Journal of Electronics and Informatics (ISSN: 2582-3825)
www.irojournals.com/iroei/

Elvis: A Highly Scalable Virtual Internet Simulator

Dheeraj Kumar Boddu

Department of Computer Science, University of Maryland, Baltimore County, USA

E-mail: dc32292@umbc.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 research describes the construction of Elvis in Rust, a new memory-safe systems programming language, and the design patterns that enabled us to reach scalability targets. Traffic in the simulation is generated from models based on user behavior research and profiling of large web servers. Additionally, a Network Description Language (NDL) was designed to describe large Internet simulations.

Keywords: Network Simulation, Rust Programming, Large-Scale Internet Simulation, TCP/IP, Scalable Networking.

1. 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. Elvis was used to explore how Rust can simplify the development of systems software. The software

DOI: https://doi.org/10.36548/jei.2024.4.006
Received: 26.12.2024, received in revised form: 22.01.2025, accepted: 01.02.2025, published: 13.02.2025

frameworks and design patterns required to construct very large-scale simulations are also explored. Some key design choices rely 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. A Network Description Language was also developed to specify very large simulations.

2. 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 the 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, in the order of tens of machines.

A common approach to researching internet simulators is to use OS mechanisms to isolate network stacks. For example, the Common Open Research Emulator (CORE)[4] uses Free BSD OS support to virtualize the kernel network structures that run on 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, scaling was achieved up to more than 100,000 machines. In a separate experiment where all machines in the simulation continuously and concurrently send and receive data, scaling reached 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 [9-11].

3. 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 goes out of scope. The potent result of this is that a Rust program that compiles has a much lower risk of leaking memory and is 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].

3.1 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 per 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 pro- vides reference-counted smart pointers R<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.

3.2 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.

3.3 Concurrency

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

Instead, the implementation was based on the Tokio [6] 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.

3.4 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. Based on experience, students new to the Elvis research group typically spend one academic quarter simply becoming comfortable with Rust.

4. Architecture

4.1 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 one 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.

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 demo each packet to based on network headers as shown in Figure.1.

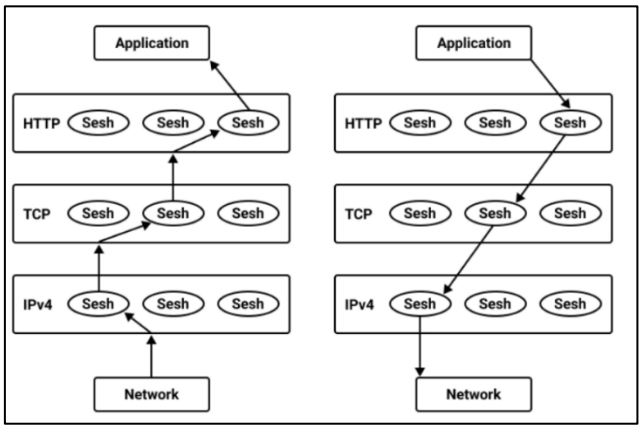

Figure 1. Receive and Send Paths in x-Kernel Protocol Graph.

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 technologies, networks can be configured with different throughput, latency, packet loss, and packet corruption characteristics. In this way, Elvis models are networking down to the data link layer. In keeping with the focus on large-scale simulation, the details of any particular physical networking protocol were omitted 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 networking as efficient as possible, these simulation uses a bespoke message data structure that allows the 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 these 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, this implementation uses zero-copy concatenation of messages and accesses bytes through an iterator interface to circumvent the need for serialization.

With this model, machines are isolated from one another without heavy-duty mechanisms such as containers. Instead, asynchronous functions were used 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.

4.2 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 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 to do so. Second, large-scale simulations would not be possible 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. 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, a language was created 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 a range of IPs may be defined in the Network sections. Following that pattern, a machine will have 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 is defined using a tabbed structure. A core declaration will be tabbed zero times, a sub declaration will be tabbed 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.

```
[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']
```

Figure 2. Basic Example Simulation.

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. 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 and then accessing and using them. Putting all of those sections together results in a complete language for defining in-depth simulations for Elvis.

4.3 Socket API

One of the goals in designing Elvis was retaining the ability to easily port existing applications into the simulation. In order to achieve this, In order to achieve this, a socket API

was needed which closely mimicked UNIX sockets, with the 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 an 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.

