MusQratt: Rust MQTT Broker on Composite OS

Aisha Mohammed and Evan Stella

George Washington University

CSCI 4243W: Capstone Design Project

December 14, 2022

Executive Summary

Innovation in technology is a large driver in societal behavior changes. Within the past ten years, we've seen how the Internet of Things has found its way into our everyday lives. Smart cars share data with manufacturers nearly instantaneously, Smart fridges know when the house residents have gone to sleep, and so on. With these innovations occurring rapidly, we must continue to optimize our implementations. Many modern cyber-physical systems that our society relies on to function, such as smart vehicles, air and spacecraft, and industrial control systems, ingest massive amounts of data and must be able to rapidly perform computations and make decisions with that data. As these systems become more advanced, and the engineers of these systems place more decision-making ability in their control, it is vital that they receive data in a timely manner, as the consequences of high latency can be severe on both human and economic scales.

MQTT, short for message queueing telemetry transport, is a cutting edge communications protocol that allows these real-time systems to communicate with each other more efficiently. Clients in a MQTT environment are categorized as publishers and/or subscribers. Publishers are clients who share data with any of their subscribers. Subscribers subscribe to a specific topic, listening for any new data to receive, and only receive information when there is information to be received. An integral component of MQTT is the broker; the broker provides all of the MQTT functionality by sending these communications to the appropriate clients. This creates a many to one connection between the broker and all the clients, as opposed to a many to many connection between all of the clients. In the real world, this can translate into a sensor in a smart city communicating with a passing vehicle about a sudden obstacle in the road in real time, where the car is a subscriber to the road sensors and the road sensors are publishers to all cars.

There are currently several MQTT implementations available. For a cloud-based implementation, Microsoft Azure and Amazon AWS have their own implementations available for developers to connect to their services' IoT hub devices. These cloud-based services provide C, Python, and JavaScript support. The most popular implementation is Mosquitto, which is implemented in C and is known to provide lightweight support for resource-constrained systems. However, Mosquitto's most discussed drawback is higher transmission latencies. In addition, both Mosquitto and cloud-based implementations have support for specific languages that provide heavier, costly run-times. Though cloud implementations provide an increased broker performance, there is still an opportunity for a more effective, safe implementation for MQTT.

We propose MusQraTT, an implementation of the MQTT broker in Rust on CompositeOS. Rust is a programming language that has built-in memory safety, high performance, and safe concurrency. Rust will fill in the gaps that languages such as C, Python, and JavaScript leave in providing safe, lightweight transmission of data within real-time,

resource-constrained systems. CompositeOS is a microkernel that prioritizes low latency and reliability. Building the Rust broker on CompositeOS allows us to remove system-level bottlenecks for performance and latency. Additionally, specific optimizations and modifications can be created to the broker and OS that prioritize a low-latency, safe implementation of MQTT that wouldn't be as achievable on an established OS, such as Linux.

Our goal is to achieve measurable improvements in transmission latency over popular competitors while creating a system suitable for resource-constrained applications on the Edge. This will include implementing an optimized MQTT broker in Rust, porting it to Composite, and creating a suite of tests that we can run on both our implementation and on competitors. The proposed technologies of Rust and Composite have built-in solutions to the current gaps in available MQTT broker implementations. Our project explores intersecting the two technologies to provide a low-latency and secure solution.

Technical Summary

Our implementation operates on CompositeOS, which is an operating system developed at GW to operate on the Edge of a network for resource constrained systems. Composite was developed with low latency and high reliability as its priorities. By using Composite, we are aiming to further reduce transmission latency rates. In addition, due to Composite's memory layout and Rust's built-in memory safety, we aim to provide a safer implementation of the MQTT broker.

One aspect of this project is developing the Rust MQTT broker. One of the many technical difficulties of a broker in this environment is creating the networking capabilities. The clients are connected and communicate with the broker over a TCP connection. The first challenge to work through is recognizing Rust's networking capabilities.

