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, Flask 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 of modification of 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 weigh 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 avoid

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, management of 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. From a philosophy of technology point of view, the choice to develop a chat app 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 produce ad revenue [8]. 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, that at least until 2016 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].

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 [1] 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"[1] (correct this since it is actually from part 1)

II. State of the art

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 [2]. 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. Conclusion

References

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

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

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

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

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

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

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