CS162 Fall 2011 - Project 2
Network Interprocess Communication
(Specification Version 2.0)

Instructors: Anthony D. Joseph and Ion Stoica
Project TAs: Karan Malik and Andrew Wang

Introduction

In the first project, you were asked to design and implement a multithreaded Go server, with separate threads for Games, Players, and Observers. In this project, you will be asked to take this one step further. Players and Observers, the clients, will run in separate processes, and will communicate with a separate GameServer process, the server. The GameServer will be multithreaded. A typical GameServer design might have one Manager thread, and two Client threads for each active client.

The main challenges here are correctly synchronizing the GameServer threads, and dealing with asynchronous client behaviour - for instance, Observers can join and leave games at any time - as well as gracefully handling player errors such as timeouts and invalid moves.

As in Project 1, we will be requiring an initial and final design document, as well as weekly group progress reports throughout. You are expected to re-use substantial amounts of code from Project 1. We are still restricting the use of built-in Java synchronization primitives, so your `Lock`, `Semaphore`, and `ThreadSafeQueue` implementations should make a reappearance.

Be warned that some amount of refactoring may be necessary due to API and structural changes. These were made with the goal of clarifying intended behavior and making the code more easily testable. We are also releasing four new code packages along with this document, which will be necessary for your implementation:

- `edu.berkeley.cs.cs162.Client`
- `edu.berkeley.cs.cs162.Writable`
- `edu.berkeley.cs.cs162.Synchronization`
- `edu.berkeley.cs.cs162.Server`

The `edu.berkeley.cs.cs162.Client` package contains a number of interfaces and abstract classes that are structured to help guide your design. You are allowed to make modifications to the abstract classes as you add shared functionality between different Players, Observers, and Players and Observers.
The `edu.berkeley.cs.cs162.Writable` package contains stubs for all the Writable types that will be passed back and forth across sockets. You should expect these to be unit tested. Feel free to add your own Writable types to this package as well.

The `edu.berkeley.cs.cs162.Synchronization` package contains a stub `ReaderWriterLock` class (see “Server Process” below for more details on this). We ask that you move your synchronization primitives (`SpinLock, Lock, Semaphore, ThreadSafeQueue`) from Project 1 into this new package.

The `edu.berkeley.cs.cs162.Server` package is where all the other miscellaneous classes from Project 1 will reside. This includes GameServer, Rules, Board, Game, and your own classes that you consider to be “Server” functionality.

You will also be making extensive use of Java TCP sockets, so please review the lecture and section notes for a refresher on network socket programming, and familiarise yourself with `java.net.Socket` and related classes. A short section on implementation tips and hints is included at the end of this document.

Overall Requirements

Unless otherwise specified, the requirements from Project 1 regarding the rules of Go carry over to Project 2. Error handling, cleanup behavior, and communication have all changed to account for the switch to a multiple process architecture.

More detail to follow, but at a high level:

- Communication between client processes (Players and Observers) and the server process (GameServer) must happen over Java TCP sockets
- Communication must follow the protocol outlined later in this document
- The `Player` and `Observer` classes now both inherit from a `BaseClient` class and implement a shared `Client` interface
- Players, Observers, and Games are no longer specified statically in a config file
- Players wait for an opponent Player to register with the GameServer before a new Game is created for the two Players
- Games no longer run in their own thread. Instead, the server spawns new Worker threads per client
- Game and Client state is now shared between Worker threads, requiring additional synchronization
- Clients and the GameServer may potentially be run on different machines
- There is no longer a `Launcher` class. Clients and the GameServer have their own `main` methods.

Server Process

The GameServer is now its own process, and it should be multithreaded. One thread - the Manager thread - should be responsible for waiting for incoming connections from clients. When a connection request from a client comes in, the Manager thread should create a new Worker thread and pass the connection to that thread.

Each Worker thread will have access to shared state in the GameServer process, such as the list of games, the list of clients, and the list of Players waiting for an opponent. You are responsible for correctly synchronizing access to this state. Your `Lock`, `Semaphore`, and `ThreadSafeQueue` implementations from Project 1 should make a reappearance here.

To receive full credit for synchronizing access to shared state in the GameServer process, we ask that you implement reader-writer locking via the provided stubbed `ReaderWriterLock` class (`edu.berkeley.cs.cs162.Synchronization`). This means that for all shared state in the GameServer, we expect reads to be allowed to happen concurrently, while restricting writes to be done by only a single thread at a time. If there are multiple readers and writers waiting for a resource, writers should get priority over readers. Note that thread cleanup becomes slightly trickier, as threads have to be sure to release locks before terminating.