For the broker to be able to communicate with the clients, bytes of encoded data called packets, contain the appropriate data for the broker to make decisions and are passed over a TCP connection. Due to the complexity of this transmission, we've decided to use an existing library to handle encoding, decoding, and parsing these packets. Developer Brian Schwind has created a publicly available crate implementation, called "mqtt_v5", for packet transmission and topic filtering within the broker. MQTT has specifications that allow it to handle various levels of Quality of Service, which is an agreement between the broker and client on the reliability of delivery of packets. In our project, we will be working with only one level of Quality of Service. The remaining two levels operate with the Eclipse Mosquitto broker.

On the broker side, we must understand the utilization of the “mqtt_v5” crate for packet handling, design the broker logic, and decide on data structures for the topic-client association. On the Composite side, the operating system must have the ability to compile and run Rust code, which it currently doesn’t. Once Rust and networking support are available on Composite, our two sides must integrate to provide our MusQraTT project.

Product Specifications

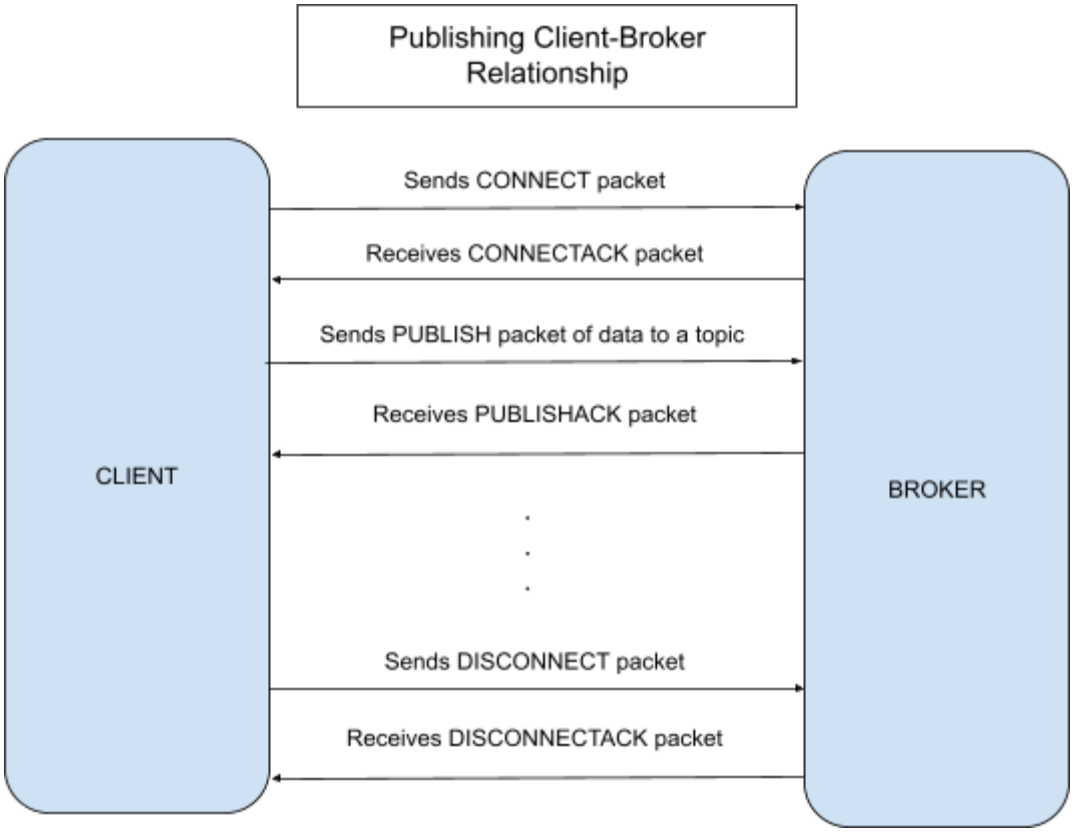
User Stories

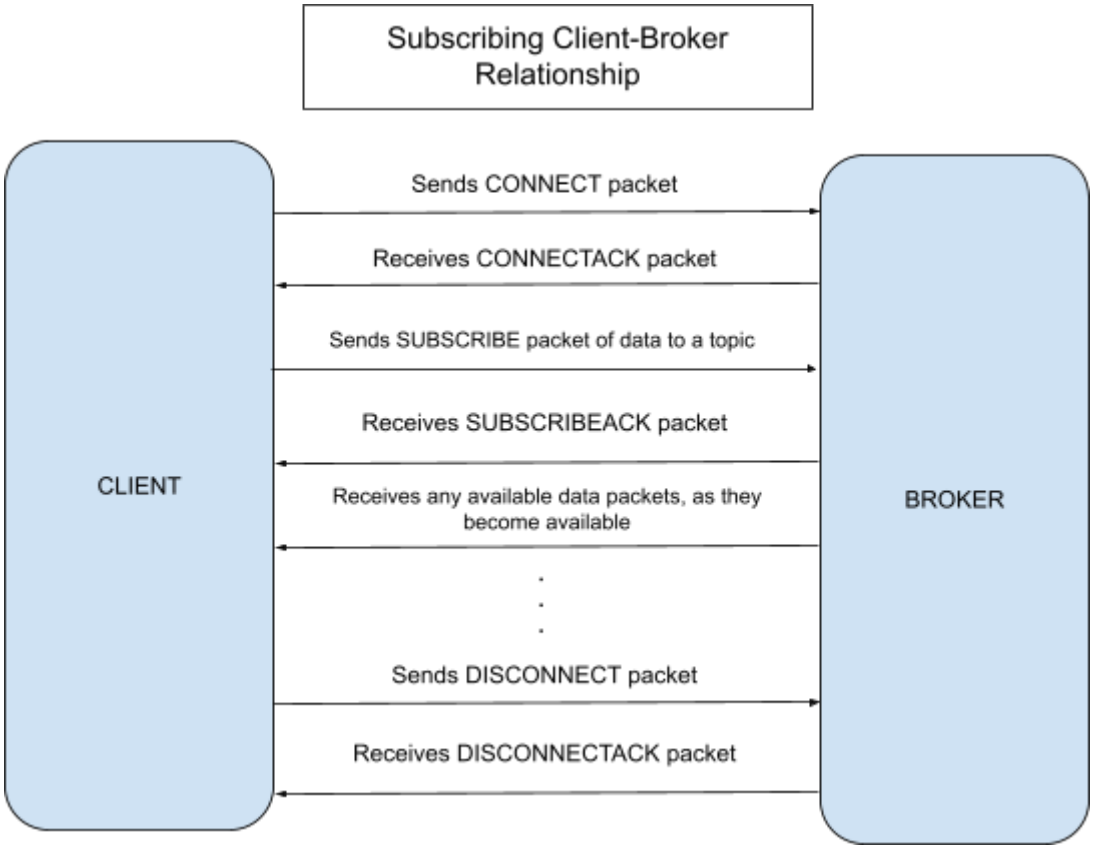
MusQraTT’s main user is developers. The MQTT protocol is increasingly becoming prevalent in Internet-of-Things configurations for its scalability and efficiency.

A developer who is seeking to build a scalable and lightweight implementation of a MQTT-protocol can MusQraTT. CompositeOS would be the primary avenue to utilize MusQraTT in order to get all of the benefits. Composite should be running on both the central server and the clients in the system. The developer is able to have as many clients as needed; they will later establish themselves to the server as publishers to or subscribers of data. MusQraTT utilizes TCP connections to send packets to ensure reliability and guaranteed deliverance. The developer’s clients must have TCP connection capacity in order to support MusQraTT.

Both publishing and subscribing clients must connect to the broker (server) to register as a viable device within the MusQraTT environment. After acknowledgment, a publisher has the ability to publish any data to any topic. As topic publications are sent to the broker and there are subscribers listening to that topic, subscribers immediately receive the published data. If there are no subscribers yet for that topic, the data packets are queued to store within the broker. After subscribing clients are acknowledged, they are able to immediately receive the queued data packets for their requested topic, if available. If there aren’t data packets ready to be sent, the client continues to listen and immediately receives data as it becomes available.

