THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

MusQraTT: An MQTT-SN Broker for the Edge

Aisha Mohammed, Evan Stella
Prof. Gabriel Parmer (Faculty mentor), Gregor Peach (Alumni mentor)
The George Washington University Computer Science

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Background / Problem

Computational workloads for cyber-physical systems and Internet-of-things (IoT) devices are becoming increasingly more sensitive to latency. Applications such as autonomous vehicles, VR/AR systems, and other real-time systems must be able to perform computations on the significant amount of data they ingest with minimal transmission latency.

MQTT-SN (1) is a publish-subscribe network protocol commonly used to facilitate communication between devices in a cyber-physical system, such as between environmental sensors and computational nodes. Within a system using MQTT, the MQTT Broker is responsible for receiving messages from data publishers and forwarding them to clients who have subscribed to receive them. Mosquitto and its MQTT-SN Gateway is one of the most commonly used non-cloud based MQTT-SN broker implementations. It is written in C and known for being lightweight enough to be run on more resource-constrained systems. One of the commonly cited drawbacks of Mosquitto is its higher transmission latency when compared to competitors. In contrast, other popular self-hosted implementations rely on cumbersome runtimes, such as HiveMQ which is implemented using Java, and Mosca which is written in JavaScript and runs on node.js.

We implemented an MQTT-SN broker using optimizations to the broker's runtime environment and design that will allow our implementation to outperform competitors specifically for latency-sensitive applications.

CompositeOS & Rust

We implemented MusQraTT broker using the Rust programming language and run on CompositeOS. These innovations allowed us to be competitive when targeting latency-sensitive applications. Rust is a language that provides high performance, memory safety, and safe concurrency. These features allowed us to optimize the latency-sensitive codepaths in the broker and lead to a more efficient design than those implemented in C and higher-level languages. Composite is a microkernel focused on low latency and reliability. By working directly with Composite, we designed an operating system optimized to remove system-level bottlenecks that hinder the broker's performance. Running our broker on a specialized OS built on Composite allowed us to make targeted system-level optimizations that we would otherwise be unable to do on Linux for example.

Removing the MQTT-SN Gateway

All popular MQTT-SN brokers required a specialized gateway to convert MQTT-SN messages to MQTT messages that the MQTT broker understands. This introduces significant transmission overheads as each packet has to make twice the number of trips through the network in order to be processed by the broker.

In MusQraTT, we cut out the MQTT-SN gateway and create a pure MQTT-SN broker. Along with our other targeted system optimizations, this allows use to drastically reduce the transmission latency induced by the broker

