Designing a Serverless SDN Controller

Grace Yue and Zachary Hilman

Presented to: Professor Craig Shue

December 16, 2022

Contents

Abstract

With the rise of remote and hybrid work arrangements, the need for secure connections and information transfer has never been higher. Even with company-issued devices, employees can still work from anywhere on any network, leaving themselves and the company vulnerable to cyberattacks. Serverless infrastructure is designed to provide functionality to a globally-distributed user base, which makes it a natural solution to managing a potentially globally-distributed workforce. In this paper, we propose a serverless software-defined networking (SDN) controller in C++ that implements a subset of the OpenFlow 1.0 specifications. We describe a software development kit (SDK) in C++ that will enable a user with minimal coding knowledge to create and customize their own SDN controllers and policies, freeing them from the limitations of a specific hardware configuration and enabling better overall performance.

Chapter 1

Introduction

As companies move to allowing their employees to work from anywhere, the need for secure company communications and intellectual property is greater than ever. Company IT departments lose both control and security assurances when employees can suddenly work from any device on any (potentially vulnerable) network. One solution is for companies to issue pre-configured company laptops to their employees; however, it can be inconvenient for employees to carry multiple devices. Another common solution is the use of virtual private networks (VPNs), which obscure a user's IP address and provide an added layer of security on public WiFi networks. However, accepting VPNs as a security solution gives a company's internal information technology and security departments less control over what an employee can do with company data.

In this paper, we propose a solution to both the security needs of a company's IT department and the convenience needs of a company's remote and hybrid employees: an software development kit (SDK) to write software-defined networking (SDN) controllers for instantaneous serverless deployment.

Software-defined networking is a modern approach to network management: essentially, the centralization of all decision-making in a given network. An SDN model consolidates the network control plane, by relying on a device called the controller to make decisions regarding all packet processing between devices. The SDN controller arbitrates the passage of packets through the network and sends its decisions to the individual network switches, while each switch caches the controller's rules in its own local flow tables (this is done to limit the number of requests that must be forwarded to the controller for decision-making, thereby enhancing network performance).

Thus, each switch is only responsible for the forwarding-plane functionalities of a network device, while all control-plane responsibilities are outsourced to the controller – our SDK would simplify the process of creating such a controller, making it more friendly to people with less extensive programming backgrounds.

A closely related protocol to SDNs is OpenFlow, an open-source protocol that runs over Transmission Control Protocol (TCP). OpenFlow provides standards and structures that simplify the work of managing an SDN. It provides message types to check for controller and switch functionalities, for a switch to request data-forwarding decisions from an SDN controller, and for the controller to reply to the switch with its decision. Among the most useful messages it provides is Flow-Mod, which both tells a network switch what to do with a given packet and tells it to store that rule in its flow tables, allowing the switch to handle similar packets without first contacting the SDN controller in the future.

Finally, designing an SDN controller specifically for serverless deployment furthers the accessibility element of the project. Serverless is an extension of cloud computing, with the goal of speeding up software deployment for widespread use. Essentially, a software development company "rents out" storage and processing power on a serverless provider's servers. When a user submits a request to the company's software, a serverless function is spun up on demand, at a data center owned by the serverless provider that is closest to the user. This has the advantage of outsourcing the management and maintenance of a software development company's own servers, as well as offering better performance for the end user, since a serverless provider could have servers in many data centers all over the country or even the world.

For employees, sending their web traffic through a serverless SDN controller would be a simple alternative to carrying a company-owned device, as their personal devices could be easily connected to a company-controlled, secure SDN. For IT departments, the underlying software development kit (SDK) would make it easy for anyone with minimal coding knowledge to create and customize their own SDN controllers, freeing them from the limitations of a specific hardware configuration's features.

Existing solutions in this area include Floodlight and Pox, other SDKs that allow developers to write custom SDN controllers. However, Pox is not designed specifically for OpenFlow, and thus is not optimized for that protocol, while Floodlight presents performance issues as it's written in

Java, an interpreted language. Further, not every serverless provider supports high-level languages like Java and Python, giving our controller an advantage as it's written in C++, a lower-level language. Using C++ also enables our users to choose from a wider variety of serverless providers, reducing vendor lock-in.

We plan to investigate whether our serverless SDN controller could achieve similar performance to a state-of-the-art Java controller such as Floodlight. Further, we seek to describe the performance characteristics of a serverless SDN controller, including aspects of scalability, latency, and overhead costs (i.e. cold starts).

Chapter 2

Background

2.1 Serverless Infrastructure

Serverless is a derivative of and expansion on cloud computing. While the cloud offers infrastructure as a service (IaaS), serverless goes a step further by offering function as a service (FaaS), allowing data to not only be retrieved but also processed at a server closer to the end user (Hassan, Barakat, and Sarhan 2021).

The rise of serverless is reflective of a greater industry trend: moving computing resources closer to the end user. Cloud computing companies like Microsoft Azure, Google Cloud Platform, and Amazon Web Services have dozens of data centers spread nationwide (and some even internationally) so that, with properly-designed technological infrastructure, a user can simply connect to the nearest data center, likely located in the nearest urban area. This contrasts with the traditional deployment model of running an application out of a single data center (typically in Ashburn, VA, which has traditionally been the default data center for all three tech giants: Microsoft, Google, and Amazon). Developers formerly had to write applications specifically designed to work across multiple regions, but serverless allows that deployment to happen effortlessly, independent of geography. This reduces application developers' obstacles down to one: state management. Coordinating potentially millions of instances across dozens of locations is nearly impossible. Serverless is built to handle this as well, as each instance of a serverless application simply issues calls to a centralized (or less distributed) database backend.

Essentially, serverless abstracts the entire backend away from the developer, providing a pay-as-you-go platform on which to execute small pieces of code (usually microservices) at scale. This saves developers the time and cost of maintaining and provisioning resources, allowing them to focus on the direct functionality of an application, while also enabling faster testing and deployment (Baldini et al. 2017).

However, serverless infrastructure also means that developers have virtually no control over the server on which their code executes, which places certain limitations on how developers can design functions designed to run in a serverless environment. Specifically, such functions are assumed to be stateless, and any need for persistent data storage must be met elsewhere (Al-Ali et al. 2018)

Serverless has some drawbacks, but its many advantages include scalability, cost-saving mechanisms, and potentially decreased latency, as a serverless application can run on any server in any data center around the world (Hassan, Barakat, and Sarhan 2021).

Designing our SDN controller as a serverless function has several benefits. First, it vastly simplifies operational costs to a customer, making it easier for companies to adopt our technology regardless of their current infrastructure. Second, most serverless providers only charge for server resources as they are actually used, allowing a customer to save money by not paying for times when requests are not being processed. Since the serverless model favors modularity and lightweight functions, and our project makes it trivial for IT professionals to write and customize SDN controllers, it is better-suited to a potential deployment model involving frequent updates and high mobility as new devices and locations are added to the network. Finally, this brings performance benefits to the user by allowing requests to potentially be processed closer to the "edge," or the end user's device (Baldini et. al).

Previous proposed expansions of serverless infrastructure include Al-Ali et al.'s "ServerlessOS," a process-oriented framework that would allow serverless processes to be scaled across not just a single server, but multiple physical servers in a given datacenter (456). Conceptually, this amounts to "making serverless more serverless," as the authors state. This has some implications for application networking. For one, in order for ServerlessOS to be feasible as proposed, data centers would require very fast internal networks, such that the cost of dynamically allocating resources across multiple servers does not overtake the performance benefit of that scaling. The authors suggest that software-defined networking techniques could be employed to ensure I/O disaggregation and proper scaling across multiple physical servers (458). The OpenFog Consortium suggests placing local fog nodes such that they can connect to other devices over computationally inexpensive local networks (OpenFog Consortium Reference Architecture: Executive Summary 4). And Masip-Bruin et al. raise the important issue of protecting sensitive data in serverless applications. Since our product will interact with users' personal web traffic, data security and privacy remain topmost concerns. In all these previous studies, network security and efficiency were declared to be critical in creating efficient, secure, and reliable serverless infrastructure.

As far as actual industry applications, Nguyen, Yang, and Chien lay out three concrete examples of serverless systems currently in use for a variety of academic and industry applications (Nguyen, Z. Yang, and Chien 2021). These include high-intensity data stream processing for the Large Hadron Collider; a distributed augmented reality/virtual reality (AR/VR) application that must respond to user action requests and synchronize an AR/VR scene across all users in a given room; and finally the Bloomberg Financial text annotation pipeline, which is effectively a distributed machine learning application. The Large Hadron Collider and Bloomberg Financial applications are both event-driven with respect to a serverless architecture. The AR/VR application is unique because every single user action triggers one invocation of a serverless function. However, all three applications achieved acceptable performance despite needing to process and synchronize huge quantities of data.

The Large Hadron Collider application's success was attributed to the high processing speeds achievable by CPUs today – examples given include Intel's Ice Lake SP and the FPGA. For the AR/VR application, the authors credit the specifications of high re-rendering rates and lax limits on the number of serverless function invocations. However, they acknowledge that meeting the application's data processing requirements is still challenging, and a truly scalable application of this type "requires careful resource allocation and load distribution design" (29). Finally, for Bloomberg Financial, the authors note how the serverless application breaks up its workflow into even smaller microservices (30). The takeaway from these three analyses is that serverless applications are more than feasible under the constraints of current technology, but they encounter special challenges when required to handle "bursty," sporadic user traffic.

In regards to future research, the authors propose "reducing function initialization overhead, employing load prediction to make proper pre-allocation, adding support for heterogeneous resources such as hardware accelerators, or bringing serverless instances closer to the caller (edge deployment), etc." (Nguyen, Z. Yang, and Chien 2021).

