Bridge Defense

In this assignment we will develop a client for a Bridge Defense game (inspired by Tower Defense games). The game has a simple rule set and a communication protocol that students must implement.

Goals:

Introduce the concept of multiplexing communication across multiple sockets.
Reinforce data encoding and decoding concepts.
Introduce the concept of error detection and packet retransmission.
Rules

In Bridge Defense, the player (your program) controls a set of cannons mounted on bridges that cross a set of parallel rivers. Rivers are numbered from 1 to 4, bridges are numbered from 1 to 8, and cannons are identified by their coordinates. The figure below shows an example game "board":

Bridge Defense Board State

The game proceed in turns. The beginning of a turn is triggered by the player (your program). In each turn, the following events happen:

New ships arrive at the first bridge. Each ship not sunk in the previous turn advances to the next bridge.

Each cannon can fire one shot per turn. A cannon can fire at a ship crossing its own bridge in any of the adjacent rivers. The table below describes the number of targets available to each cannon in the example board above:

Cannon	Possible targets
(1, 0)	1
(3, 0)	0
(N, 1)	2
(2, 2)	1
(3, 3)	0
(N, 4)	0

The game has three ship types, each requiring a different number of shots to sink:

Ship	Shots to sink
Frigate	1
Destroyer	2
Battleship	3

For the purposes of scoring, the servers track three metrics. The lower the values, the better:

The number of ships that were not sunk and crossed the last bridge
The number of protocol messages received by the servers (see below)
The total time the program took to complete the game
Protocol

The protocol used in this assignment uses the UDP transport protocol and encodes messages using JSON. All messages are a JSON object that contains, at least, the auth and type fields. The auth field should contain a Group Authentication Sequence (GAS), generated using a client for the authentication protocol in assignment 1. The type field should be one of the message types defined below. Both type and auth fields contain values that are strings. Different messages may have additional fields, which will be specified below together with the semantics for each message. The client program is responsible for implementing a timer to detect transmission errors and retransmit lost packets.

The protocol considers that each river is controlled by a separate server. The client program must connect and run the protocol with all four servers concurrently. Operations such as authentication, reception of ship information, and confirmation of cannon shots should be coordinated with each server separately.

Authentication Request [authreq]

A client starts a connection with the server sending an authentication request with type equal to authreq, passing in the Group Authentication Sequence (GAS):

```
{
   "type": "authreq",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... "
}
```

The client should send an authentication request and receive a response from each of the four game servers.

Note that a server identifies clients by the GAS in the auth field. Each client needs to open four sockets and authenticate with the four different servers using the same GAS across all servers.

If you execute two instances of your clients (each instance opens four sockets), each instance will need to use a different GAS or conflicts will happen on the server side when handling client messages. Use a different GAS for each client instance in execution.

Authentication Response [authresp]

A server answers an authentication request with an authentication response with type equal to authresp containing the client's GAS in the auth field. The response also contains a status field with value 0 in case the authentication succeeded (i.e., the GAS is valid) and with value 1 in case the authentication failed. The response also contains a river field indicating which river is controlled by that server. For example:

```
{
   "type": "authresp",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "status": 0
   "river": 1
}
```

A client should track authentication state with each server individually. A client should proceed to the next steps only after authenticating with all four servers.

Cannon Placement Request [getcannons]

The client program should start the game after authentication sending a request for cannon placements. This should be done after the client has successfully authenticated with all game servers. A getcannons request has only the type and auth fields:

```
{
   "type": "getcannons",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
}
```

The client can send the getcannons request to any server, and getting a response from any server is enough: the placement is shared across all servers and remains fixed throughout the game.

Cannon Placement Response [cannons]

Any server responds to a getcannons request with a response containing cannon placements. The response contains a cannons field that is a list of coordinates indicating the bridge and the river where each cannon is located. For the cannons in the example board above, the server would send the following response:

```
{
   "type": "cannons",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "cannons": [[1, 0], [3, 0], [N, 1], [2, 2], [3, 3], [N, 4]]
}
```

The river coordinate (y) of a cannon ranges between 0 and 4, as in the example board state above. A cannon can fire at ships on adjacent rivers. For example, a cannon at coordinate (X, 1) can fire at ships crossing bridge X on rivers 1 and 2. Cannons with river coordinates 0 and 4 can only fire at ships in rivers 1 and 4, respectively.

