Elvis: A Highly Scalable Virtual Internet Simulator

Tim Harding, Jacob Hollands, Mitchell Thompson, Logan Giddings, Robin Preble, See-Mong Tan
Western Washington University
hardint, hollanj9, thomp288, giddinl2, prebler, see-mong.tan@wwu.edu

***Abstract*—*Elvis* is a highly-scalable virtual Internet simulator that can simulate up to a hundred thousand networked machines communicating over TCP/IP on a single off-the-shelf desktop computer. This paper describes the construction of *Elvis* in Rust, a new memory-safe systems programming language, and the design patterns that enabled us to reach our scalability targets. Traffic in the simulation is generated from models based on user behavior research and profiling of large web servers. We also designed a Network Description Language (NDL) to describe large Internet simulations.**

***Index Terms*—Virtual Internet Simulator, Rust, Network Description Language**

I. Introduction

Elvis is a highly scalable simulation of a virtual Internet. It runs cross-platform on Linux, Mac, and Windows computers, and is shown to accommodate simulations of more than twenty thousand machines sending and receiving TCP and UDP traffic on a standard, off-the-shelf Linux workstation.

Elvis is intended as a research tool, both for developing new networking protocols and for evaluating those protocols in a large network. It is also intended as a pedagogical tool for students to explore networking scenarios that are infeasible to realize without simulation, such as Distributed Denial of Service (DDOS) attacks.

Elvis is written in Rust [1], a memory-safe systems programming language. We used *Elvis* to explore how Rust can simplify the development of systems software. We explore the software frameworks and design patterns required to construct very large scale simulations. Some key design choices are relying on language constructs for protocol stack isolation rather than OS-level virtualization, true zero copy of memory data throughout the simulation, as well as predicating concurrency on lightweight user-level coroutines rather than kernel threads. We also developed a Network Description Language that allowed us to specify very large simulations.

II. Related Work

The x-Kernel was an early approach to network simulation [8]. The x-Kernel architecture exemplified a highly-modular approach to implementing networking protocols in operating systems. Later work extended this into running the x-Kernel as a simulation. However, the goal was more focused on simulations to test protocol implementations rather than large scale simulations of an Internet.

Many commercial programs exist that allow users to construct simulated networks and learn how to configure network devices from different manufacturers. For example, the Cisco Packet Tracer provides a drag-and-drop interface to create networks of Cisco-specific devices [3]. Simulation sizes are typically small, on the order of tens of machines.

A common approach of researching internet simulators is to use OS mechanisms to isolate network stacks. For example, the Common Open Research Emulator (CORE) [4] uses FreeBSD OS support to virtualize the kernel network structures that run in separate virtualized jails. CORE reaches scales of a hundred virtual machines on one desktop computer.

A further example is the SEED Internet Emulator [5], a Python library that implements autonomous systems and routers, including protocols like BGP. The system is geared toward cybersecurity pedagogy and allows users to set up a virtual Internet that can be used to emulate scenarios like BGP poisoning or blockchain attacks. SEED is based on Docker [2] containers to provide isolation between machines in the simulation.

Elvis differs from previous work in that high scalability is a primary goal of the simulation. In one experiment wherein many machines send a single UDP message to a single receiver, we were able to scale up to more than 100,000 machines. In a separate experiment where all machines in the simulation continuously and concurrently send and receive data, we were able to scale up to 20,000 machines in a single simulation before memory limits were encountered. *Elvis* also provides ground-up, parallelizable constructions of key protocols including TCP/IP, DNS, and DHCP. rather than rely on existing OS implementations and virtualization technologies for isolation between machines. Instead, *Elvis* runs entirely in user space, vastly reducing both memory usage and overhead due to context switches. Concurrency is achieved with the Tokio [6] lightweight coroutine library.

III. Rust

Rust is a general purpose programming language that is focused on performance and memory-safety. Unlike languages like Java or C# that maintain memory-safety with a garbage collector, Rust maintains memory-safety by requiring that all references point to valid memory through the use of the compiler.

Memory in Rust is reclaimed when owning variables go out of scope. The potent result of this is that a Rust program that compiles has a much lower risk of leaking memory, and also guaranteed to never have a segmentation fault due to the invalid access of memory. Without a garbage collector, Rust is also as fast as C or C++ [7].

A. Rust and Memory Safety

The Rust borrow checker tracks the lifetimes of pointers, making sure that all references are valid and a heap allocation is not freed until all references are released.

