An experimental comparison of the performance between different web concurrency paradigms

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 and Go, handle the creation and management of a running backend 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. Modern web frameworks mostly try to handle concurrent client connections in user-space under the premise that handling concurrency in kernel-space is too costly. The problem of blindly relying on a web framework without understanding its internal architecture is that it might not be the most efficient choice for a web application that has to deal with multiple concurrent connections. The aim of this paper is to provide an experimental comparison in the CPU utilization efficiency of two completely different concurrency-handling paradigms: a multi-process implementation in C and a *goroutine*-based non-preemptively scheduled web service in Go. In order to see if there is a performance penalty for handling concurrency in web applications primarily in kernel-space, rather than in user-space, as most modern web frameworks tend to do nowadays.**

***Index Terms*—concurrency, backend development, Go, C, software architecture, systems programming, instant messaging, software engineering**

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 an event-driven 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 providing the user with the full spectrum of system-level primitives that can enable concurrency, in favor 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, supply chain attacks, a cumbersome management of patches for vulnerabilities and difficulties recreating the same behavior of an application between the development and production environments [1].

In the early days of web development, some of the literature comparing different concurrency paradigms favored threads over an event-driven architecture, primarily due to the better readability and maintainability that threads allegedly provide [2][3]. While at the same time acknowledging without experimental evidence that threads can have a higher CPU usage overhead due to context-switching [2]. Nonetheless, both [2] and [3] expected that future improvements in compiler integration of threads and the development of frameworks that make use of '*cooperative threading*' with user-space context-switching and small dynamic stack sizes would improve the performance of multi-threaded systems.

Eventually, multi-core platforms became ubiquitous and many languages like Go, Erlang and Elixir popularized cooperative threading as a way of getting more performance in concurrent web applications through, among other methods, parallelization and context-switching in user-space. [4] presents an actual implementation of primitives for the LLVM compiler which support context-switching for coroutines and lightweight threads in a language-agnostic way. [4] experimentally evaluates the performance of different cooperative threading implementations with various context-switching benchmarks, the compiler-integrated architecture proposed in the paper very frequently outperforms other more popular alternatives as Go, Haskell, Erlang and POSIX threads.

Many sources claim that kernel-space context-switching like with processes and threads, should most of the times represent a very noticeable performance penalty in comparison to other concurrency approaches centered around user-space context-switching, like cooperative threads i.e. non-preemptively scheduled threads, without providing any experimental confirmation for those claims [2][3][5][6]. Meanwhile, other sources that present experimental data, do not work with benchmarks directly related to concurrency in the context of web applications [4].

The motivation of this paper is two-fold, on the one hand it primarily strives to gather experimental data to test the assumption that multi-process concurrent web applications, due to their mostly kernel-centered context-switching, should be outperformed in terms of CPU usage by coroutine-based context-switching taking place in user-space. For that matter, the CPU usage of the same concurrent web application will be compared between an implementation in C and an imple-

mentation in Go.

The secondary goal of this work is to develop a stand-alone, almost dependency-free, command-line interface chat service independent of current backend frameworks. In order to put into practice the concurrency handling primitives tested and to explore the challenges and advantages of different system-level networking architectures. Current web frameworks do not offer the ability to implement backend services in a multi-procedural and dependency-free way.

The chat service backend will be entirely developed using C, due to the fact that this language provides all possible system calls (syscalls) capable of directly interacting with kernel concurrency primitives in Unix systems. Moreover, the requirement of using the least amount of 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 both ubiquitous on modern Unix systems.

From a more philosophical point of view, the choice to develop an instant messaging (IM) application capable of being self-hosted by the user is 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 mean to increase ad revenue [7]. 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 backend is a centralized and closed platform [8]. Also, at least until 2016, it allowed some user metadata to traverse through Google cloud services [9] and has already handed user metadata to law enforcement authorities in the past [10]. Furthermore, all major social media platforms offer some kind of IM capabilities, but at the cost of allowing that the users' data be thoroughly harvested for some kind of value generation [11]. 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 [12]) is a way of regaining control over the most sensitive data and metadata produced from our daily communication needs, while simultaneously shielding it against commercialization.

II. State of the art

Arguably, Go is very well suited to be used as a state of the art reference of a web-centered language which handles concurrency through user-space context-switching, especially when compared to a multi-procedural web application implemented in C. 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" [13]. 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 [14][15][16]. Therefore, it is a great candidate as a point of reference of how modern server-side network concurrency can be handled [17].

