An experimental comparison between the performance of 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 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, 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 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*', i.e. user-space context-switching and small dynamic stack sizes, would improve the performance of multi-threaded applications.

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. Furthermore, there have been advances in implementing compiler integration of cooperative threading [4][20]. An actual implementation of primitives for the LLVM compiler which support user-space context-switching for coroutines and lightweight threads in a language-agnostic way is presented by Dolan et al. [4]. The paper experimentally evaluates the performance of different cooperative threading implementations with various context-switching benchmarks. The compiler-integrated architecture proposed in the paper outperforms Go, Haskell, Erlang and POSIX-threads in various metrics.

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][20].

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 applications with 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 implementation 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.

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 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.

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]. 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 [9]. 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 open-source GitHub repository [10]) is a way of regaining control over the most sensitive data and metadata produced by our daily communication needs.

II. Go as a state of the art non-preemptively scheduled language

Go is a state of the art web-centered language which exclusively handles concurrency through user-space context-switching. It 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*" [11].

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 [12][13][14]. Therefore, it is a great candidate as a point of reference of how modern server-side network concurrency can be handled exclusively in user-space [15], particularly when compared to a preemptively scheduled application in its 'close-relative' C.

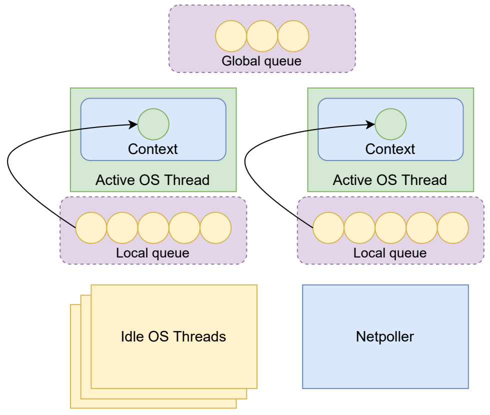

Fig. 1. High level depiction of the runtime environment in Go. *Goroutines* are depicted as circles, OS threads as rectangles. Active goroutines running on a *context* are green, idle goroutines waiting in a queue are yellow. The same color semantics apply to active and idle OS threads.

A. Goroutines

The idiomatic way of dealing with client connections in a Go server, is by spawning a new *goroutine* that handles each client concurrently [16][17]. Goroutines are very lightweight concurrent subroutines supervised by the Go *runtime* exclusively 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].

Goroutines are non-preemptive, i.e. they are not interrupted by the operating system's (OS) scheduler to run other goroutines. They have defined *points of entry* where they can be suspended or activated by Go's 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 can 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 manage 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: goroutines, contexts 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 underlying 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. 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 [16] (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.

After getting to know all the complex concurrency abstractions and runtime system that was developed to allegedly perform well in highly-concurrent network systems, one might ask himself: how does this state of the art concurrency paradigm compare performance-wise with a traditional multi-procedural framework?

III. Problem Statement

As previously stated in the introduction, there is ample literature that affirms that kernel-space context-switching in multi-procedural programs tends to have a worse CPU-utilization performance than user-space context-switching, but none of the bibliographic sources provide experimental data to corroborate this, show how big the difference is or perform a benchmark for network-related concurrency handling [2][3][4][5][6].

This work strives to first design the architecture of a multi-procedural concurrent IM application written in C and, then, create a test that makes a performance comparison between the preemptively scheduled C application and an equivalent non-preemptively scheduled IM application in Go possible, to address the following research questions (RQ):

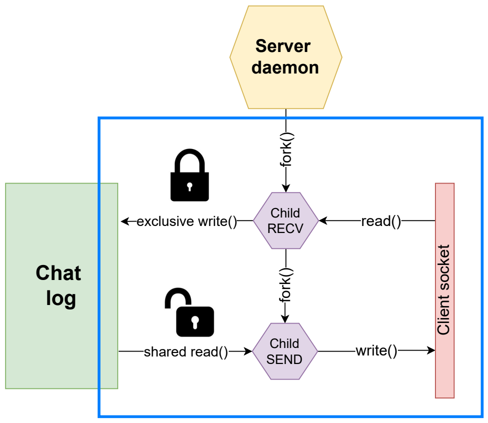

Fig. 2. Server's back-end architectural overview. The blue rectangle denotes the process cluster created exclusively for each client taking part in a chat service.

RQ 1: How does the mean CPU usage of a multi-procedural C application compares to the CPU usage of a similar non-preemptively scheduled program in Go while handling concurrent network connections? RQ1 serves as an experimental comparison of the performance of context-switching in user-space and in kernel-space.

RQ 2: How stable over a period of time is the CPU usage of the two models being compared, when receiving a constant load? This question is relevant to more accurately be able to provision servers for high load scenarios.

