Exploring the design space of self-made back-end services

Eduardo Rodriguez Fernandez
Chair for Data Processing, Technical University of Munich
eduardo.rodriguez@tum.de

***Abstract*—Most of the popular modern web-development frameworks like Node.js or Go, handle the creation and management of a running back-end service, in a mostly abstracted high-level way that does not allow a developer much freedom to modify the inherent system architecture of the server. Such an inflexible and abstracted, often plug-and-play, server implementation helps to facilitate web development by concealing the system-level design choices from the end user. The problem of blindly relying on a web framework without understanding its internal architecture is that it might not be the most suitable choice for a particular web application, which then ends up having unnecessarily bloated and difficult to maintain dependency-prone services. The aim of this paper is to explore the design space of a dependency-free, cloud-deployable back-end service purely written in C, through a literature review and an actual software implementation. In order to compare the advantages and challenges, in terms of both performance and software development ease, of a highly abstracted back-end framework and a self-made low-level service.**

***Index Terms*—back-end services, concurrency, C, databases, software architecture, systems programming, software engineering, design principles**

I. Introduction

When using any of the most popular modern web development frameworks the inherent system architecture supporting multiple concurrent client connections is often concealed from the developer. Moreover, the system-level design choices to handle concurrent connections tend to be immutable, so that, for instance, a framework based on a single-threaded event loop architecture, like Node.js, cannot be modified or configured to work in a multi-threaded or multi-procedural way. It could be argued that modern web development ecosystems have entirely renounced to providing the user with the full spectrum of system-level primitives that can enable concurrency, in favour of abstracting the complexity of concurrent systems away from the framework's APIs and making portability invisible to the developer.

Another aspect that characterizes some of these frameworks, is that the developer ends up having many different library or packet dependencies from a diverse range of sources, which are sometimes fundamental to enable basic functionality or enhance the capabilities of the framework. With an increasing number of dependencies numerous issues can arise, e.g. mutual incompatibilities between different package versions, a cumbersome management of patches for vulnerabilities and difficulties recreating the same behaviour of an application between the development and the production environments.

The goal of this work is to develop a stand-alone, almost dependency-free, command-line interface chat service independent of current back-end frameworks. In order to explore the challenges and advantages of different system-level networking architectures. There is an unexplored technological gap, because current web frameworks do not offer the ability to implement back-end services in a multi-procedural and dependency-free way.

The chat service back-end will be entirely developed using C, due to the fact that this language provides all possible syscalls capable of directly interacting with kernel concurrency primitives in Unix systems. Moreover, the secondary goal of requiring as little dependencies as possible for this application fits well with a development environment consisting of only gcc as a compiler and the C standard library, which are ubiquitous in Unix systems nowadays.

From a more philosophical point of view, the choice to develop a messaging application capable of being self-hosted by the user was a very deliberate decision. Although, this is not the main goal of this work, it is motivated by the fact that there are no good mainstream alternatives for messaging services. WhatsApp fails miserably as a suitable option since it coerces users to remain on its platform to indiscriminately harvest metadata as a means to increase ad revenue [1]. Signal seems to be a viable alternative at first glance. But it is actually as vulnerable as WhatsApp to fail catastrophically regarding its availability, since its back-end is a centralized and closed platform [2]. Also, at least until 2016 it allowed some user metadata to traverse through Google cloud services [3] and has already handed user metadata to law enforcement authorities in the past [4]. To some extent, more than a coding exercise, the fully functional chat app that came out of this work (which can be found in this GitHub repository [5]) is a way of regaining control over the most sensitive data and metadata produced from our daily communication needs, and shielding it against commercialization.

II. State of the art

C is a good choice of language in order to experiment with system-level concurrent programming primitives, since all of the available primitives in any Unix kernel can be used inside a program developed in C and its portability should be possible. Therefore, it makes sense to take a look at how a reference server implementation in Go handles concurrent connections, for many reasons: Go is a very popular modern language

used widely to provide back-end services, it was designed to provide a simple to implement concurrency model, and it was developed by some of the original pioneers who created C at Bell Labs, it is no surprise that it is described as a ”C-like language” or as ”C for the 21st century” [6]. ——————————

Describe how the systems architecture of other popular web frameworks is, for instance Go and its goroutines.

Describe the concurrency model of Golang, and its goroutines. Explicitly mentioning that most developers have no knowledge or even control, over how this is done [7]. Make clear that since we are developing the chat app in C, it makes sense to compare it with Go, since Go is C for the 21st Century/ it was developed by the Bell Labs architects of C (maybe find a good reference from the Go philosophy statement from its creators that talks about this). And Go also being a concurrency-first language with a very simple syntax for handling concurrency.

If, another main discussion in the methodology would be the async own database implementation, it would be interesting to talk about locks in any db system, and use the data book as a major reference.

Might as well read and describe how nginx works ?????

III. Problem statement

objectives, goals, unsolved issues

IV. Methodology

Describe that a chat app is being used as an example implementation. Argue why the architectural design of the chat app is more IO-bound that CPU-bound [8] and therefore works best with a traditional pre-emptive scheduler.

”If you have a program that is focused on IO-Bound work, then context switches are going to be an advantage. Once a Thread moves into a Waiting state, another Thread in a Runnable state is there to take its place. This allows the core to always be doing work. This is one of the most important aspects of scheduling. Dont allow a core to go idle if there is work (Threads in a Runnable state) to be done.

If your program is focused on CPU-Bound work, then context switches are going to be a performance nightmare. Since the Thead always has work to do, the context switch is stopping that work from progressing. This situation is in stark contrast with what happens with an IO-Bound workload”[8] (correct this since it is actually from part 1)

Mention that there were also portability issues, and that a different behaviour between the development environment and the production or deployment environment was seen. A system call was been compiled differently (the default flags were being used, which where different between systems) so working entirely with gcc and the the standard C library is not a guaranty for automatic perfect portability. In fact, it was very cumbersome to debug the faulty behaviour, since it had to be done using strace, in order to find the misbehaving syscall. Conclusion, thinking that using only C, gcc and the stadlib is working almost dependency-free is a fallacy or an illusion, debugging unexplained behaviour will still be arduous.

V. Conclusion

References

[1] R. Kumar, “WhatsApp and the domestication of users,” https://seirdy.one/2021/01/27/whatsapp-and-the-domestication-of-users.html, January 2021.

[2] M. Hodgson, “On privacy versus freedom,” https://matrix.org/blog/2020/01/02/on-privacy-versus-freedom, January 2020.

[3] J. Edge, “The perils of federated protocols,” https://lwn.net/Articles/687294/, May 2016.

[4] B. M. Kaufman, “New documents reveal government effort to impose secrecy on encryption company,” https://www.aclu.org/blog/national-security/secrecy/new-documents-reveal-government-effort-impose-secrecy-encryption, October 2016.

[5] E. Rodriguez Fernandez, “papayaChat: a self-hosted CLI chat service for the cloud written in C,” https://github.com/erodrigufer/papayaChat, 2022.

[6] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Addison-Wesley, October 2015.

[7] K. Cox-Buday, *Concurrency in Go*. Sebastopol, California: O’Reilly, August 2017.

[8] W. Kennedy, “Scheduling in Go,” https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html, August 2018.

[9] M. Kerrisk, *The Linux Programming Interface*. San Francisco: No Starch Press, 2010.

[10] M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, California: O’Reilly, March 2017.

[11] R. C. Seacord, *Effective C: an introduction to professional C programming*. San Francisco: No Starch Press, 2020.