4.4 Web Traffic

In order to create realistic simulations of the Internet within Elvis, realistic traffic was required. One primary category of traffic is web traffic between clients and servers. To do this, first step is 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 was then used to model the distribution of web pages on that type of website. 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.

A web scraper was created in Rust that recursively traverses a website and outputs the links and images on each page as well as the size. yahoo.com was chosen for testing this scraper since it is a large site and most of the links on Yahoo lead to other pages on the site. Over 200,000 pages were scraped.

When analyzed, each attribute measured revealed specific trends and patterns, the majority of pages tended to fall within a narrow range of values, with the rest fairly scattered and without a clear distribution. This made it difficult to find a simple mathematical model of the distribution. Instead, a program was created 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 from which data was gathered. Figure 3 depicts the distribution of webpage sizes on yahoo.com.

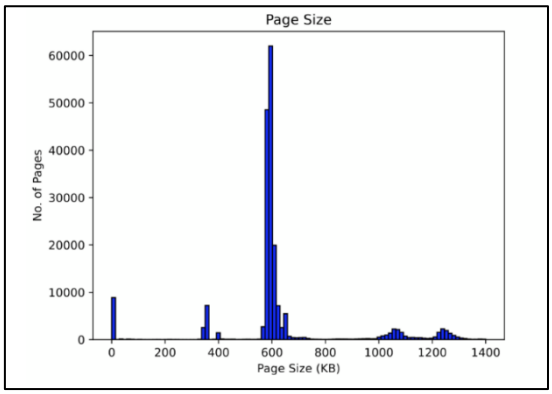

Figure 3. Distribution of Web Page Sizes on Yahoo.com

Further work led to the development of an application that mimics how users typically be- have when browsing web pages.

5. Experimental Results

5.1 Scalability

To test Elvis' scalability, a variety of simulations were run 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 the overall system load low and simulate 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.

A robust testing system was needed to accomplish these simulations. A Bash script and Python-based system were designed to run 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.

Low Bandwidth Simulations: The first set of simulations ran 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 the same amount of messages as the number of machines created. This was tested on machine counts ranging from one thousand to one million.

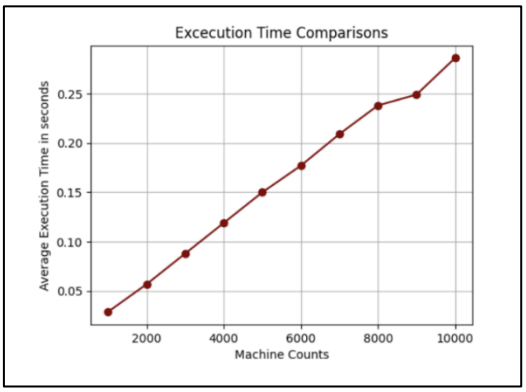

Figure 4. Execution Times of Low Bandwidth Simulation with Machine Counts from 1,000 to 10,000.

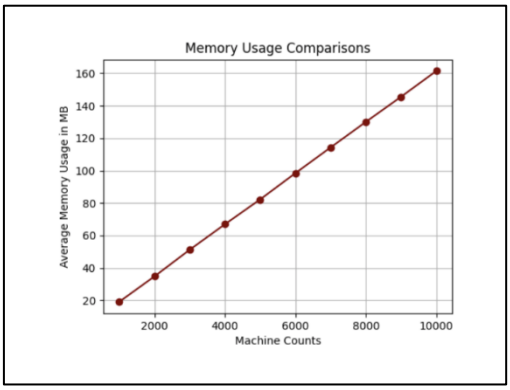

Figure 5. Memory Usage of low Bandwidth Simulation with Machine Counts from 1,000 to 10,000.

As seen in Figures 4 through 5, 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.

High-Bandwidth Simulations: The next set of simulations ran to test Elvis's 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

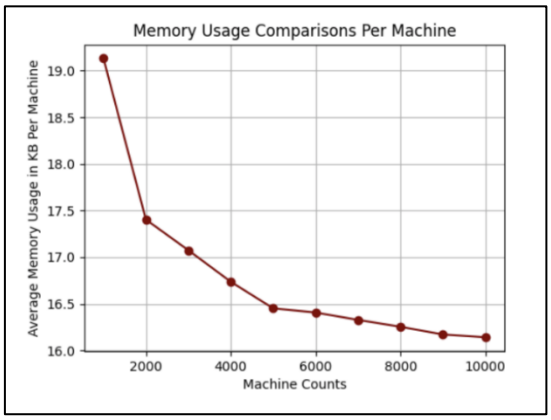

Figure 6. Memory Usage Per Machine of High Bandwidth Simulation with Machine Counts from 1,000 to 10,000.