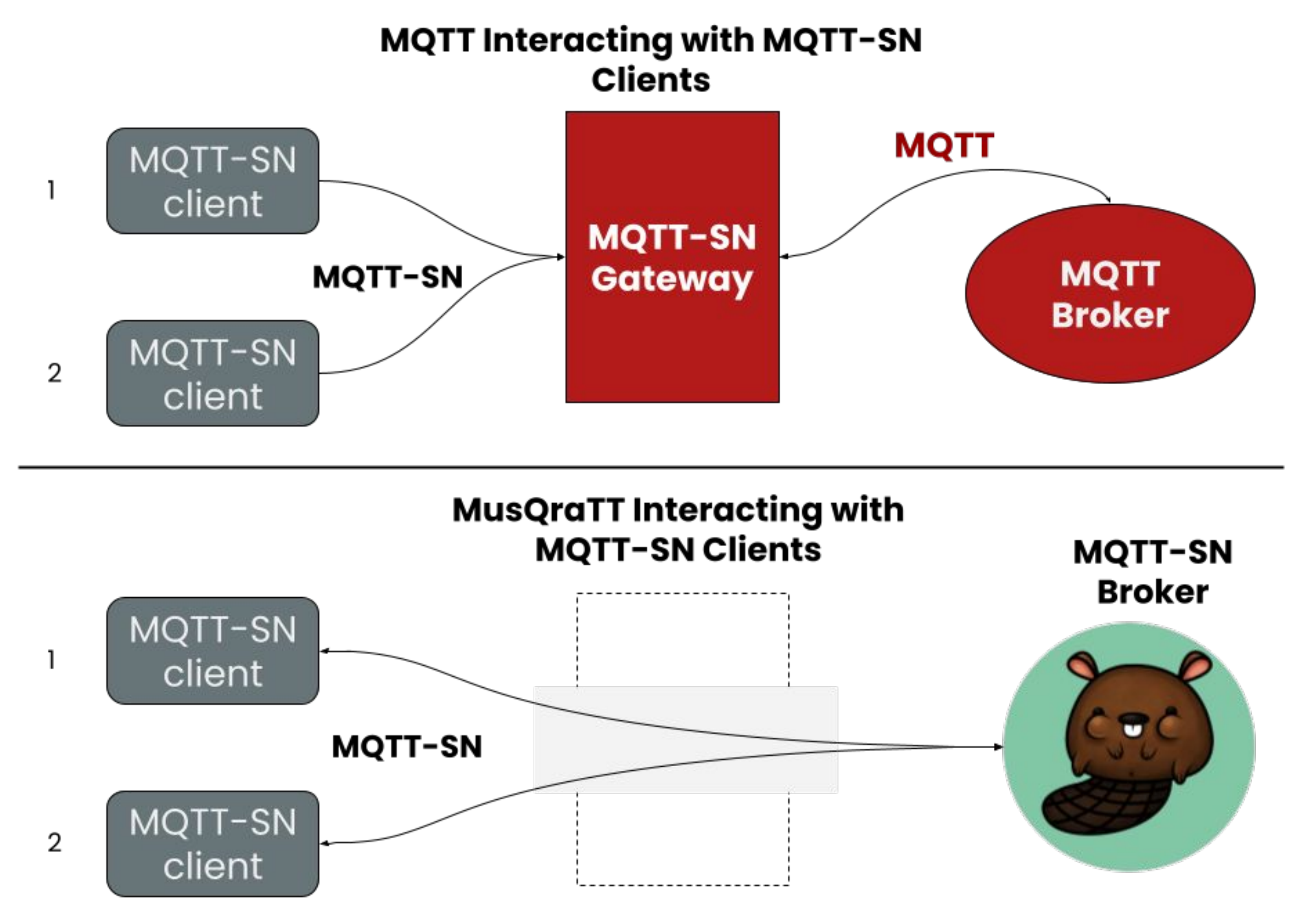


Figure 1: Removing the MQTT-SN Gateway.

Results

We measured MusQraTT's performance against Mosquitto and it's MQTT-SN Gateway. We measured the average transmission latency with increasing number of verbose clients on the network.