2.2 C++

C++ famously got its start as computer scientist Bjarne Stroustrup's Ph.D project, known as "C with Classes." In a document detailing the history of C++, Stroustrup himself credits the object-oriented programming language Simula as being part of his inspiration, saying, "C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming." Flexibility was an important element that was preserved in creating "C with Classes" based on C. Since C is so low-level, it allows the programmer to potentially perform unsafe memory operations. Some modern languages, such as Python, smooth this over by performing memory management for the programmer, but C with Classes kept those potentially "unsafe" features intact (Stroustrup 1995).

C with Classes was formalized in April 1980 (Stroustrup 1982), a version of the language that implements familiar object-oriented concepts such as classes and inheritance. The next significant updates to the language came in 1983 (Albatross n.d.), when polymorphism and function overloading was added; this is when the language first shed the name "C with Classes" and became known as "C++" in its own right. Two years later, C++ was released commercially for the first time, but there was no official standard for the language at that point, prompting Stroustrup to publish his own reference, "The C++ Programming Language" and a second edition (to include new features that had been added to the language) four years after that. Finally, in 1998, a group at the International Organization for Standardization (ISO) formalized C++ into ISO/IEC 14882:1998, the document that formed the basis for every C++ standard thereafter (ISO 1998).

C++'s popularity peaked in 2003 and has generally declined since, but it is by no means a dying language. In the December 2022 TIOBE Index, which seeks to measure the popularity of programming languages by examining search query data, Python ranks as the most popular language, but C and C++ rank second and third. Both C and C++ show greater increases in popularity from the previous cycle's data than Python does (4.77% and 4.21% respectively, compared to Python's increase of 3.76%) (TIOBE 2022). Software developers cite C++'s speed and performance benefits, versatility (Exterman 2021), and its ability to update itself based on programmers' needs (Castro 2021), as reasons the language remains in widespread use today.

Originating as an object-oriented extension to the C programming language, C++ eventually grew into a massive language of its own, supporting nearly every coding paradigm while also maintaining remarkably fast native performance. Modern C++ features allow programmers to write elegant, high-performance code that is difficult to replicate in other languages. Below we describe some of the many features in C++ that we anticipate being particularly useful to our project.

2.2.1 Constexpr

Constexpr, short for constant expression, is a feature introduced in C++11 that allows the programmer to write functions to be evaluated and precomputed at compile time. This had been possible in one form or another since C++98, but that was using template metaprogramming, a much harder technique. Constexpr allows using standard syntax function marked as constexpr to write code computed at compile time. This project will benefit from this by precomputing OpenFlow packet structures and embedding the binary directly into the software, so that responding to packets simply involves a memory copy operation.

2.2.2 Templates

Templates are a C++ feature that at first glance seems similar to generics from other languages like Java and Rust, however they are much more powerful. When templates get used by other code in the program, two (or more) distinct versions of the entire function or class are generated as if they were completely separate. For example, if you use a std::vector<int> and std::vector<std::string> in the same program, it would be as if two completely different classes had been written. This, combined with non-type template parameters (allowing numbers, strings, and even composite types to be passed into functions) will allow this project to support an accessible low-code solution, which will be outlined in more detail later in this paper.

2.3 Rust

Rust is a low-level systems language with security and safety checks built in, allowing it to make certain memory guarantees with low overhead (*Rust Programming Language* 2022). Its performance and power efficiency were third only to C and C++ when benchmarked using programming problems from the Computer Language Benchmark Game repository (Pereira et al. 2021) Rust also transpiles to WebAssembly, enabling performance that is orders of magnitude faster than even the most optimized JavaScript — but still at least four times slower than native software (Piepera et al. 2021).

We briefly considered Rust as an alternative to C++ for our project, as Rust, as both languages have similar features: low-level efficiency, performance, and versatility. Like C++, Rust does not offer services such as garbage collection, which allows it to retain high native performance and run on embedded devices. It also offers similar compiler warnings and type systems to C++, with the addition of safety constructs including the concept of "ownership," which has implications for scope and memory safety (*Rust By Example: Ownership and moves* n.d.). Rust originally sought to eliminate the memory vulnerabilities and the potential for "unsafe" behavior that is present in both C and C++, two of the most widely-used low-level languages (TIOBE 2022). An oft-quoted figure is that nearly three-quarters of CVEs (Common Vulnerabilities and Exposures) relate to memory exploits in C++. The Microsoft Security Response Center performed an analysis of all of Microsoft's reported security vulnerabilities dating from 2004 and found that 70% of those vulnerabilities were caused by C/C++ memory corruption bugs (MSRC 2019). However, as the article itself states, many of those CVEs can be eliminated by simply enabling compiler warnings and marking them as errors, in addition to increasing developer education in the area. Many developers treat C++ as merely "C with Classes" and ignore many of C++'s low-level features which, if used, could help eliminate entire classes of memory bugs. Thus, we believe that these arguments in favor of Rust do not disqualify C++ as being the best choice for our project. In fact, for our use case, Rust is at several disadvantages to C++.

First, Rust allows programmers to upload and share their work with others using a package registry called Crates.io (*Crates.io* n.d.). Having access to other users' code on crates.io initially seems like it would enable a lot of functionality; however, since anyone can upload their code to

Crates.io, the projects published there could be poorly designed, poorly maintained, or even open to potential license violations and security vulnerabilities. Thus, we would need to thoroughly inspect any and all dependencies, with a strong preference for more recently-published code in order to ensure quality and security. Additionally, when writing high-performance code, Rust's borrow checker may make it impossible for us to achieve maximum performance on the system, due to the memory safety-related rules enforced by the language. Thus, our code might have needed to employ Rust's unsafe construct, in places where human reasoning about safety was appropriate.

In the end, the Microsoft Security Response Center's summary provides the best argument for our choice to use C++: "C++...is blisteringly fast, it has a small memory and disk footprint, it's mature, it's execution predictable, its platform applicably [sic] is almost unparalleled and you can use it without having to install additional components" (MSRC 2019). Taken together, C++'s advantages, as outlined here and in the previous section, far outweigh its downsides.

2.4 WebAssembly (WASM)

WebAssembly was the end product of a long history of attempted browser optimizations. As the world wide web increased in popularity and users began to run more programs directly from their browsers, the need for code that could execute quickly in a browser environment grew. Readers may already be familiar with some of the older, earlier solutions, including Java applets and Flash.

Oracle's official documentation describes Java applets as "a special kind of Java program that a browser with Java technology can download from the internet and run, and as programs that "enabled richer functionality through a browser plugin at a time when browser capabilities were very limited, and provided centralized distribution of applications without requiring users to install or update applications locally." Applets were expected to execute in a browser environment. However, this created performance concerns; if an applet required a different version of the Java runtime environment (JRE) than the one available on the user's machine, it would have to install the new JRE from scratch, forcing the user to wait until the download process completed. Moreover, Java applets came with a host of security concerns. Aside from vulnerabilities in the software itself, Java applets were difficult to sandbox, containing a number of edge cases that, if missed, could be exploited by bad actors. Although most browsers automatically blocked untrusted applets as a security measure, the user still had the power to disable this protection and allow the applet to run anyway. And since not all users were aware of the difference between "trusted" and "untrusted" Java applets, this meant that they might unknowingly enable untrusted Java applets, posing substantial security risks.

The final blow to Java applets as a web interactivity solution came with the rise of mobile devices (*How do I get Java for Mobile device?* N.d.). Most mobile browsers don't support the plugins that are necessary for Java applets to execute, meaning that any company that used Java applets for important parts of its website's functionality would effectively cut itself out of the mobile device user market. As Oracle puts it, "The rise of web usage on mobile device browsers, typically without support for plugins, increasingly led browser makers to want to restrict and remove standards-based plugin support from their products, as they tried to unify the set of features available across desktop and mobile versions" (Java Client Roadmap Update). By 2017, with the release of JDK (Java Development Kit) 9, Oracle announced that applets were being deprecated (*JDK 9 Release Notes - Deprecated APIs, Features, and Options* 2017). Java applets, while they had served their purpose in the early days of the web, had been definitively phased out of the technology industry.

Next came Flash. In 1996, Macromedia began distributing the player component of Macromedia Flash 1.0 as a free browser plugin. By 2005, Flash appeared in more web browsers than any other plugin competitor, including Java applets, and that same year, Flash was acquired by Adobe, underscoring its popularity. Contemporary news articles called it "a potentially formidable competitor in the race to build powerful Web-based applications" and noted that "[m]ore than 98 percent of personal computers connected to the Web have some version of the Flash player installed, according to Macromedia, and more than 100 equipment manufacturers are building Flash into their devices" (*Just a Flash in the Web video pan?* 2005). Flash was lighter, more portable, and considered more usable and accessible than Java applets.

However, web standards continued to evolve throughout the early 2000s, and as with Java applets, the rise of mobile devices damaged the popularity of existing web technologies for desktop. When Apple released the first iPhone in 2007 without support for Flash, Steve Jobs justified this decision in a high-profile letter, "Symantec recently highlighted Flash for having one of the worst security records in 2009. We also know first hand that Flash is the number one reason Macs crash. We have been working with Adobe to fix these problems, but they have persisted for several years now.” (Keplek 2015) Indeed, Flash Player malware was very common, including the Flashback, Open Lotus, InstallCore, and MacDownloader attacks, which all prompted users to install what appeared to be a legitimate update to Adobe Flash Player on their computers.