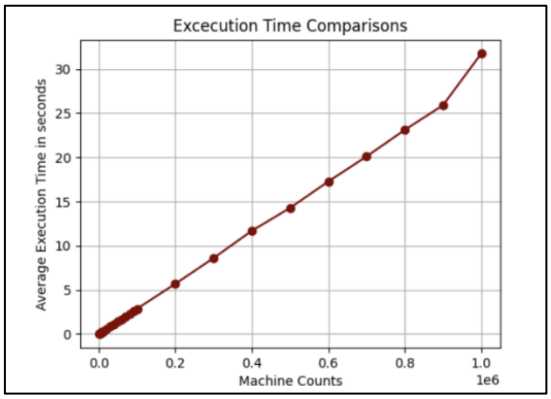

Figure 7. Execution times of High Bandwidth Simulation with Machine Counts from 1,000 to 1,000,000.

generate machines in such a way that for concurrent full usage of Elvis could be identified, rather than just the single message per machine case. Figures 6 and 7 depicts the memory usage and the execution times per machine of high bandwidth.

In Figures 8 and 14 the pattern differs from the low bandwidth tests. From 1,000 to 20,000 machines, it follows a similar linear pattern of growth. However, passing the 20,000-machine mark, the system has reached the full usage of 16 GB of RAM. At that point Elvis requires more physical memory than is available, resulting in 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 cannot 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. It is believed that with more available memory, this simulation could easily be run, but it may require upwards of 32 GB of memory.

Another factor to consider is the CPU usage of the simulations. Figure 8 shows that as it reaches that memory limit of 16 GB, CPU usage skyrockets. This is due to the resulting thrashing.

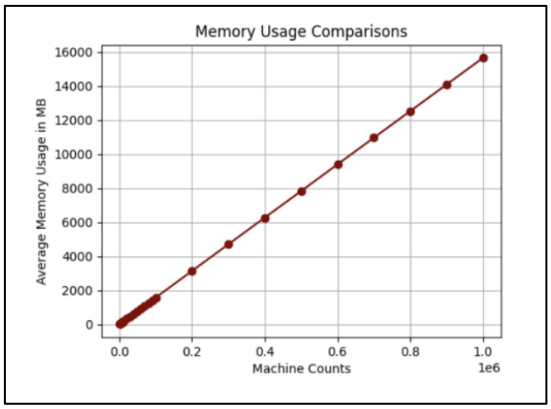

Figure 8. Memory Usage of Low Bandwidth Simulation with Machine Counts from 1,000 to 1,000,000.

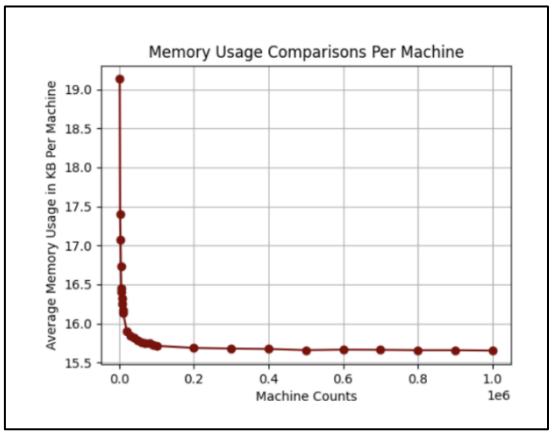

Figure 9. Memory Usage Per Machine of High Bandwidth Simulation with Machine Counts from 1,000 to 1,000,000.

Aside from that anomaly, it is important to note that the average CPU usage before the spike settles 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 25 percent to 42 percent, as found in the high-bandwidth versions.