Developers are able to set the expiry time for their client connection to the broker and for their published data packets to ensure efficient use of resources. If this option is not utilized, as this is an option for personalized configuration within MusQraTT, developers must add a disconnecting ability for their clients. As a topic loses its last subscriber, its existence also expires.


Publishing Client-Broker Relationship
CLIENT
BROKER
Sends CONNECT packet
Receives CONNECTACK packet
Sends PUBLISH packet of data to a topic
Receives PUBLISHACK packet
Sends DISCONNECT packet
Receives DISCONNECTACK packet


Subscribing Client-Broker Relationship
CLIENT
BROKER
Sends CONNECT packet
Receives CONNECTACK packet
Sends SUBSCRIBE packet of data to a topic
Receives SUBSCRIBEACK packet
Receives any available data packets, as they become available
Sends DISCONNECT packet
Receives DISCONNECTACK packet

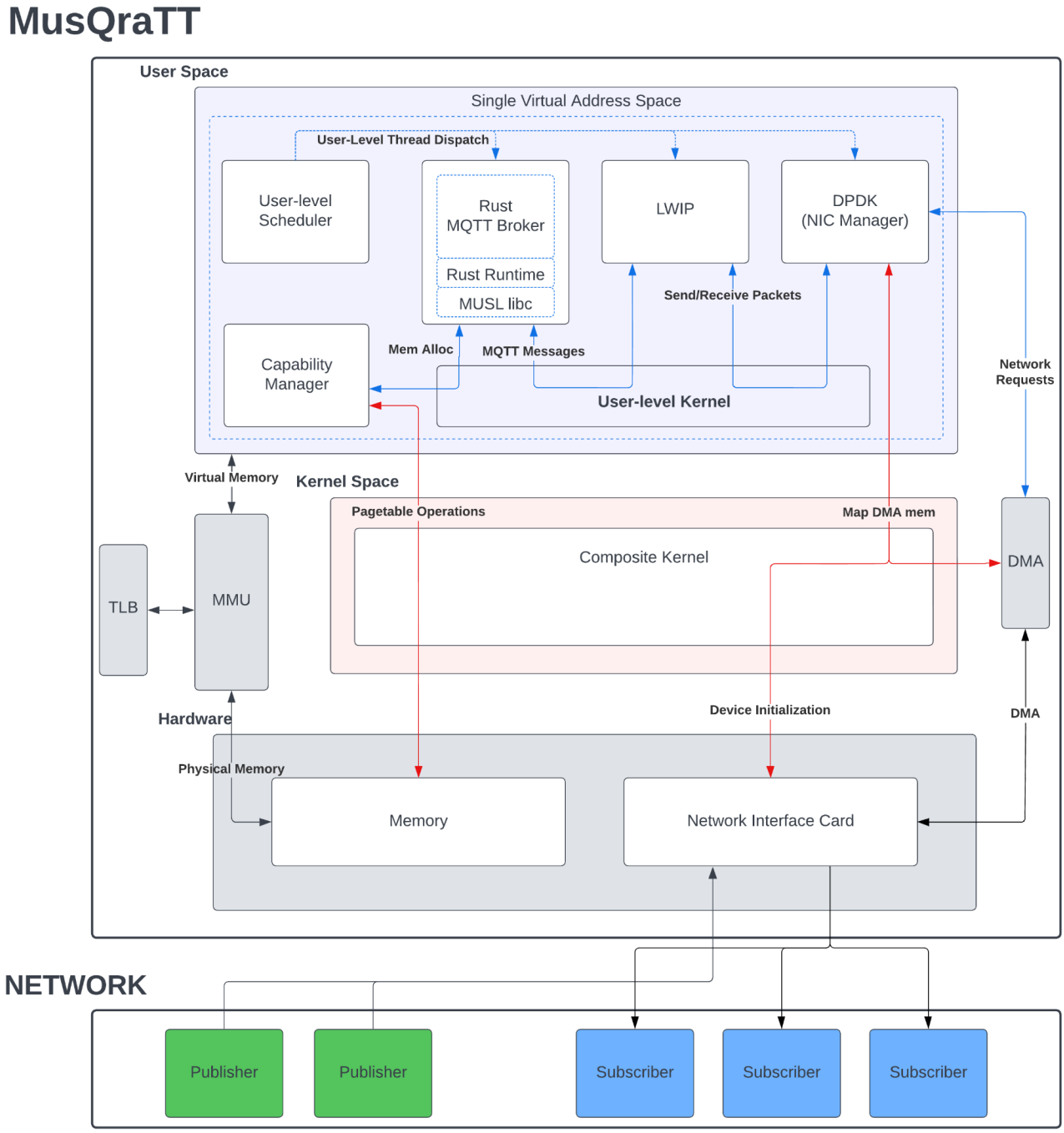
Command Interface

For the client to communicate with the broker, the following commands can be utilized:

- connect [client_id]
    - The client can establish itself as a viable resource to the broker by making this command. The client will be registered to the broker under the client_id parameter.
- disconnect [client_id]
    - To end a connection to the broker, this command will disconnect the client from and delete the client as a viable resource to the broker.
- sub [topic_name]
    - The client subscribes to a topic, topic_name. From now on, as data becomes available, the client will receive packets of information published to this topic.
- pub [topic_name] [packet_id]
    - The client publishes the packet of packet_id to a topic, topic_name. With this command, all subscribers to a topic will receive the packet of data.
- unsub [topic_name]
    - The client can unsubscribe from a topic, topic_name. From now on, they no longer receive data from this topic.

Technical Specifications

MusQraTT encompasses two key technical components: an MQTT Broker written in Rust, and its runtime environment: a custom operating system built on the Composite Kernel. Every aspect of MusQraTT is designed to reduce transmission latency.



The Broker is written in Rust to take advantage of the language's built-in memory and concurrency safety, and the compiler optimizations that these features allow for. A significant advantage we have over our biggest competitor, Mosquitto, is that our broker will be multithreaded. A significant portion of the early project implementation will be implementing Rust support for CompositeOS. Updates to Composite must be made to support the Rust