Additional requirements

- For this project, the server should limit the number of connected Clients to 100
    - Additional Client connection attempts should be denied by closing the Client's socket
    - Make this number easily configurable, since it might be tweaked later.
- There should be at most `2n+c` threads running within the GameServer for `n` connected Clients, where `c` is a small constant
- A Client and the GameServer should do all of their communication over just two sockets
    - One socket is for Client -> GameServer messages
    - One socket is for GameServer -> Client messages
    - This is necessary since some messages require synchronous replies
- The GameServer must correctly clean up leftover state when a Game terminates or a Client leaves.
- Clients can leave explicitly (via a message) or implicitly (via a timeout or closing the socket). The GameServer must correctly handle both cases.
- While we are not explicitly testing for this, part of your design document grade will be based on the your GameServer's architecture in relation to scalability, efficient use of threads, and performance.
- GameServers will be passed in the IP address and port to bind to via args, e.g:
    - `java GameServer 192.168.1.100 8000`

Client Processes

As in the first project, we have two types of clients in the system: Observers and Players. They will now be running in separate processes instead of separate threads, and will be communicating with the GameServer over Java sockets.

We have refactored `Observer` and `Player` to both inherit from a `BaseClient` class that implements a parent `Client` interface to make the roles and responsibilities of the different client processes more clear. To this end, we have provided framework code for the `Client`, `BaseClient`, `Observer`, and `Player` classes, as well as stubs for `PrintingObserver`, `HumanPlayer`, and `MachinePlayer`. This code is available in the `edu.berkeley.cs.cs162.Client` package.

Observers

- Observers can observe multiple games at once
- Observers can join and leave games at any time

Players

- Players do not observe games
- Players can play in at most one game at a time
- After a `gameOver` message is sent for their game, Players must `waitForGame` again to play in another game
- `MachinePlayer` should continually wait for and play in new games, terminating if it times out when trying to reach the `GameServer`

Operating assumptions

- Clients will provide globally unique names when connecting to the server
- Clients will be passed via command line arguments their name and the IP address and port number of the GameServer like so (example - IP: 192.168.1.100, port: 8888, name: “Andrew”):
    - `java HumanPlayer 192.168.1.100 8888 Andrew`

Client-Server Protocol

An important part of this project will be carefully implementing the client-server communication protocol defined in this specification. The protocol is defined as a number of different messages, which originate from either the client or the server. Some of these messages synchronously wait for a response from the receiver, while others are asynchronous and do not expect a response. Responses are either the data the sender is expecting, or a predefined error code.

As part of this protocol, we are also requiring you to write serialization and deserialization routines for a number of different datatypes. Serialization and deserialization are used to have a consistent bit-level format when transmitting data across the network. The goal here is that different group’s implementations of Clients and the GameServer are interoperable, since they all adhere to the same protocol specification and serialization format.

Code in the package `edu.berkeley.cs.cs162.Writable` defines a number of important constants and classes to help you implement the protocol. Each message type is uniquely identified by a byte opcode - all of these, along with status and error codes for responses to messages, are defined in `MessageProtocol.java`. We also provide a `Message` abstract class that we expect you to subclass for each of the message types specified below. Finally, to assist in serialization and deserialization of messages and their parameters, we provide the `Writable` interface, which defines two methods: `readFrom()` and `writeTo()`. Classes which need to be included as message parameters or return values should implement `Writable`.

Serialization Format

A typical message exchange consists of two phases. The sender sends a message consisting of a single byte opcode and a variable length set of parameters, in the order specified. The receiver may respond, depending on the message, with a variable length set of return values or a status or error code.

The parameters and return values in many instances include classes or primitive types. To serialize and deserialize primitive types, use the Java classes `java.io.DataInputStream` and `java.io.DataOutputStream`. We provide directions on serializing and deserializing lists and classes below.

Classes that need to be serialized should implement `Writable`. To that end, you should create or modify the following classes used to represent the data that each message will contain, and implement serialization for each of them by outputting their fields in the order listed:

- `ClientInfo` - Uniquely identifies a client (either Player or Observer)
  - `String name`
  - `byte playerType` (see `MessageProtocol.java`)
- `GameInfo` - Uniquely identifies an ongoing game
  - `String name`
- `Location` - Specifies the (x, y) coordinates on a board
  - `int x`
  - `int y`
- `StoneColorInfo` - The color of a stone (black, white, or none)
  - `byte color` (see `MessageProtocol.java`)
- `BoardInfo` - The entire state of the board
  - `StoneColorInfo[][] board`: a list of lists of StoneColorInfos. Serialize by row, then column, in increasing array index order.