5.2 TCP Performance

On the test machine, data transfer over TCP achieves a rate of 1GB per 2.6 seconds on a single thread, approximately three times faster than an ideal gigabit Internet connection. Significant potential for performance improvement remains. Enabling multi-threading currently reduces throughput by 33%, indicating that the existing parallelism approach introduces unnecessary contention and leaves a great deal of performance on the table. Profiling results show that only 30% of CPU time spent in Elvis code, with the other 70% being shared between the Tokio async runtime and system calls. For future work, Future efforts will focus on refactoring the

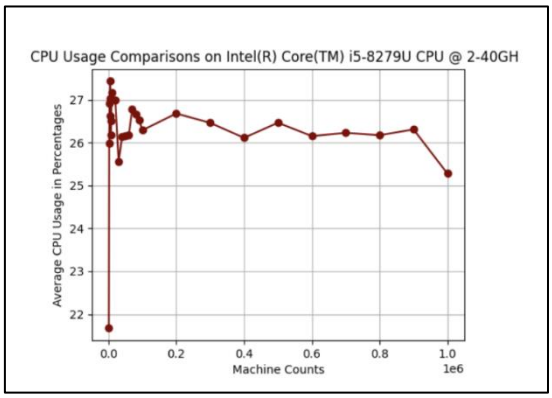

Figure 10. CPU usage of Low Bandwidth Simulation with Machine Counts from 1,000 to 1,000,000.

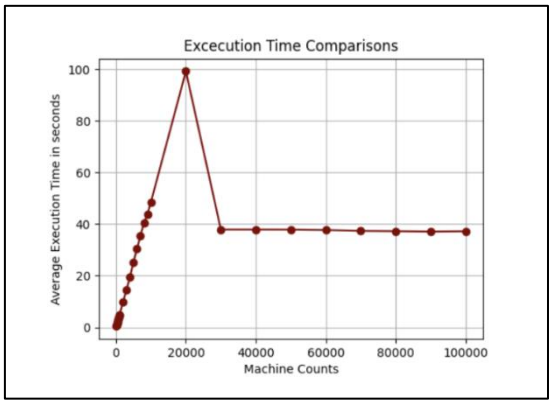

Figure 11. Execution time of High Bandwidth Simulation with Machine Counts from 1,000 to 100,000

core simulation to maximize CPU utilization within Elvis and improve multi-core efficiency. At the time of publication, An initial implementation of a custom task system already achieves 170% of the throughput of the Tokio-based approach described in this study. This demonstrates the potential for substantial gains in TCP throughput.

5.3 Socket API Performance

The performance impact of socket usage was evaluated by comparing simulations that utilize the Socket API with those that do not. The runtimes of several simulations are shown in Table 1.

Table 1. Socket Performance

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

The usage of sockets is expected to slow down simulations since there are several blocking functions in the implementation, with accept() and rice() as the most notable. These functions

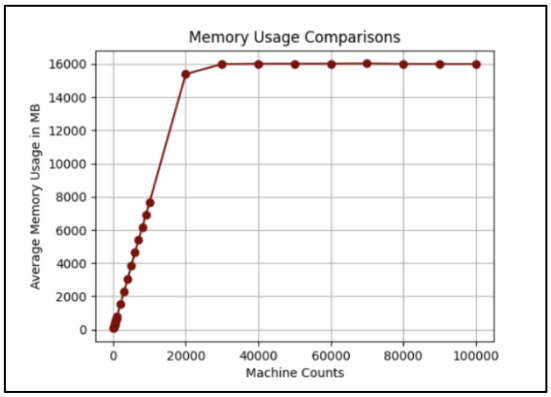

Figure 12. Memory usage of Low Bandwidth Simulation with Machine Counts from 1,000 to 100,000

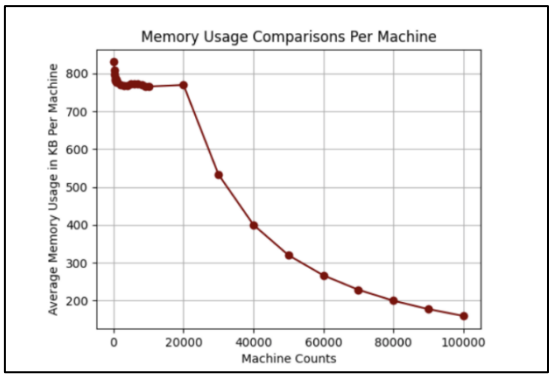

Figure 13. Memory Usage Per Machine of High Bandwidth Simulation With Machine Counts from 1,000 to 100,000

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 the 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.

6. Limitations of the Approach

The 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 showed 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 that provide isolation between nodes.

While memory safety is enforced, the current approach lacks a mechanism to divide resources between nodes, a capability present in more heavyweight, container-based solutions. For example, assigning higher priority to specific nodes or tasks is not yet possible. Since scalability is predicated on green-threaded cooperative co-routines, it is entirely possible that one task may run for long periods, depriving other tasks of execution time.