By 2015, YouTube, one of the first major web platforms to adopt Flash back in 2005, dropped support for Flash. The same year, Mozilla disabled Flash plugins by default in its browsers. Finally, in 2017, Adobe announced that they would “stop updating and distributing the Flash Player at the end of 2020 and encourage content creators to migrate any existing Flash content to these new open formats,” explaining that “as open standards like HTML5, WebGL and WebAssembly have matured over the past several years, most now provide many of the capabilities and functionalities that plugins pioneered and have become a viable alternative for content on the web” (“Flash and The Future of Interactive Content” 2017).

Both of these once-prominent technologies came to be replaced by a still more prominent language, JavaScript. After some effort to adapt existing languages for the browser, Netscape Communications redirected its effort into developing a lightweight scripting language for the browser instead, and thus JavaScript was created in 1995. Netscape submitted the new language to ECMA International.

In September 2008, Google released its first version of Chrome running V8, a just-in-time JavaScript compiler aimed at making JavaScript faster to execute in the browser. The same month, Safari released SquirrelFish Extreme, a competitor to V8. Webkit development team leader Maciej Stachowiak wrote on SquirrelFish’s optimizations, explaining that it employed “bytecode optimizations, polymorphic inline caching, a lightweight ‘context threaded’ JIT compiler, and a new regular expression engine that uses our JIT infrastructure” to produce a very high-performance JavaScript engine. And in 2009, Mozilla released TraceMonkey with Firefox 3.5, with its own JIT compiler for JavaScript. JavaScript was a very widespread language on the web, and with these three major tech companies competing to make it faster, JavaScript’s browser performance suddenly grew by leaps and bounds. At the end of 2009, the ECMAScript 5 standard was released, which represented the combined work of all major contributors (open-source and corporate) on JavaScript up to that point (*Ecma International approves major revision of ECMAScript* 2009).

Numerous JavaScript browser optimizations followed, the first of which was the idea of “just-in-time” compilation. Compilation is a necessary step to translate JavaScript into code that is understandable by the user's machine. However, Internet users could be using any browser and any machine, and compilation is specific to the user's machine, making it necessarily a slower process. Just-in-time compilation is similar to "lazy loading" images on a webpage — namely, compiling a function to machine code only when it is actually called by the web application. After a function has been called for the first time and compiled, subsequent calls are also faster since the JIT compiler will have cached the machine code and be able to execute it immediately. This tends to save time on startup (since a JIT compiler will not attempt to compile every single function when the page loads), but does have performance implications for the first time each function is called.

Some JIT compilers incorporate an element of AOT (ahead-of-time) compilation as well, implementing AOT compilation to bytecode, then compiling that bytecode to machine code using JIT logic. This adds some lag due to the AOT compilation, but improves performance once page load is completed, since the JIT compilation process from bytecode to machine code is now faster.

By 2010, work on optimizations continued, ultimately producing NativeClient, or NaCl, which sought to provide faster, close-to-native web performance without sacrificing the security of the runtime environment. However, NaCl worked by sandboxing x86-32 machine code, limiting its portability.

Later in 2010, Portable Native Client (PNaCl) was released. It shipped LLVM bytecode, which was then compiled to NaCl specific to the client's machine. Native Client was developed with security in mind, and in that regard it prevents developers from calling web APIs. Native Client applications could only access specific plugin APIs, namely the Pepper Plugin API. However, the functionality of the Pepper API was already available in other web APIs, so in a sense, the developers of Pepper had to reinvent the wheel just to make Native Client work.

Another predecessor to WebAssembly was asm.js, allowing low-level, statically-typed languages to be transpiled to JavaScript. Although asm.js allowed languages like C to be run in the browser with significant optimization, the tradeoff was that it limited language functionality to certain features that would be efficient when transpiled. asm.js enjoyed some popularity around 2013, when both Google's Octane 2.0 benchmark and Unity added support for asm.js. Unlike NaCl, asm.js could use Web APIs and runs directly in the DOM.

Finally, in 2015, the development of WebAssembly was announced. WebAssembly is a newer technology aimed at overcoming all the aforementioned limitations of JavaScript and making high-performance web software possible. In fact, the initial implementation of WebAssembly was based on asm.js' features.

In 2022, JavaScript engines are highly optimized, with key contenders including V8 (from Google Chromium and its many derivatives), SpiderMonkey (from Mozilla Firefox), and JavaScriptCore (from Apple's WebKit, which powers Safari and many embedded devices). All of these engines employ advanced techniques such as Just-in-Time (JIT) compilation to execute JavaScript as efficiently as possible.

Starting in the early 2010s, the Node.js project (which packaged Google's V8 engine with several libraries) enabled programmers to execute JavaScript in the context of a server or desktop application. This was later extended to serverless computing, which was initially seen as a great breakthrough. However, developers soon began encountering the same problems as client-side web browsers: namely, JavaScript's performance and cost, this time manifesting as increased time and money spent on the serverless platform. JavaScript still executes orders of magnitude slower than native code.

As a result, WebAssembly was created. It uses a bytecode approach similar to Java, where code from native languages (C, C++, and Rust, among others) is compiled into WebAssembly instructions. Each serverless function then begins execution with a small JavaScript shim, which loads the WebAssembly binary into memory and jumps to it. For a browser, WebAssembly bytecode is easier to parse compared to a full language like JavaScript. This means that a highly-optimized JIT compiler (typically LLVM) can execute web application code at speeds only four to eight times slower than native, orders of magnitude better than simple JavaScript.

Overall, WebAssembly is crucial to ensure that our serverless SDN controller can process, route, and return requests efficiently, to make this system not only comparable to "normal" SDN controllers, but also viable on serverless infrastructure.

2.5 Port-Based Policy

In the context of the Internet and software-defined networks, "ports" refer not to a physical point on the device but to an abstract 16-bit value that can identify what type of traffic is being received. Ports are classified as layer 4 – the transport layer – in the traditional seven-layer open-systems intercommunication (OSI) network model. They determine to which process or program a packet is directed (*What a computer port? | Ports in networking* n.d.). Certain protocols default to using certain ports for the sake of predictability; for example, File Transfer Protocol (FTP) servers use port 21, Secure Shell (SSH) uses port 22, Simple Mail Transfer Protocol (SMTP) uses port 25, Domain Name Server (DNS) uses port 53, etc. As outlined above, source and destination port numbers are one of the identifying factors that can be used to determine rules for packet handling.

One application of our SDN controller would be to provide port-based policy; for example, blocking all FTP traffic or blocking all SSH connections. One example would be the SSHBlocker-Controller, which we will use our SDK to implement an SDN controller policy for.

2.6 Software Defined Networking (SDN)

To understand software-defined networking, it is first necessary to understand computer networks on a general level. Computer networks consist of any number of devices, switches, and routers, each of which must be able to communicate with the others by sending packets. Networks are conceptualized as having two planes of functionality: the control plane and the data (or sometimes forwarding) plane. The control plane determines rules for packets' direction and flow through the network; such rules can be based on the packets' origin port or device, the destination port or device, the packet type, or any combination of those factors. The control plane also determines network topology – that is, a "bird's-eye" view of the data flow through the whole network (*What is the control plane? Control plane vs data plane* n.d.). Meanwhile, the data plane is responsible for implementing the rules given by the control plane. The control plane deals with network protocols, while the data plane deals with packet-level operations such as decrementing the time-to-live and verifying checksums in packet headers. Since the control plane is far more abstract than the data plane, it was a natural next step to seek to separate them.

As previously mentioned, software-defined networking enables a programmer to abstract a network's functionality away from the hardware. SDN as a concept for the web can be said to originate in April 2004, when the International Engineering Task Force (IETF) published a memo about a proposed "Forwarding and Control Element Separation (ForCES) Framework." After distinguishing the data plane and control plane by identifying "control elements" (CEs) and "forwarding elements"

(FEs) in a basic network diagram, the memo proposes the logical and physical separation of those elements. A connection between a CE and FE could be initiated by either entity; it would begin with a security handshake, followed by an exchange of information regarding what packet types and processing each network element could support. The authors also acknowledge the need for ongoing authentication throughout the CE-FE communication process; individual packets must undergo continuous authentication, and if they fail this process, they should be dropped and a security alert put out. Several elements from this 2004 memo would be familiar to programmers working with SDNs today, including the redirection of packets by an FE to a CE for decision-making, the CE querying FEs for relevant network and packet statistics, and the FE sending messages to the CE regarding major network events, such as dropped packets or port failures. The memo goes on to summarize the basic concept behind software-defined networking: "In general, the forwarding plane carries out the bulk of the per-packet processing that is required at line speed, while the control plane carries most of the computationally complex operations that are typical of the control and signaling protocols" (L. Yang et al. 2004).

Also reflected here is one of the primary concerns that initially prevented SDNs from gaining widespread appeal: cybersecurity. The authors acknowledge that "the physical separation of two entities usually results in a potentially insecure link between the two entities" (L. Yang et al. 2004), and thus they dedicate nearly seven pages to addressing potential security risks in the proposed separation of a network's control and data plane. Message flooding, impersonation, and replay attacks are all potential vulnerabilities, as well as the security of network data itself, like the confidentiality or integrity of packet contents.

Still, a few years later, in 2007, Ethane was developed by a research group at Stanford University. Ethane can be categorized as a software-defined networking project because it utilizes "a centralized controller that manages the admittance and routing of flows" and is implemented in both hardware and software (Casado et al. 2007). It also added a network manager-friendly element in that it emphasized managing networks at a high level – determining policy based on "users, hosts, and access points" rather than based on "low-level and often dynamically-allocated addresses" (Casado et al. 2007). Ethane's proposed model of Switches and Controllers (capitalized as per the original paper) bears strong resemblance to the Open vSwitches and Open SDN Controllers of today. An Ethane Switch's flow table consists of "a Header (to match packets against), an Action (to tell the switch what to do with the packet), and Per-Flow Data [such as packet and byte counters]" (Casado et al. 2007).

