PEBBLE: AN MQTT-BASED COMMUNICATION ARCHITECTURE FOR HETEROGENEOUS MULTI-ROBOT SYSTEMS

by

Andrew C. Mattson

A Project Submitted in Partial Fulfillment of the Requirements
for the Degree of

Master of Science
in
Computer Science

University of Alaska Fairbanks
May 2026

APPROVED:

Dr. Orion S. Lawlor, Committee Chair
Dr. Glenn G. Chappell, Committee Member
Dr. Sangram K. Jena, Committee Member
Dr. Orion S. Lawlor, Chair
Department of Computer Science
William Schnabel, Dean
College of Engineering and Mines
Emeline Jones, Director
Graduate School & Interdisciplinary Studies

Copyright

Abstract

Pebble is a local-first MQTT communication architecture for heterogeneous robotic systems. It uses a robot-local broker for low-bandwidth control, telemetry, discovery, and service coordination, and selectively bridges those topics to a remote broker for off-robot teleoperation and monitoring. On Linux-based robots, single-owner daemons manage exclusive resources such as serial links, cameras, and microphones; on MCU-only robots, a reduced version of the same MQTT topic contract is implemented directly in firmware. The architecture was exercised on four physical robot platforms (Goob, Fred, MiniLAMP, and BenJr). Archived MQTT traces and controlled router-impairment experiments show that the shared topic contract supports teleoperation, retained state publication, autonomy coordination, and media control across these platforms. The evaluation also identifies the current limits of the implementation: performance depends strongly on network conditions, the current MQTT audio path is bandwidth-inefficient because it is uncompressed, and the architecture is not intended for hard real-time control. These results indicate that MQTT can serve as a practical interoperability layer across a heterogeneous robot fleet.

Acknowledgements

I'd like to thank Dr. Lawlor for his extensive input and guidance throughout this project. The knowledge gained from his Robotics & 3D Printing (CS F453) course directly contributed to the content of this project. I'd also like to thank the other members of my committee, Dr. Chappell and Dr. Jena, for their invaluable feedback.

Thank you also to my family for helping me reach this point, and specifically my wife, Katherine, for supporting me during my work in every way imaginable.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

This work is motivated by the growing role of robots as networked systems rather than isolated machines. The way increasingly capable robots communicate and coordinate is central to their usefulness. Robots matter because they extend human capability into places and situations where people are limited by risk, distance, or endurance. In harsh environments such as space or Alaska, they can perform dangerous, repetitive, or physically demanding work more safely and consistently than humans. Advances in sensing, computation, and artificial intelligence are making it possible for robots to operate in spaces designed for people, handling fragile objects, adapting to variation, and supporting tasks that would otherwise require significant human effort.

Cloud and off-robot compute allow robotic systems to move some computationally demanding workloads away from the robot itself and onto more capable remote resources (Hu et al., 2012; Kehoe et al., 2015). That shift, however, makes communication performance more important. Robotics algorithms such as Simultaneous Localization and Mapping (SLAM) and other perception pipelines can impose substantial computational demands, while many embedded platforms remain constrained by energy, size, compute budget, or thermal limits (Campos et al., 2021; Ji et al., 2021; Matthee et al., 2024). Networking can move some of that work to external resources, but only when the communication fabric remains practical under those same constraints (Hu et al., 2012; Kehoe et al., 2015). MQTT provides a lightweight and scalable mechanism to bridge this gap between resource-constrained robots and off-robot computation (OASIS, 2014; OASIS, 2019).

MQTT is a lightweight, publish-subscribe network protocol designed for constrained devices and low-bandwidth, high-latency, or unreliable networks (OASIS, 2014; OASIS, 2019). Unlike traditional request-response models, MQTT relies on a central broker to route messages between publishers (who send data) and subscribers (who receive it) based on hierarchical topics. This decoupling allows robots to share telemetry and receive commands without maintaining direct, point-to-point connections.

Robot Operating System (ROS) is a widely used robotics middleware framework that provides a collection of tools, libraries, and conventions for building complex robot behavior (Quigley et al., 2009; Macenski et al., 2022). While ROS offers a powerful ecosystem, it also

brings framework-specific runtime conventions, interfaces, and deployment patterns that are well suited to larger robot software stacks but not always necessary for smaller embedded participants. For resource-constrained robots, a thinner MQTT-based fabric can therefore be attractive as an interoperability layer across heterogeneous systems (Quigley et al., 2009; Macenski et al., 2022).

Some keywords present throughout this paper include:

1. **Heterogeneous:** Composed of non-identical hardware and software configurations
2. **Interoperability:** The capacity for diverse systems to communicate and collaborate through a shared standard
3. **Publish/Subscribe:** A communication model where systems send messages to topics instead of directly to each other
4. **IoT:** The Internet of Things; network-connected physical devices that exchange data and commands (Greer et al., 2019)
5. **Broker:** A server that receives messages and routes them to the right subscribers
6. **QoS:** Quality of Service; MQTT's selectable message-delivery guarantee level
7. **Teleoperation:** A robot controlled remotely by a human operator

For some systems, interoperability is not optional. Swarm robots (Gielis et al., 2022) and other multi-robot systems (Kolling et al., 2016) require inter-robot communication at scale to support coordination and collective decision-making. González Bonifaz et al. (2018) likewise identify device-to-device communication, control centralization, and scalability as recurring challenges in multi-robot systems. In extraterrestrial environments, where communication infrastructure is sparse, robots may also need to operate as interoperable participants in a broader network. LunaNet, a networking architecture proposed by NASA for lunar exploration, is explicitly framed as an extensible communications infrastructure connecting robots, towers, landers, satellites, and other nodes (Israel et al., 2020). In such settings, MQTT offers a broker-mediated way to publish needed information and subscribe selectively while retaining centralized security controls (Mukhandi et al., 2019).

Modern robotics frequently uses off-robot software for teleoperation, where a remote operator monitors first-person video and issues manual overrides. Traditionally, this requires a direct, often fragile link between the robot and the controller. By using MQTT as a bridge, these components become modular; a web-based dashboard can subscribe to a robot's video-stream

topic and publish to its drive-control topic without either component needing to know the other's network address (Atmoko & Yang, 2018; Mukhandi et al., 2019).

This report describes the Pebble architecture and implementations evaluated during the project period. Later repository additions are outside the scope of this report. Improvements to the software stack described are likely to continue well into the future.

Chapter 2: Prior Work

Zegarra et al. (2024) propose an Internet of Robotic Things (IoRT) middleware that adapts IoT architectural patterns to robots. Their design includes a central server managing client publications and subscriptions in a way that closely resembles an MQTT-style pub-sub model. Their system includes four physical robots and one simulated robot with different body plans and custom middleware, making it the closest discovered architecture to Pebble.

Das et al. (2025) propose an IoT-enabled mobile robotic system built around cloud MQTT. Their robot uses an ESP32-CAM, three servo motors, and an ultrasonic sensor, with MQTT carrying commands and telemetry. They argue that this architecture can reduce power consumption, simplify packet handling, and lower bandwidth usage. They also note a practical problem that is directly relevant here: when both ends of a robotic system require substantial manual network configuration, modification, extension, and reuse become unnecessarily difficult.

Koodtalang et al. (2022) describe multi-robot control over MQTT with Zigbee modules extending network range in outdoor environments. Their Pi-hosted MQTT broker coordinates both robot-to-robot and robot-to-cloud communication. Using kit-built tracked robots, they report real-time display of environmental and robot-status data, and they also employ compact message identifiers that function similarly to a simple topic scheme.