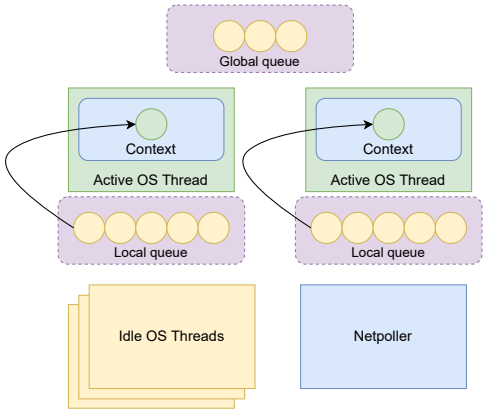

Fig. 1. High level depiction of the runtime environment in Go. *Goroutines* are depicted as circles, OS threads as rectangles. Active goroutines running on a *context* are green, idle goroutines waiting in a queue are yellow. The same color semantics apply to active and idle OS threads.

If one of the aims 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 immutable architecture to handle concurrency. In the case of Go, the syntax to handle the creation of concurrent workloads (so-called "*goroutines*") and the language constructs for communication channels between the goroutines are 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 [16].

A. Goroutines

The idiomatic way of dealing with client connections in Go, either in an HTTP(s) server or through solely raw TCP communication, is by spawning a new goroutine that handles each client concurrently [18][19][20]. 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 [4][5].

More concretely, goroutines are very lightweight concurrent subroutines supervised by the Go *runtime* in user-space. Their memory footprint is very small [4], the assigned stack memory by default is only a few kilobytes at their creation [5]. 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 user-space. Since a context-switch between

goroutines happens in user-space and the runtime decides which data should be persistent between context-switches, it should be orders of magnitude faster than context-switching between OS threads [5] or between OS processes [6]. 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 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 a context in one of the OS threads, as illustrated in figure 1. The contexts are in charge of managing the scheduling of the queues and the data required by the different goroutines.

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. 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 by the scheduler at the given *points of entry* 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 would translate to one blocked OS thread, consequently defeating Go's goal of keeping context-switches primarily in user-space.

Therefore, Go handles network connections in a way that avoids using too many system resources. When a goroutine tries to read from a client socket which would block, a special perpetually running thread called the "*netpoller*" is notified of this fact [18] (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 continually polls all network sockets and notifies a context when an operation on the socket would not block, 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.

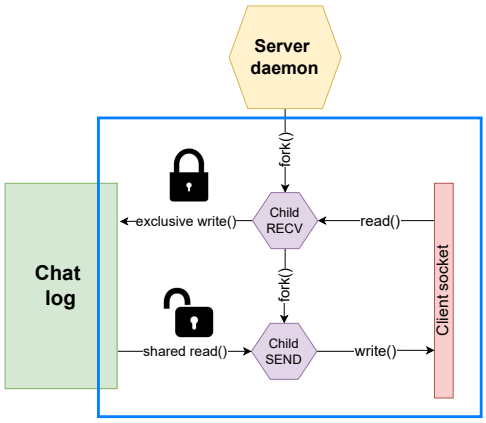

Fig. 2. Server's back-end architectural overview. The blue rectangle denotes the process cluster created exclusively for each client taking part in a chat service.

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 comparison 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?

How do two different concurrency approaches like processes and goroutines compare performance-wise (CPU utilization) with a same implementation?

Is there a difference in the coefficient of variation of both concurrency models? Does one of them has a more stable CPU utilization than the other one?

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 with the same source file in FreeBSD or in a Linux distribution.

The server fundamentally requires a process working as a daemon accepting incoming connection attempts from clients. 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 [21]. 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 in threads are generally faster than for processes, since processes have an inherently more complex memory layout than threads [6].

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.

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 command line tools like *kill*, *ps*, *ss* or *sockstat*, and *top*.

A. Transactional isolation and atomicity

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. Deploying the chat server into a cloud server is as easy as pulling the repository, compiling and running the binary, there is no need to first install and configure a (No)SQL server.

Nonetheless, this implies that the chat program has to fulfil some data safety guarantees that would otherwise be outsourced entirely to the database management system (DBMS). Mainstream DBMS have the ability to perform a series of reads and writes to the underlying data system as a single "*logical unit*" [22], a so-called "*transaction*". A transaction is useful as a way of ensuring atomicity and isolation within a distributed system.

Since the incoming messages from the clients will all be centrally stored in a log file and messages from different clients can arrive at any time simultaneously to the server, a transactional mechanism is implemented to avoid race conditions.