Essentially, rather than use a network switch with a performant but highly specialized piece of custom silicon, instead the network can be programmed according to a set of policies. Critically, these rules are not configured directly on the hardware device, like settings on a typical managed switch. Instead the rules reside on a central server, which each network switch can refer to in order to know how to handle a packet. This allows near-limitless network flexibility, but also incurs a speed and performance cost due to the "lookup" process between the many network switches and the centralized SDN controller. In an effort to minimize this cost, the switch also has the ability to cache rules sent back by the controller for common traffic. This way, the switch does not need to escalate requests to the SDN controller for every single web request; it can simply handle the packet based on its own locally-stored rules, resulting in a performance benefit.

Ultimately, the rapid growth, integration, and expansion of networks – along with the publication of standards such as OpenFlow, the development of the Internet of Things, and the rise of edge computing – spurred the rise of SDNs. Originating in the research and academic sphere, SDNs now broke into the industry. SDNs' programmability and flexibility of the control plane allows them to be easily modified and updated to account for factors such as energy efficiency (Oliveira, Xavier-de-Souza, and Silveira 2021) or path length (Assefa and Özkasap 2020), which are becoming ever more important as networks continue to expand in size. In fact, some researchers have made proposals based on the underlying concept of SDNs: namely, even further logical abstraction of various network components, such as dividing a single control plane into two levels in order to avoid bottlenecks in the control planes of important Internet of Things (IoT) device gateways (Ren et al. 2018). SDN-based models have also been proposed for autonomous vehicle networks in the growing IoT (Stojmenovic 2014).

Overall, software-defined networking is a powerful new trend in the industry. There currently exist multiple different open-source implementations of SDN controllers; however, most are far from production-grade or are missing important features.

One of the most popular SDN controllers is Pox, based on an earlier popular controller called Nox. Nox was written in C/C++, while Pox is written in Python (*POX Manual Documentation* 2020). Nox is the oldest of the controllers we will discuss here, meaning that it would likely be unsafe to use in production due to its unmaintained codebase and lack of support for the current specification (this could open it up to various security vulnerabilities). Pox has some of the same issues as its predecessor Nox, as its codebase is slightly unmaintained. Furthermore, Pox being written in Python, an interpreted language, comes with a performance cost compared to Nox. In addition to Nox and Pox, there is another popular controller called Floodlight, written in Java. Floodlight is the most regularly-maintained controller of the three, but like Pox, it suffers from being written in an interpreted language (*Floodlight SDN OpenFlow Controller* 2021), which is not ideal if an organization wants to route lots of traffic over the OpenFlow protocol alone.

One of the biggest issues with both Pox and Floodlight (which we aim to solve in our project) is the learning curve and existing coding knowledge required to produce an easily-configurable, user-friendly OpenFlow controller framework. We believe that it should not require advanced knowledge of Java to perform basic SDN-related tasks. Between Pox and Floodlight, the industry bar is set very low for a workable solution to easily create OpenFlow controllers.

In addition to making tasks easier for software developers, there is also a huge gain to be made on the performance side (i.e. better performance for users). According to a study conducted by Abasyn University, Pox is significantly less performant than a similar Python-based SDN controller called Ryu, and these controllers are both handicapped by being written in an interpreted language. Thus, there is likely a large performance gain to be realized by using C++, even with the added overhead of WebAssembly (Naimullah et al. 2021).

Chapter 3

Approach

3.1 Programming Language

When originally conceiving this project, we had planned on using Rust. This is because out of the four languages officially supported by Fastly, which are JavaScript, AssemblyScript, Go and Rust, Rust is the best option. However, as we began experimenting and planning the project, Rust proved to be unsuitable for this use case. This is for several reasons:

1. **Classes** Rust's trait/impl syntax does not lend itself to human-readable code, which was a primary goal in our SDK design. In C++, we can simply design consumers that implement a standard class without having to inherit or implement a trait or base class, then provide detailed error messages about any malformed structures.

2. **Templates** While Rust supports generics, and recent versions of Rust have added non-type parameters, these features are still orders of magnitude behind their C++ counterparts in terms of power and functionality. C++'s templates make it simple to implement the out-of-order optional parameter processing, a crucial feature to make our SDK user-friendly.

3. **Constexpr** Rust provides some support for compile-time operations, but it does not measure up to C++'s modernized constexpr. C++ allows pre-computation of OpenFlow packets, which will optimize the controller's performance and enable it to quickly respond to Packet-IN messages.

For all these reasons, we made the decision to pivot away from Rust to C++ for this project. With the help of a toolchain called wasi-sdk, it is possible to compile C++ into WebAssembly. This, combined with the documentation provided by Cloudflare, allows us to write our own C++ SDK, which will then be used to write our low-code OpenFlow controller SDK.

3.2 Design Overview

The project has a very simple design; it essentially acts as a middleware between the OpenFlow protocol, the interface exposed by Cloudflare, and the user interface. Most of our design decisions are dictated by the OpenFlow specification; it would be redundant to expose functionality that cannot be implemented or to neglect to implement a core functionality in the specification.

As such, we have chosen the following design:

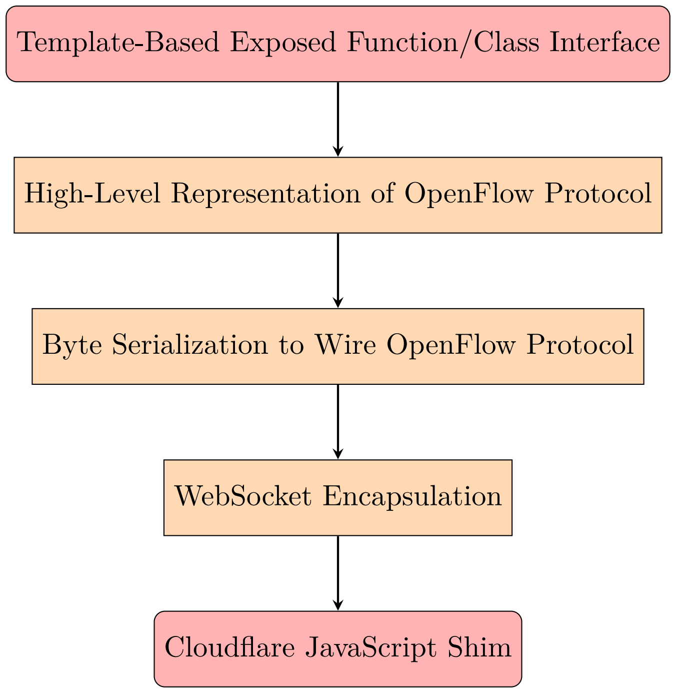

Template-Based Exposed Function/Class Interface
High-Level Representation of OpenFlow Protocol
Byte Serialization to Wire OpenFlow Protocol
WebSocket Encapsulation
Cloudflare JavaScript Shim

Figure 1. The five layers of our SDN controller design.

The overall flow between the client and server is shown below.

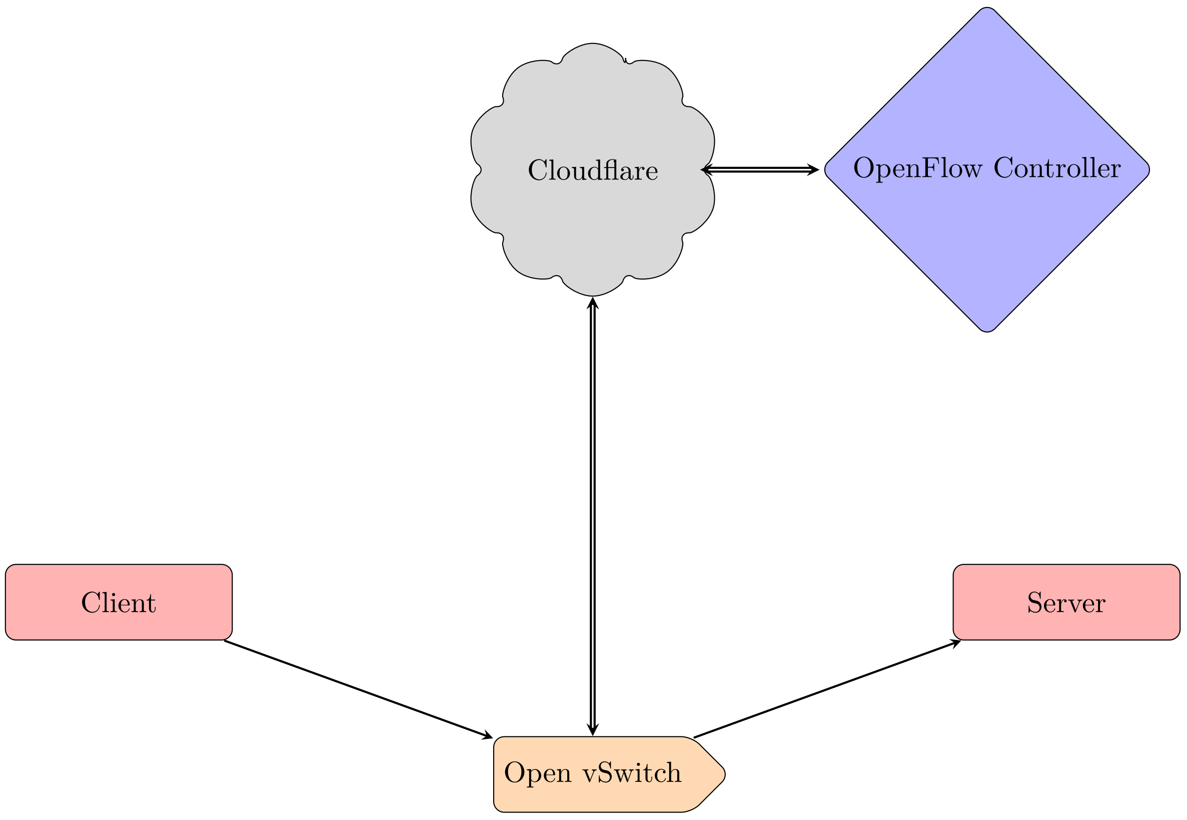


Figure 2. Client-server interactions with our SDN controller.