González Bonifaz et al. (2018) describe an MQTT-based architecture in which multiple robots of the same body plan communicate with each other and with controller processes for autonomous navigation and teleoperation. They define a principle-topic construction similar in spirit to the MQTT topic standard used in this report, although theirs is tailored specifically to one robot body plan and telemetry set. Their evaluation is framed around standard IoT concerns such as distribution, interaction, security, scalability, and energy efficiency. Their results also highlight a limitation that remains relevant here: communication quality depends strongly on connection stability. Mukhandi et al. (2019) refine that point by showing that, in their setting, network status mattered more than the added security mechanisms.

Mukhandi et al. (2019) examine MQTT as a way to close a networking security gap in ROS-based robot systems. Their architecture places MQTT between patrolling robots and remote clients so that robots and clients do not connect directly. They report that MQTT provided appropriate encryption and authentication support while also accommodating both simple sensor values and richer data such as camera images. Their results further indicate that, even when strong security mechanisms such as TLS and X.509 certificates are enabled, network status and Wi-Fi quality remain the primary communication bottlenecks.

Atmoko and Yang (2018) provide another example of MQTT used successfully in robotics. Their system enables an industrial robot arm to be monitored and controlled from laptops or mobile devices, with ROS bridged through MQTT. They frame real-time responsiveness as a system requirement and conclude that the observed responsiveness was sufficient for their intended application.

In summary, IoT-based infrastructures for robotics (and, in particular, MQTT) have already been proven to be effective as a layer for secure, low-latency communication in environments with reasonable network stability. Existing applications use it as a communication layer within their niche architectures. Only one source, Zegarra et al. (2024), built a more general-purpose, MQTT-based architecture that operates consistently across heterogeneous robotic body plans and environments.

Chapter 3: The Pebble Architecture

Cloud-based robotics changes the dominant constraints of modern autonomous systems by making remote compute more available while increasing dependence on communication performance (Hu et al., 2012; Kehoe et al., 2015). In that setting, communication becomes a primary systems concern. While ROS remains a dominant robotics middleware, its richer framework model and deployment conventions are not always an ideal fit for the constrained embedded hardware found in smaller or more specialized robotic units (Quigley et al., 2009; Macenski et al., 2022). In a multi-robot fleet where hardware ranges from powerful single-board computers to simple microcontrollers, the absence of a lightweight, shared communication standard often leads to "one-off" integrations that are harder to maintain and scale (González Bonifaz et al., 2018; Zegarra et al., 2024).

Beyond the challenges of hardware diversity, practical field robotics must also contend with exclusive hardware ownership and the inherent unreliability of wireless networks. In traditional monolithic designs, multiple processes often compete for exclusive or priority-sensitive access to hardware endpoints, such as a serial link to a motor controller, a camera device, or an audio capture/playback device (Linux Media Documentation, 2026; Linux Media Documentation, 2026; ALSA Project, 2026). When these scripts are written in isolation, such resource contention becomes a frequent cause of system failure. Furthermore, relying strictly on a remote-path communication model creates a fragile system; if the connection to the central cloud broker is intermittent, the robot's internal components may lose the ability to coordinate even basic local tasks. There is a clear need for a communication fabric that is globally interoperable yet locally resilient (Gielis et al., 2022; Mukhandi et al., 2019).

To address these challenges, this project introduces Pebble, an MQTT-based architecture designed to provide a unified communication and control layer for heterogeneous systems. Pebble moves away from monolithic control structures in favor of a decentralized publish-subscribe model (OASIS, 2014; OASIS, 2019). By using a negotiator-daemon pattern and a robot-local brokerage system, Pebble keeps high-rate internal traffic on-robot while exposing a standardized JSON topic-and-payload contract for remote teleoperation and data logging. This approach allows diverse platforms, ranging from the Strandbeast-style walker BenJr to the Linux-based Goob, to communicate through a shared standard. The following sections detail the

technical implementation of this architecture, beginning with the shift to a local-first MQTT structure.

3.1 Our Implementation

We seek to demonstrate Pebble's ability to work on many types of robots, by implementing it on a diverse set of robots. We describe our results on four robots: **Goob, Fred, MiniLAMP, and BenJr**. All four components communicate through the same remote MQTT broker, but each have different hardware implementations.

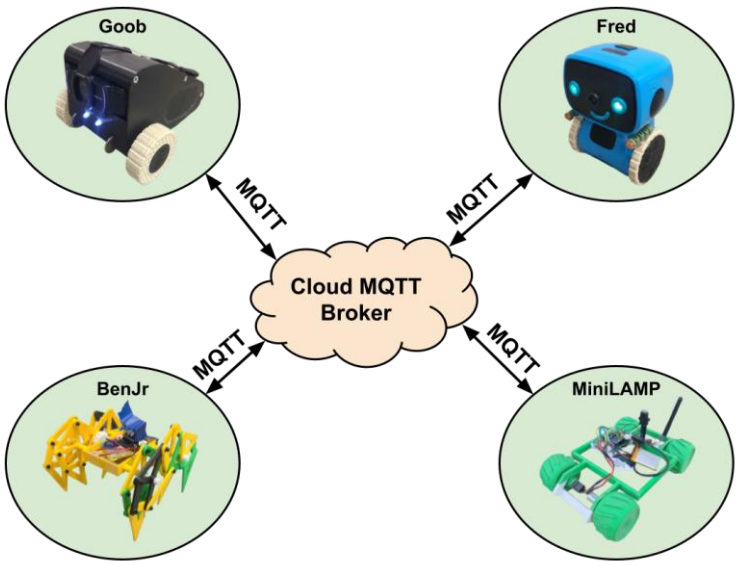

Figure 3.1: Pebble test platforms: Goob, Fred, MiniLAMP, and BenJr.

Goob and Fred share the same general hardware structure. Each have Seeeduino XIAO SAMD21 microcontrollers (MCUs) that convert non-USB peripheral hardware (motors, lights, sensors) into USB serial for Pi-style computer boards to access. Said Pi computers run the full Linux Pebble software backend.

Figure 3.2: Goob, the primary Pebble development platform.

Goob (above) is the primary development and testing platform for the Pebble architecture.

Brain & Logic: An **Orange Pi Zero 2W** (Quad-core Cortex-A53) that handles the full suite of local services, including the `mqtt_bridge` and `av_daemon`.

Peripheral Hardware: A **Seeeduino XIAO SAMD21** MCU acts as a bridge to non-usb hardware. The MCU converts USB serial streams into PWM signals for Goob's white LED headlights and DRV8833-driven motors.

Media Devices: A **Logitech C922** webcam and a USB speaker.

Body Plan: A fully **3D-printed chassis** with two driven wheels and one roller ball. Goob was the original "Pebble" concept platform.

Figure 3.3: Fred, a Pebble platform built on a commercial toy base.

Fred (above) is a friendlier-appearing counterpart to Goob, sharing a nearly identical internal architecture in a drastically different body plan.

Brain & Logic: A **Raspberry Pi Zero 2W** (Quad-core Cortex-A53). It runs the full Pebble software stack, including the local broker and negotiators. This helps demonstrate the architecture's portability across different ARM-based single-board computers.

Peripheral Hardware: A **Seeeduino XIAO SAMD21** MCU serves as the hardware bridge. Identical to Goob's, it manages the serial link between the Pi and the robot's DRV8833-controlled motors and full RGB LED lighting.

Media Devices: An **Arducam** and a WM8960 microphone/speaker hat.

Physicality: Built on a modified "GILOBABY" toy robot base. Fred serves as an example of integrating the Pebble architecture into pre-existing commercial mechanical designs.

Figure 3.4: MiniLAMP, an MCU-based Pebble platform.

MiniLAMP is a scaled-down version of the Aurora Robotics Lab's Lunar Autonomous Modular Platform (LAMP) concept robot.

Brain & Logic: A **Seeeduino ESP32-C6** MCU. It lacks a local broker and connects directly to the remote MQTT server to process drive commands and publish heartbeats.

Peripheral Hardware: Integrated voltage divider to measure battery voltage and two DRV8833 motor drivers. The MCU maps battery voltage to MQTT telemetry topics.