Turn State Request [getturn]

The client program should advance the state of the game by sending a getturn request to servers. Turns are numbered from zero and should be requested in order. The turn number is indicated by the turn field. After obtaining cannon placements, the client should start the game requesting the state of turn zero:

```
{
   "type": "getturn",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "turn": 0
}
```

When the client is ready to proceed to the next turn (see below), the client should send getturn messages to receive the state of the next turn. For example, after firing shots and receiving shot confirmations for turn zero, a client should request the next turn:

```
{
   "type": "getturn",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "turn": 1
}
```

Clients should send getturn messages to each server separately. Clients should reconstruct the turn state from all servers by retransmitting getturn request as needed before proceeding to the next stages.

Turn State Response [state]

A server answers one getturn request with multiple state responses. Each state message refers to a specific turn (indicated by the turn field) and a specific bridge (indicated by the bridge field). A state message also contains a ships field that contains a list of ships crossing a bridge at a given turn. (Note that multiple ships may be crossing a bridge in the same river.) The river is derived by the server sending the message, as each server handles a single river. When no ships arrive at a bridge at a given turn, the server sends an empty list in the ships field. For example, the server which controls river 2 in the example will answer a getturn request with the following messages (assuming the game is on turn 133):

```
{
   "type": "state",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "turn": 133,
   "bridge": 1,
   "ships": [{ ... }]
}
{
   "type": "state",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "turn": 133,
   "bridge": 2,
   "ships": []
}
{
   "type": "state",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "turn": 133,
   "bridge": 3,
   "ships": []
}
{
   "type": "state",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "turn": 133,
   "bridge": 4,
   "ships": [{ ... }]
}
```

A ship is encoded as a JSON object (dictionary) with three fields. The id field is a unique integer identifier for a ship. A ship's id is kept fixed as the ship advances bridges across turns. The hull field is a string that identifies the type of ship, and can contain either frigate, destroyer or battleship. The hits field is an integer indicating how many shots have hit that ship. (Frigates, destroyers, and battleships are sunk with 1, 2, or 3 hits, respectively.) For example:

```
{
   "id": 51,
   "hull": "destroyer",
   "hits": 1
}
```

Shot Message [shot]

After reconstructing the state of a turn, the client should decide which ships each cannon should fire at, and send a shot message to the server responsible for managing the ship being fired at, as determined by the river the ship is on. A shot message contains a cannon field with a 2-element list indicating the cannon firing the shot, as well as an id field indicating the ship being fired at. For example:


```
{
   "type": "shot",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "cannon": [2, 2],
   "id": 51
}
```


Shot Result [shotresp]

A shot message is answered by the server with a response indicating if a shot is valid or if an error occurred. A shot result message has type equal to shotresp, the cannon identifier in a cannon field, the target ship's id, and a status field with value 0 if the shot is valid and 1 otherwise. In case of an error, the message contains an additional description field to explain the error. For example:


```
{
   "type": "shotresp",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "cannon": [2, 2],
   "id": 51,
   "status": 0
}
{
   "type": "shotresp",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "cannon": [3, 0],
   "id": 51,
   "status": 1,
   "description": "out of range"
}
```


Game Over Message [gameover]

Servers that receive invalid messages from clients or without an ongoing game will send a gameover message to clients. After receiving a gameover message, the client should terminate the current game and start a new one (starting with authentication).

The server also sends a gameover message to a getturn message when the game ends.

A gameover message has a status field with value 0 if the game has completed successfully, and with value 1 in case of early termination. The gameover message also contains a score field containing a JSON object with metadata about the game. If the game is terminated early (status equal to 1), then a description field is added to the gameover message to diagnose the reason.


```
{
   "type": "gameover",
   "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... ",
   "status": 0,
```

```
    "score": { ... }
}
```
Servers answer clients without an active game with a gameover message with a description of "client unknown". Many of these messages may be received after a game is terminated early due to an error. As a result, when a game it terminated early by a server, the client should retrieve the first gameover message, which contains the primary cause for the early termination.

