Jared VanEnkevort

John Sarkela

2 December 2021

Senior Project

To introduce my senior project, I'd like to start by discussing the motivations behind my choice of application type, a packet capture tool for macOS. Most of my practical Computer Science experience is in web development, and I've always had an interest regarding the inner workings of the internet, more specifically the underlying protocols and technologies that facilitate our modern world's primary mode of communication. This interest regarding the underlying working of computer networks is in contrast to the world of modern Web Development; in my experience, you're more concerned with how aesthetically pleasing the front end of the application is, as opposed to the underlying protocols that support the communication. To further bolster my interest in the subject, while brainstorming possible Senior Project topics, I recalled my enjoyment of a Network Programming class I had attended at Northern two year earlier. I thoroughly enjoyed the introduction to the basic theory and application of computer network programming. On the other hand, this recollection of my enjoyable experience in Network Programming further solidified my adverness towards developing a project akin to the web apps I create for work.

The greatest influence on my choice of targeting the macOS platform is relatively simple; I had acquired a Mac recently and gathered that developing an application on macOS using Apple developed libraries, languages & tools would result in an application that feels quite optimized & stable. On top of reaping the benefits of an application developed natively for macOS, I also decided to target Apple's latest ARM architecture base hardware. This decision

would be accompanied with numerous benefits; CPU usage and time would be significantly reduced when running on ARM architecture, battery usage significantly improves, and would differentiate it from the most popular packet analysis tool on macOS, Wireshark. Due to the vast amount of functionality Wireshark supports, it hasn't been ported to run as a native ARM application. While my application is a far cry (feature wise) from Wireshark, the core features I envisioned are as follows; it enables the user to select a network device, capture packets for a period of time, view various details regarding the packers and save them to file for later use (possibly including some extra features such as touch bar integration).

After deciding upon a packet capture tool built natively to run on Apple's ARM architecture, I began researching what tools will comprise the technology stack of my application. With my requirements in mind, I began looking for a framework or library that would allow me to utilize a network device to capture packets, the "backend" of the application. I spent a good amount of time sifting through Apple's developer documentation, looking for a macOS API that would allow me to get hold of a network device. I exclusively looked through Apple's Swift API for macOS, as I knew I was going to utilize SwiftUI for the application's UI. That would've allowed for a large amount of consistency throughout the codebase, but unfortunately it seems no such library exists on macOS. Consequently, I began looking into various C & C++ libraries as a remedy; due to macOS being a unix based operating system, libpcap seemed like a viable option. Due to concerns regarding time constraints, I opted to continue looking into more abstracted libraries, as I was worried the time and effort involved using a C library such as libpcap would prove to be too much complexity. This increased complexity was exacerbated by my decision to implement a gui, which led to the C++ library Pcapplusplus. Pcapplusplus conolidates libpcap and it's associated libraries into one API. This

library provided the lower level access I needed to network devices, while having a relatively object oriented API alleviating a fair amount of the development workload.

As previously mentioned, I chose SwiftUI as the GUI for the application, as it would facilitate the creation of a GUI within macOS with relative ease, while running performantly as a native application. Swift was one of the two languages involved in this project that I was completely inexperienced with, next to Objective C. While SwiftUI itself shared similarities to UI frameworks I had experience with such as Vuejs (such as data binding), it was the first declarative UI framework I had utilized since JavaFx. Its declarative nature expedited the learning experience in many ways, and hindered it in other ways. The Swift language itself was quite different from anything I had utilized before, including features from many other languages; it included the ‘var’ and ‘let’ syntax from javascript, it also included extremely concise syntax for iterating over collections, similar to python (i.e ‘for item in collection’, ‘for n in 1..5’) and the option character ‘?’ , allowing execution for a variable to be skipped if the value is nil. On the other hand,the force unwrap character ‘!’ forces swift to execute a value, implying the programmer believes the value should never be nil. It should be noted I could’ve used Apple’s legacy Objective C UI frameworks, but I opted not to as Swift is Apple’s successor to the former language, and is what Apple uses to build their future technologies upon.

As mentioned, Objective C was a language I utilized which was necessary due to the use of a C++ framework; as it would act as an intermediary between the Swift runtime and C++ runtime. On a side note, to my surprise Objective C shared a fair amount of syntactic attributes with the smalltalk language (most notably in use the of blocks for messages/selectors).I found Objective C to be significantly “lower level” when compared to Swift, as it involves memory management, pointers etc. Throughout the project, there is little plain Objective C in the project;

instead a variation called “Objective C++ was used”. These Objective C++ classes would act as wrapper classes, encapsulating the C++ framework classes. Mixing C++ source code was quite interesting and advantageous, but would prove to be accompanied by a few significant caevates. Moving on from languages and libraries, the only IDE I used to write & debug was Xcode. The choice was quite simple, as Xcode provided the most straightforward and well documented manner of developing a SwiftUI/ macOS application. I was quite pleased with how extensive Xcode’s capabilities are regarding performance profiling and application debugging. Most notably, Xcode made creating the Objective C++ bridging classes relatively painless.