Media Devices: None. MiniLAMP is paired with Pitcam (described below).

Physicality: A fully 3D-printed chassis. Currently deployed for testing at the UAF experimental farm.

Paired with MiniLAMP is **Pitcam**, a Raspberry Pi 4B that manages a simple WebRTC webcam stream (photo of webcam view below) via a Starlink satellite connection. It allows users to teleoperate MiniLAMP with lunar-analog latency from across campus. Pitcam's mast also houses MiniLAMP's charging boom, which has a magnetically attaching USB-C charging cable. Pitcam does not incorporate the full Pebble stack, only sending a heartbeat and listening for commands to start/stop the video stream.

Figure 3.5: Pitcam webcam view paired with MiniLAMP.

Figure 3.6: BenJr, a Strandbeast-style Pebble walker.

BenJr (above) is a Strandbeast-style legged walker, and the only robot using legs.

Brain & Logic: A LOLIN WEMOS D1 Mini clone (ESP8266-based MCU). As another "constrained tier" platform, it lacks a local broker and connects directly to the remote MQTT server to process drive commands and publish heartbeats.

Peripheral Hardware: I2C motor and battery controller hats.

Media Devices: None. BenJr helps test the bare minimum requirements for a robot to participate in the Pebble communication fabric.

Physicality: A 3D-printed featuring Strandbeast-style legs.

3.2 Robot-Local MQTT

Up to now, we have considered MQTT as a solution for inter-robot communication, but MQTT can also be used *within* a robot to allow devices with exclusive communication endpoints to communicate with multiple processes. For example, a robot might have a serial device connected via USB, which presents one serial endpoint (a byte stream) typically only accessible by one process at a time (and, if it is accessible by multiple processes, can result in race

conditions). What if multiple processes need to exchange data with this serial device? In such a situation, a robot-local MQTT broker greatly simplifies the problem. A single simple negotiator process can claim the serial endpoint and subscribe to a topic, or set of topics, through which other processes send data headed to the serial device. The negotiator would also publish outgoing serial data to a known topic, or set of topics.

Below is the Pebble architecture *before* introducing robot-local MQTT.

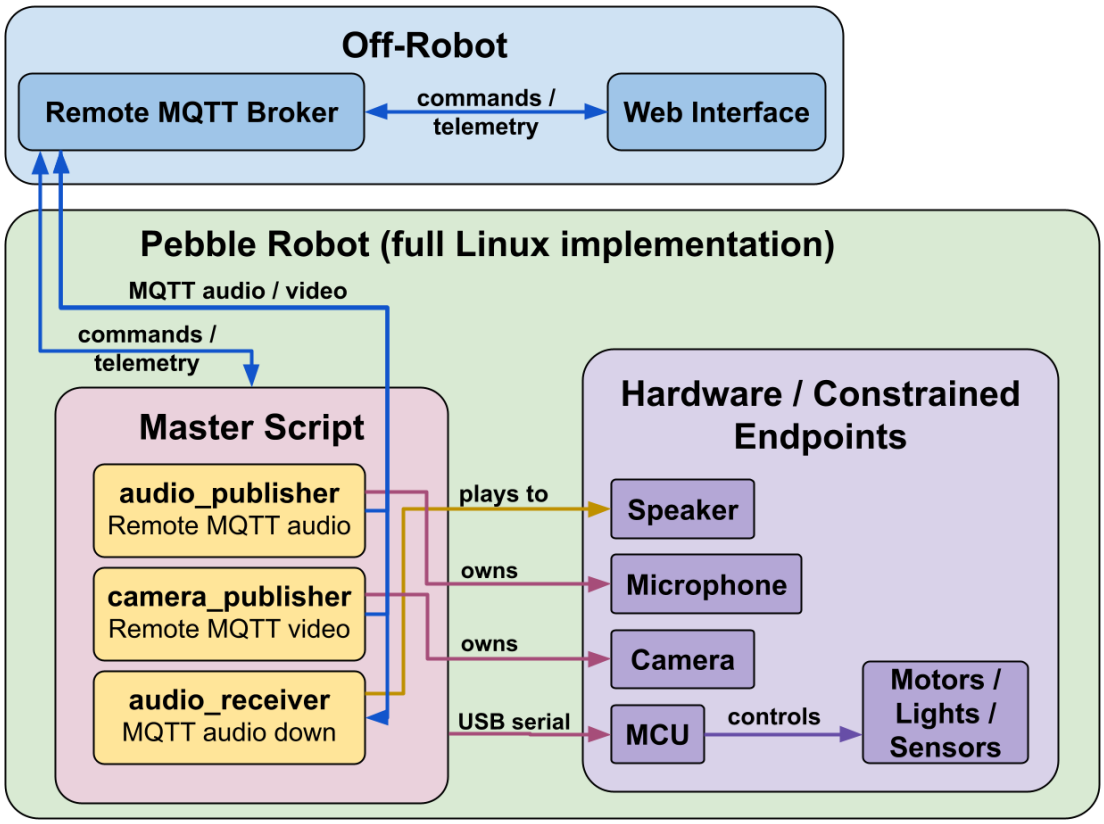

Figure 3.7: Pebble architecture before robot-local MQTT.

It should be noted that robots without a full Linux-based software stack (like robots with just a WiFi-capable MCU) do *not* implement a robot-local MQTT broker and rely fully on the remote broker. As such, their architecture resembles that above, but typically without audio or video due to high bandwidth requirements, which means the architecture is greatly simplified and acceptable for those platforms.

To reduce complexity and increase extensibility, we shifted to a robot-local MQTT structure. Below is the current Pebble architecture diagram for Goob and Fred, which integrates robot-local MQTT, allowing multiple scripts to work together and greatly increasing extensibility.

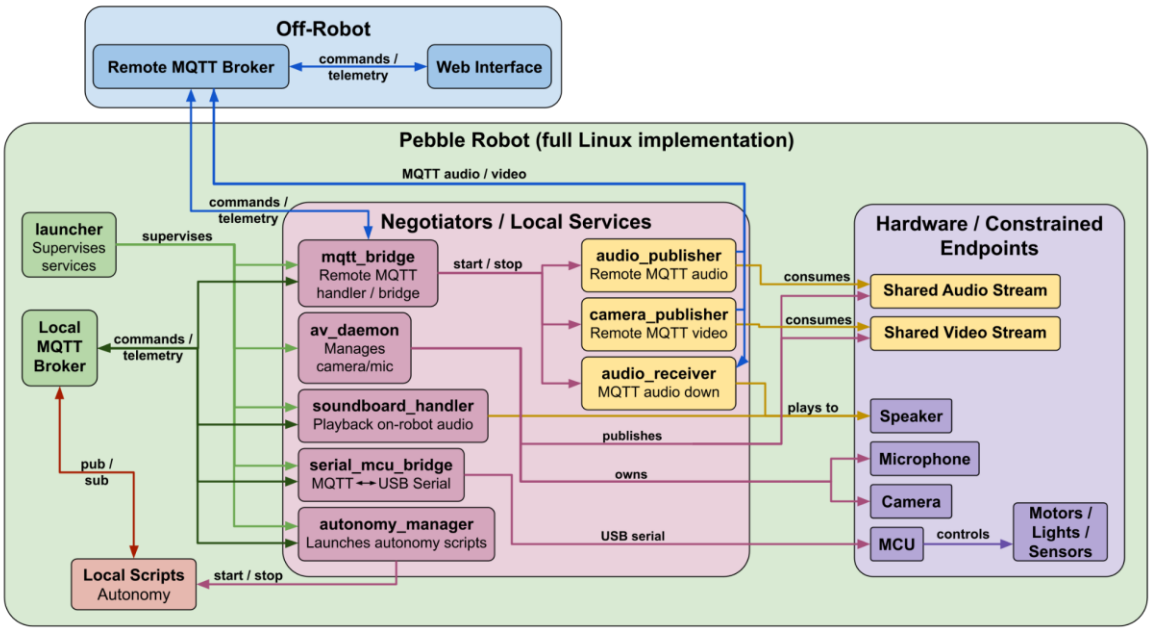

Figure 3.8: Pebble architecture with robot-local MQTT and negotiator processes.