3.3 Controller-Client Procedure

3.3.1 Step 1: Template-Based Representation of Protocol

Each device is assumed to have the equivalent of an Open vSwitch (OVS) pre-installed. OVSes act as network switches in an SDN context; their primary task is to store data in flow tables, allowing for optimized network performance. Flow tables are versatile; they can store instructions based on source or destination IP addresses, device MAC addresses, ports, or request types. In addition to information needed to match against certain requests, flow table entries also include (1) instructions on what to do when a match is found, and (2) priority levels. (A flow table can store multiple actions for a given packet match, but by default, only the highest-priority one will be executed). Using the flow table, an OVS could decide to drop the packet, forward the packet, or do nothing. Flow tables use enumerated datatypes including oxm_ofb_match_fields for flow match headers, ofp_instruction_type for steps to take with the packet (such as setting up the next flow table in the pipeline, adding new actions to the given flow table entry, and executing or deleting actions stored in the entry), and ofp_action_type for possible actions to take on the packet (including switching ports, modifying headers, and pushing or popping various tags).

If an OVS receives a request that does not have an entry in the flow table, it sends it "up" to the SDN controller. For purposes of our project, the controller is deployed on Cloudflare's serverless platform.

An OVS communicates with an SDN controller using certain types of packets, which are dictated by the OpenFlow protocol specifications (McKeown et al. 2008). The exact packet types we plan to implement will be detailed below, but we will focus on two important types for now. When an OVS pings the SDN controller, the controller responds with either a Packet-OUT or a Flow-MOD message. Both types of messages tell the OVS what to do with the initial request, but the messages differ in their longevity.

A Flow-MOD packet tells the OVS what to do with the initial packet, then tells it to store the given behavior in its flow table. Thus, the OVS will no longer need to query the SDN controller for this particular behavior case, as it will have already cached that information. This is very useful for making permanent changes to network flow and forwarding, such as a network-wide security policy. Flow-MOD messages can also be used to modify or delete entries in a given OVS' flow table.

In contrast, a Packet-OUT message will only apply its instructions to that specific request. Upon receiving a Packet-OUT, the OVS will implement the behavior sent by the SDN, but it will not store that information in its flow table. In practice, this can be used for one-off cases – for example, a network administrator might need to send a given packet to every single switch on a network, but he would not want every single request thereafter to be sent to every switch on the network. Thus, a Packet-OUT response, rather than a Flow-MOD, would suit this purpose.

3.3.2 Step 2: High-Level Representation of Protocol

OpenFlow packets are serialized and sent as variable-length bit arrays over the wire. However, that data format is difficult to work with, so we will design a set of C++ classes that directly mirror the OpenFlow packet types necessary for this application (see above list), then deserialize and serialize packets to and from these classes.

Our deserialization function could produce as many as thirteen different return types, for each of the packet classes we plan to implement. Since return types are not an element of the function signature (which dictates overloading behavior), this situation would normally necessitate thirteen different functions, where each function returns one of the thirteen packet types. This is where C++'s template functionality proves extremely useful. Using C++, we can overload function return types and eliminate a potentially huge amount of duplicated code.

Our serialization function does not need to be templated, as it can simply overload parameters, accept any packet object, and have the same return type of a variable-length byte array regardless.

This object-oriented representation of the OpenFlow protocol at the packet level draws substantially on C++'s more modern, advanced features to produce intuitive code.

3.3.3 Step 3: Byte Serialization to Wire Protocol

OpenFlow messages, as defined in the OpenFlow protocol specification, contain a byte indicating the base packet type, such as OFPT_HELLO or OFPT_FLOWMOD. If necessary, it also supplies other information in subsequent bytes, such as a byte that indicates if an OFPT_FLOWMOD message is of type add, modify, or delete.

For the actual serialization, the code will take the higher-level structures and classes (modeled on OpenFlow messages) from the previous section and transform them into raw bytes at compile time. In order to do this, we must first determine what OpenFlow packet types will be supported, and we settled on the following thirteen.

For each packet, we provide a brief description of its functionality in the context of an SDN controller-Open vSwitch infrastructure.

1. **OFPT_HELLO** The OpenFlow hello packet is used to start a handshake with a controller, and optionally contains extra pre-connection information (such as versions of the OpenFlow protocol supported). We will implement this because it is required to create a connection, and we will also provide basic version bitmap checking to prevent invalid clients from connecting.

2. **OFPT_ERROR** The OpenFlow error packet is used to signal an error that occurred on the switch to the controller to allow the controller to log, react, and mitigate potential issues. We will implement this due to the triviality of adding it, and have the controller output aggregated error logs.

3. **OFPT_ECHO_REQUEST** The Echo Request is a packet sent by the switch to the server to verify the state of the connection and/or keep it alive with TCP. Optionally, the client can also send additional arbitrary data to have the server echo it back. We will implement this type of packet because it will be an easy way to check connection in testing.

4. **OFPT_ECHO_REPLY** The Echo Reply is the counterpart to the Echo Request, it is what the server sends back to the client with the arbitrary data as requested. We will implement this because it is needed for Echo Request to work properly.

   From the OpenFlow specification on the OFPT_ECHO_REQUEST and OFPT_ECHO_REPLY message types: *The body of the message is undefined and simply contains uninterpreted data that is to be echoed back to the requester. The requester matches the reply with the transaction id from the OpenFlow header.*

5. **OFPT_FEATURES_REQUEST** The Features request is sent by the controller as part of the handshaking process to determine what features and functionalities are supported by the switch so the controller can modulate its behavior in response. We will implement this because it is necessary for a proper handshake and we can reject clients that do not support all of the features we require.

   From the OpenFlow specification on the OFPT_FEATURES_REQUEST messages types: *This message does not contain a body beyond the OpenFlow header.*

6. **OFPT_FEATURES_REPLY** The Features Reply is sent by the switch in response to the request. We need to parse this as part of the handshake and to verify all the requested features we need to support work on the switch.

7. **OFPT_GET_CONFIG_REQUEST** The SDN controller uses this packet type to request configuration parameters from the switch. We are implementing this so that our SDN controller will have access to information about the switch's current state.

8. **OFPT_GET_CONFIG_REPLY** The switch uses this packet type to send configuration parameter information to the SDN controller. This will be necessary to respond to packets of type Get Config Request.

9. **OFPT_SET_CONFIG** The SDN controller uses this packet type to update configuration parameters on the switch. Once the controller has requested the current configuration parameters using Get Config Request, it will then be able to modify them as required using this packet type.

10. **OFPT_PACKET_IN** The switch uses this packet type to send information to the SDN controller. This is critical to the function of our end product.

11. **OFPT_FLOW_REMOVED** The switch uses this packet type to notify the SDN controller that an entry has been removed from its flow tables, if the SDN controller has requested such notification. Since we will implement the Flow Mod packet type as part of our application, the Flow Removed packet is a natural dependency.

12. **OFPT_PACKET_OUT** This packet type is used by the SDN controller to send information to the switches. This is critical to the communication function of our application.

13. **OFPT_FLOW_MOD** This packet type is used by the SDN controller to add, update, or delete entries from flow tables stored on switches in the network. This is the most complex of the packet types we plan to implement, but it will greatly increase efficiency by preventing every single Packet In from needing to be elevated to the SDN controller.

    These are the thirteen packet types our SDN controller will implement.

3.3.4 Step 4: WebSocket Encapsulation

To actually send the packets-turned-byte arrays between the server (a serverless worker in this case) and client (a device that implements the OpenFlow protocol), we plan to use WebSockets, a protocol for enabling bidirectional communication between web servers and clients. In addition to being a common web communications protocol, WebSockets operate over TCP, but they also employ TLS encryption for secure messaging over the wire. A WebSocket connection is initiated when the client sends an HTTP request with an upgrade header; once the server responds with status code 101 ("Switching Protocols"), the WebSocket connection is initiated.

In our project, this layer will need to receive the byte array from the above layer and send it using the WebSocket protocol. The server will maintain a WebSocket connection to the client, while the client connects to a TCP proxy to send socket messages. The next section will discuss how WebSockets are configured on the server side in JavaScript.

3.3.5 Step 5: Cloudflare JavaScript Shim

Planned Configuration with Cloudflare

In order to utilize Cloudflare's worker runtime, the first code executed must be in standard JavaScript. This is simply a limitation of Cloudflare's engine, and actually makes it easier to interface with their APIs.

First, the client will make a standard HTTP request with the upgrade header set. This signals to the server (the Cloudflare serverless worker) that the HTTP connection should be upgraded to a WebSocket connection. Without this, the server will reject any request with a error code 426, which means that an upgrade is required in order to proceed.