To serialize a list, output the following items in order:

- `int numItems`: the number of items in the list, serialized as a primitive type
- `numItems items`, each serialized individually either as a `Writable` or a primitive type

As an example, if we have a list of two `Locations`, [L1 L2], we would output the following in order:

- `int 2`

- `int L1.x`
- `int L1.y`
- `int L2.x`
- `int L2.y`

with each `int` serialized using `java.io.DataOutputStream`.

String serialization is considered a special case of list serialization, with the string treated as a list of characters. To serialize strings, output the length of the string as an `int` first, followed by the characters. Use `DataOutputStream.writeChars()` for the latter.

As an example, the string "abc" would be serialized as follows:

- `int 3`
- `char a`
- `char b`
- `char c`

3-way Paired Handshake Protocol

Clients and the GameServer will communicate over a pair of sockets. One socket will be used for client-to-server messages, and one for server-to-client messages. The need for paired sockets is because the Client needs to be able to send new messages to the server, even if the server is waiting for a reply to a message it sent. As a concrete example, imagine that the Server sends a `getMove` message to Player, and is waiting for a `Location` where the Player is moving in response. If the Player instead wants to forfeit (by sending a disconnect message, an operation that can be done at any time), this could not be supported by just a single socket. In short, having two sockets, one for each direction of messages being sent, allows both the client and server to send synchronous messages at the same time.

The Client and GameServer should use the following protocol to logically link a pair of sockets together as being from the same Client. Take note that before clients and the server exchange any of the messages described in the tables below, they will need to complete this 3-way paired handshake. That is, before a `connect` message can be sent, the 3-way handshake must first be completed.

This protocol bears a striking resemblance to TCP's 3-way handshake protocol. We are using the same terminology here. Every message exchanged in this protocol consists only of `ints`.

- Client opens two sockets with the server
- Client generates a random `int`, `rand1`
- Client sends SYNs:
    - On both sockets: send `rand1`
- When the server sees `rand1` from two different sockets, it can pair them as being from the same client

- Server generates two random `ints`, `rand2` and `rand3`, such that `rand2` < `rand3`.
- Server arbitrarily designates one socket as socket 1 for client->server communication and another as socket 2 for server->client communication.
- Server sends SYNACKs:
  - On socket 1 (client->server): send `rand2`, followed by `rand1+1`
  - On socket 2 (server->client): send `rand3`, followed by `rand1+1`
- Since `rand2` < `rand3`, the client now also knows which is socket 1 and which is socket 2.
- Client sends ACKs:
  - socket 1 (client->server): send `rand2+1`
  - socket 2 (server->client): send `rand3+1`

After the handshaking process is complete, the random SYN and ACK numbers will not be used for anything else (unlike handshaking in TCP). Errors at any point during handshaking can be resolved by closing the sockets. The server should discard dangling sockets after a 3 second timeout, if the handshaking process does not complete. The normal 3s socket timeout also applies to each message exchanged, not to the entire handshaking process.

Finally, note that it is not necessary for you to account for potential random number collision.

After the 100 client limit is reached, the server should also refuse socket connections from additional clients. Below the 100 client limit, any number of half-open connections is still allowed (within reason), though they are still subject to the 3s timeout.

Client-to-server messages

Opcode	Parameters	Type	Reply	Description

`connect`	ClientInfo `player`	sync	`STATUS_OK`
			-or-
			`ERROR_REJECTED`
				Connects to the game server.
				
				`ERROR_REJECTED` is returned if an already connected client tries to connect again.

`disconnect`		async		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 can close down the Client's sockets.

`waitForGame`		sync	`STATUS_OK`
			-or-
			`ERROR_UNCONNECTED`
				For players. Signals that the player wants to play in the next game created.

`listGames`		sync	`STATUS_OK, [GameInfo g1, GameInfo g2, ...]`
			-or-
			`ERROR_UNCONNECTED`
				For observers. Lists the games in progress that the observer can watch.

`join`	`GameInfo game`	sync	`STATUS_OK, BoardInfo board, ClientInfo blackPlayer, ClientInfo whitePlayer`
			-or-
			`ERROR_INVALID_GAME`
			-or-
			`ERROR_UNCONNECTED`
				For observers. Tells the server that the observer wants to join the given game.

`leave`	`GameInfo game`	sync	`STATUS_OK`
			-or-
			`ERROR_INVALID_GAME`
			-or-
			`ERROR_UNCONNECTED`
				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.