Speaking of pain, my senior project wasn’t free of any trials and tribulations. The first task I tackled was a sort of proof of concept; I attempted to link and compile the Pcapplusplus library, so I could then test it in a standalone Xcode C++ project. This was easier said than done as I had never utilized a 3rd party C++ framework in such a manner. I first attempted to install the precompiled binaries using a package manager, but never advanced the project into a useful state. After failing to properly utilize the pre-compiled binaries, I downloaded Pcapplusplus and compiled it for ARM64 architecture. This proved to be the proper decision, as I was able to run and compile the sample provided with the framework. Unfortunately, ensuring all of the proper flags and various options set in the provided makefile were set in Xcode’s build settings were not as simple as I originally envisioned. These difficulties were only exacerbated by the two other languages runtimes within the project. For instance, in the compile settings, there are three different sections for options informing the compiler of how you want to compile Swift files, Objective C files, and plain C++. Much of my time was spent tinkering with various compile flags and options, slowly whittling away at the esoteric compiler errors.

Before I discuss the issues involved in getting the Objective C++ wrapper classes to work, I'd like to explain the general structure of the app. I envisioned the user opening up the application and being greeted by a list of network devices to select. To achieve this, the main SwiftUI view would use a wrapper class I created to retrieve a collection of network devices wrappers displaying them in a list. As a result, the "Main Wrapper" that is responsible for retrieving the network devices for the wrappers classes would need to include C++ header files. Naively, I assumed I could include the C++ header in the Objective C wrapper's header. Immediately after compiling, many nonsensical compile errors appeared, stating files such as "vector" and "iostream" didn't exist. As it turns out, when utilizing objective C++ bridging classes, one must keep their Objective C headers "clean"; What this means is that any usage of C++ source code must be kept in the wrapper class's implementation file, the .mm file. Note that the implementation file extension is .mm, not .m. This extension informs the Objective C compiler to expect a mix of Objective C source code and C++ source code within the implementation file. This is fine, as long as the header files themselves are kept free of any reference to C++ code; the header files of the wrapper classes is what you "expose" to the Swift classes, which are completely interoperable with Objective C, but not C++ whatsoever. As a result, much of the wrapper code became just a tiny bit trickier. For example, both the network device wrappers and the packet wrappers need to hold onto a pointer to the aforementioned Pcapplusplus classes. Due to the paradigm of "keeping the headers clean", the pointer to the wrapper's respective device need to be of type void *. Whenever I reference the device or packet pointer in the implementation file, I have to always cast it to its proper type.

On the subject of difficulties caused by mixing Objective C and C++, any piece of data retrieved from the pcapplusplus lib had to be transferred to an Objective C data type, which

Swift can “consume”. Whenever I retrieve a device name, description, mac address etc it would have to be transformed into an NSString, NSInteger, a respective Wrapper class etc. This issue first appeared when retrieving the network devices, where pcapplusplus would return a C++ vector of pcppdevices. To pass something SwiftUI can work with, I allocated a new Wrapper for each device in the vector and added it to an NSMutablearray, which I return in the appropriate method call. After successfully acquiring an instance of each device, I attempted to initiate a packet capture on a device; I immediately discovered macOS refused my application access to the device files corresponding to each network adapter. After a fair amount of troubleshooting, I noticed Xcode has a project “sandbox” setting. Among the security settings included in “sandbox” mode, by default Xcode doesn’t allow the app to read and write from network devices, a prudent security decision on Apple’s part. I simply removed the sandbox restrictions, which may cause issues with deploying and packaging the application, but fortunately resolved my device access issues.