Although the new architecture introduces more processes, it reduces code concentration and clarifies ownership boundaries. The previous architecture housed *all* logic within **a single master bridge script**, including device ownership, cloud routing, and application logic, which resulted in **thousands of lines of code**. The current solution instead applies stronger abstraction and encapsulation. Each subsystem is controlled by a single daemon process that abstracts one domain of control. These daemons translate between locally-brokered MQTT topics or, for higher bandwidth data like audio and video, shared-memory transport via GStreamer and one constrained endpoint or task (e.g., a serial port, a remote MQTT broker, or autonomy logic) (GStreamer, 2026). Other processes then interact with those daemons through MQTT contracts. This contract-based composition allows components to be replaced or restarted independently without rewriting the rest of the stack.

The robot-local MQTT architecture also resembles ROS in that independent processes communicate through a publish/subscribe fabric rather than direct point-to-point calls, but its interoperability contract is thinner because it consists primarily of MQTT topics and payload schemas (Quigley et al., 2009; Macenski et al., 2022; OASIS, 2014; OASIS, 2019). Unlike ROS, there are no framework-specific package or graph-runtime requirements that must be adopted before a component can participate. In practice, each service can remain a small plain program with minimal boilerplate, while common MQTT connection, topic, and serialization code is

centralized in a shared library, so new modules mostly implement domain logic instead of infrastructure scaffolding. The result is modularity and composability with a smaller operational footprint and fewer moving parts to maintain on constrained robot hardware. This also allows one robot to incorporate programs written in different languages, so long as the runtime is supported by the hardware and the process can communicate through MQTT.

3.3 Negotiator Processes

Before examining the individual negotiator processes, it is important to mention the top-level launcher process. Rather than starting each service manually, the launcher reads the robot configuration, starts the enabled Pebble services as child processes, republishes a retained capabilities payload to the robot-local MQTT broker, and forwards structured runtime logs onto MQTT for inspection by remote tools. It also supervises those child services during operation, restarting them after unexpected exits and enforcing a controlled shutdown sequence when the robot is stopped. As a result, each service remains narrowly focused on one resource or task, with startup order, supervision, logging, and service recovery handled centrally.

In our implementation, a **`serial_mcu_bridge`** process exclusively owns the MCU USB serial endpoint and maps MQTT command topics (incoming MCU peripheral data, like motor or light commands) to serial frames, while publishing parsed telemetry back to outgoing topics. The result is that autonomy scripts, incoming web controls, and supervisory services no longer compete for serial access, and they no longer require knowledge of serial framing details. They only need to publish or subscribe to known topics, which reduces both integration complexity and risk of race conditions at the hardware boundary.

The same principle is applied to single-owner media resources (camera and microphone). An **`av_daemon`** process is responsible for owning those device endpoints and exposing shared outputs through GStreamer shared-memory transport (GStreamer, 2026). MQTT media publisher scripts then consume those shared streams instead of opening `/dev/video*` or ALSA devices directly. This allows multiple consumers (for example, remote video publishing and local autonomy perception) to coexist without direct contention for exclusive devices, which was previously a common failure mode when two processes attempted to access the same camera or audio endpoint simultaneously (Linux Media Documentation, 2026; Linux Media Documentation, 2026; ALSA Project, 2026).

Another benefit of this architecture is clean separation of local and remote concerns. Local services interact only with the local broker. A dedicated **`mqtt_bridge`** then handles all off-robot broker negotiation, including heartbeat publication and a selective mirroring policy from the local broker to a remote broker. This setup encapsulates all remote MQTT logic into one place, rather than duplicating that logic across every subsystem. In practice, this also simplifies operational changes: updating remote broker hostnames, credentials, or TLS behavior becomes a single-service configuration change instead of a fleet-wide code edit across multiple scripts.

The same pattern is extended to operator tooling through the **`soundboard_handler`** service. Rather than allowing arbitrary processes to manage local audio playback directly, this service subscribes to a local MQTT soundboard command topic, scans a configured directory of .wav files, and publishes retained file-list and playback-status topics for the web interface. It then owns the actual playback subprocess, so operators can trigger or stop sounds through the same topic contract used elsewhere in Pebble while the robot retains one controlled point of access to local audio playback.

A similar supervisory role is performed by **`autonomy_manager`**. This service scans the repository's autonomy scripts, publishes retained metadata describing the available autonomy files and their configurable arguments, and accepts local MQTT commands to start or stop one autonomy behavior at a time. It also manages script dependencies and publishes retained runtime status, allowing operator interfaces to discover what autonomy behaviors exist, inspect their effective configuration, and observe which processes are currently running without requiring direct process management on the robot.

3.4 MQTT Standard

The MQTT standard is the contract that allows heterogeneous Pebble components to participate in a shared control architecture without requiring a common programming language, operating system, or hardware abstraction framework. In Pebble, every component is identified by a common base topic of the form:

```
{system}/{type}/{id}
```

Operational topics are divided into "incoming" commands and "outgoing" telemetry, status, and discovery messages:

```
{system}/{type}/{id}/{incoming|outgoing}/{metric}
```

For example, a robot named "goob" in the "pebble" system would use topics such as:

1. `pebble/robots/goob/incoming/drive-values`
2. `pebble/robots/goob/outgoing/charge-status`

This naming scheme makes the communication surface predictable enough that Linux robots, MCU-only robots, operator interfaces, and autonomy scripts can all participate in the same network without per-robot integration logic.

Pebble intentionally separates control topics from state topics. Incoming topics express operator or autonomy intent, such as drive commands, light commands, media enable requests, or soundboard and autonomy control. Outgoing topics express heartbeat, charging state, logs, discovery information, and higher-level outputs such as odometry or AprilTag detections. Most Pebble payloads are JSON. Lightweight commands may be sent either as a direct object or wrapped beneath a "value" field, which keeps the topic contract usable across both firmware-scale and Linux-scale implementations.

The standard also supports discovery and supervision through retained topics. At startup, the launcher publishes a retained `outgoing/capabilities` payload that describes the robot's available controls and topic names. Other retained topics, such as `soundboard-files`, `soundboard-status`, `autonomy-files`, and `autonomy-status`, allow late-joining operator interfaces to reconstruct the robot's current state without requiring out-of-band discovery logic. In practice, this retained-topic layer is what allows one web interface to adapt to multiple robots with different capabilities while preserving one shared MQTT contract.

Below are the different families of topics incorporated in the MQTT standard and their purposes. The full MQTT standard definition is available in the appendix.

Table 3.1: MQTT topic families used in the Pebble standard.

Topic Family	Example Topic	Purpose
Motion control	`.../incoming/drive-values`	Drive requests from teleop or autonomy
Lighting control	`.../incoming/lights-solid`	LED state control

Media control	`.../incoming/front-camera`	Start/stop remote video publishing
Audio control	`.../incoming/audio`	Start or stop MQTT audio publisher or receiver
Supervisor flags	`.../incoming/flags/mqtt-video`	Retained control over mirroring, media, reboot, and update actions
Heartbeat	`.../outgoing/online`	Liveness monitoring and basic latency estimation
Discovery	`.../outgoing/capabilities`	Advertises supported controls and topic names
Logs	`.../outgoing/logs`	Structured runtime logs for operator tooling
Device state	`.../outgoing/charging-status`	Reports physical state from robot hardware
Media	`.../outgoing/front-camera`	Remote video stream using keyframes and delta frames

Autonomy/perception	.../outgoing/wheel-odometry	Higher-level outputs shared with the UI or other scripts

3.5 Video and Audio Over MQTT

The current MQTT media path is split between local capture and direct broker publishing. The av_daemon service is the sole camera and microphone owner, and continuously exports raw BGR video and PCM16LE audio to local GStreamer shared-memory sockets. The mqtt_bridge service supervises media rather than carrying the media payloads itself by watching the retained .../incoming/flags/mqtt-video and .../incoming/flags/mqtt-audio topics plus the .../incoming/front-camera and .../incoming/audio command topics, and starts or stops the video publisher and audio publisher/receiver subprocesses when streaming is enabled and requested.