RQ 3: How portable across Unix platforms is the IM application developed in C with minimal dependencies? In other words can the program be built and executed without any changes across different Unix platforms?

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 the daemon handles the given client separately in a unique child process.

The server will mostly have a workload without any long-lasting computations and with many context-switches while handling concurrent network packets coming from multiple clients and writing the messages received from the users into files in the server's filesystem. Such a workload benefits from the use of a preemptive scheduler, since the processes are constantly changing alternatively between a blocked and an unblocked state in an unpredictable manner. As soon as a client stops sending network packets to an active socket, the preemptive scheduler can switch from the now blocked socket to execute any other runnable process. Hence, the context-switching is actually advantageous for these kinds of workloads, whereas in computationally heavy programs (e.g. intensive long-running sequential computations) context-switching becomes a performance bottleneck [18].

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].

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 file, 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 file 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.

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 of 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 cloning 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 DBMSs have the ability to perform a series of reads and writes to the underlying data system as a single "*logical unit*", a so-called "*transaction*". A transaction is useful as a way of ensuring atomicity and isolation within a distributed system [19].

Since the incoming messages from the clients will all be centrally stored in a single 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 sending messages to the server should not over-write their messages. Furthermore, it should not be possible for any client to read the file that stores the messages during a write-operation to avoid reading incomplete data, and, conversely, it should not be possible to write to the file, while another client is reading from it. Finally, unlimited concurrent reading operations from multiple clients are permitted on the file, since reading from it does not have any side effects on the stored data.

To satisfy these criteria, the server creates and opens the message-handling 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 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 server, the child process Child RECV tries to place an exclusive lock on the chat file. If no other lock is currently placed on the file, Child RECV can write exclusively into the file, 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 file would block on the call to *flock*, until the process holding the exclusive lock releases it. All the necessary writes to the file would then be sequentially scheduled.

Conversely, when the server sends new messages to a client through Child SEND, it has to read the messages from the filesystem. 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 file is a secure isolated task performed in an atomic way.

V. Experiments

To evaluate the performance of a concurrency paradigm based upon context-switching in user-space and another one based upon context-switching in kernel-space, the same very rudimentary IM application was developed in both C and Go. The C program handles concurrent client network connections by spawning a new process per client, while the Go application creates a goroutine for each client.

A TCP load generator running in a cloud server is used to send a constant load of network packets (1Mbps) per concurrent connection for exactly 60 seconds to another server running the application being tested. Each application accepts a given number of concurrent client connections per test, receives and reads the clients' packets (which simulate text messages from the users in a real IM application) and writes the text in the packets to *stdout*. To not overwhelm the system

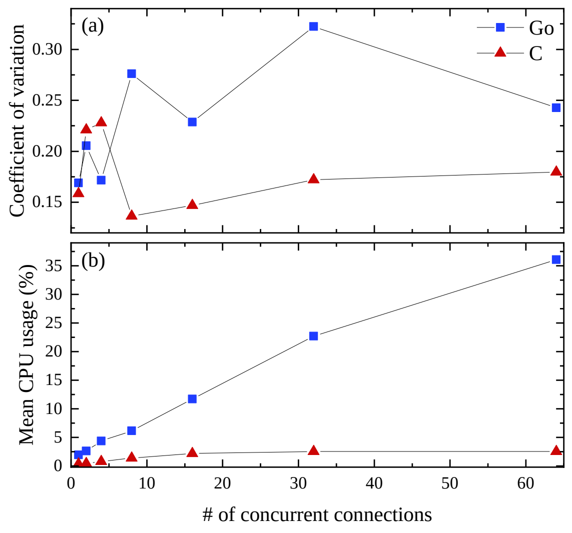

Fig. 3. (a) compares the development of the coefficient of variation C_v depending on the number of concurrent connections being handled by the two different concurrency platforms. (b) compares the mean CPU usage value (%) in relationship to the number of concurrent connections.

with filesystem writes and focus primarily in the performance of the concurrency handling mechanisms in a real-life environment, the data being output to stdout is immediately discarded to */dev/null*.

The CPU usage of the IM application is measured and recorded using *top(1)* with a sampling rate of 500ms. The number of active concurrent clients is incremented in each new test, starting at 1 and going all the way up to 64 concurrent connections.

A. *Test environment*

Both the TCP load generator and the application under test were executed in cloud virtual machines inside the same region running FreeBSD 13.0-RELEASE-p11 with 1vCPU and 2GB RAM. The versions of software used for these experiments were: for the TCP load generator *tcpkali v1.1.1*, for the C compiler *GCC v10.4* and for the Go compiler *Go v1.18.2*.