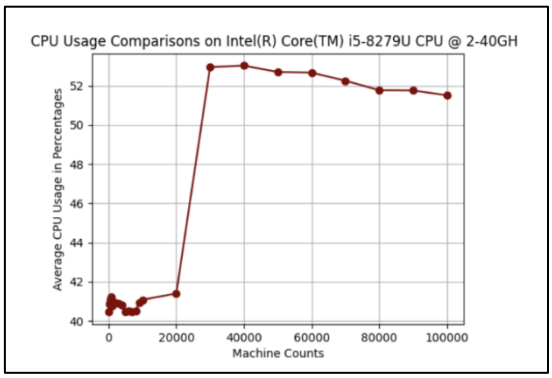

Figure 14. CPU usage of High Bandwidth Simulation with Machine Counts from 1,000 to 100,000

Ongoing research is focused on developing a custom co-routine runtime to address these issues, as well as challenges related to TCP and socket performance.

7. Conclusion

Elvis is a highly scalable virtual Internet simulator developed in Rust, designed to simulate large-scale network environments on commodity hardware. This study has demonstrated that Elvis can support up to 100,000 simulated machines communicating over

TCP/IP while maintaining efficiency through lightweight concurrency, zero-copy message handling, and memory- safe design principles. Unlike traditional simulation approaches that rely on OS-level virtualization, Elvis utilizes Rust's built-in safety guarantees and Tokio's coroutine-based concurrency model to maximize performance and scalability. The introduction of the Network Description Language (NDL) further enhances usability by allowing researchers and educators to specify complex network topologies with ease. Experimental results highlight Elvis's ability to handle both high- and low-bandwidth simulations, revealing that memory, rather than CPU, is the primary constraint in large-scale simulations. While the system can process high volumes of UDP and TCP traffic, additional optimizations, such as improved parallelism and resource allocation strategies, are necessary to enhance performance further the study also identified challenges in integrating a socket API that balances ease of use with execution efficiency, as similar trade-offs have been observed in other networking emulators such as CORE and SEED. These insights open new directions for future work, including extending protocol support to DNS, DHCP, ICMP, and web-based simulations that emulate real-world Internet traffic patterns. Future enhancements to Elvis will focus on increasing simulation fidelity, incorporating machine learning-based network behavior modeling, and expanding support for distributed execution across multiple physical machines. Previous research in network simulation scalability, such as the ns-3 simulator, has shown that efficient resource management is essential for performance, which will be a key area of improvement in Elvis. Furthermore, integrating Elvis with existing cybersecurity and network analysis frameworks could significantly benefit researchers studying network resilience, distributed denial- of-service (DDoS) attacks, and large-scale data transmission protocols. Overall, Elvis provides a powerful foundation for scalable network research, enhancing innovation in Internet simulation, protocol development, and network education.

References

[1] Steve Kalanick and Carol Nichols. The Rust Programming Language. No Starch Press, 2020.

[2] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: Linux Journal 2014.239 (2014), 2.

[3] Tracer, Cisco Packet. "Cisco Packet Tracer." URL: http://www. cisco. com/web/learning/netacad/coursecatalog/PacketTracer. html (2013).

[4] Ahrenholz, Jeff, Claudiu Danilov, Thomas R. Henderson, and Jae H. Kim. "CORE: A real-time network emulator." In MILCOM 2008-2008 IEEE Military Communications Conference, IEEE, 2008. 1-7.

[5] Du, Wenliang, and Honghao Zeng. "The SEED internet emulator and its applications in cybersecurity education." arXiv preprint arXiv:2201.03135 (2022).

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

[7] Ivanov, Nikolay. "Is rust c++-fast? benchmarking system languages on everyday routines." arXiv preprint arXiv:2209.09127 (2022).

[8] Druschel, Peter, Mark B. Abbott, Michael A. Pagels, and Larry L. Peterson. "Network subsystem design." IEEE network 7, no. 4 (1993): 8-17.

[9] Jakob Nielsen. How Long Do Users Stay on Web Pages? 2011. URL: https://www. nngroup.com/articles/how-long-do-users-stay-on-web-pages/.

[10] Riley, George F., and Thomas R. Henderson. "The ns-3 network simulator." In Modeling and tools for network simulation, pp. 15-34. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[11] M. Fiedler, T. Hossfeld, and P. Tran-Gia. “A Generic Quantitative Relationship Between Quality of Experience and Quality of Service”. In: IEEE Network24.2 (2010), 36– 41.