standard runtime, which is required for any non-trivial Rust code to run on the system. The Rust runtime sits on top of MUSL libc, the C standard library implementation used by CompositeOS. An additional compatibility layer provides an implementation for the POSIX system APIs used by the C standard library.

On a system using a Monolithic kernel, these system API calls would trap into the kernel, which would provide the requested service. CompositeOS allows us to implement most system services at user-level. Thus, on Composite, these POSIX API calls abstract RPC calls to other services in the system that are executing at user-level. On MusQraTT, these services include a capability manager which manages system resources, including memory, for all components in the system, a scheduler, a TCP/IP stack, and a Network Interface Card driver.

Aside from Composite's custom scheduler and capability manager, we are using two third-party frameworks in the system. We will be using LwIP, which is an open-source TCP/IP protocol suite designed for embedded systems for MusQraTT's TCP/IP stack. We chose LwIP because it is a popular, well-tested, multithreaded TCP/IP stack that was designed for the types of applications we are targeting. We are using the Data Plane Development Kit (DPDK), an open-source set of libraries used to implement packet processing, that allows us to move Network Interface Card management to user-level. Most critically, DPDK allows us to access the NIC directly, and thus send and receive data on the network, without ever dropping into kernel mode. DPDK is an open-source and well-trusted solution that is used commonly on Linux to achieve the same goal of reducing transmission latency that is caused by user/kernel mode switches.

We will be taking advantage of CompositeOS's User-level Kernel, an experimental feature still in development that allows multiple components in the system to cohabitate in a single pagetable but still have strong isolation between them. This allows these components to make RPC calls to each other without having to trap into the kernel to switch page tables and flush the TLB. Being able to make RPC calls in the system without inducing system call latency, TLB misses, and pagetable walks will dramatically reduce the overall system latency during MQTT message transmission.