Server-to-client messages

Opcode	Parameters	Type	Reply	Description

`gameStart`	`GameInfo game, BoardInfo board, ClientInfo blackPlayer, ClientInfo whitePlayer`	sync	For Players:
			
			`STATUS_OK`
			-or-
			`ERROR_REJECTED`
			
			For Observers:
			
			When sending to an observer, no reply is expected.
				Tells two players that they 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. `ERROR_REJECTED` is sent by the client if it doesn't want to play in the game.
`gameOver`	`GameInfo game, double blackScore, double whiteScore, ClientInfo winner, byte reason` Extra error parameters: `ClientInfo player, String errorMsg`	async		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, [Location capturedStone1, Location capturedStone2, …]`	async		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.
`getMove`		sync	`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`. See later sections on “Error Handling and Fault Tolerance” and “Go Game Logistics”.

Error Handling and Fault Tolerance

All socket communication should be set by default to use a timeout of 3 seconds. See the `java.net.Socket.connect()` and `setSoTimeout()` methods. This socket timeout applies to the initial connect(), writing a message, and reading the response to a synchronous message. Put another way, the socket timeout is relevant for situations where the server or client is unable to make expected contact within the timeout. In the case of a timeout, relevant server-side state should be cleaned up, and all sockets associated with the client should be explicitly closed.

On the server side, if there is a socket timeout, it should drop the client - that is, close the client’s sockets and clean up state. If the client that times out is a Player that was playing in a game, this results in a forfeiture and ending of the game. In the specific instance that a timeout happens when waiting for a reply to a `gameStart` message, or if the response to the `gameStart` message is `ERROR_REJECTED`, the offending player should be considered to have forfeited and the game should be ended. Finally, if the client that times out is an Observer, no externally visible action needs to be taken besides cleaning up server-side state.

Besides socket timeouts, the server will also need to enforce move timeouts on players in games. Clients register as either a Human, Machine, or Observer when initially connecting to the server. When the server sends a `getMove` message, `HumanPlayers` are allocated 30 seconds and `MachinePlayers` 2 seconds from the time the server starts sending the message and finishes receiving its response. This differs from the normal socket timeout of 3 seconds, and might need to be enforced differently for correctness. The actual remote procedure call (e.g., the complete round trip of sending the message and receiving the response) can take more than 30 seconds in the case of a very slow client, but will still result in a forfeit.

If a player fails to respond within the given time, a `gameOver` message with the correct `reason` parameter and the relevant optional parameters should be sent to all other participants of the game. The server also closes down the client sockets and cleans up state in the same manner as in a normal timeout.

After a client completes the 3-way handshake, it must send a `connect` message before sending any other messages, besides a `disconnect` message. In the case of invalid behavior here, the server should reply back with `ERROR_UNCONNECTED`, and simply close the client’s sockets and clean up state.

In the case that the server receives an invalid message opcode, or a corrupted or otherwise unparseable message, the server should again simply close the client's sockets and clean up state. This is essentially the only way to recover when the server does not know what to expect next from the client. In the event that the server receives a mangled stream of bytes, it has no way of knowing which byte is the opcode of the next message. Thus, it is best off terminating the connection and starting over.

The server is trusted enough to not send the client mangled messages, but you should add the same type of cleanup behavior to `MachinePlayer`, where the client will close down sockets and cleanly exit in the case of a timeout.

Example client-server message exchange

We are going to walk through an example exchange between two Players P1 and P2, an Observer O1, and a GameServer process G1. This will show the expected messages between entities. Some parameters, return values, and status codes are elided for simplicity; refer to the actual specification for the actual and complete message formats.

First, P1 and P2 both complete the 3-way handshake with G1.

Next, P1 and P2 register with G1 via connect messages:

```
P1: connect pinfo1 → G1
G1: OK → P1

P2: connect pinfo2 → G1
G1: OK → P2
```

Next, P1 tells G1 it wants to play in the next game:

```
P1: waitForGame → G1
G1: OK → P1
```

Next, P2 tells G1 it wants to play in the next game. Since P1 is also waiting, G1 starts a new game for the two players:

```
P2: waitForGame → G1
G1: OK → P2
G1: gameStart → P1
G1: gameStart → P2
```

G1 now explicitly requests a move from P1:

```
G1: getMove → P1
P1: OK, loc → G1
```

G1 broadcasts the move to both P1 and P2:

```
G1: makeMove loc → P1
```

```
G1: makeMove loc → P2
```

Now G1 explicitly requests a move from P2:

```
G1: getMove → P2
P2: OK, loc → G1
```

… and so on.

Observer O1 now connects, and lists the available games:

```
O1: connect oinfo1 → G1
G1: OK → O1
O1: listGames → G1
G1: [GameInfo g1] → O1
```

O1 now knows about the game that P1 and P2 are playing in, and asks to join it:

```
O1: join g1 → G1
G1: OK, Board, pinfo1, pinfo2 → O1
```

O1 now has the state of the game up to that point, and receives all the move updates.