Only one variable can "own" a pointer. Assigning the pointer to another variable transfers ownership of the pointer, and the pointer can no longer be accessed with the original variable. This is conceptually similar to C++'s unique_ptr smart pointer, but Rust enforces this strictly at compile time. Temporary references to a pointer may be passed to functions to use, but the compiler ensures that the lifetimes of those references never exceed the lifetime of the owning variable. When the owning variable goes out of scope, the memory that the pointer references is freed.

In situations where it is critical for multiple references to exist for shared data, Rust provides reference counted smart pointers Rc<T> (reference counted pointers of type T). For reference counted pointers that work across threads, Rust provides Arc<T> (atomically reference counted pointers of type T). *Elvis* makes liberal use of these language features.

B. *Unsafe Rust*

Programmers can circumvent Rust's memory-safety by marking a region of code "unsafe". This is usually needed when Rust calls into C-code, which is by nature unsafe. The onus is on the programmer to make sure that memory safety is preserved in unsafe Rust code.

Elvis makes no use of unsafe Rust. Memory-safety is maintained throughout the simulation.

C. *Concurrency*

Rust provides native support for concurrency through the thread:: module. In Unix, this is a wrapper on top of POSIX threads.

We chose instead to base our implementation on the Tokio [**?**] library. Tokio provides users with the ability for asynchronous programming using async/await functions. More important to *Elvis*, Tokio allows user level coroutines to be multiplexed on top of kernel threads. Coroutine resource usage is significantly lower than threads that require both a user-level stack as well as a kernel-level stack.

D. *Trade Offs*

The drawback to Rust is that the programming paradigm requires a much steeper learning curve. Programmers new to Rust often report "fighting the borrow checker". This situation improves with experience but is certainly a real impediment. In our experience, students new to the *Elvis* research group typically spend one academic quarter simply becoming comfortable with Rust.

IV. Architecture

A. *Elvis Core*

The two main constructs in *Elvis* are machines and networks. Machines model some device running an operating system, such as a computer or smartphone. Each machine is modeled after the x-Kernel design. A machine is a container for a set of protocol objects which interact with one another through an abstract interface. Some standard protocols that are included in most machines are IPv4, UDP, and TCP. User applications such as client and server programs are also modeled as protocols for uniformity. Each protocol can create sessions, which are objects that represent a particular network connection. Sessions are created either when an upstream protocol makes an active open to a remote host or when a packet comes in for which there is no existing connection and an upstream protocol is listening, as in the case of server programs. Sessions form a chain with each contributing protocol providing a link.

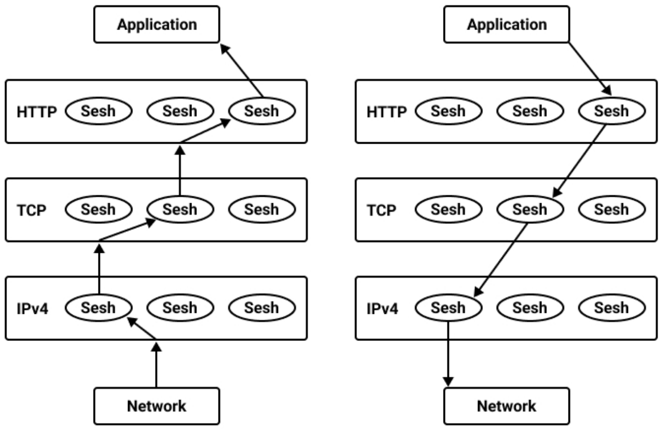


Fig. 1: Receive and send paths in x-Kernel protocol graph.

For example, a user application may hold a TCP session which in turn holds an IPv4 session. Sessions take charge of sending packets by appending headers and forwarding the packets to downstream sessions, while protocols receive incoming packets and decide which session to demux each packet to based on network headers.

Machines are connected to networks, which are the conduit through which packet traffic passes. Networks are designed to abstractly model a variety of real-world networking technologies, such as Ethernet, Wi-Fi, and point-to-point communication. To simulate a variety of underlying technology, networks can be configured with different throughput, latency, packet loss, and packet corruption characteristics. In this way, *Elvis* models networking down to the data link layer. In keeping with our focus on large-scale simulation, we choose to omit the details of any particular physical networking protocol for the sake of performance and uniformity. Instead, an *Elvis* network provides functionality that is common to most data link protocols, such as unicast, multicast, broadcast, and standard frame header information such as MAC addresses.

In order to make our networking as efficient as possible, our simulation uses a bespoke message data structure that allows addition and removal of headers, slicing, and sharing without copying or moving bytes. This helps us avoid serialization and deserialization of network traffic for efficiency gains. All of our protocols are written from scratch to take advantage of this data structure. For example, where most TCP implementations expect flat byte arrays for input and copy segment text into ring buffers, our implementation uses zero-copy concatenation of messages and accesses bytes through an iterator interface to circumvent the need for serialization.

