CS162 Fall 2011 - Project 2
Network Interprocess Communication

(Specification Version 1.1)

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, with one Manager thread, and one Client thread for each active client.

The main challenges here are correctly synchronizing the GameServer threads, and dealing with asynchronous client behaviour - Observers can join and leave games at any time, among other things - 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 three 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`

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.

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 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 a new Worker thread 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. That Worker thread is then responsible for handling all further communication with the client.

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
- There should be at most `n+1` threads running within GameServer for `n` connected Clients
- A Client and the GameServer should do all of their communication over a single socket
- 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.

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 a Java socket.

We have refactored `Observer` and `Player` to both implement 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`, `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 the IP address and port number of the GameServer like so (example - IP 192.168.1.100, port 8888):

    - `java HumanPlayer 192.168.1.100 8888`

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 are synchronous and expect 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`
- `StoneColor` - The color of a stone (black, white, or none)
    - `byte color` (see `MessageProtocol.java`)
- `BoardInfo` - The entire state of the board
    - `StoneColor[][] board`: a list of lists of `StoneColors`. 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. We will only be dealing with ASCII encoded strings for this project. You could potentially use DataOutputStream.writeChars() and DataInputStream.readFully() here.

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

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

Client-to-server messages

Opcode	Parameters	Type	Reply	Description

connect	ClientInfo player	sync	STATUS_OK -or- ERROR_REJECTED	Connects to the game server.
disconnect		sync	STATUS_OK -or- ERROR_UNCONNECTED	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.
waitForGame		sync	STATUS_OK -or- ERROR_UNCONNECTED	For players. Signals that the player wants to play in the next game.
sendMove	byte moveType, Location loc	sync	STATUS_OK -or- ERROR_UNCONNECTED	For players. Makes the given move for the given name. Valid values of moveType are specified in MessageProtocol.java In this case, the “STATUS_OK” reply is essentially an ack. If the move is illegal, the server will send a playerError message to the player.
listGames		sync	STATUS_OK, [GameInfo g1, GameInfo g2, ...] -or- ERROR_UNCONNECTED	For observers. Lists the games in progress that the observer can join.
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.

Server-to-client messages

Opcode	Parameters	Type	Reply	Description
gameStart	GameInfo game, ClientInfo blackPlayer, ClientInfo whitePlayer	sync	STATUS_OK -or- ERROR_REJECTED	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. 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	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 error code sent in the preceding playerError message.
makeMove	GameInfo game, ClientInfo player, Location loc, int	async		Broadcasts that player placed a stone at loc, and to expect numCaptured stoneCaptured messages

	`numCaptured`			
`stoneCaptured`	`GameInfo game, ClientInfo player, Location loc`	async		Broadcasts that the stone at `loc` was captured by `Client` player
`playerError`	`GameInfo game, ClientInfo player, byte errorNum, String errorMsg`	async		Broadcast when `player` makes an error that results in the end of the game. Immediately followed by a `gameOver` message. The possible values for `errorNum` are specified in `MessageProtocol.java`. `errorMsg` is an optional human-readable string, related to the error.

Error Handling and Fault Tolerance

All sockets should be set to use a timeout of 3 seconds. See the `java.net.Socket.connect()` method, which takes a timeout. Clients and the Server should handle both these timeouts and explicit closing of the socket.

Players may optionally retry any operation if they are unable to communicate with the server, up to a total of three times. Since this is an optional feature, it's up to you to decide the semantics.

The server should retry communication with the client, but should drop the client (that is, close the client's socket and clean up state) if the connection times out a total of three times. If the client that times out is a Player that was playing in a game or waiting for a game, this results in a forfeiture and ending of the game. If the client that times out is an Observer, no externally visible action needs to be taken besides cleaning up server-side state.

The server will also need to enforce move timeouts on players in games. When the server sends a `makeMove` message, it should expect a `sendMove` message within 30 seconds for `HumanPlayers`, and within 2 seconds for `MachinePlayers`. Clients register as either a Human, Machine, or Observer when initially connecting to the server. (See "Go Game Logistics" for a description of the expected behaviour upon such a timeout.)

In the case that the server receives a valid message, but before a `connect` message - that is, it

responds with the status code `ERROR_UNCONNECTED`, it should after sending this response again simply close the client's socket and clean up state.

In the case that the server receives an invalid message opcode, or a corrupted or otherwise unparsable message, the server should again simply close the client's socket 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. Suppose the server receives a mangled stream of bytes - then how does it know which byte is the opcode of the next message? Hence, it is best off terminating the connection and starting over.

The server is trusted enough to not send the client mangled messages, but you may need to add the same disconnect/reconnect logic as the server to the client to assist in debugging, since again, it is hard to recover with this protocol after the first mangled message.

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 spec for the actual and complete message formats.

First, 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
```

P1 and P2 now start to play the game, with the active player sending a `sendMove`, and the server broadcasting it to everyone:

```
P1: sendMove → G1
G1: makeMove → P1
G1: makeMove → P2
```

… 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: sendMove → 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

- When a player times out, the GameServer should send a `playerError` message, followed by a `gameOver` message.
- When a player makes an invalid move, the GameServer should not send a `makeMove` message. Instead, it should send a `playerError` message, followed by a `gameOver` message.
- Here are some of the events that should cause the server to send a `playerError` message followed by a `gameOver` message:
    - A player makes a move out of turn
    - A player places a stone in an invalid board location
    - A player violates the Ko rule, as described in Project 1 (not on Wikipedia!)
    - A player hits their move timeout

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, error, 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