A multi-threaded game server

Florian Suess / 187147214

***Abstract*—Implementing a multithreaded game server suitable for the well known two-player chess game.**

I. IMPLEMENTATION

We begin with the restriction of having to use nothing more than a simple Berkeley socket abstraction [1]. Thankfully C#'s .NET implementation offers quite some customizability and offers TCP obliging protocol out of the box. Considering our client is a full fledged modern browser, we can assert quickly that we encounter a robust FIFO communication stream of packets. The implementation includes support for keeping sockets alive without any manual intervention either.

A. Handling concurrent live sockets

We host a main thread that goes through a establish, hand-off, establish loop. Each "establish" contains the process of listening on a port for client connection. Each new connection created gets immediately handed off onto a thread. We queue up to 10 pending connections that arrive simulatenously, to cater for an overrun of the main thread before refusing any further piling of requests.

The instantiation of the thread, takes with it the newly minted socket connection. At this point the main thread does not see it again as the entire lifecycle of the socket connection will be handled by this child process. Indeed we respect the request to keep requests alive, and hence these sockets are configured accordingly [2]. The thread continouly reads the stream of bytes in a buffered manner. Now to reading the request...

B. Implementing a bare minimum subset of HTTP/1.1

We submit to the transfer protocol format choice our set of modern browsers wouldn't dear deviate from; 'HTTP/1.1' [3].

There are two main abstractions implemented that closely hug 'HTTP/1.1' semantics, you'll notice a 'Request' class; hosting the responsibilities of understanding a request, coupled with the 'ResponseWriter' class; hosting the responsibilities around writing a response back to the client. This particular pattern has been drilled into me whenever handling web serving systems - for me at least inspired directly from the implementation details of Go's net/http package. [4]

1) Request: There seemed to be no low level utility in .Net 6 that I could find that, independently from our socket choice, that would make sense of an incoming HTTP/1.1 shape stream of bytes for me and so, reluctantly, we do a bit of re-implementing. As the socket receives bytes from a client, we look for a new line delimitation between the header and the potential request body. Byte sequence '13, 10, 13, 10'. We assume the connection has yet to receive the entire request until we encounter this unique sequence.

Upon receiving we now preliminarily check the headers 'Content-Length' header (extracted via regex), and use this value to determine the size of the body to receive. If the header is omitted we can safely assume an empty body and signal to the process that the client message has been fully received.

Once the message is "received" we break down the header, completely ignoring the importance of the request body, using it instead to solely derive the intended route. We do however introduce some formality by restricting to the set of 'POST/GET' requests (as per relevant regex shows).

Here we can actually understand the requests provided by the client and can deduce what is "intended". The set of functionalities provided are then inside a business logic layer matched with the intention found. We will touch on all of this later.

2) ResponseWriter: Lets now turn our focus on what happens when formulating a response. Due to our limited ability to modify the behaviour of a modern browsers HTTP client, we must naturally adopt HTTP/1.1 valid response form. Recall our restriction of basic HTTP/1.1 utilities in .NET6 when forced to use a synchronous socket as they come at head again. With the first line of the header reserved to show concisely the status of the serving of the request, we are given a bone via 'HttpStatusCode' [5]. The easy availability allows easy responses without an ad-hoc implementation of the available HTTP/1.1 status codes. We indicate the 'Content-Length' alongside some feeble COR's headers. Notably we adopt JSON encoding outgoing payloads for the body portion of any sent response.

C. Componentization

In confidence I followed a path of identifying components to help with the clarity of the growing codebase[6]. Delineation of the responsibilities re: request handling vs business logic helped establish clear direction forward. 'handler' clearly operates on the raw consumption and construction of requests, 'mux' to map request intention to capabilities and in a completely separate class lives 'GameState' that lives in complete isolation focusing on the core business logic side of things.

The strong boundary between implementation details and the 'GameState' allowed for concurrency control instruments to live closer to the center of the system which ultimately mediate the integrity of our game facilitation, reducing the size of this critical section was in central focus to putting performance on some pedestal.

D. concurrency control instrumentation

Given we can draw a clear boundary between 'GameState' and the rest of the system. We define a mutex to live here. The

public methods are responsible for obtaining a mutex lock, as the code path related to reading and writing game state concurrently is indeed simplest to remain mutually exclusive between threads. Albeit some rudimentary validation is done outside of the 'GameState', including the aforementioned mechanics behind interpretting the request and sending a response, operations on anything related to 'GameState' are strictly locked to concurrency issues.