Video is published by the camera_publisher service launched by the mqtt_bridge service, which reads from the shared video socket, optionally rotates/resizes frames, and sends .../outgoing/front-camera messages as JSON containing a frame id, timestamp, and a base64-encoded zlib-compressed JPEG. It sends periodic full keyframes with delta frames in between, and the web UI reconstructs those frames server-side before relaying them to the browser. Audio is published by the audio_publisher service also launched by the mqtt_bridge service, which reads the shared audio socket, chunks PCM16LE audio, and publishes .../outgoing/audio as binary PBAT packets with sample rate, channel count, sequence number, and timestamp in the header. For talk-back, the web UI encodes browser microphone PCM into the same PBAT format and publishes it to .../incoming/audio-stream. That topic is consumed directly by the audio_receiver service rather than mirrored through mqtt_bridge, and the receiver plays it to ALSA with a small packet-loss concealment window. Having high-rate media be handled by independent services keeps that data from clogging the local MQTT broker.

3.6 Preliminary Benefits and Tradeoffs

This local-first model also improves robot behavior on unstable networks. If WAN connectivity drops, local control loops can still function through localhost MQTT. In previous

centralized/cloud-forwarded patterns, remote-path dependency could delay or block local actions even when all physical hardware was healthy.

From a software engineering perspective, this approach also improves testability and fault isolation. Each negotiator process has a narrow interface (topic inputs and outputs), which makes unit and integration tests easier to define. Service restarts become less disruptive because failure domains are smaller (e.g., a crash in video logic does not imply failure in motor control). A lightweight launcher and systemd supervision can then enforce restart policies per process while keeping deployment as a single managed unit.

Using Python brings a number of benefits, too. Since the Pebble system is written in Python, this architecture is very portable. The libraries used in this project have excellent cross-platform support. System-specific commands and device paths can be set within the configuration file, and `launcher` management is not restricted to systemd.

There are tradeoffs. Introducing a local broker and multiple daemons increases process count and requires strict topic and payload governance. This governance is manageable by incorporating an MQTT topic structure, and the increased process count is worth the gains in modularity, offline capability, and maintainability. For Pebble-class robots specifically, the local MQTT with negotiators pattern has created a reliable foundation for future features, especially autonomy, that must share hardware resources safely without reintroducing control bottlenecks. However, the asynchronous nature of MQTT means that certain robotic pipelines, especially AI-driven ones that expect synchronous telemetry, are not suited to the negotiator process architecture. Such systems would be better off incorporating the `mqtt_bridge` process and an intermediary process to relay commands and telemetry over a local MQTT broker, keeping the synchronous core intact while also allowing for MQTT-based communication.

Chapter 4: Experimental Analysis

This section evaluates Pebble through two complementary evidence paths. First, archived MQTT traffic was used to characterize steady-state bandwidth, heartbeat behavior, and the relative cost of control traffic versus media traffic during real operation. Second, controlled router-based impairment tests were performed on an isolated OpenWrt network to evaluate how Pebble media and heartbeat traffic behave under constrained bandwidth and packet loss. Together, these analyses provide both a long-horizon view of normal system behavior and a short-horizon view of failure modes under deliberately degraded network conditions.

4.1 Archived MQTT Traffic

The archived data itself supports one of the most important practical claims in the project: Pebble's shared MQTT standard is not limited to simple heartbeat or drive topics. The same archive contains retained discovery topics, structured logs, soundboard state, autonomy state, and higher-level perception outputs. That demonstrates that the architecture is already being used as a general-purpose robot communications fabric rather than a simple bridge for teleoperation.

Archived MQTT traffic was collected in MongoDB and separated into active sessions and idle-online periods. This made it possible to distinguish routine background traffic, such as heartbeats and retained-state publications, from session-scoped traffic generated by teleoperation, video, audio, or autonomy usage. The archive analysis shows that control and status traffic remain small compared with media traffic, while the media cost itself is highly asymmetric: the current MQTT video path is relatively bandwidth-efficient in the collected captures, but the current MQTT audio path is much more expensive because it is not using a modern compressed codec.

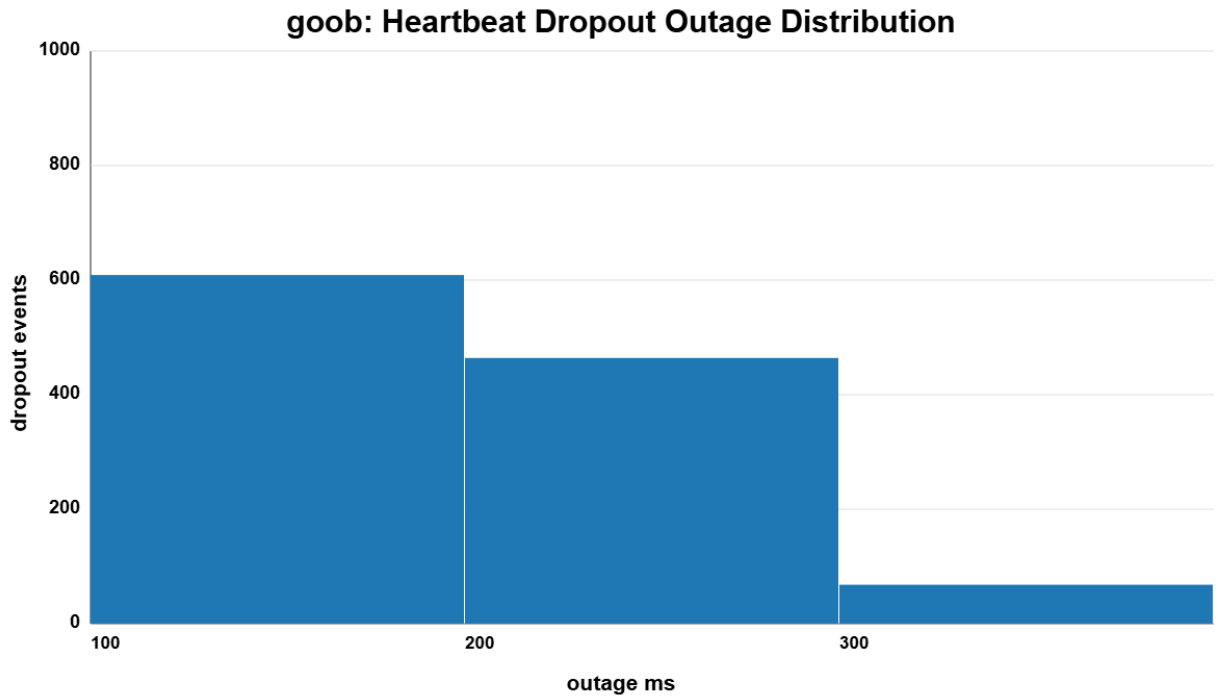

Figure 4.1: Goob heartbeat outage duration distribution from archived MQTT traffic.

The Goob and Fred traces showed near-identical heartbeat-outage distributions, so only Goob's data is discussed here for brevity. The chart above plots heartbeat outage events against outage duration. Most recorded outages clustered below 300 ms. Within the archived operating conditions represented by this dataset, that pattern indicates that heartbeat interruptions were usually brief. Because these observations come from archived field operation rather than controlled fault injection, the data do not isolate root cause, but the short durations are more consistent with transient network disruption than with sustained Pebble-side failure.