With this model, we are able to isolate machines from one another without heavy-duty mechanisms such as containers. Instead, we use asynchronous functions to deliver packets over networks and avoid context switches. Because of Rust's guarantees, users need not worry that unsafe memory accesses will break isolation such that a malfunction in one machine can affect another.

B. Network Description Language

Previously, simulations in *Elvis* required a great deal of manual setup using Rust. The solution to this was a Network Description Language (NDL) to allow for not only easier setup, but also for the possibility of large scale simulations.

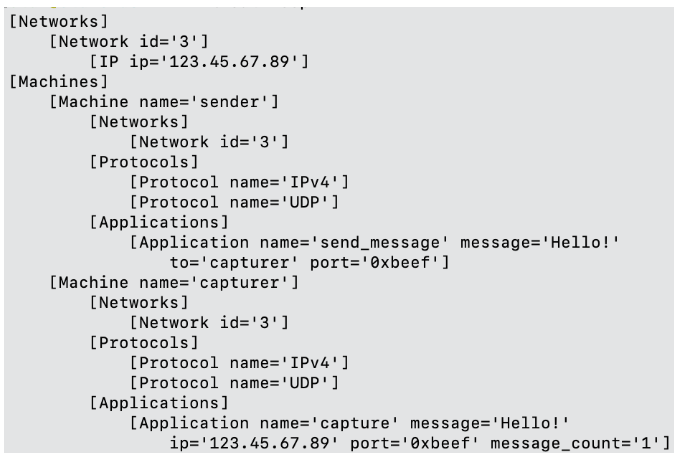
Using Rust to define these simulations would cause core problems for users for two main reasons. First, not all end users will be fluent in Rust, and due to the intricacies of how simulations must be defined this is a major limiting factor. A user would have to manually define each part of the simulation in Rust and be familiar with each part of *Elvis* in order to do so. Second, large scale simulations would not be impossible to define, but tremendously difficult and time consuming. Each machine, and application written out and defined in Rust would mean some of the larger scale simulations would need hundreds if not thousands of lines of code just to be run. Our NDL simplifies the process of programming by enabling the creation of easily definable and reproducible sections. This, in turn, facilitates the rapid development of large-scale simulations.

Typical NDLs use a variety of different languages to define their protocols. Some choices initially considered were XML and JSON, however given the idea of protocols being contained within parts of the simulations we decided to create a language using tabbed blocks. This allows nesting of sub protocols and definitions within other sections, and allows for repeatability of these sections.

To define a simulation, two core components are required: a set of Networks and a set of Machines. Within those there can be as many Network and Machine sections as needed. Each Network can currently contain either statically defined single IPs or a range of IPs. As many single IPs or range of IPs may be defined in the Network sections. Following that pattern, a Machine will contain a similar structure. Each Machine must contain Protocols, Applications, and Networks. These three components help clearly define the location and function that the Machine will serve. Protocols such as UDP or TCP may be used, a set of Applications such as sending or receiving messages may be used, and finally a set of Networks the machine is on must be defined. See Figure 2 for an example of the NDL.

Each of those sections are defined using a tabbed structure. A core declaration will be tabbed zero times, a sub declaration one time, and so on. For example, a Networks section will be at zero tabs, a Network defined in that section will be at one tab, and each IP definition for that Network will be tabbed twice.

Arguments for subsections can be defined freely. Other than the core needs of a specific application or protocol, such as the name of the protocol or the IP range of a network, users can define any such argument needed and it will be read in. This argument then gets stored with the rest of the arguments, core or otherwise, and can be accessed in the generator code. Users have no extra steps in defining new applications or protocols for use other than adding checks for those new applications or arguments, then accessing and using them.



```
[Networks]
    [Network id='3']
        [IP ip='123.45.67.89']
[Machines]
    [Machine name='sender']
        [Networks]
            [Network id='3']
        [Protocols]
            [Protocol name='IPv4']
            [Protocol name='UDP']
        [Applications]
            [Application name='send_message' message='Hello!'
                to='capturer' port='0xbeef']
    [Machine name='capturer']
        [Networks]
            [Network id='3']
        [Protocols]
            [Protocol name='IPv4']
            [Protocol name='UDP']
        [Applications]
            [Application name='capture' message='Hello!'
                ip='123.45.67.89' port='0xbeef' message_count='1']
```

Fig. 2: Basic example simulation.

Putting all of those sections together results in a complete language for defining in depth simulations for *Elvis*.

