message.
leave	GameInfo game	STATUS_OK -or- ERROR_INVALID_GAME -or- ERROR_UNCONNECTED -or- ERROR_REJECTED	For observers. Tells the server that the observer wants to leave the given game. After this, the server should send at most one more message related to that game to the observer. This allows the message currently being sent to be flushed. ERROR_REJECTED is returned if a Player sends this message.

Server-to-Client Messages

Opcode	Parameters	Reply	Description
gameStart	GameInfo game,	STATUS_OK	Tells two players that they

	BoardInfo board, ClientInfo blackPlayer, ClientInfo whitePlayer		are playing against each other in a new game. blackPlayer is assigned as the black player, and moves first. whitePlayer is the white player. board is the initial board. The server picks the size, and clients should be able to handle any size between 3 and 19. Observers may also receive this message, if they join after a game is created, but before it's started.
gameOver	GameInfo game, double blackScore, double whiteScore, ClientInfo winner, byte reason Extra error parameters: ClientInfo player, String errorMsg	STATUS_OK	Broadcasts that a game is over. winner is the winner of the game. 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. 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 that provides the error message.
makeMove	GameInfo game, ClientInfo player, byte moveType, Location loc,	STATUS_OK	Broadcasts that player placed a stone at loc. There is a moveType parameter (MOVE_STONE,

	`[Location capturedStone1, Location capturedStone2, …]`		`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.
`getMove`		`STATUS_OK, byte moveType, Location loc`	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.

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. This should include some discussion of transactions, synchronization, connections, and demonstrative code for saving a new move in the database (including actual SQL statements).
- A diagram of the four 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. Please make sure to analyze both strengths and weaknesses of the system in your description.
- 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/

Specification Changelog

Version 1.0

- Initial release

Version 1.1

- Updated "EC2" section with information about and directions to using our scripts to manage EC2 instances. Added new sub-section "Using Our EC2 Scripts".
- Updated schema for the `clients` table. Specifically, changed "`text password not null`" to "`text passwordHash not null`", in order to make it clearer that the password should never be stored as plaintext.
- Fixed a typo with listing the primary key fields that auto-increment, under "Database Schema". Now it is specified that "(`clientId`, `gameId`, `moveId`, `stoneId`) will auto-increment as new rows are added".
- Added a clarification about game names under "Client-Server Protocol". Each game's name should be set to its `gameId` (automatically assigned by the database), as a string.
- Updated the `register` message, specifying that `ERROR_REJECTED` is also sent if register is sent at any point after a `connect`, such as during a game.
- Clarified that ASCII should be used for encoding characters when converting strings to byte arrays under the section "Secure Authentication".

Version 1.2

- Fixed a typo in the "Design Document" section referring to "three tables". It now says "four tables".
- AMI changes made: ports 30000-65535 have been opened, additional requested packages have been installed, and error with permissions on JAR files fixed.
- Database file location now clearly specified: it should be in the Server package.
- Specified that the database should be created by the GameServer if it does not exist; if it does, it should be read from to look for unfinished games etc.
- Specified that the results of a `getMove` should be committed to the database before state is externalized to clients: that is, before `makeMoves` and/or `gameOvers`.
- Clarified what should happen in the case that a player disconnects within the one minute

alloted for a second player to reconnect to continue a partially played game under “Fault Tolerance and Recovery”. Players cannot explicitly forfeit a partially played game until both players have reconnected and the game has resumed.

- Minor change to design document requirements. Requirements now explicitly ask you to analyze both the strengths and weaknesses of our security and authentication scheme.

Version 1.3

- Specified that when a `changePassword` is sent, the `ClientInfo` parameter passed must match that used by the client used to connect. `ERROR_REJECTED` is to be sent by the server if it does not.
- Specified the exact command used to initiate a database connection, to make it clear that the database should live in the `GameServer`'s current working directory - a change from the Server/ directory previously specified.
- Changed encoding for strings to be used for hashing to UTF-16 from ASCII. This is to keep consistency with Java's internal string encoding, and with how strings are presently encoded on the wire.
- Added a sample hash-and-salt sequence, with hash values, to make it clear how passwords should be hashed. Conversion to hex is specified for hashes.
- Specified that the one minute reconnection timer is to begin when players send a `waitForGame`, and not upon their sending a `connect`.
- Specified that observers only get to observe games that have both players. `listGames` should only return games that are actually active - that is, have both players present.