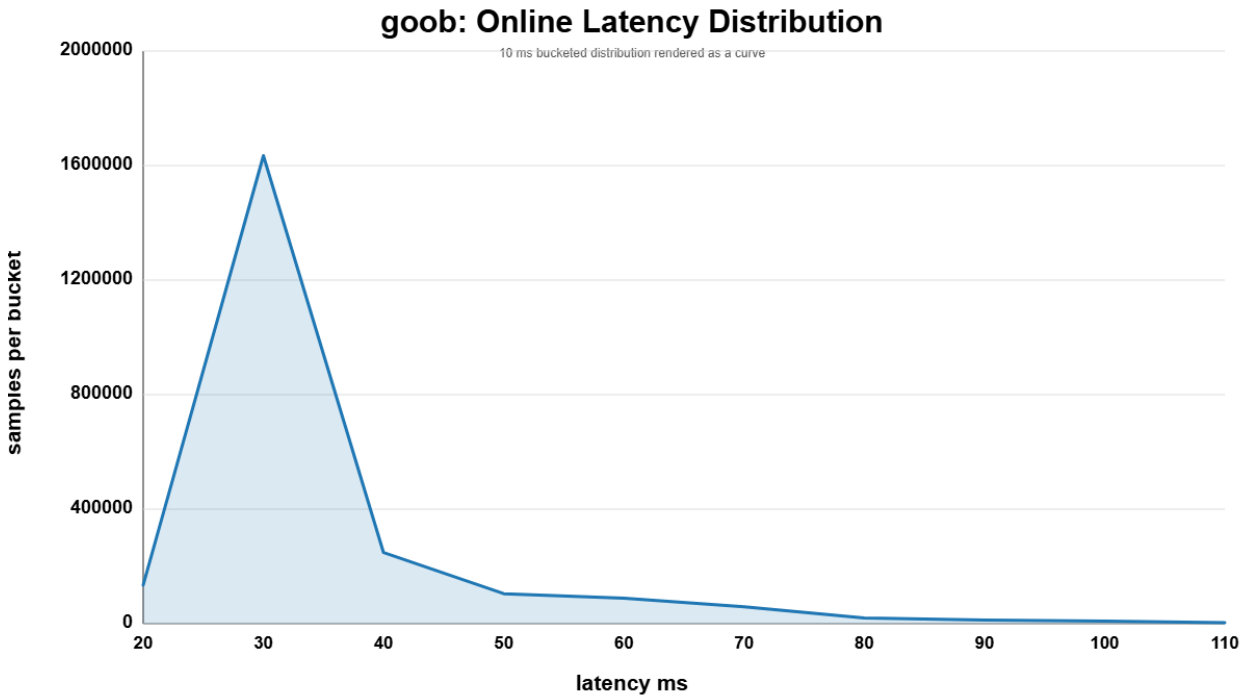

Figure 4.2: Goob latency distribution from archived MQTT traffic.

Above is Goob's latency distribution. Most latency is 20-40 ms, and given the network path (Aurora Robotics Lab network across the internet to a Fairbanks-located remote broker and back), the majority of this latency comes from the network. Fred's latency distribution is identical.

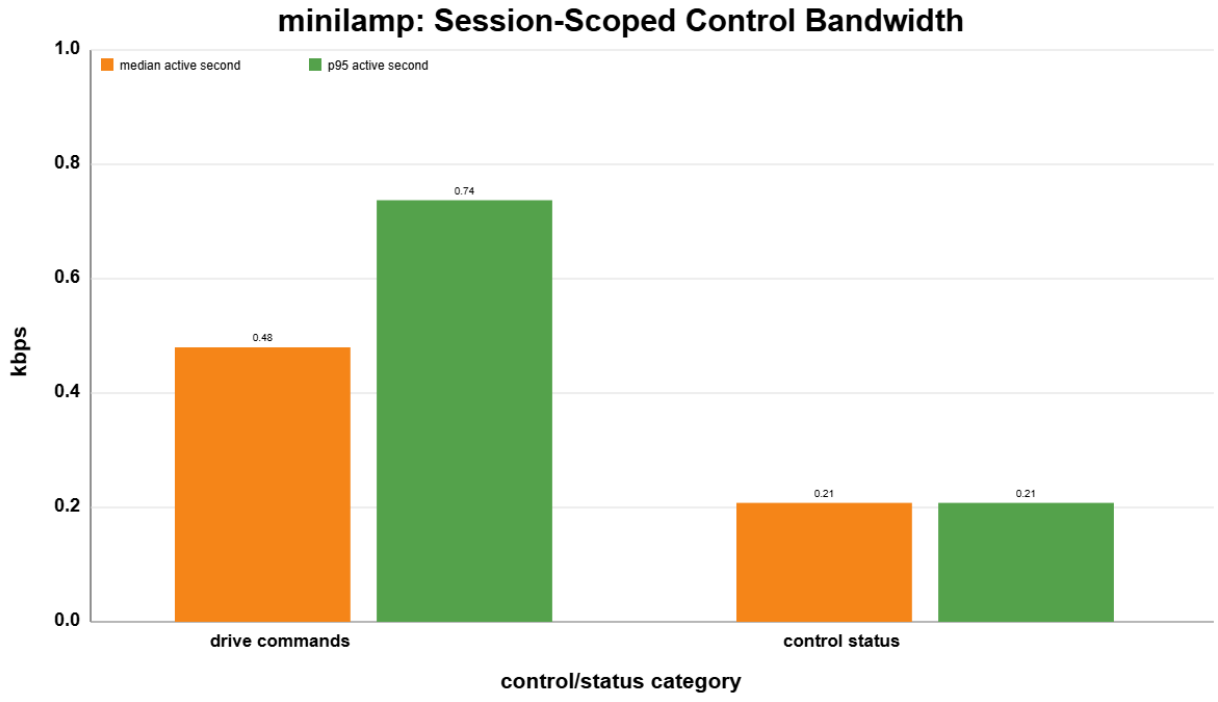

Figure 4.3: MiniLAMP bandwidth measurements from archived MQTT traffic.