E. the client

We make use of the fetch API, passing in the option to keep connections alive and guide the user through UI cosmetics around the variety of functionalities this server provides.

I won't go into depth of each route, I followed closely the spec recommended in the assignment overview for your ease of understanding. Nevertheless, the seperate key options we have are:

- Registering the client with the server.
- Pairing the registered client with a game on the server.
- The ability to submit a next move.
- The ability to check for the next move.
- Quitting the current game.

Each operation was implemented in tight accordance with 'HTTP/1.1', agreeing on tactical status codes (in particular '404', '423') to guide certain scenarios concisely.

The client stream lines a few operations, aiming to be more intuitive for the user. For example, drag and drop pieces would automatically submit moves. And when anticipating opponent moves ie, it's the opponents turn - we poll on a frequent basis the operation allowing the client to check for a next move.

II. EVALUATION

Some key design decisions worth discssuing.

A. keeping connections alive

You could use a series of mature SLO's[7] to easily highlight that maintaining adequate performance is not a tough feat to accomplish for accommodating a casual game of chess. Performance is the mantra behind keeping connections alive - as when done correctly cuts down relevant round trip "handshake" requests between a client and server reducing the overall time to send and receive requests - an optimisation that lives on the extreme. The cost intuitively being more strain on the server, the resources required to maintain many concurrent live but idle connections, a notable consideration when we're dealing with real world scenarios, where the frequency of interaction between any client and the server is relatively low, but the breadth of connected clients is sizeable. It's natural to then prioritise the servers ability to handle many multiple requests over serving each request quickly. A simple send, receive, close handling of requests would have been more than sufficient in my eyes.

B. use of low level abstractions

I re-invented quite a chunk of the 'HTTP/1.1' specification over TCP. In practice, this is a fruitless endeavour as many abstractions over these mechanics have evolved over the decades. I spent about 70% of my time re-implementing basic communication between client and server for an overall implementation that is naive, in grande contrast among mature abstractions out there already. I believe it would have been perhaps better to focus on the robustness of the system.

C. stream lining of UX

I believe the choices made to control user behaviour, disabling certain functionalities, introduced polling logic etc... was indeed a very good choice. Not only does it improve overall system robustness, it genuinely improves the UX of the game. The game itself still feels slightly wonky, and I wish some more effort was spent in the direction of a more snappier and guided UX. I also felt the UX validation complements well the already strong request validation in the server, and so overall I'm very confident in the behaviours of the game presented.

APPENDIX

A. running the application

Unzip the directory, open the 'index.html' file via a browser. Spin up the server you can do this by running 'dotnet run –project server', tested on 'dotnet –version' '¿=6.0.202'. The server listens on 'localhost://8080'.

B. playing the game

Client must 'register', server will issue a username. Then the client must 'pair' to match up with someone. If the game is in "waiting" state, we require the client to periodically 'pair me' again. Once the game is in "progress" state, you will know who's turn it is, and what peices you are playing. There is no move validation here. When it is your turn, to make a move simply drag and drop a chess piece. If it's not your turn the client periodically polls for the next move. At any time you can 'quit', which exits the client registered user from the game.

REFERENCES

[1] Microsoft, "Socket class reference, implements the berkeley sockets interface." 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socket?view=net-6.0

[2] ——, "The set of different configurable options for instances of system.net.socket," 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.net.sockets.socketoptionname?view=net-6.0

[3] W3, "The hypertext transfer protocol – http/1.1," 1999. [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616.html

[4] G. Team, "The standard library net/http package," 2022. [Online]. Available: https://pkg.go.dev/net/http

[5] Microsoft, "Http status code reference, provides the list of status codes according to http/1.1," 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.net.httpstatuscode?view=net-6.0

[6] M. Fowler, "Software components are things that are independently replaceable and upgradeable," 2015. [Online]. Available: https://martinfowler.com/bliki/SoftwareComponent.html

[7] Atlassian, "Sla vs. slo vs. sli: What's the difference?" 2022. [Online]. Available: https://www.atlassian.com/incident-management/kpis/sla-vs-slo-vs-sli