Additionally, at this phase we will also verify an authorization header set by the client to validate access. After this request, Cloudflare will complete the necessary WebSocket negotiation and protocol upgrade, then hand over control to a Cloudflare Durable Object. A Durable Object is Cloudflare's persistent state offering, which is absolutely critical to our project. Cloudflare's documentation states that "[w]ithout Durable Objects, there is no way to send an event to the specific Worker holding a WebSocket," which is exactly the task our SDN controller will need to achieve. Cloudflare's Durable Objects allow state information to be stored in memory while WebSocket messages are handled. Cloudflare picks the nearest data center to the client to store each Durable Object's data at, and all the programmer has to do is register a standard JavaScript function as the event handler.

In addition, Cloudflare provides two critical guarantees about Durable Objects that make them important to our project. First, Cloudflare guarantees that Durable Objects have globally unique IDs, and that each object with a given ID only exists in one place at any given time. This is a critical feature for managing multiple client connections simultaneously. Second, Cloudflare's Durable Objects block concurrent operations while callbacks are executed, allowing asynchronous code (such as waiting on WebSocket communications) to run safely.

After passing control from a serverless worker to a new Durable Object, we simply need to bind into C++ to start calling code in our actual SDK. This process uses a module part of the WebAssembly interface called 'ccall' which allows calling arbitrary C functions by specifying how to call them. This is pretty limited in the types of the parameters and return types, but all that is needed to be passed is a byte array (the OpenFlow data) and any other information we may need to persist or gather to serve the request (i.e. state of connection, client data, authorization, etc.). After this, execution will jump to our C++ code, which will decode the byte array into an object and handle the type of OpenFlow packet accordingly. Whether or not the object is a valid OpenFlow packet will be determined using our C++ parser functions.

Project Pivot: Amendment to Step 5

As work on the project progressed, we found it necessary to make changes to the platforms and technologies used. As has already been stated, we changed our chosen language from Rust to C++. When attempting to deploy the final product, we encountered significant technical issues involving environment dependencies, which could not be resolved in the project timeframe. Thus, we modified Layer 5 to instead be a containerized TCP server that would handle the WebSocket. In our minds, this was the best compromise available, especially since serverless providers such as AWS (*Deploy Docker Containers on Amazon ECS* n.d.) allow for direct deployment of Docker containers on their platforms. However, once the environment dependency issue is resolved, we believe it would be possible to deploy our existing application on Cloudflare as planned.

With this pivot came updates to our research questions as well. Initially we had set out to compare our serverless SDN controller, writtten in C++, to controllers written in other languages, such as Floodlight and POX. We also sought to describe other performance characteristics of such a serverless SDN controller, including aspects of scalability, latency, and overhead costs (i.e. cold starts). This was scaled down to testing our containerized controller against a POX controller (an open-source OpenFlow SDN controller written in Python), taking latency as our primary metric of performance. (Due to the change from Cloudflare deployment to containerization, we chose to drop the scalability metric, and due to the nature of "spinning up" and "spinning down" Docker containers, we chose not to measure overhead costs).

The updated design, featuring the containerized TCP server as the final stage of the application, is the one referred to from this point onward.

3.4 Core Interface

Much thought went into ensuring that the interface used by people writing controllers (presumably IT personnel, among others) would be highly accessible and policy-based, with as low a technical learning curve as possible. Using C++'s templates, we were able to write an interface that allows a programmer to specify optional parameters in any order, a feature that goes a long way towards achieving our goal of accessibility. Additionally, since all of the packet data is known at compile time, it is possible to pre-generate the OpenFlow packet data and embed this directly into the binary for better performance.

For example, imagine an IT employee who wanted to write a controller to block all traffic destined for port 22 (the default port for SSH traffic). With our SDK, we would aim to have the employee write something like the following:

```
// Each controller is represented by a class.
class SSHBlockerController {
    // The constructor can be used to create rules
       applied immediately upon vSwitch connection.
    explicit SSHBlockerController(OpenFlow::Connection&
        connection) {
        connection.Send<FlowMod>(Block, DestPort<22>);
    }
};
```

As shown in this example, the code is extremely concise and human-readable, and the policy can easily be interpreted from the controller code, even by non-technical personnel. This design will enable our SDK to have broad market appeal for people with little to no coding background, while not restraining experienced programmers and still allowing them to write potentially advanced controllers if they wish.

Below is a sequence diagram illustrating the flow of the finished product.

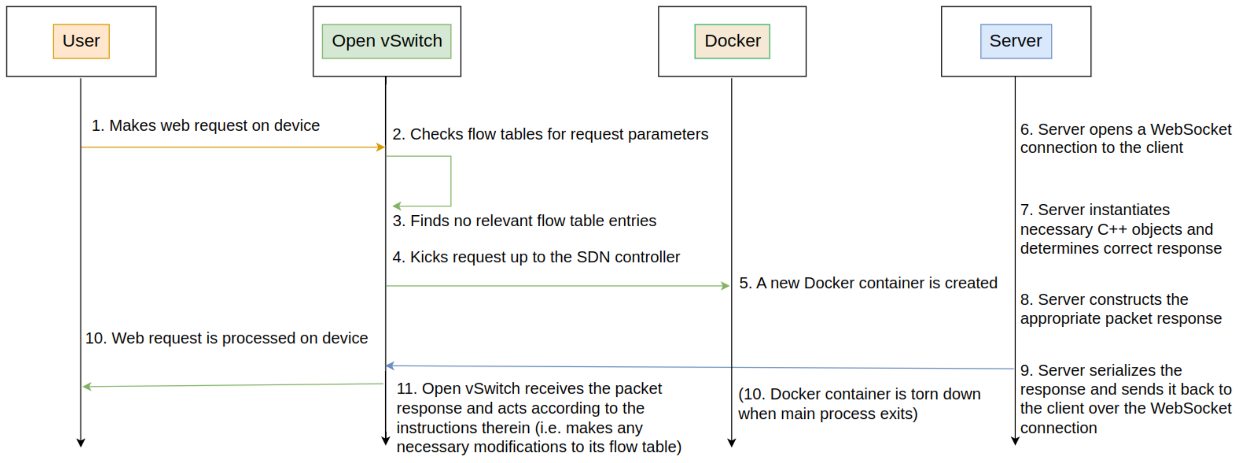

User
Open vSwitch
Docker
Server
1. Makes web request on device
2. Checks flow tables for request parameters
3. Finds no relevant flow table entries
4. Kicks request up to the SDN controller
5. A new Docker container is created
6. Server opens a WebSocket connection to the client
7. Server instantiates necessary C++ objects and determines correct response
8. Server constructs the appropriate packet response
9. Server serializes the response and sends it back to the client over the WebSocket connection
10. Web request is processed on device
11. Open vSwitch receives the packet response and acts according to the instructions therein (i.e. makes any necessary modifications to its flow table)
(10. Docker container is torn down when main process exits)

Figure 3. Software sequence diagram.

3.5 Testing

Building and deploying the SDN controller is only half of the project; we next need to test the controller against existing alternatives for OpenFlow SDN controllers. These include, as mentioned above, Pox, Floodlight, and Ryu. Our testing methodology will be largely informed by prior work in this area.

Bholebawa and Dalal (2018) conducted a performance analysis of Pox versus Floodlight, two open-source SDN controllers. Their tests involved a custom network topology, scripted in Python; the design ensured that each switch was connected to an equal number of hosts for testing purposes. They then ran an ICMP connectivity test by sending messages between two specific end hosts in a network using either a Floodlight or Pox SDN controller. They found that Floodlight had significantly lower round-trip-time (RTT) than Pox in all network topologies tested (single, linear, tree, and the custom Python topology described earlier). They also measured throughput as a metric of overall network performance and found that while the performance gap was greater for the custom topology than for the tree topology, the Floodlight controller consistently outperformed the Pox one on both the tree and custom topologies.

Altangerel, Chuluuntsetseg, and Yamkhin (2019) ran a similar experiment, but in addition to testing Pox and Floodlight, they also tested Opendaylight. Like Bholebawa and Dalal, their testing configuration used a simulation network in Mininet. The metric was likewise the RTT for the first packet of flow, which makes sense because an Open vSwitch could potentially handle subsequent packets without ever connecting to the controller. A PC running Mininet (in either tree or a custom Python-scripted mesh topology) served as the "client" in this scenario, while a virtual machine ran the SDN controller. They found that all three SDN controllers (Pox, Floodlight, and Opendaylight) performed much better in a mesh topology. Once again, though, Floodlight outperformed Pox, yielding the lowest RTT of all three controllers. However, Opendaylight also outperformed Pox, trailing Floodlight closely in terms of RTT. This held true for both the mesh and tree topologies, and for testing with packets of size 1000 bytes and 3000 bytes. Then, using iperf, a command-line utility for network diagnostics, they measured both TCP and UDP throughput over the various controllers and network topologies. Once again, Pox's throughput was generally much lower than Floodlight's, with Opendaylight's throughput noticeably fluctuating (sometimes greater than Floodlight's, sometimes lower than Pox's) over time. However, the three controllers' performance differed more greatly when measuring TCP than UDP throughput. Interestingly, Rowshanrad, Abdi, and Keshtgari (2016) found that Opendaylight actually outperformed Floodlight in certain scenarios – specifically, low load, tree-topology networks at 50% bandwidth cross traffic. However, in high-traffic networks, Floodlight performed better on average. OpenDaylight only outperformed Floodlight in high-traffic, single-topology networks; the authors reason that this is due to OpenDaylight's superior packet loss handling.

Our testing setup will be very similar to these previous studies, using Mininet to simulate three testing scenarios meant to check for latency, controller efficiency, and a particular use case illustrated in Chapter 2 in the form of the SSHBlockerController.