C. Applications and the Socket API

One of our goals in designing *Elvis* was retaining the ability to easily port existing applications into the simulation. In order to achieve this, we needed a socket API that closely mimicked UNIX sockets, with functionality of creating sockets, using sockets to connect to a server machine or listen and accept incoming client connections, as well as sending and receiving messages over the network.

A challenge in doing this is the fact that the machines in *Elvis* operate using a x-Kernel style protocol stack, which is incompatible with writing UNIX style applications. The *Elvis* socket API serves as an interface between the two. A second drawback is that UNIX style sockets have undesirable aspects to their implementation, such as pointer casting. *Elvis* presents a modernized implementation while retaining all the functionality needed.

The resulting socket API is familiar and easy to use for anyone who has experience writing server-client applications using UNIX sockets, with the only noticeable differences being syntactical ones and the fact that applications utilizing it are written in Rust instead of C. This allows for convenient conversion and porting of existing UNIX server-client applications into *Elvis* applications.

Note that it is not the goal of *Elvis* to support full OS-level functionality in the simulation. *Elvis* applications are limited to using the *Elvis* Socket API to create network connections, and send/receive data.

D. Web Traffic

In order to create realistic simulations of the Internet within *Elvis*, we require realistic traffic. One primary category of traffic is web traffic between clients and servers. To do this, we first have to characterize servers on the internet.

That means gathering data from top servers on the size, number of links, number of images, and the size of images for each page on a web server. The distribution of that data will then be used to model the distribution of web pages on that type of website. Our simulated web servers will generate html pages for "users" to browse based on the distribution of size, number of links, and images sizes for that category of website.

We started by creating a web scraper in Rust that recursively traverses a website and outputs the links and images on each page as well as the size. We chose yahoo.com to test this scraper since it is a large site and most of the links on Yahoo lead to other pages on the site. We scraped over 200,000 pages.

When analyzed, we found that for each attribute measured, the majority of pages tended to fall within a narrow range of values with the rest being fairly scattered without a clear distribution. This made it difficult to find a simple mathematical model of the distribution. Instead, we created a program that went through the data for each page characteristic (size, number of links, etc.) and sorted it into buckets while keeping track of how many pages fell into each bucket. This information was saved in a .csv, which can then be passed into the web server program to inform the characteristics of the html pages it generates so the web server can mimic the servers we gathered data from.

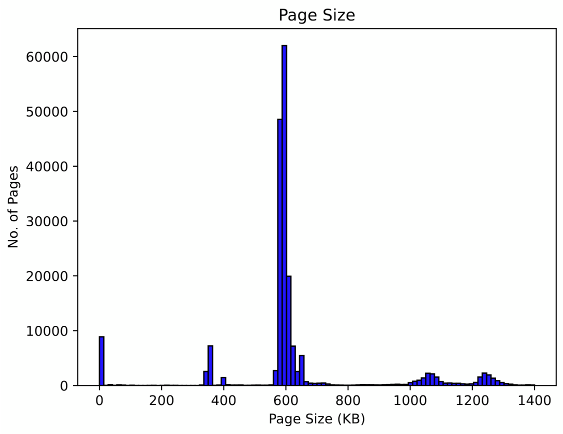


Fig. 3: Distribution of web page sizes on Yahoo.com.

In further work, we developed an application that mimics how users typically behave when browsing web pages.

V. Experimental Results

A. Scalability

To test *Elvis*' scalability we ran a variety of simulations on two core simulation types. The two types used were a low bandwidth high machine count simulation, and a high bandwidth lower machine count simulation. The low bandwidth focuses on a higher machine count, but does not keep the machines running concurrently, meaning the machines send messages one at a time to keep overall system load low and simulates an environment where users could be connecting to a server and then disconnecting when they are done. The high bandwidth focuses on keeping the machines concurrent, leading to a lower overall machine count. This means that the machines are all trying to send their 1,000 messages at once to the server. This better simulates a massive load on *Elvis*.

To accomplish these simulations, a robust testing system was needed. We designed a Bash script and Python-based system that runs various simulations and tracks memory usage, CPU usage, and execution times of the simulations. This data is then compiled into JSON for storage which is then used to generate graphs automatically using Python's MatPlotLib. Additionally, all the following tests were run on an Intel Core i5-8279U CPU, with 16 GB of RAM.

1) Low Bandwidth Simulations: The first set of simulations ran was to test *Elvis* capabilities with lots of machines all running at once. To do this the simulations were designed to generate a set amount of machines all sending a message to a single machine. That machine is then configured to receive that same amount of messages as the number of machines created. This was tested on machine counts ranging from one thousand to one million.