Immediately after resolving the above issue, I began attempting to capture packets. As I didn’t want the main UI thread to halt when capturing packets, I utilized pcpp’s built in asynchronous capture callback function. This function was called within the pcpp device wrapper, where it allocates a new packet wrapper and adds it to the device wrapper’s packet array; this array is what the SwiftUI view uses to display each packet on the screen. Fascinatingly enough, the Objective C++ was completely fine with me simply adding the function to the Objective C class’s implementation file, with caveats. For starters, the asynchronous function requires you to pass a void pointer to it, which provides “context”. In my case, I ended up passing a pointer the “self ”(the PcapDeviceWrapper class calling the capture function). As I was simply passing a pointer to the function, I had to cast the pointer to a

pcapdevice wrapper using ” __bridge”. This needs to be used due to me recasting the void pointer, which results in a pointer to a pointer. Besides the issues of passing a pointer to the device instance, and issues with how Objective C passes pointers, if I didn’t dynamically allocate a new copy of each packet, I would intermittently receive memory access violation errors. To try and avoid memory leaks, I deallocated each dynamically allocated packet when the packets array was emptied. On the other hand loading and saving packets to and from file went quite smoothly, as the pcpp framework provided a very simple file reader and writer class. As an aside, I feel these issues wouldn’t have caused me nearly as much hardship if I wasn’t attempting to use 3 languages at once. On top of that, Objective C++ adds an extra bit of unexpected behavior.

Finally, the subtleties of Swift & my inexperience with the language imposed a significant roadblock on my project’s progress. Originally, I attempted to start the packet capture within the view that was intended to display them. As I would find out, when the view would load it would simply create a copy of the array retrieved from the device wrapper, not holding onto the NSMutable array by reference. Due to this the view would contain an empty NSMutablearray, as the array was empty when the view was created . I attempted to remedy this by creating a separate NSArray for the view to hold onto, but this only displaced the issue, as it was still creating the array by value and not reference. After a bit of investigation I discovered that NSObjects support Key Value Observing, where an observer can be notified of property changes within an object. After some mild success with an attempt to implement KVO compliant properties on my devicewrapper, I decided to implement a cruder method; at a specific time interval (say every second), I would simply reassign the Swift UI view’s packet array. To do so I called the packet wrapper’s getPacketArray() method, which returns a pointer to its packet array. This course of action led me into discovering another significant characteristic of Swift; structs

are of value type, a consequence of which is that a struct cannot mutate its own members as easily as a class. I experimented with the timer that reassigns the packet array every n seconds; unfortunately, no matter where I placed it within the SwiftUI view, it would incur various runtime or compile time issues. Most of these involved mutating the member of the struct within a closure, whether it be within a button's closure, or the function itself which is called every given interval. After a significant amount of deliberation, I elected to make a design compromise due to time constraints and other external factors beyond the scope of this project; I opted to prompt the user to start capturing packets, which is conveyed by an activity animation. Once the user stops the capture, they can view the packets within another Swift view. This allows the view to be passed a new copy of the packets array every time it is invoked,which will never mutate when that view is open. This solution wasn't exempt from issues, as I discovered when attempting to display packets within a VStack. Despite using a LazyVStack to contain the packet views, the performance was horrendous; upon opening the packet view after capturing packets whilst relatively high traffic activities were occurring (i.e streaming high quality video), the memory usage would peak to dozens of gigabytes. After a few mins of troubleshooting, I observed that the LazyVstack was encapsulating the *entire* container the ForEach loop sat within. As this ForEach loop is responsible for rendering the packet views, the LazyStack encapsulating its parent was problematic. Once I moved the LazyVStack within the container, it properly rendered only the packet views within the scrolling viewport, not all 20,000. Despite being a quick fix, the solution was less obvious then one would think, as I assumed the LazyVstack would handle conditionally rendering *all* child items contained within it.

Overall, I'd say I am satisfied with the resulting project; I created a macOS application that allows a user to select a network device, from which they can capture packets on. My

application allowed the aforementioned packets to be saved to file, where they can be opened for later use. Furthermore, the application and all its related processes run natively on Apple's ARM based processors. While the packet capturing process isn't as interactive as I envisioned, it is all contained within a fairly user-friendly UI, and more importantly a quite performant and efficient application. On Top of the previously mentioned feature,it's all facilitated by three different language runtimes running within the application. I also made extensive use of the Wrapper and bridge design patterns, encapsulating the C++ frameworks and various Objective C classes, facilitating communication with the SwiftUI components. If I was to rethink how I should approach this problem , I would spend a significantly greater amount of time trying to formally learn Objective C and Swift. Many of my hardships came from a lack of understanding the subtitles of the aforementioned languages, as my learning experience was more exploratory rather than formal; I simply discovered the features of the languages as they cropped up in my use cases. Unfortunately, due to the time constraints and the use of 3 languages within the project combined with external factors, I had a difficult time devoting time to properly learning the fundamentals of each language. Whether it be contending with the various subtitles of Swift and Objective C, resolving benign compile errors,resolving performance issues or implementing design patterns, I believe I accomplished much of which I set out to do and solved many of the various issues I came upon. This all culminated in a fairly beneficial learning experience, and a project I enjoyed developing.