Chapter 4

Implementation

Overall, the design of the system is very simple, though some allowances had to be made to enable local testing.

The primary entry point to the system is a function called `HandleControllerPacket` which takes in a byte buffer and a mutable reference to a State object, and returns a byte buffer. The 'State' object is technically opaque to users of this function (such as the JavaScript shim or local main), but in practice just keeps track of variables for the state machine and policy.

For local testing, a `main()` function was implemented, which binds a TCP socket, accepts clients, and processes reads and writes on behalf of the entrypoint previously mentioned. The only logic here is detecting the size out of OpenFlow packets to ensure buffers passed to `HandleControllerPacket` are one complete packet. In a serverless setting, this function would not exist.

Next is the `Connection` object which exposes functions to send `FlowMod` or `PacketOut` objects, which are queued and sent to the switch. These objects are just basic structures based on what OpenFlow allows. The `Connection` object is passed to the callbacks in the `Controller` object, which is virtual, and it exposes two callbacks, one for switch connection and one for `PacketIn` events. The `PacketIn` event also receives a basic structure containing the data from the event.

As part of the internals of the handler, two template functions called `SerializePacket` and `DeserializePacket` are used. These convert the basic structure types to and from byte buffers ready for the socket.

Finally, the consumer of the code must provide the controllers that will be used to generate policy. This is done via an unimplemented function called `UserMain` that returns an array of function objects that produce `Controller` objects. The function objects are needed to allow multiple instances of a `Controller` for each vSwitch.

Chapter 5

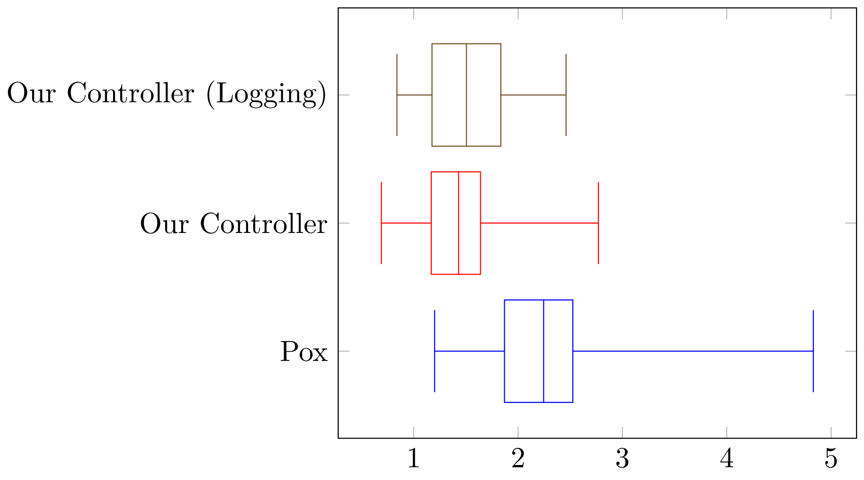
Results

To test our implemented controller, we added a small shim over the core library that listens over TCP, breaks the stream into discrete OpenFlow packets, and passes these packets to the `HandleControllerPacket` described previously.

For a test, we determined that the best way to do it would be to avoid installing any `FlowMod` rules and require every packet to be sent `PacketIn` to the controller so it forces a `PacketOut`. Thus, each and every packet flows through the controller for a better benchmark.

To complete this test, we rigged up a basic controller in both our controller and Pox, where it simply waits for a `PacketIn`, and then emits an identical `PacketOut`. One potential issue with this is that in order to more accurately simulate the serverless environment we would have to also simulate the cold start of the controller if it was evicted from the worker cache, and it was hypothesized docker could be used for this, but it is not actually a really meaningful test. First of all, Cloudflare's workers system does not use Docker, and instead uses a lightweight isolating runtime based on V8, a JavaScript engine. The way this works is that the bulk of the cold start time is spent on loading the JavaScript into memory and getting it into a form ready to execute, which is orders of magnitude less than the docker startup, which involves pulling a large image and then setting up the system to run the processes in the container, followed by any in-container initialization prior to user code. Additionally, for the computation, we cannot simply add the cold-start time to every call, due to the fact that Cloudflare caches workers in memory and also has measures in place with durable objects to avoid cold-starts. Thus, for our testing, we simply tested the raw performance of the controller as described. (Note that in a production scenario with constant controller queries, there would not be cold starts).

The results of the testing were in line with our hypothesis that our controller would be faster than Pox. Over a span of 50 ICMP ping packets, all sent over `PacketIn`/`PacketOut`, the average for Pox was 2.34 ms, with a standard deviation of 0.805. Compared to our controller which had an average of 1.44 ms, with a standard deviation of 0.427. Finally, some users may consider it important to be able to log using the controller. In Cloudflare, simply printing to output is captured and printed to a logging management system or storage solution. Thus, we ran the same test on the same controller but with a basic print statement added to the `PacketIn` evaluator. This produced an average of 1.51 ms, with a standard deviation of 0.394.



Chapter 6

Discussion

Due to the time constraints on the project, we had to compromise and test our OpenFlow controller core directly over a local TCP server. This did produce valid results, and our controller did indeed work better than Pox, but the testing conditions and results were not as compelling as originally intended. Future work would definitely include deploying our controller to a cloud serverless provider that supports all the necessary features, such as Cloudflare.

Overall, we speculate that our controller, once deployed on serverless infrastructure, would outperform a typical SDN controller like Pox or Floodlight running on a local machine. This is for several reasons. First, the raw controller code is more performant by nature of being a compiled language, even with the overhead imposed by WebAssembly and the JavaScript shim that would be required to deploy to a serverless provider like Cloudflare. In addition to this, the stateless, serverless nature of the software would allow technically infinite scaling of our SDN controller to occur, as needed to meet demand. The only limitation would be the platform itself, and given the level of traffic handled by the majority of organizations, it would be very unlikely that they would ever reach a cloud provider's limit of service.

The project ended up using a library-style architecture that was better-suited to the serverless environment. On serverless platforms, typically there is no `main` function (or similar entrypoint) into the code. Instead, certain functions are invoked from JavaScript or another language directly, similar to how a Foreign Function Interface (FFI) works but with WebAssembly and other security measures. To adapt the project into a testable state, we had to write a `main` function that performs the same invocations as a JavaScript or other shim would. This led to a somewhat unusual architecture for a desktop application, but, in theory, such architecture would enable easier deployment to serverless, as there would be very little modification required.

We originally planned to implement the project using Fastly's Compute@Edge platform. However, this proved technically impossible; there were difficulties managing the longer-lived connection required for the Open vSwitch , as it was not compatible with standard, stateless HTTP. The problem is that if a tunneling mechanism was used to send OpenFlow packets in HTTP bodies, then that would successfully convey the data to the server — but it would pose issues when future packets come in, as the state established by previous handshakes and OpenFlow packets would somehow have to be made persistent. The two primary options to handle this would be either through the use of HTTP cookies, or referring to a centralized database that could store state based on some session identifier. Neither of these work for our use case. HTTP cookies would be insufficient, as the OpenFlow protocol does not have an existing analog of this functionality, making it very challenging to implement. A centralized database would, in theory, fix this; however, it would defeat the purpose of designig our controller to be serverless, as the benefits gained by distributing and localizing the processing would be countered by the added network hops of each individual worker connecting to the central database. The solution we chose involved switching to stateful WebSockets (rather than stateless HTTP) as the communication protocol. This ensures a single session for all of the packets, and Cloudflare's implementation of Durable Objects wiht WebSockets also provides several other service guarantees that simplified our task.

During testing, it was decided not to impose the overhead of spinning up a Docker container to the testing. This is because in a production serverless scenario, it would have been irrelevant. Docker startup is much slower than serverless for several reasons. First is the time required to pull a much larger image (i.e. at least 20-30 MB for Docker, as opposed to under 5 MB for WebAssembly code and a JavaScript shim), along with unpacking it and creating a cgroup, which involves the kernel. This stands in contrast to the optimized runtime that Cloudflare advertises, which simply loads the JavaScript shim and assets from a cached server in the local data center network, then loads it into memory and begins execution (*How Workers works* 2022). Additionally, a true cold start would likely not occur very often in production, as Cloudflare attempts to keep code in memory for as long as possible before eviction, and a typical OpenFlow controller would receive enough traffic to keep it alive during business hours. The only noticeable startup cost when running on a serverless provider would have likely been the Cloudflare Durable Object creation, which in our case was untestable, since that is not possible to simulate locally.

For the original architecture we were planning, the only bottleneck to scaling would be the Cloudflare Durable Objects themselves, as Cloudflare provides synchronization of its Durable Objects to ensure that only one object with a given ID is running at any given time anywhere globally (*Using Durable Objects* 2022). Fortunately, our design uses one Durable Object for each connection of a given Open vSwitch, so a theoretically unlimited number of vSwitches can connect at once, and each will have a worker running in the (logically) closest serverless data center. Additionally, since the SDN controller policy is embedded as code, there is no centralized database to increase latency. Finally, if logging traffic is desired (for example, in the IT department application mentioned at the start of the paper), Cloudflare provides a built-in logging solution.

