# MusQraTT

An MQTT Broker for the Edge

## 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.

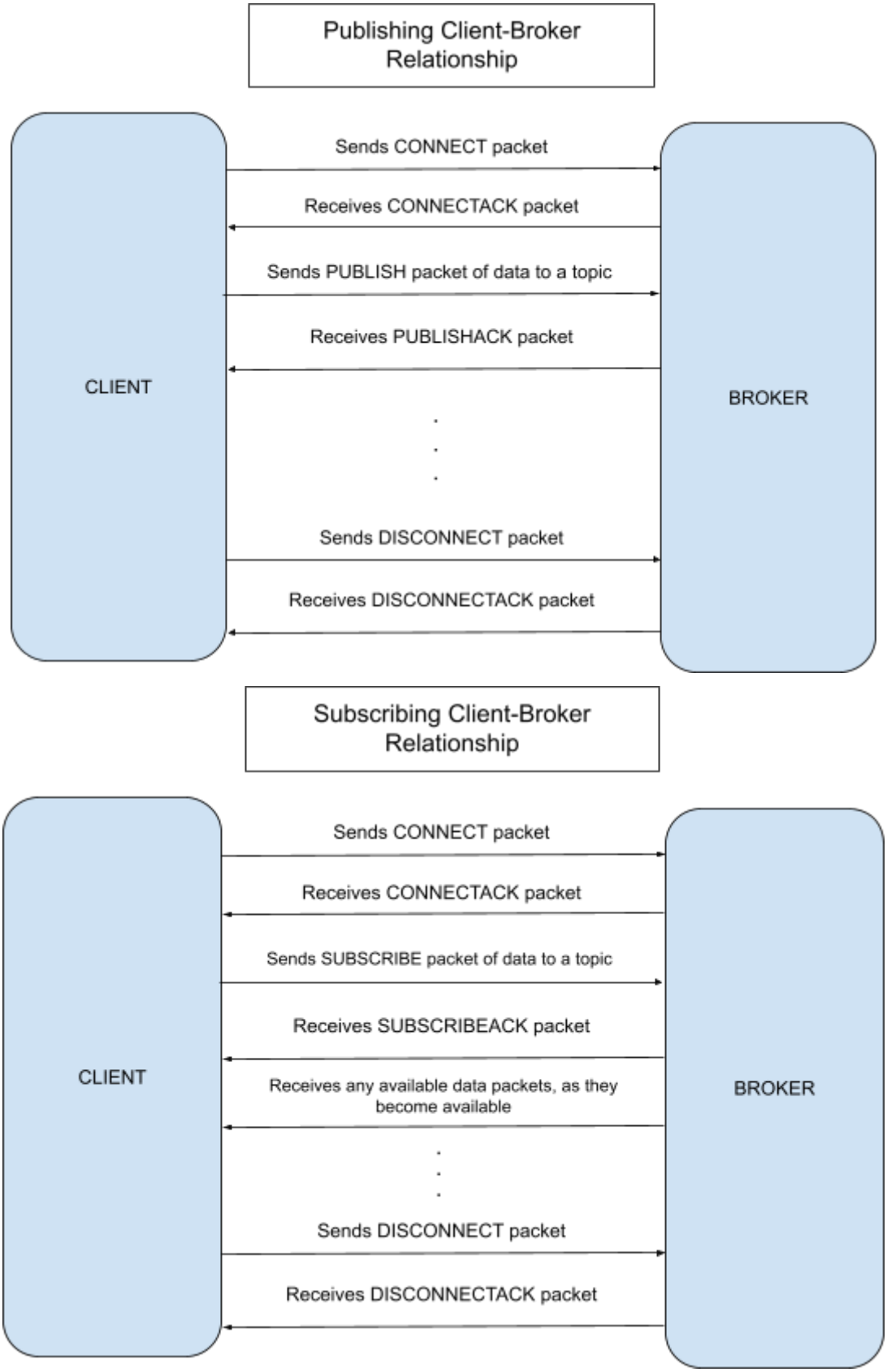
A developer who is seeking to build a scalable and lightweight implementation of a MQTT-protocol can MusQraTT. CompositeOS would be the primary avenue to utilize MusQraTT in order to get all of the benefits. Composite should be running on both the central server and the clients in the system. The developer 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.