Therefore, the data management system must fulfil the following four requirements. Multiple clients simultaneously writing to the log file should not over-write their messages. Furthermore, it should not be possible to read the log file during a write-operation from another client to avoid reading incomplete data, and, conversely, it should not be possible to write to the log, while another client is reading from it. Finally, unlimited concurrent reading operations from multiple clients are permitted on the log file, since reading from the log file does not have any side effects on the stored data.

To satisfy these criteria, the server creates and opens the central log file with the "*O_APPEND*" flag (append mode), so that before each write to the file the offset is positioned at the end of the file and the write operation is subsequently performed in a single atomic operation [6]. Thus, old data written to the log cannot get corrupted by new writes to the file.

Moreover, a file locking system is implemented using the *flock* syscall, in order to fulfil the previously mentioned four

requirements. Two different types of locks can be placed on a file: a shared lock and an exclusive lock. When multiple clients try to simultaneously send messages to the chat log, the child process Child RECV tries to place an exclusive lock on the chat log file. If no other lock is currently placed on the file, Child RECV can write exclusively into the chat log until the placed lock is released. Meanwhile, no other process can write or read from the file, the other Child RECV processes trying to write to the chat log file would block on the call to *flock*, until the process holding the exclusive lock releases it. All the necessary writes to the chat log file would be scheduled sequentially.

Conversely, when the server sends new messages to a client through Child SEND, it has to read the messages from the chat log. Hence, the child process places a shared lock on the chat log and reads from it. Any other process trying to simultaneously read from the same file can place another shared lock and read from the file, but a process trying to write to the file would block when placing the exclusive lock, until all shared locks have been released.

This file handling architecture makes sending messages to multiple clients a highly parallelizable operation, while writing to the chat log is a secure isolated task performed in an atomic way.

B. Multicasting with Unix IPC

The system should deliver new messages instantly through the respective Child SEND process of every client each time a new message is received, in other words, every message must be multicasted to all participating clients. The number of connected clients can change over time and it is unlimited.

The two main IPC (inter-process communication) mechanisms capable of multicasting to multiple processes are sockets and POSIX signals [6]. Signals are used in this implementation, because from a software engineering perspective configuring sockets to handle multicasting is less portable and requires more code-refactoring.

The Unix standard *SIGUSR1* signal, which is reserved for ”user defined behaviour”, can be used to synchronously inform a group of processes about an event, in this case, the need to send new messages to the clients.

By choosing to multicast with signals the possibility to work with threads, instead of with processes, is eliminated. Since all threads in a process share the same signal dispositions and for this design a different signal disposition between parent and child processes is a requirement. Moreover, this architecture does not require a centralized administrative process constantly monitoring which clients are currently online, to whom the messages should be delivered. Instead, it outsources this task to the kernel which manages signal distribution.

C. Portability issues

Even though, only portable Unix syscalls are used and the number of dependencies is reduced to the utmost minimum of GCC and the C standard library, some portability issues arise when deploying in a multi-platform fashion.

The backend was compiled with GCC and tested in two different Debian-based distros: Ubuntu and Kali Linux, the latter was a 32-bit system, and in FreeBSD 13.0. A single compilation difference between the two Debian platforms rendered the backend service completely useless in one instance. The root cause of the faulty behaviour was then established using a syscall and signals monitoring tool like *strace* in a very cumbersome process. The bug was caused by a difference in the default flags assigned by the compiler to the syscall assigning a signal disposition.

The same code that worked flawlessly in Ubuntu listened for clients in FreeBSD using solely IPv6, which changed the behaviour of the application massively and required code refactoring to enforce IPv4 in a cross-platform fashion.

Thus, it is illusory to think that restricting the dependencies to the bare minimum of the C standard library and GCC will make the code perfectly compatible across GNU/Linux systems. Debugging unexplained behaviour will still be arduous.

D. Security

The chat server runs as a long-lasting daemon, therefore leaving at least one port open to the public internet from which the clients will establish a connection with the server. It must be taken for granted that the open port will eventually be discovered by web-scanning botnets that periodically scan targeted hitlists (specially, known IP ranges from cloud providers) or random IP ranges [23]. Accordingly, it is necessary to secure the application to not compromise the infrastructure.

The backend service is not run inside a container, since this would reduce the portability across Unix systems, e.g. the BSDs do not have kernel features such as cgroups and Linux namespaces, which make natively running typical container management systems like Docker impossible.