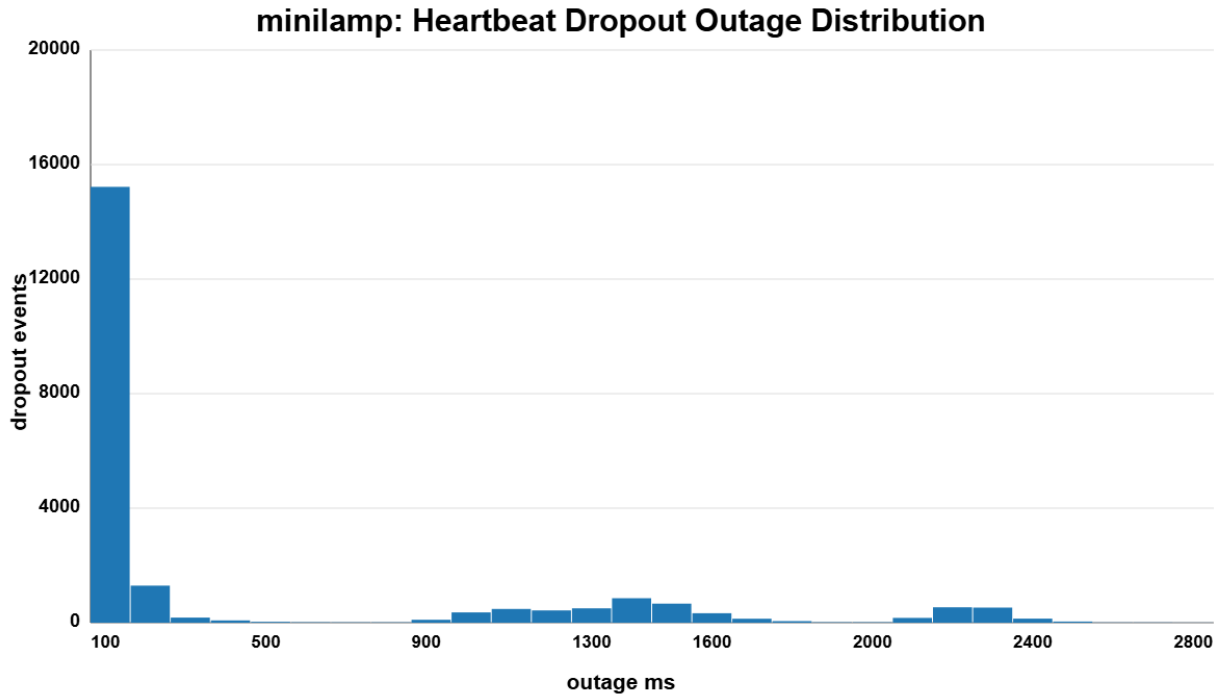

Figure 4.4: MiniLAMP heartbeat outage duration distribution from archived MQTT traffic.

Moving onto the MCU-based robots, the results were similar between MiniLAMP and BenJr, as they were for Goob and Fred, since they are also very similar architectures. Above are the bandwidth measurements for MiniLAMP. All robots shared identical drive and control bandwidths since the packet structure across each is identicalAbove is MiniLAMP's heartbeat dropout outage distribution. Similar to Goob's results, the majority of dropouts are for very short durations (<300ms). Slight bumps at one and two seconds are visible, which might be due to the Starlink connection. None of these dropouts were observed during operation, and there was no effect on operability.

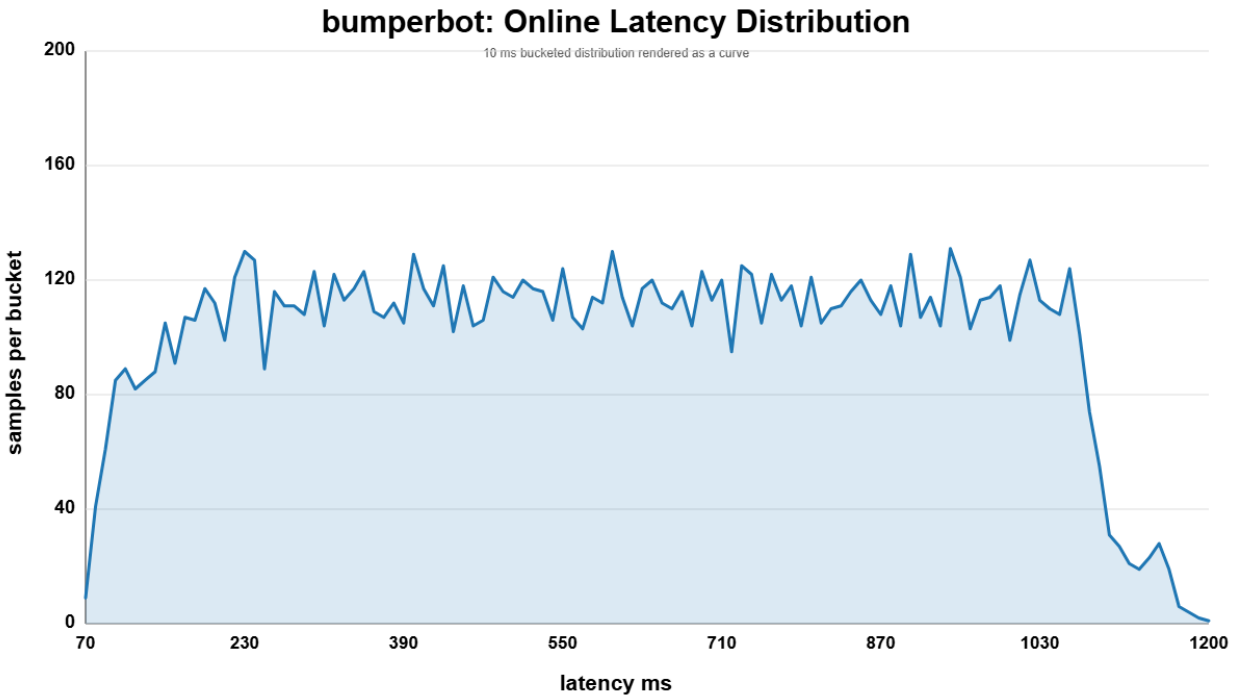

Figure 4.5: BenJr latency distribution from archived MQTT traffic.

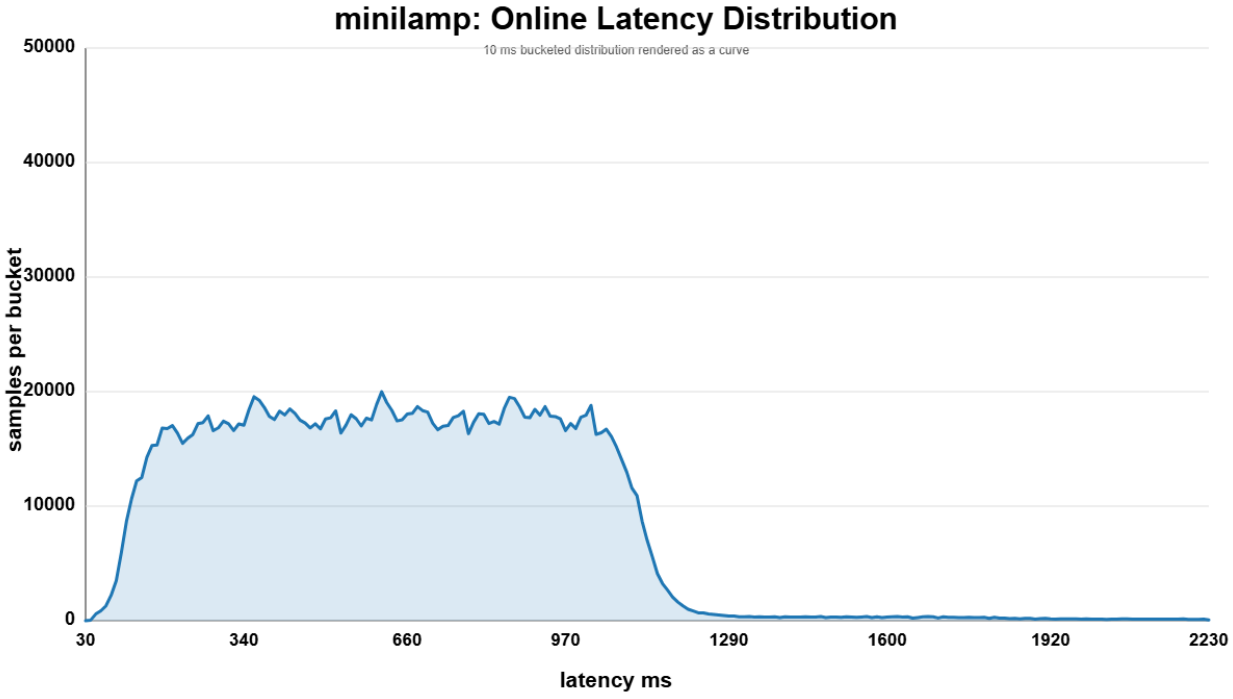

Figure 4.6: MiniLAMP latency distribution from archived MQTT traffic.

Above is the latency distribution for both BenJr, previously known as BumperBot (the name the database stores), and MiniLAMP. Interestingly, for each, the latency is consistently between ~100 ms and ~1100 ms. It is possible the MCU-based robots inherently have a greater latency

range due to simultaneously handling all logic on far fewer threads than Pi-based systems. Regardless, this range still resulted in functional robots. MiniLAMP, despite going over a Starlink connection, has impressive results.

