Exploring the low level design space of 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.

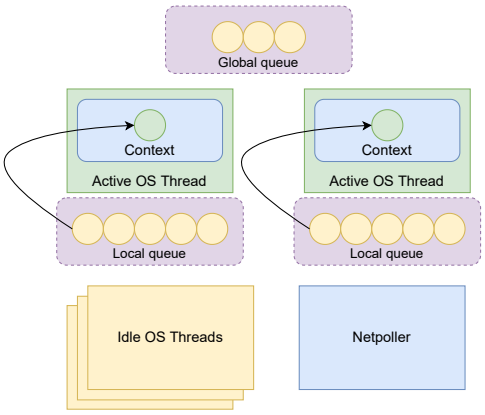

Fig. 1. High level depiction of the runtime environment in Go. *Goroutines* are depicted as circles, OS threads as rectangles. Active goroutines running on a *context* are green, idle goroutines waiting in a queue are yellow. The same color semantics apply to active and idle OS threads.

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 [11].

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 [11]. 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 context-switches, it is orders of magnitude faster than context-switching between OS threads [11] or between OS processes [12]. 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, the operations performed in kernel space to make the transition happen and, possibly, the shifting of memory blocks during the transition.

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, as illustrated in figure 1. The contexts are in charge of managing the scheduling of the goroutine queues.

Parallelism in the system is achieved by having multiple contexts (in fig. 1 only two contexts are simultaneously running, depicted as blue rectangles), 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 (illustrated as green rectangles) coupled with contexts and another set of idle threads (yellow rectangles). 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 netpoller thread is depicted as a blue rectangle in fig. 1). 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 standard library implementation of an HTTP(s) server, it will be discussed in this paper what other Unix system primitives could potentially be used to handle concurrency on a server.

The topic is first handled with a literature review of the advantages and disadvantages of different architectures. Afterwards, specific challenges regarding the implementation of the chat application will be covered, namely: how to guarantee a secure continuous operation of the server daemon, after opening a public port, what has to be considered when implementing a dependency-free database in a concurrent distributed system and is the application more easily portable when only having the standard C library as a dependency and compiling only with gcc?

IV. Implementation

The requirements for the chat application are that an undefined number of participants can simultaneously exchange text messages in a chatroom. Furthermore, the communication might be asynchronous, so that the participants can read messages sent to them while they were not connected to the server. The application should use the least amount of dependencies as possible to enable portability across Unix systems, i.e. the chat server should compile natively to FreeBSD or a Linux distribution with the same source file.

The server fundamentally requires a process working as a daemon accepting incoming connection attempts from clients.

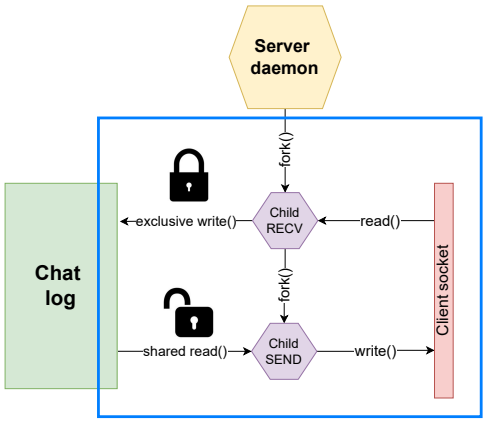

Fig. 2. Server's back-end architectural overview. The blue rectangle denotes the process cluster created exclusively for each client taking part in a chat service.

For each accepted client connection there are multiple possibilities regarding the architecture of the server. The daemon could handle each client separately in a unique thread or child process.

The server will mostly have an IO-bound workload, consisting of handling asynchronous network packets and writing the messages from the users into files in the server's filesystem. An IO-bound workload benefits from the use of a pre-emptive scheduler, since the threads or processes are constantly changing alternatively between a blocked and an unblocked state in an unpredictable manner. As soon as a client goes silent the pre-emptive scheduler can run any other runnable process [13]. Hence, the context-switching is actually advantageous for IO-bound workloads, whereas in CPU-bound workloads (e.g. intensive long-running sequential computations) it becomes a performance bottleneck.

Therefore, handling each client connection separately by forking a child process seems like a good fit for the kind of workload that is expected. Nonetheless, it must be acknowledged that a counterargument against using processes is that thread creation and context-switching times are generally faster than for processes, since processes have an inherently more complex memory layout than threads [12].

However, other reasons settled the decision towards processes instead of threads. Before going into these reasons, it makes sense to review the architecture that was actually implemented as a solution. Figure 2 shows a high-level representation of how the server handles every client connection. After successfully authenticating a client, the server daemon calls the *fork* syscall and creates a new child process exclusively for the new client.

This child process, called "*Child RECV*" in fig. 2, inherits a copy of the newly established socket which handles the client. Child RECV is responsible for reading any incoming messages from the client, writing these messages in a concurrently-safe way into a central chat log, sending a multicast signal to let all other clients know that there is a new message and creating a further child process called "*Child SEND*". *Child SEND* also inherits the client's socket in order to send the messages stored in the central chat log at an appropriate time to the

client. The blue rectangle depicted in figure 2 comprises a single process cluster for a particular client. For every client connected simultaneously to the server there is one of this process clusters running concurrently.

One reason to choose processes over threads is that compartmentalizing the different clients into separate processes has the intrinsic advantage of granting more availability in case of a distributed denial of service (DDoS) attack or simply heavy traffic in the server's public-facing daemon. If for any reason, the daemon is getting cluttered with connection attempts, to a point in which the high load threatens to affect the communication performance of the clients already participating inside a chat service, then the daemon's process can be temporarily stopped, or even killed, so that the public-facing port is closed. Since each client's process cluster works completely independently from the daemon accepting new clients and from all other clients' process clusters, the service can continue uninterrupted for all clients already connected.

From a system administration perspective it is also very convenient to handle each client connection with different processes, since it makes the monitoring and administration of system resource utilization in a *per-client* granular way easy through the use of tools like *kill*, *ps* and *top*.

A. Transactional isolation

Although the data model of the chat application would fit well within a relational database, since the types of the data fields for every message exchange are immutable and translatable into the data types used in relational databases, the system intentionally avoids using any kind of external database system. This design decision makes the application more easily portable and deployable, due to the fact that the same executable of the chat server entirely handles the message storing and retrieving for all data exchanges.

B. Portability issues

Even when using only POSIX compliant syscalls the server does not compile entirely the same in all platforms. Mention problem between Debian (Ubuntu and Kali) distributions, in which strace had to be used to pinpoint a compilation difference.

C. Security

We are not containerizing the application, since it would not do it portable across many Unix systems, e.g. the BSDs, so we have to guarantee the security of running an application with an Internet facing open port. -¿ Authentication system and running the server through a system user with very limited permissions. The system user technique limits the amount of damage that could possibly be generated, because the system user should not have that many permissions.

---

* Refer to how it is not possible to handle signals with threads, and therefore the decision went towards using processes.

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

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

[13] W. Kennedy, "Scheduling in Go: OS scheduler," https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part1.html, August 2018.

[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.