An alternative way of sandboxing the application without containers is running the service as a specially created system user account with the minimum amount possible of privileges. Thereupon, even if the service is compromised the amount of possible damage is limited to the reduced capabilities of the system user. Furthermore, a long authentication token is required to access any given chat session.

V. Experiments

A. Test environment

Add specifics of measurements: FreeBSD version, vCPU and memory stats, GCC version, (maybe even stdlib version?). Version of tcpkali and of golang compiler (v 1.18.2).

Measure mean latency between instances in the FRA region?.

VI. Conclusion

Creating a backend service from the ground up gives the developer the freedom of choosing between a thread-oriented, process-oriented, pre-emptively or non-pre-emptively scheduled architecture. The chat service developed in this work consists of a mainly IO-bound workload, so that it

benefits from a pre-emptively scheduled architecture which switches processes handling clients as soon as they block [21]. Nonetheless, it must be acknowledged that the development productivity, particularly because of the debugging of syscalls with strace, is not high compared to a high-level framework with mature networking libraries.

Furthermore, although the software was exclusively tested on Debian-based platforms, the dependency restriction of only using the C standard library and compiling with GCC did not guarantee an entirely bug-free portability between Unix platforms. A different default initialization of the parameters of some syscalls by the compiler generated difficult to find sources of faulty and unequal behaviour between platforms. Therefore, even when reducing dependencies to a bare minimum, it is an illusion to think that fully portable code can easily be generated, so that to some extent the appeal and reasoning behind OS-virtualization (container management systems) can be better grasped.

On the other hand, since the data model used in this implementation is immutable and uncomplicated, the implementation of an asynchronous data management system with transactional guarantees delivers benefits due to the reduction of system dependencies. A deployment in the cloud or in a local system is very expeditious and does not require the management and configuration of a DBMS.

The software created in this project, distributed through a public repository with a AGPL license (GNU Affero General Public License), delivers a functional command-line chat application that gives the user the possibility to self-host its chat service and regain full control over the management of its instant messaging data and metadata. Furthermore, it allows its users to avoid vendor lock-in effects and the single points of failure of an IM application with a centralized architecture like Signal and WhatsApp, since its minimal amount of dependencies facilitate a prompt native deployment in any Unix system. The permissive open source license of the project allows further collaboration in order to improve the application in regards such as end to end encryption, UX and further tests in more platforms, among others.

REFERENCES

[1] P.-H. Kamp, "A Generation Lost in the Bazaar: Quality happens only when someone is responsible for it." *ACM Queue*, vol. 10, no. 8, p. 2023, August 2012. [Online]. Available: https://doi.org/10.1145/2346916.2349257

[2] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS'03. USA: USENIX Association, 2003, p. 4.

[3] A. Gustafsson, "Threads without the pain: Multithreaded programming need not be so angst-ridden." *ACM Queue*, vol. 3, no. 9, p. 3441, November 2005. [Online]. Available: https://doi.org/10.1145/1105664.1105678

[4] S. Dolan, S. Muralidharan, and D. Gregg, "Compiler support for lightweight context switching," *ACM Transactions in Architecture and Code Optimization*, vol. 9, no. 4, January 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400695

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

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

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

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

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

[10] 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.

[11] D. Choudhery and C. K. Leung, "Social media mining: Prediction of box office revenue," in *Proceedings of the 21st International Database Engineering; Applications Symposium*, ser. IDEAS 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 2029. [Online]. Available: https://doi.org/10.1145/3105831.3105854

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

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

[14] R. Pike, "Go at Google," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 56. [Online]. Available: https://doi.org/10.1145/2384716.2384720

[15] S. Ajmani, "Program your next server in Go," in *Applicative 2016*, ser. Applicative 2016. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2959689.2960078

[16] M. Chabbi and M. K. Ramanathan, "A study of real-world data races in Golang," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 474489. [Online]. Available: https://doi.org/10.1145/3519939.3523720

[17] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson, "The Go programming language and environment," *Communications of the ACM*, vol. 65, no. 5, p. 7078, April 2022. [Online]. Available: https://doi.org/10.1145/3488716

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

[19] "Go's standard library's net package. Listener type," https://pkg.go.dev/net#Listener, go1.18.3.

[20] "Go's standard library's net/http package. Serve() function," https://pkg.go.dev/net/http#Serve, go1.18.3.

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

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

[23] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, p. 3953, Apr. 2004. [Online]. Available: https://doi.org/10.1145/997150.997156

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

[25] 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.

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