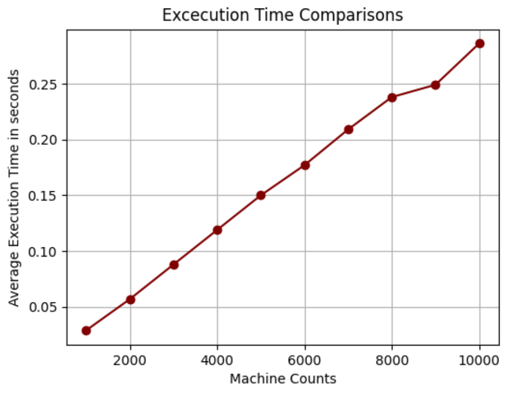


Fig. 4: Execution times of low bandwidth simulation with machine counts from 1,000 to 10,000.

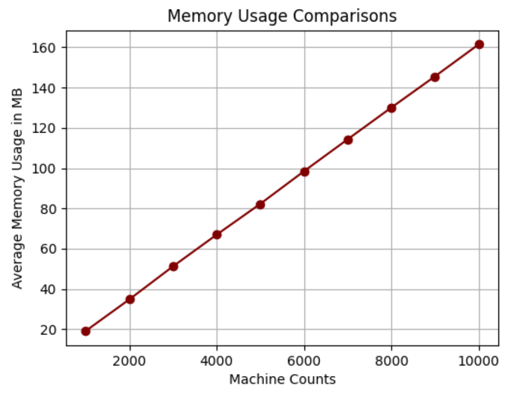


Fig. 5: Memory usage of low bandwidth simulation with machine counts from 1,000 to 10,000.

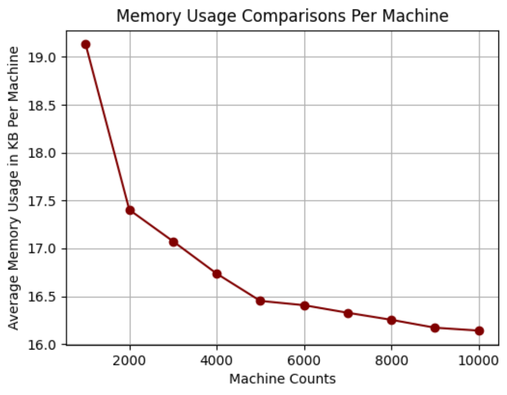


Fig. 6: Memory usage per machine of low bandwidth simulation with machine counts from 1,000 to 10,000.

As seen in Figures 4 through 6, low bandwidth simulations with 1,000 to 10,000 machines start to develop a distinct pattern. As machine count increases overall execution time and memory usage increases linearly. Memory usage per machine in the simulation decreases as the simulation's overhead is amortized over more machines. The average memory usage per simulated machine is slightly above 16 KB.

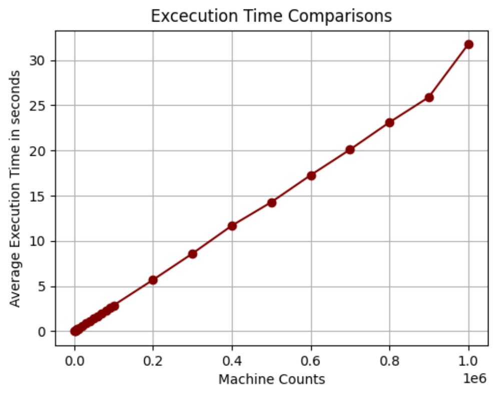


Fig. 7: Execution times of low bandwidth simulation with machine counts from 1,000 to 1,000,000.

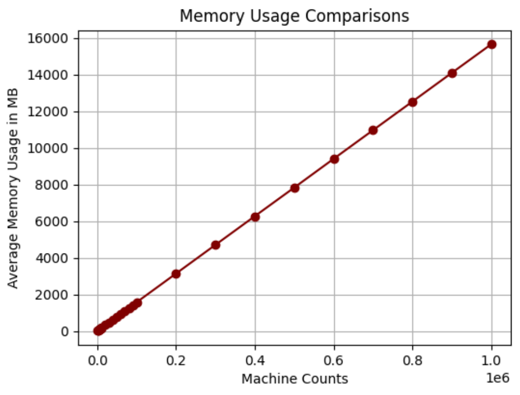


Fig. 8: Memory usage of low bandwidth simulation with machine counts from 1,000 to 1,000,000.