VI. Discussion

A. *CPU usage and load handling stability*

The results, as depicted in fig. 3(b), show an almost linear increase in the CPU usage mean value in the Go implementation, while the multi-process C implementation considerably decreases the rate of growth of its CPU utilization after 16 concurrent connections. Overall, the C implementation handles the load more efficiently at all levels of concurrent connections. The results might seem counter-intuitive and baffling at first glance, since many literature sources warn against the performance penalties incurred by handling context-switches in kernel-space [5][6].

Nonetheless, in a comparison based on programs similar to real-life deployed network applications, like in these experiments, the CPU usage is influenced not only by the context-switches, but also by inherent traits of the platform running the application. In this case, the actual performance gains attributable to user-space context-switching might be too small compared to the overhead of the Go runtime and the periodic garbage collection (GC) it performs. Both the runtime and GC are required to handle the complexity of the user-space context-switching, among other things.

Moreover, the CPU usage of the non-preemptively scheduled implementation has an overall higher coefficient of variation (C_v), calculated by dividing the standard deviation (σ) of a measurement by its mean value (μ). This value allows us to answer RQ2 and compare the stability of the CPU usage of both platforms, a higher C_v implies that the data points of a measurement are on average spread farther from the mean value as in a measurement with a lower C_v.

Especially at high loads, as can be seen in fig. 3(a), the multi-process implementation achieves a more stable CPU utilization (lower C_v), which does not oscillate so far from the mean value as the Go implementation. This insight in the stability of the CPU usage of a particular paradigm has relevance to servers that periodically work close to the maximum CPU usage during high load events, since an application with a smaller C_v will react more predictably and securely close to the maximum possible CPU usage of a platform by not putting an even bigger burden on the server with CPU load spikes.

B. *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 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 Unix systems. Debugging unexplained behaviour will still be arduous.

VII. Conclusion

Creating a backend service from the ground up gives the developer the freedom of choosing between a thread-oriented, process-oriented, preemptively or non-preemptively scheduled architecture. For some applications that are expected to work at almost the full capacity of the CPU usage of a machine, it makes sense to consider porting their code to a multi-process implementation in C, instead of developing in a platform with a more modern and complex concurrency paradigm, like Go. Since, the results of our measurements show a more stable CPU utilization in the multi-procedural implementation in C, with a much lower coefficient of variation under high loads. Nonetheless, it must be acknowledged that the development productivity while programming low-level network concurrency, as in C, is not as high as compared to a high-level framework with mature networking libraries.

Furthermore, at all levels of concurrent network connections tested, the C implementation with context-switching in kernel-space, instead of in user-space as with the Go application, had a significantly lower percentage of CPU utilization. The non-preemptively scheduled Go application shows an almost linear increase in the CPU utilization proportional to an increase in the number of concurrent connections, while the C program drastically reduces the rate of CPU usage increase after 16 concurrent client connections. In a comparison based on programs similar to real-life deployed network applications, like in this case, the CPU usage is influenced not only by the context-switches, but also by inherent traits of the platform running the application. In this case, the actual performance gains attributable to user-space context-switching dwindle compared to the overhead of the Go runtime and GC.

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. 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.

Finally, the software developed in this project [10], 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.

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. (2021, January) WhatsApp and the domestication of users. Visited on 2022-06-27. [Online]. Available: https://seirdy.one/2021/01/27/whatsapp-and-the-domestication-of-users.html

[8] M. Hodgson. On privacy versus freedom. Visited on 2022-06-27. [Online]. Available: https://matrix.org/blog/2020/01/02/on-privacy-versus-freedom

[9] 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: ACM, 2017, p. 2029. [Online]. Available: https://doi.org/10.1145/3105831.3105854

[10] E. Rodriguez Fernandez. (2022) papayaChat: a self-hosted CLI chat service for the cloud written in C. Visited on 2022-06-27. [Online]. Available: https://github.com/erodrigufer/papayaChat

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

[12] 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: ACM, 2012, p. 56. [Online]. Available: https://doi.org/10.1145/2384716.2384720

[13] S. Ajmani, “Program your next server in Go,” in *Applicative 2016*, ser. Applicative 2016. New York, NY, USA: ACM, 2016. [Online]. Available: https://doi.org/10.1145/2959689.2960078

[14] 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

[15] 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

[16] D. Morsing. (2013, September) The Go netpoller. Visited on 2022-06-27. [Online]. Available: https://morsmachine.dk/netpoller

[17] (go1.18.3) Go’s standard library’s net package. Listener type. Visited on 2022-06-27. [Online]. Available: https://pkg.go.dev/net#Listener

[18] W. Kennedy. (2018, August) Scheduling in Go: OS scheduler. Visited on 2022-06-27. [Online]. Available: https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part1.html

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

[20] P. Jskelinen, P. Kellomki, J. Takala, H. Kultala, and M. Lepist, “Reducing context switch overhead with compiler-assisted threading,” in *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, vol. 2, 2008, pp. 461–466.