# MusQraTT

An MQTT Broker for the Edge



# MusQraTT

An MQTT Broker for the Edge

**Command Interface**

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

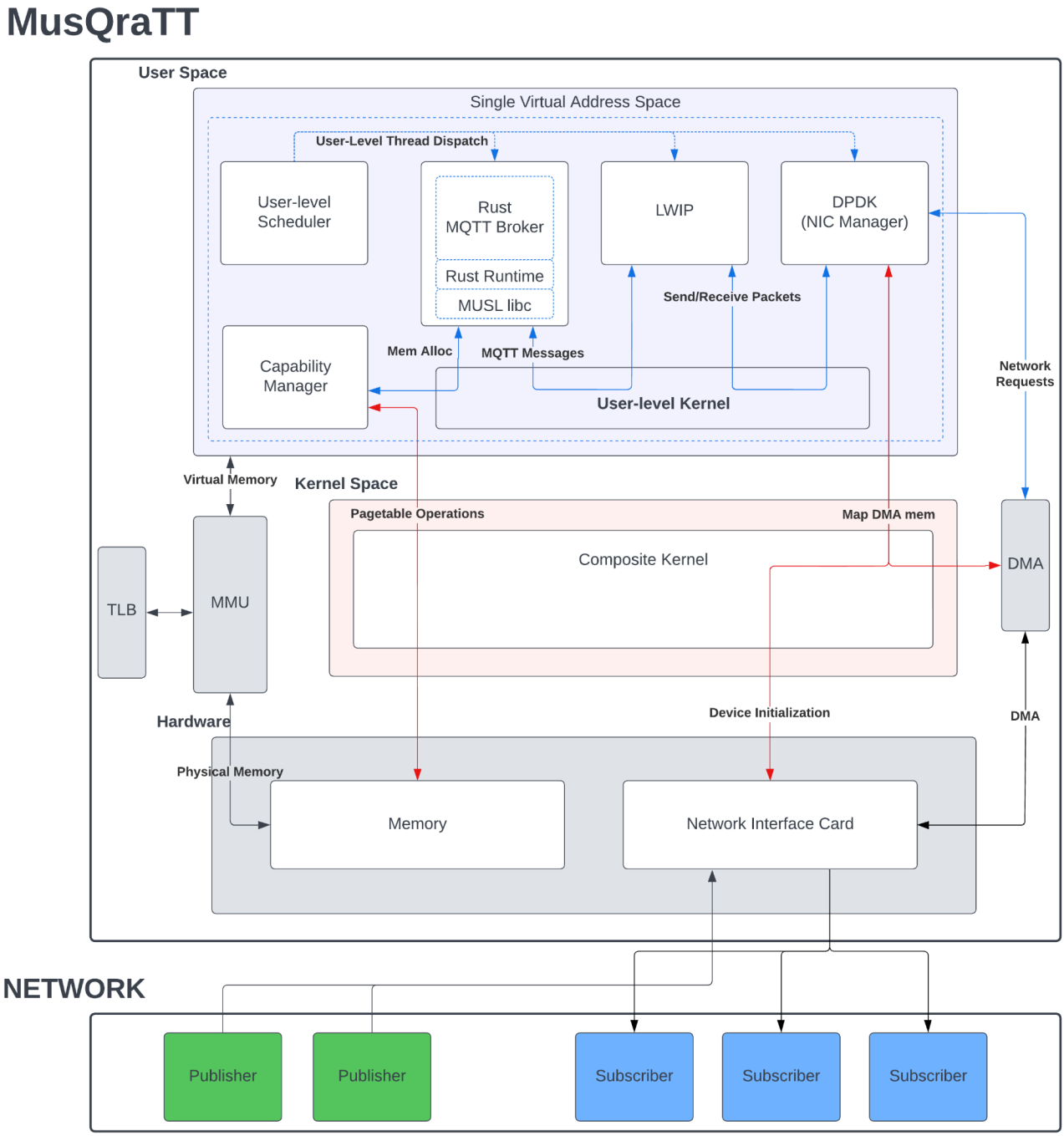
- connect [client_id]
    - The client can establish itself as a viable resource to the broker by making this command. The client will be registered to the broker under the client_id parameter.
- disconnect [client_id]
    - To end a connection to the broker, this command will disconnect the client from 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.

# MusQraTT

An MQTT Broker for the Edge

### 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.



# MusQraTT

An MQTT Broker for the Edge

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.