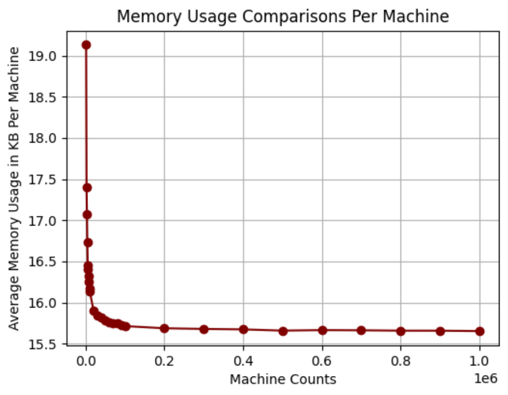


Fig. 9: Memory usage per machine of low bandwidth simulation with machine counts from 1,000 to 1,000,000.

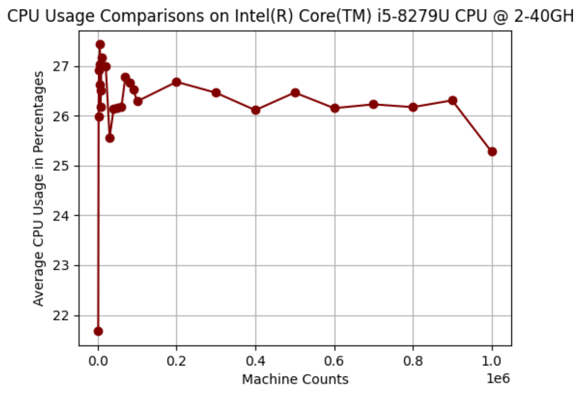


Fig. 10: CPU usage of low bandwidth simulation with machine counts from 1,000 to 1,000,000.

In Figures 7 through 9 the previously noticed patterns become more prevalent. Once again, as machine count increases, overall execution time and memory usage increase linearly and memory usage per machine in the simulation decreases due to amortization of the core invariant simulation resources. One interesting part to notice is that machine specific memory usage plateaus as we approach the system's physical memory limit.

2) High Bandwidth Simulations: The next set of simulations ran was to test *Elvis* capabilities with lots of machines all running at once; however, each machine now sends 1000 messages. This means that the total count of messages sent is machine count multiplied by 1000 and it also means that each machine lives for longer within the simulation.. The goal with this was to generate machines in such a way that we could find the limits for concurrent full usage of *Elvis* rather than just the single message per machine case.

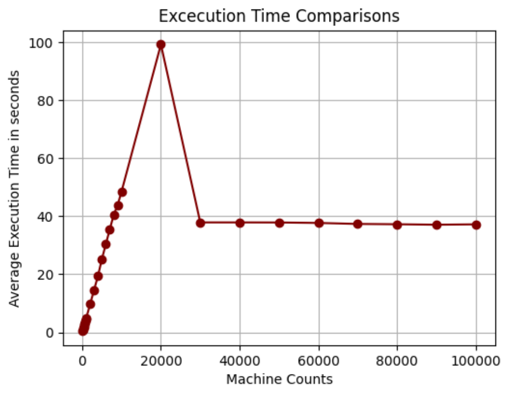


Fig. 11: Execution time of high bandwidth simulation with machine counts from 1,000 to 100,000

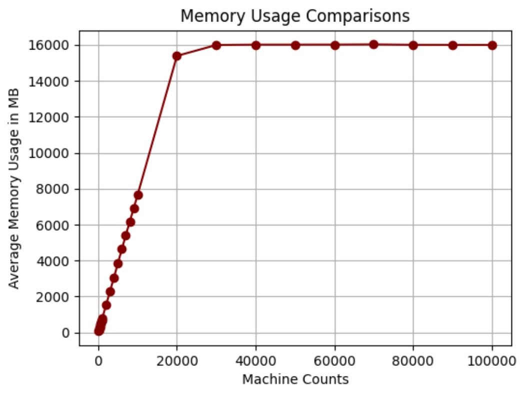


Fig. 12: Memory usage of high bandwidth simulation with machine counts from 1,000 to 100,000

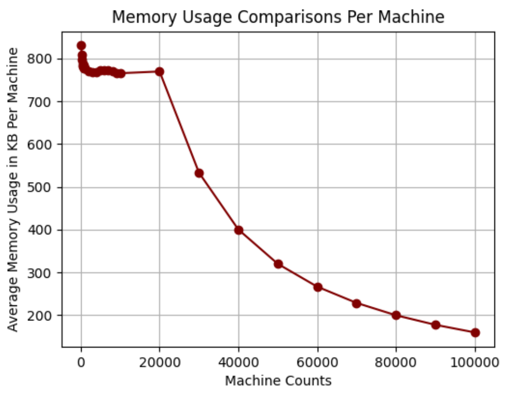


Fig. 13: Memory Usage per machine of high bandwidth simulation with machine counts from 1,000 to 100,000