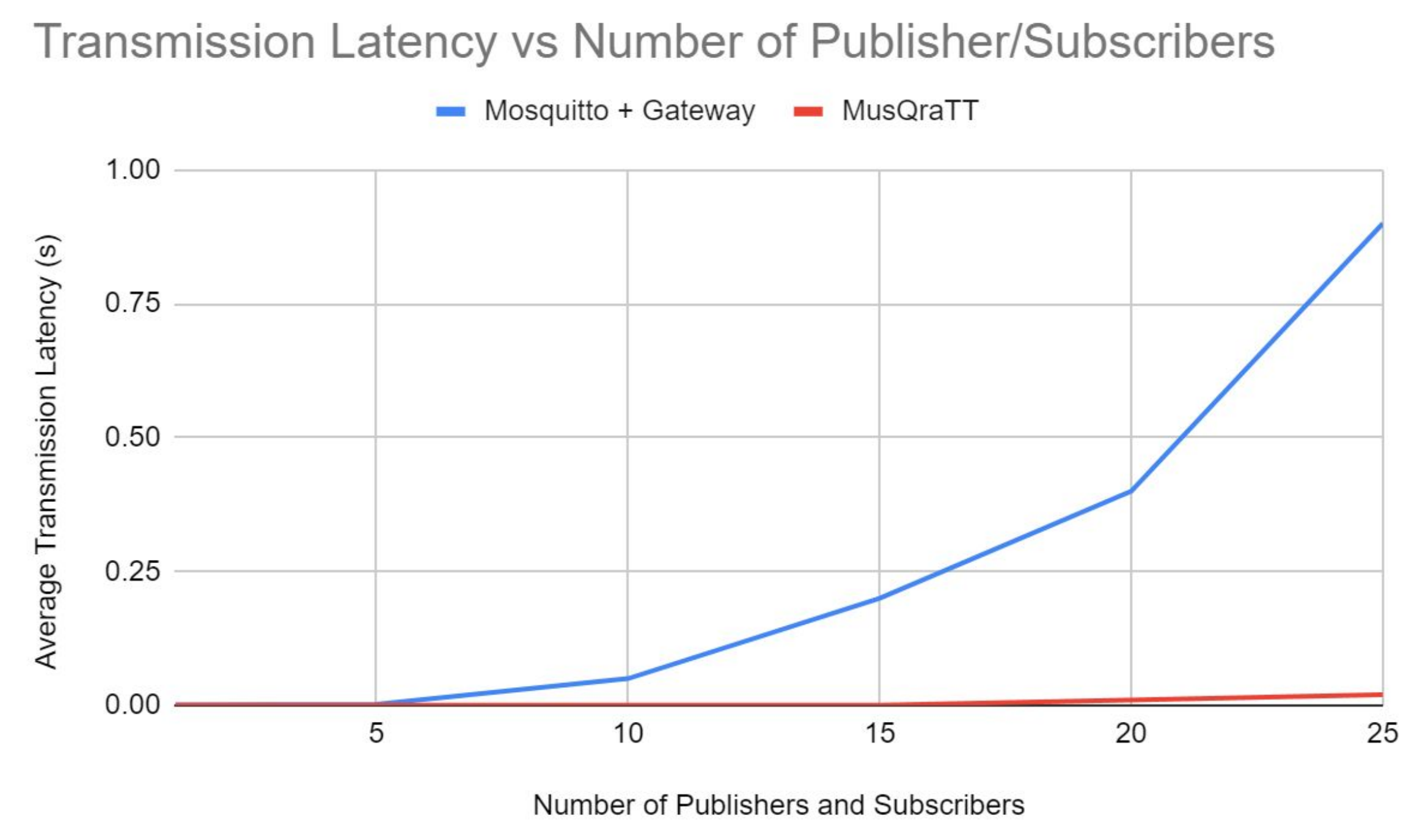


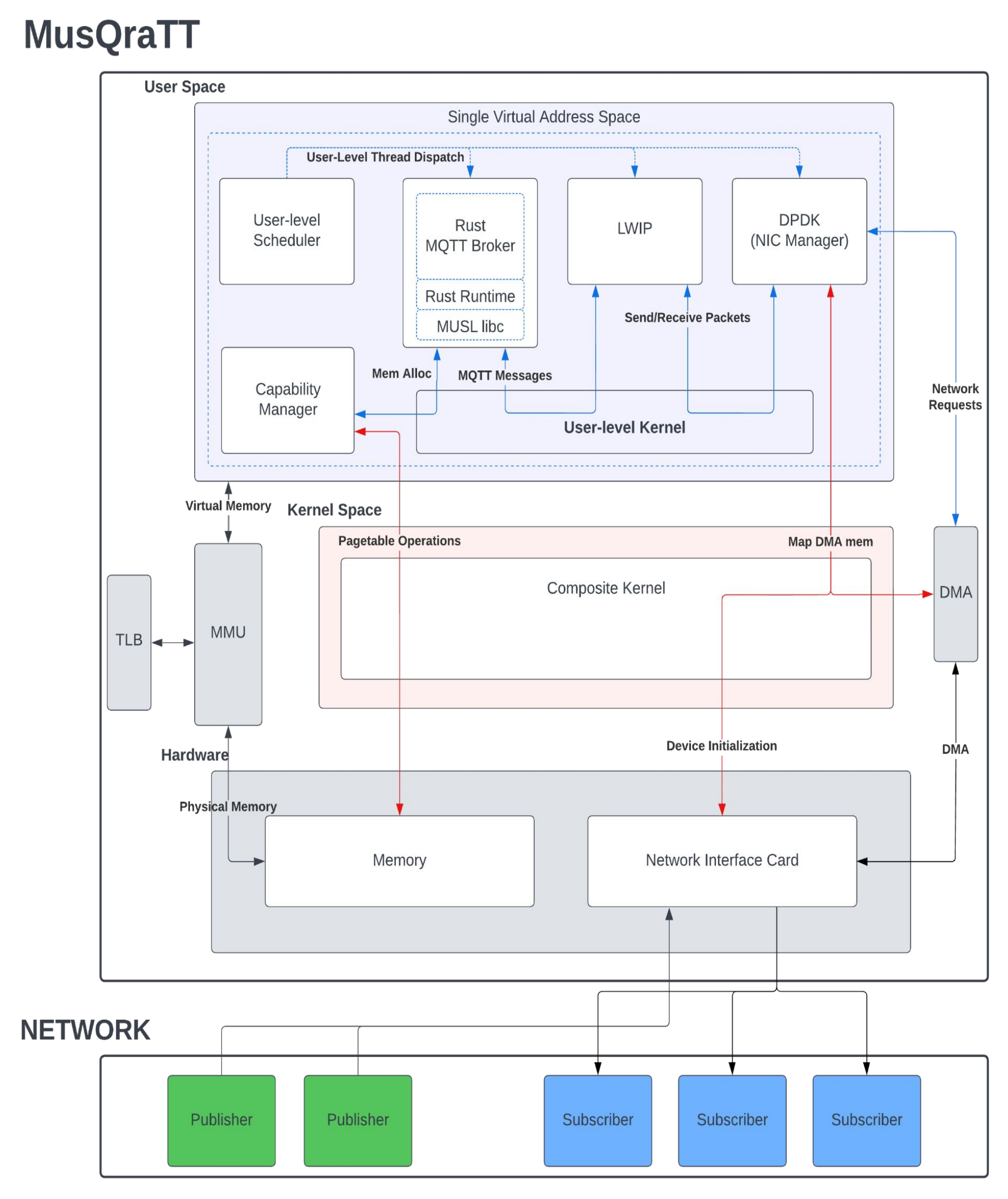
Figure 3: Average Transmission Latency measured versus the number of clients on the network. Each client published packets every 10ms to the broker. The latency is measured as the round-trip time between 1 publisher sending a message and receiving from the broker