Pretend now that P1 and P2 pass in sequence, and the game is over. G1 now needs to terminate the game.

```
P2: OK, MOVE_PASS, loc → G1 # this is the second pass
G1: makeMove → P1
G1: makeMove → P2
G1: makeMove → O1
G1: gameOver → P1
G1: gameOver → P2
G1: gameOver → O1
```

P1 and P2 now need to `waitForGame` again before they play in another game. O1 can `listGame` again and start observing additional games.

Go Game Logistics

- The black player plays first.
- When a player makes an invalid move, the GameServer should send a `gameOver` message, with the reason set to `PLAYER_INVALID_MOVE`. These and other reasons are defined in `MessageProtocol`.
- Note that we are using the Ko rule as described in Project 1 (not on Wikipedia!)

Design Document

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

- A high-level description of your GameServer, and how you had to adapt it to a networked environment
- A high-level architecture diagram of the GameServer and a number of Clients, showing what is a process vs. thread, socket connections, and important bits of global state (e.g. arrays, queues)
- A state diagram depicting the behavior of the client. Transitions between states should be the messages and responses specified in the protocol section. Include initial and termination states.
- A description of how you designed your serialization and deserialization classes, and how they are used in your system.
- A description of how you plan to implement reader-writer locking and fill in the `ReaderWriterLock` class. Details of `SpinLock`, `Lock`, `Semaphore`, and `ThreadSafeQueue` can be elided since they are Project 1 material.
- Descriptions of your `Player` and `Observer` classes.
- An testing plan that covers the essential classes and different aspects of system behavior. This means both unit tests and integration tests.

Hints

Implementation

- A portion of your grade is going to be based on code style. Make sure to comment your code, use sensible variable names, indent correctly, etc. Define a group policy and stick to it.
- The `flush()` method can be used to force a `Socket` to send the contents of its send buffer. This is useful when sending small amounts of information and then waiting for a reply.
- Look at the `ServerSocket` and `Socket` Java classes.

Andrew wrote a little multithreaded server (`Provider.java`) and client (`Requester.java`) which follows the basic architecture that should be used for the multithreaded Go server. The Provider listens on a `ServerSocket` and spawns a new worker thread for every new connection. It might be useful as a reference.

See here: https://github.com/umbrant/iowt/tree/master/iowt-java-mt

Testing

- Try testing with another team's server and client classes. This helps ensure interoperability and adherence to the specification.
- Test each `Writable` class's serialization and deserialization routines separately
- Go down the list of errors that each message could return, and ensure that they are being handled correctly.
- Ensure timeouts are being handled correctly, both in the client and server processes
- Make sure you are enforcing the game mechanics, as described both in Project 1 and in

this project

Specification Changelog

Version 1.0

- Initial release

Version 1.1

- Minor fixes and clarifications
- List all of the classes within Synchronization package
- Clarify global vs. game timeouts
- Specify String serialization as a subcase of list serialization
- Modify “sendMove” message to have a new parameter for pass moves vs. normal

Version 1.2

- Specified behavior when client returns `ERROR_REJECTED` to `gameStart` message
- Further clarified socket timeouts
- Added `BoardInfo` as part of `gameStart` message
- Removed need for Observer ack when it receives a `gameStart`
- Removed requirement for error states in design doc state diagram
- Specified cmdline args for GameServer IP addr and port #
- Renamed `StoneColor Writable` to `StoneColorInfo` to prevent conflict with `StoneColor enum` from project 1 code
- Added `BaseClient` abstract class that implements `Client`. `Player` and `Observer` now extend `BaseClient`.
- Client.`getPlayerInfo()` renamed to `Client.getClientInfo()`
- Made `Writeable.readFrom()` and `writeTo()` throw `IOException`

Version 2.0

- Raise the limit on # of threads to be 2n+c, n is the # clients, c is a small constant.
- `makeMove` has a new `moveType` parameter for pass moves
- New `edu.berkeley.cs.cs162.Server` package for server stuff
- Removed mention of `DataInputStream.readFully()`, since Java uses UTF-16 (2 bytes), not ASCII (1 byte).
- Client now takes the client’s name as a command line argument.
- Instead of having `stoneCaptured` messages, the list of captured `Locations` is passed as part of the `makeMove` message
- The `sendMove` message (client to server) has been functionally replaced with `getMove` (server to client)
- The `playerError` message has been folded into `gameOver`, since it was redundant
- The disconnect message no longer needs an “ack”.
- Further clarified player move timeouts vs. normal socket timeouts