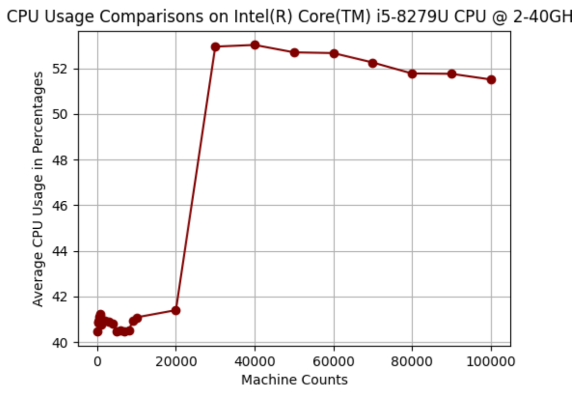


Fig. 14: CPU usage of high bandwidth simulation with machine counts from 1,000 to 100,000

In Figures 11 through 13 the pattern differs from the low bandwidth tests in an interesting way. From 1,000 to 20,000 machines, it follows a similar linear pattern of growth. However, passing the 20,000 machine mark our system reaches the full usage of the 16 GB of RAM. At that point *Elvis* requires more physical memory than is available, and we see the effects of continuous paging as the system enters a thrashing state. Execution time becomes more static as the system can only handle so much at once, along with memory usage which stays pegged at 16 GB for the remaining simulations. Interestingly, memory usage per machine starts to decrease rapidly at this point, as they can not use more than the 16 GB available, and the CPU begins paging. This is why these simulations were only scaled up to 100,000 machines rather than 1,000,000. We believe that with more available memory this simulation could easily be run, but may require upwards of 32 GB of memory.

Another factor to consider is the CPU usage of the simulations. In figure 11 it shows that as we hit that memory limit of 16 GB, CPU usage skyrockets. This is due to the resulting thrashing that occurs. Aside from that anomaly, it is important to note that the average CPU usage before the spike sits around forty to forty-two percent for the high-bandwidth simulations. This does not grow linearly alongside the machine counts, but rather grows more in line with the bandwidth the simulation uses. From the low-bandwidth to high-bandwidth simulations for similar machine counts the average usage grows from the 25 percent to 42 percent usage found in the high-bandwidth versions.

B. TCP Performance

On our test machine, we are able to transfer data over TCP at a rate of 1GB per 2.6s on a single thread, roughly thrice as fast as an ideal gigabit Internet connection. We believe that there is significant room for improvement in performance for several reasons. First, we note that enabling multithreading currently decreases throughput by 33%. This suggests that our current approach to parallelism causes unnecessary contention and leaves a great deal of performance on the table. Additionally, profiling shows that we spend only 30% of CPU time in *Elvis* code, with the other 70% being shared between the Tokio async runtime and system calls. For future work, we intend to invest effort into refactoring the core simulation to spend more time in *Elvis*code and make better use of multiple CPUs. At the time of publication, we have an initial implementation of a custom task system that gives 170% the throughput of the Tokio system described in this paper. This demonstrates the potential for substantial gains in TCP throughput.

C. Socket API Performance

We tested the performance of simulations that use our Socket API against the performance of simulations that do not in order to see how heavily the usage of sockets affects performance. The runtimes of several simulations are shown in Table I.

TABLE I: Socket Performance

	Basic	**Ping Pong**	**Server Client**
No Sockets	0.063 ms	1.55 ms	4.15 ms
Sockets	0.094 ms	2.30 ms	5.00 ms

Usage of sockets is expected to slow down simulations since there are several blocking functions in our implementation, with accept() and recv() as the most notable. These functions block when waiting for an incoming connection or when waiting for an incoming message, respectively. The runtimes in the above table indicate that usage of our socket API can cause as much as a 50% increase in runtime for simple simulations like Basic and Ping Pong, and as much as a 20% increase for more complex simulations like Server Client. These metrics indicate that performance is not optimal. Work is in progress to reduce *Elvis*'s Socket overhead.

VI. Limitations of the Approach

Our original goal was 50,000 nodes in the simulation. While the low bandwidth test demonstrated that double that amount could be achieved, the more realistic high bandwidth test showded that memory limits were reached on a 16G workstation with 20,000 nodes. The experiments nevertheless demonstrate the feasibility of large scale simulations with memory safe language constructs providing isolation between nodes.