Game Termination Request [quit]

A client can terminate a game at any point by sending a termination request to any server. In particular, clients must terminate a game after receiving a gameover message with a status of zero after completing a game. A quit message received by any server will terminate the game in all servers immediately. A quit message contains only the auth and type fields:

```
{
    "type": "quit",
    "auth": "ifs4:1:2c3b... +ifs4:2:cf87... +e51d06... "
}
```
Control and Execution

The control and execution of the protocol, and thus the advancement of the game, is the responsibility of clients. Clients should authenticate with servers, request the cannon placements, advance turns, and compute shots.

Note that communication between clients and servers is subject to errors, and messages can be lost. In case a message is lost in either direction, clients should retransmit the corresponding request to obtain the responses.

The servers also track of the game's play time, which requires timely retransmissions, and the number of received requests, which penalizes spurious retransmissions. The retransmission algorithm is one important factor in your assignment's grading. You should design a retransmission algorithm that attempts to minimize the waiting time as well as the number of retransmissions. Describe your algorithm in your documentation.

Implementation Details

Servers

The servers will run on the class's virtual machine. We will run four sets of servers in three different "difficulty" settings. The different sets of servers will allow students to test against different servers during development and avoid impacting each other if the servers crash.

Easy difficulty servers will run on ports starting at 5111X, 5112X, 5113X, and 5114X, where X varies between 1 and 4. Remember that a client should connect to 4 different server ports (one per river). These ports should be consecutive and have the same initial 4 digits; in other words, only the rightmost digit of the port numbers should change. For example, a client can connect to the servers on ports 51121, 51122, 51123, and 51124.

Medium difficulty servers will run on ports starting at 5121X, 5122X, 5123X, and 5124X, where X varies between 1 and 4. Medium difficulty emulate the loss of messages received from clients, and thus require retransmissions. This emulates message corruption in real networks.

Hard difficulty servers will run on ports starting at 5131X, 5132X, 5133X, and 5134X, where X varies between 1 and 4. Hard difficulty servers emulate the loss of messages received from and sent to clients.

Easy difficulty server sets (choose one set):
Set 1: 51111, 51112, 51113, 51114
Set 2: 51121, 51122, 51123, 51124
Set 3: 51131, 51132, 51133, 51134
Set 4: 51141, 51142, 51143, 51144

Medium difficulty server sets (choose one set):
Set 1: 51211, 51212, 51213, 51214
Set 2: 51221, 51222, 51223, 51224
Set 3: 51231, 51232, 51233, 51234
Set 4: 51221, 51222, 51223, 51224

Hard difficulty server sets (choose one set):
Set 1: 51311, 51312, 51313, 51314
Set 2: 51321, 51322, 51323, 51324
Set 3: 51331, 51332, 51333, 51334
Set 4: 51321, 51322, 51323, 51324
User Interface

The client program should receive the server's address, the first port number (port numbers are assumed sequential), and the Group Authentication Sequence to be used for authentication:

./client <hostname> <port1> <GAS>
The GAS should be transmitted as defined in assignment 1. Client programs should support both IPv4 and IPv6.

Suggestions

Implement one request at a time. Start with authreq, then getcannons, then getturn, and finally the shot message. Test each request by sending them to the easy servers and inspecting the responses.

Ambiguities, Errors, and Extra Credit

Whenever a server crashes, it stays 1 minute offline before restarting. The downtime allows the student to detect that the server has crashed. Identifying reproducible server crashes and documenting how to crash the server is worth extra credit. The amount of extra credit for reporting bugs depends on the severity, subtlety, and complexity of the issue.

Grading

You should submit the source code for your client program, a PDF with documentation. You should also execute your client against the easy, medium, and hard difficulty servers before the deadline using both IPv4 and IPv6 protocols.

Source Code

Submit your source code on the assignment page. For implementations in compiled languages, provide documentation on how to compile your program.

Documentation

Your documentation should contain at most 4 pages in 10pt font. You should describe any challenge you encountered during the implementation. In particular, you should describe (1) the strategy you used to reconstruct the state of a turn when sending getturn messages, (2) your retransmission algorithm, (3) how you manage concurrent communication with the four servers, and (4) the strategy you used to decide which shots to take. Explicitly, state the GAS that should be considered for grading your games. The documentation is worth 4 points.

Games

An automated grading script will process server logs to check your client's operation. Completed games are worth 11 points: 4 points in the easy difficulty and 3 points on medium and hard difficulties.