Figure 3: The Average Transmission Latency for Mosquitto increases drastically since the Mosquitto is unable to send out published packets faster than it receives them. MusQraTT is able to do so, leading to much lower latency overall.

System-level Optimizations

MusQraTT uses CompositeOS as its runtime environment, a microkernel developed by the George Washington University that allowed us to design an optimized operating system that builds off years of research into performant and fault-tolerant OS design. We chose Composite over Linux because it allowed us to make targeted optimizations to key system services. One of these optimizations is to dramatically reduce the amount of system calls made during an MQTT message transmission by moving all message and packet processing into user-level.

We leveraged a significant innovation in operating system design from CompositeOS: process segregation and IPC done completely in user-level. CompositeOS's User-Level Kernel allows multiple components to share a pagetable, allowing low-cost IPC between system level services without incurring the cost of a system call. Use Composite, MusQraTT can send packets to an arbitrary number of subscribers with little-to-no kernel interaction. Compare this to Linux, where several system calls must be made per message to subscriber.



Broker and key system services running at user-level

System calls replaced with IPC calls between services

IPC facilitated by Composite's User-Level Kernel, Eliminating costly context switches

Figure 2: Overview of MusQraTT System Architecture.

Conclusion

MQTT-SN can be a suitable platform for latency-sensitive IoT and Edge computing applications. A performant Broker can enable this protocol to perform at levels sufficient to be utilized by systems that require ultra-low communication latency.

MQTT-SN is held back by an ecosystem that has neglected IoT applications. The current industry standard, Mosquitto is a crucial example of this. MusQraTT is a step towards this flexible protocol having more wide-spread adoption.

mosquitto

Future Research

- MusQraTT currently supports only a subset of the MQTT-SN protocol standard. Expanding MusQraTT to support the entire standard would be required in future research.
- The performance tests done test Mosquitto and MusQraTT under a very specific use-case. More robust performance testing is required.
- More robust performance testing will inevitably show parts of MusQraTT that require further optimization. Further optimization is a key next step in future research.
- Implementation of multicast packet transmission at the operating system level was a stretch goal that we did not hit on this iteration but would be a priority in further improving MusQraTT's performance.

References

References
1. Stanford-Clark, Andy, and Hong Linh Truong. "Mqtt for sensor networks (mqtt-sn) protocol specification." International business machines (IBM) Corporation version 1.2 (2013): 1-28.