Cloudflare also ended up not being viable, but for a more subtle reason than with Fastly. It was possible to build a WebAssembly bundle for the platform, and it is possible to use WebSockets with Cloudflare's Durable Objects as well. This solves the issue of statefulness that we originally encountered on Fastly's platform. The issue with Cloudflare is based on Emscripten's runtime. Emscripten is a compiler toolchain and other assorted software that allows C/C++ code to be compiled into JavaScript and WebAssembly. However, Emscripten is designed for use in browsers on websites or on a Node.js server, and as such, it depends on certain functions and objects in a browser or Node environment in order to properly load and execute WebAssembly. Normally, this is a non-issue with Cloudflare Workers as, by default, Cloudflare servers emulate the necessary variables that allow Emscripten-compiled code to work. However, due to how Durable Objects are created and executed, this emulation is not present at the time our code would need to be loaded, which is inside of a Durable Object callback. Thus, it is not possible to run Emscripten WebAssembly in the manner necessary for the project, necessitating our switch to a local TCP server for testing. In the future, this bug could be fixed, or an alternative C++ compiler could be used.

While our paper speculates on features and mechanisms used by Cloudflare's workers system, it is not possible to know it all or fix bugs or issues that arise while using it. Most of the information is derived from Cloudflare's documentation (*Security model* 2022). Cloudflare's serverless workers platform is very promising for the future of cloud workloads due to its extensive feature set, low pricing, good documentation and tooling, and the future potential for it to be open sourced. However, in the current state of serverless, between Cloudflare and Fastly's platforms, we were simply not able to properly deploy a serverless OpenFlow controller.

In addition to the performance and deployment speed advantages of running an OpenFlow controller over serverless, it is also likely more secure than non-serverless infrastructure. For example, a single server running in a local data center is more vulnerable to a distributed denial-of-service (DDoS) attack than a serverless provider (likely with many servers in many data centers across the world) would be. Additionally, it is typically easier to secure access to resources with cloud providers, as cybersecurity is typically managed by the cloud provider itself. However, there is one serious drawback, which is vendor lock-in and the introduction of a third party in data management, so cybersecurity engineers should carefully consider backup plans and mitigation measures in the case that connectivity cannot be established with the provider.

In conclusion, it is unfortunate that several technical limitations prevented us from deploying a serverless OpenFlow controller on Cloudflare. However, the work accomplished was compelling in demonstrating how serverless infrastructure could improve the ease of deployment, ease of coding, performance, and reliability of OpenFlow SDN controllers. Currently, many OpenFlow controllers are deployed in virtual machines or on bare metal in a local data center or server (on premises). This is typically because this architecture enables low-latency connections from switches that allow the network to function effectively. However, using serverless infrastructure allows easier management, deployment, and maintenance of software applications. Further, the widespread deployment of serverless data centers coupled with the locality provided by providers has the potential to compete with traditional data center controllers. Future work could expand upon this by building and deploying a truly serverless SDN controller, whether by deploying our current controller or by writing and deploying a new one on serverless infrastructure.

Bibliography

Albatross (n.d.). *History of C++*. URL: `https://cplusplus.com/info/history/`.

Al-Ali, Zaid et al. (2018). "Making Serverless Computing More Serverless". In: *2018 IEEE 11th International Conference on Cloud Computing*, pp. 455–459. DOI: `DOI10.1109/CLOUD.2018.00064`.

Assefa, Beakal Gizachew and Öznur Özkasap (Feb. 13, 2020). "RESDN: A Novel Metric and Method for Energy Efficient Routing in Software Defined Networks". In: *IEEE Transactions on Network and Service Management* 17 (2). DOI: `10.1109/TNSM.2020.2973621`.

Baldini, Ioana et al. (June 10, 2017). "Serverless Computing: Current Trends and Open Problems". In: DOI: `https://doi.org/10.48550/arXiv.1706.03178`.

Casado, Martìn et al. (Aug. 27, 2007). "Ethane: Taking Control of the Enterprise". In: pp. 1–12. DOI: `https://doi.org/10.1145/1282380.1282382`.

Castro, Santiago (June 15, 2021). *6 Reasons C++ Is Still In Use Today.* URL: `http://blog.jobsity.com/6-reasons-c-is-still-in-use-today`.

Crates.io (n.d.). URL: `https://crates.io/`.

Deploy Docker Containers on Amazon ECS (n.d.). URL: `https://aws.amazon.com/getting-started/hands-on/deploy-docker-containers/`.

Ecma International approves major revision of ECMAScript (Dec. 15, 2009). URL: `https://www.ecma-international.org/news/ecma-international-approves-major-revision-of-ecmascript/`.

Exterman, Dori (July 26, 2021). *More than 35 years later, why is C++ still so popular?* URL: `https://sdtimes.com/softwaredev/nearly-a-quarter-century-later-why-is-c-still-so-popular/`.

"Flash and The Future of Interactive Content" (July 25, 2017). In: URL: `https://theblog.adobe.com/adobe-flash-update/`.

Floodlight SDN OpenFlow Controller (May 25, 2021). Project Floodlight. URL: `https://github.com/floodlight/floodlight` (visited on 10/28/2022).

Hassan, Hassan B., Saman A. Barakat, and Qusay I. Sarhan (2021). "Survey on serverless computing". In: *Journal of Cloud Computing* 10. DOI: `https://doi.org/10.1186/s13677-021-00253-7`.

How do I get Java for Mobile device? (N.d.). URL: `https://www.java.com/en/download/help/java_mobile.html`.

How Workers works (2022). Cloudflare. URL: `https://developers.cloudflare.com/workers/learning/how-workers-works` (visited on 12/15/2022).

ISO (1998). *ISO/IEC 14882:1998 Programming languages — C++*. URL: `https://www.iso.org/standard/25845.html`.

JDK 9 Release Notes - Deprecated APIs, Features, and Options (2017). URL: `https://www.oracle.com/java/technologies/javase/9-deprecated-features.html`.

Just a Flash in the Web video pan? (Aug. 2, 2005). URL: `http://news.zdnet.co.uk/internet/0%2C1000000097%2C39211831%2C00.htm`.

Keplek, Patrick (July 14, 2015). *The Death Of Adobe Flash Is Coming, And Game Developers Are Worried*. URL: `https://kotaku.com/the-death-of-flash-is-coming-and-not-everyones-happy-1717824387`.

McKeown, Nick et al. (Apr. 2008). "OpenFlow: Enabling Innovation in Campus Networks". In: *ACM SIGCOMM Computer Communication Review* 38 (2).

MSRC (July 16, 2019). *A Proactive Approach to More Secure Code*. URL: `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`.

Naimullah et al. (2021). "Performance Analysis of POX and RYU Based on Dijkstra's Algorithm for Software Defined Networking". In: DOI: `https://doi.org/10.1007/978-3-030-77246-8_3`.

Nguyen, Hai Duc, Zhifei Yang, and Andrew A. Chien (June 25, 2021). "Motivating High Performance Serverless Workloads". In: *HiPS '21 Proceedings of the 1st Workshop on High Performance Serverless Computing*, pp. 25–32. DOI: `https://dl.acm.org/doi/10.1145/3452413.3464786`.

Oliveira, Tadeu, Samuel Xavier-de-Souza, and Luiz Silveira (May 28, 2021). “Improving Energy Efficiency on SDN Control-Plane Using Multi-Core Controllers”. In: *Energy Efficiency in Cloud and Edge Computing* 14 (11). DOI: https://doi.org/10.3390/en14113161.

Pereira, Rui et al. (May 1, 2021). “Ranking programming languages by energy efficiency”. In: 201 (2021). DOI: https://doi.org/10.1016/j.scico.2021.102609.

Piepera, Ricardo et al. (2021). “High-level and efficient structured stream parallelism for rust on multi-cores”. In: 65 (2021). DOI: https://doi.org/10.1016/j.cola.2021.101054.

POX Manual Documentation (May 20, 2020). NOX Repo. URL: https://noxrepo.github.io/pox-doc/html/ (visited on 10/28/2022).

Ren, Wei et al. (Dec. 18, 2018). “A Novel Control Plane Optimization Strategy for Important Nodes in SDN-IoT Networks”. In: *IEEE Internet of Things Journal* 6 (2). DOI: 10.1109/JIOT.2018.2888504.

Rust By Example: Ownership and moves (n.d.). URL: https://doc.rust-lang.org/stable/rust-by-example/scope/move.html?highlight=ownership#ownership-and-moves.

Rust Programming Language (Oct. 24, 2022). Rust Team. URL: https://www.rust-lang.org/ (visited on 10/28/2022).

Security model (2022). Cloudflare. URL: https://developers.cloudflare.com/workers/learning/security-model/ (visited on 12/15/2022).

Stojmenovic, I. (2014). “Fog computing: A cloud to the ground support for smart things and machine-to-machine networks”. In: *2014 Australasian Telecommunication Networks and Applications Conference (ATNAC)*. DOI: 10.1109/ATNAC.2014.7020884.

Stroustrup, Bjarne (Jan. 1, 1982). “Classes: an abstract data type facility for the C language”. In: 17 (1), pp. 42–51. DOI: https://dl.acm.org/doi/10.1145/947886.947893.

— (1995). *A History of C++: 1979-1991*. URL: https://www.stroustrup.com/hopl2.pdf.

TIOBE (2022). *TIOBE Index for December 2022*. URL: https://www.tiobe.com/tiobe-index/.

Using Durable Objects (2022). Cloudflare. URL: https://developers.cloudflare.com/workers/learning/using-durable-objects/ (visited on 12/15/2022).

What a computer port? | Ports in networking (n.d.). URL: https://www.cloudflare.com/learning/network-layer/what-is-a-computer-port/.

What is the control plane? Control plane vs data plane (n.d.). URL: https://www.cloudflare.com/learning/network-layer/what-is-the-control-plane/.

Yang, Lily et al. (Apr. 2004). “Forwarding and Control Element Separation (ForCES) Framework”. In: *SIGCOMM ’07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*. URL: https://datatracker.ietf.org/doc/rfc3746/.