While memory safety is enforced, our approach has currently no way to divide resources between nodes that the more heavyweight, container-based approaches are able to. For example, we are currently unable to assign more priority to one node to the next, or indeed for one task to the next. Since scalability is predicated on green threaded cooperative coroutines, it is entirely possible that one task may run for long periods, depriving other tasks of execution time. We are currently investigating the use of a custom coroutine runtime that may address these issues, as well as that seen in TCP and Socket performance.

VII. Summary

Elvis was constructed to explore issues in building a highly scalable Internet simulation. Our exploration drew on x-Kernel [8] for the modularization of protocols. We extended the idea of zero copy of network messages in protocol processing to the entire simulation – once a network message is constructed, the message data is truly zero copy from the sending application all the way to the receiving application.

Rather than virtualize the network stack using OS-level features, we constructed the simulation so that the protocol stack does not assume it is the only stack. Each simulated Machine owns its own protocol stack. Rust memory-safety means that protocol processing in one machine cannot corrupt the memory space of other machines, effectively giving us isolation through language level constructs.

Concurrency was achieved through the use of Tokio, a lightweight coroutine library that multiplexes user level green threads on top of kernel threads. An *Elvis* simulation with a hundred thousand machines will have at least several hundred thousand coroutines active.

Constructing a large simulation is impossible without a Network Description Language (NDL). The *Elvis* NDL allows the specification of very large simulations containing multiple machines and the applications running on them.

Our experimental results indicate that it is feasible to run a basic simulation of a hundred thousand clients sending single UDP messages to a receiver on a single off-the-shelf desktop computer without thrashing. A more representative simulation where clients continuously and concurrently send data showed that we could scale up to 20,000 client machines before memory limits were encountered. The results also showed that *Elvis* is memory-constrained more than CPU-constrained with present hardware technology, e.g. the high bandwidth tests demonstrated that the 16 GB memory limit was reached while CPU usage was still at approximately 45%.

Our experience of implementing in Rust was that the learning curve was steep in the beginning when starting on *Elvis* for motivated graduate and undergraduate students both. One academic quarter of preparation and experience was necessary before students were ready to contribute productively on the codebase.

VIII. Future Work

Near term future work in *Elvis* includes fleshing out the protocol stack with DNS, DHCP and ICMP, and experimenting with web servers that mimic the behavior of large web servers like Yahoo and browser client simulations that mimic the behavior of users. We are also keen to explore video streaming on our simulations of the Internet. The socket API bindings we developed should permit the easy porting of networking applications to *Elvis*, and we look forward to enriching the simulation with a larger variety of applications.

IX. Acknowledgements

Our thanks to Min Ngyuen and Kelly McElroy for their early contributions to *Elvis*. Western Washington University and Dr. Filip Jagodzinski graciously donated lab space. Dr. Erik Fretheim and Vipul Kumar provided the initial inspiration for *Elvis*. Dr. Aran Clauson and Dr. Phil Nelson provided invaluable comments.

References

[1] Klabnick, Steve and Nichols, Carol. 202. The Rust Programming Language. No Starch Press

[2] Merkel, D. 2014. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.

[3] Frezzo, D., Wang, M., Chen, M., Anderson, B., Hou, J., Hoang, T., Deng, T., Vinco, P., Nguyen, T., Le, T. and Ghoghari, N., 2010. Cisco Packet Tracer. Cisco Networking Academy.

[4] Ahrenholz, Jeff., Henderson, Thomas., Kim, Jae H. November 2008. CORE: Real time network emulator. In MILCOM 2008 - IEEE Military Communications Conference 2008.

[5] Du, Wenliang., Zeng, Honghao. Jan 2022. The SEED Internet Emulator and Its Applications in Cybersecurity Education. arXiv preprint arXiv:2201.03135, 2022.

[6] Lerche, C. ”Announcing Tokio 1.0”. Retrieved December 11, 2022. https://tokio.rs/blog/2020-12-tokio-1-0.

[7] Ivanov, N. ”Is Rust C++-fast? Benchmarking System Languages on Everyday Routines”. arXiv preprint arXiv:2209.09127, 2022 - arxiv.org.

[8] Druschel, P., Abbott, M. B., Pagels, M., and Peterson, L. L. Network subsystem design. IEEE Network (Special Issue on End-System Support for High Speed Networks), 7(4):8-17, July 1993.

[9] Nielson, Jakob. ”How Long Do Users Stay on Web Pages?” Nielson Norman Group, Jakob Nielson, September 11, 2011. https://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/.

X. Author Information

Tim Harding, Logan Giddings and Robin Preble are undergraduates at Western Washington University. Jacob Hollands and Mitchell Thompson are Masters candidates at Western Washington University. See-Mong Tan is an instructor at Western Washington University.