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, Go, 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

As mentioned previously, choosing C as the sole development language is a deliberate design decision, since it provides all possible concurrency primitives natively in Unix systems (threads, message queues, processes, file locks, etc.), while only requiring the standard library. This fact provides as much freedom as possible to experiment and explore diverse concurrent back-end architectures, while also making the code easily portable to all Unix systems.

Arguably, Go is very well suited to be used as a state of the art comparison to this paper's proposed implementation in C. First of all, Go was created at Google in 2010 by some of the computer science pioneers that originally came up with Unix and C at Bell Labs, so it is no surprise that Go has been described as a "C-like language" or as "C for the 21st Century" [6]. Furthermore, it was created with "built-in concurrency" to tackle modern large distributed infrastructure problems and it is currently widely used at all network traffic levels as a server side service provider [7]. Therefore, it is a great candidate as a point of reference of how modern server side network concurrency can be handled, from which a totally different architecture based on blocking processes can be developed.

If the main goal of this paper is to open a developer's eyes to the many different concurrency paradigms that can be used for server side development, then the philosophy of Go (and for that matter, also of other popular frameworks like Node.js) is the antithesis of this work, because these frameworks provide an inflexible architecture that handles concurrency. In the case of Go, the syntax to handle the creation of concurrent workloads (so-called "*goroutines*") and of communication channels between the goroutines is so simple that an unaware or beginner programmer might be completely oblivious of the scheduling work being performed under the hood by the Go runtime, or even of the fact that its code is running concurrently.

A. Goroutines

The idiomatic way of dealing with client connections in Go, either in an HTTP server or through solely raw TCP communication, is by spawning a new goroutine that handles each client concurrently [8][9][10]. From a software engineering perspective this is a very practical approach, since it elevates the level of abstraction that the programmer has to deal with, so that it is unnecessary to directly intervene in memory synchronization and the management of a thread pool. This should have as a consequence gains in developer productivity, with the trade-off that there is less design freedom. The pledge of Go is that the runtime will single-handedly manage the scheduling of goroutines in the most effective way possible and that goroutines are so lightweight that the developer should not worry upfront about the amount of goroutines that would simultaneously be spawned [12].

Goroutines are very lightweight concurrent subroutines supervised by the Go *runtime* in userspace. Their memory footprint is very small, the assigned memory by default is only a few kilobytes at their creation [12]. From the perspective of the kernel goroutines are non-preemptive, i.e. they are not interrupted by the OS scheduler to run other goroutines. They have defined *points of entry* where they can be suspended or activated by the runtime scheduler, which is entirely running in userspace. Since a context-switch between goroutines happens in userspace and the runtime decides which data should be persistent between goroutines, it is orders of magnitude faster than context-switching between OS threads [12] or between OS processes [13]. A context-switch between OS threads or processes is a costly operation in terms of both the kernel-side data structures needed to maintain all threads and processes, and the operations performed in kernel space to make the transition happen.

B. Runtime scheduler

Each Go executable is compiled with its own statically linked runtime environment in charge of scheduling the goroutines, garbage collection and other tasks. The system model that describes the runtime scheduler consists of three main elements: all statically and dynamically called goroutines, a context and the OS threads where the goroutines are run. Goroutines are placed by the runtime in either the local queue of a context or in the global queue pending to be run by the context in one of the OS threads. The contexts are in charge of managing the scheduling of the goroutine queues.

Parallelism in the system is achieved by having multiple contexts, each using a different core of the processor through different OS threads, in order to run the goroutines waiting in their queues. The runtime manages a set of working threads coupled with contexts and another set of idle threads. If a goroutine performs a syscall that would block, e.g. listens for clients on a TCP socket, the overlying OS thread in which the context is executing the goroutine would also have to block. In this scenario, the blocking thread is decoupled from the context, so that the context can re-activate one of the idle threads and keep working with other non-blocking goroutines.

As long as the goroutines running in the contexts do not call a blocking system call, the different goroutines in the queues can be freely interchanged at the given *points of entry* by the scheduler within the same set of OS threads. This, as previously stated, avoids a costly context-switch in kernel space.

Nonetheless, blocking syscalls for networking are handled in a special way by the runtime. As previously stated, Go idiomatically creates a new goroutine for each client connected to a server. If the server were to have thousands of simultaneously connected clients and most of the clients were to call blocking system calls at the same time, it would then have to create a unique blocked OS thread for each client. This state would be very costly because every blocked client goroutine translates to one blocked OS thread, consequently defeating Go's goal of keeping context-switches primarily in userspace.

Therefore, Go handles network connections in a way that avoids using too many system resources. First, when a new connection is accepted by the server, its file descriptor is set in non-blocking mode, which means that if I/O is not possible in the network socket, the syscall returns an error, instead of automatically blocking. Hence, when a goroutine tries to perform I/O in a network socket and it returns an error, the goroutine notifies a special perpetually running thread called the "*netpoller*", which polls the status of network sockets[8]. The goroutine which could not perform its network operation is placed back on a queue (making its OS thread once again free to run another goroutine). The netpoller notifies a context when it is again possible to perform I/O in the file descriptor, so that the goroutine can be scheduled back in the future. Thereupon, the runtime environment avoids overloading the kernel with unnecessarily too many blocked threads for the client connections.

III. Problem statement

After having seen how modern web frameworks provide their users with an immutable low level architecture to handle concurrent client connections, exemplified by Go's the standard library implementation of an HTTP server.

objectives, goals, unsolved issues

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.

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 [11] 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"[11] (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 stdlib 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] R. Pike, "Go at Google: Language design in the service of Software Engineering," https://talks.golang.org/2012/splash.article, 2012.

[8] D. Morsing, "The Go netpoller," https://morsmachine.dk/netpoller, September 2013.

[9] "Go's standard library's net package. Listener type," https://pkg.go.dev/net#Listener, go1.17.8.

[10] "Go's standard library's net/http package. Serve() function," https://pkg.go.dev/net/http#Serve, go1.17.8.

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

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

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

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

[15] D. Morsing, "The Go scheduler," https://morsmachine.dk/go-scheduler, June 2013.

[16] C. Siebenmann, "The Go runtime scheduler's clever way of dealing with system calls," https://utcc.utoronto.ca/~cks/space/blog/programming/GoSchedulerAndSyscalls, December 2019.

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