Fred and Goob are the only robots that support video and audio. Below is a table of bandwidth measurements from the archived data for each.

Table 4.1: Archived media bandwidth measurements for Goob and Fred.

Robot	Video median active-second kbps	Video p95 kbps	Audio median active-second kbps	Audio p95 kbps
Fred	172.46	283.99	265.60	270.91
Goob	181.18	588.79	521.60	532.03

These archived results also support a cautious comparison against common RTC media ranges. In the collected captures, Pebble's MQTT video publishers operated well below common VGA WebRTC bandwidth guidance, but the current audio path consumed far more bandwidth than Opus-based audio in WebRTC-style systems. That does not prove that MQTT is categorically superior to WebRTC; rather, it shows that Pebble's current video transport is efficient at its chosen quality level while the current audio transport is the clearest remaining bandwidth weakness.

4.2 Router Impairment Tests

The second evidence path was a set of controlled OpenWrt impairment experiments. In these trials, only the robot uplink MQTT path to the remote broker was shaped, using bandwidth caps, packet loss, and combined conditions. For Fred and Goob, each trial applied one impairment condition, observed heartbeat traffic, enabled MQTT video, disabled video and observed residual traffic, enabled MQTT audio, disabled audio and observed residual traffic, and, where available, issued a soundboard playback command. This produced a repeatable dataset for comparing media robustness across impairment levels without issuing unsafe drive commands during testing.

The trials went as follows:

1. **base-start:** Control, no impairment

2. **Bandwidth impairment** in five steps: 1024k, 512k, 256k, 128k, and 64k limits
3. **Packet loss** in four steps: 1%, 2%, 5%, and 10% loss
4. **Combined trials:** 256k+1%, 128k+2%, and 64k+5%
5. **base-end:** Control, no impairment

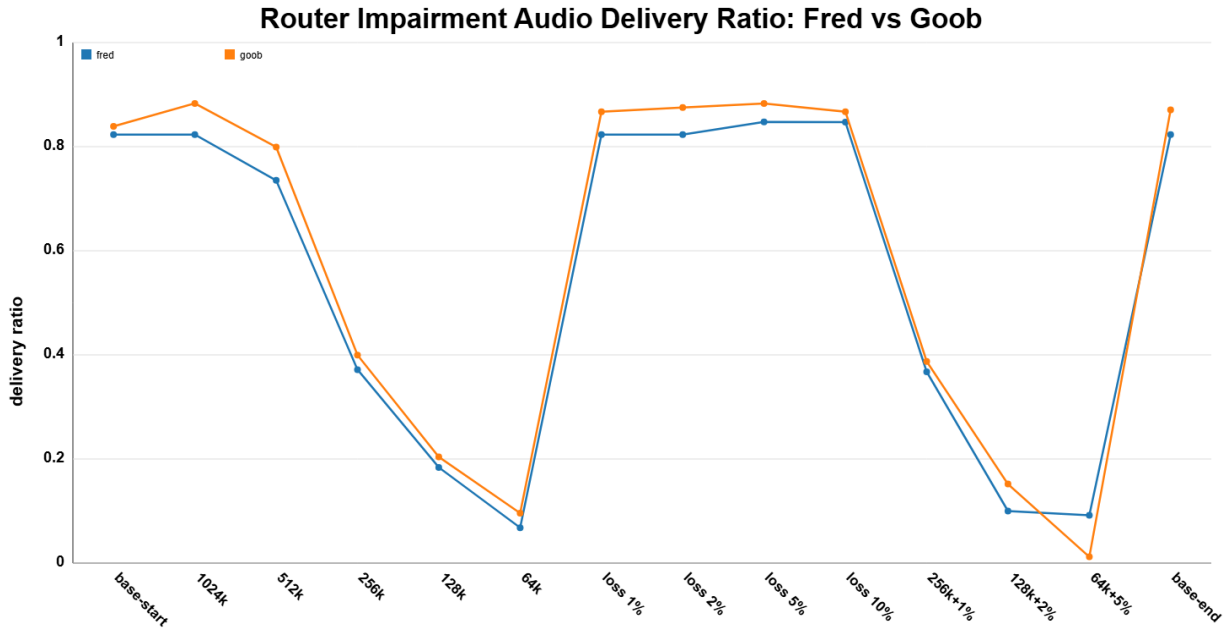

Figure 4.7: Audio delivery over the router impairment trials.

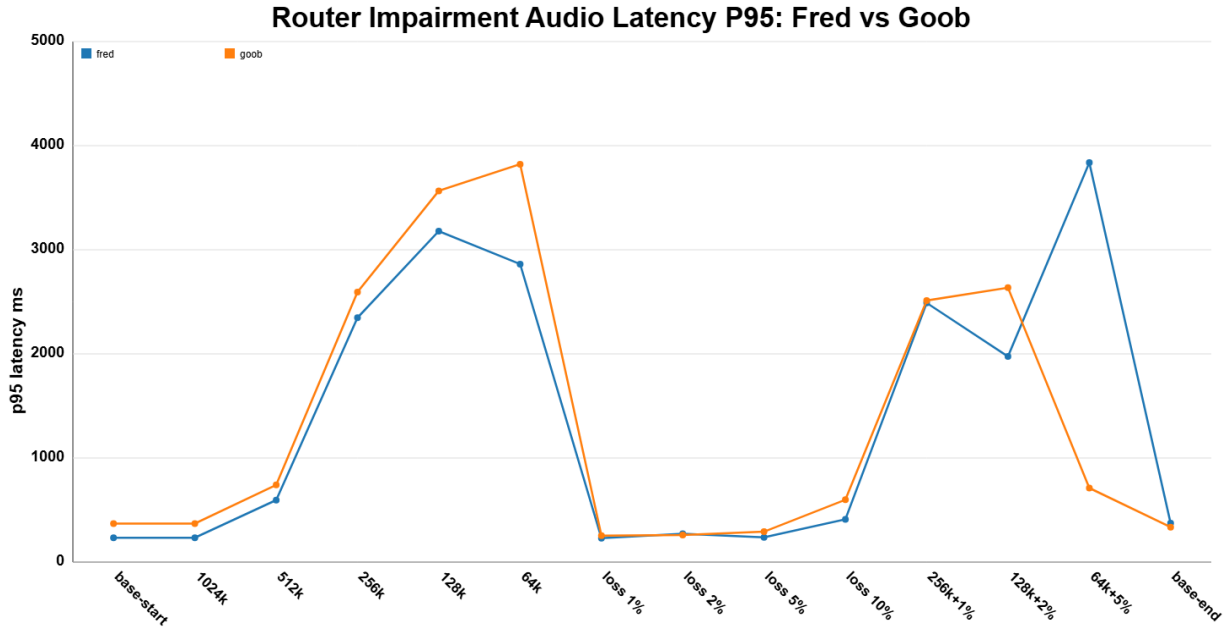

Figure 4.8: Audio latency over the router impairment trials.

The above graphs show audio delivery and latency over the course of the trial. In these runs, delivery saturated near ~80%, reflecting inherent losses in the test configuration. As expected, both lower bandwidth caps and higher packet-loss settings reduced performance, but the bandwidth cap was the main limiter, especially for audio. In the two-robot run, audio delivery and latency remained comparatively stable at 1024 kbit/s, degraded noticeably at 512 kbit/s, and dropped sharply by 256 kbit/s. Video followed the same threshold pattern on both robots. In practical terms, the data indicate that this MQTT media path can operate on moderate-bandwidth links, but performance margin decreases rapidly once the uplink is constrained below roughly 512 kbit/s.

Chapter 5: Conclusion

Pebble demonstrates that an MQTT-based architecture can provide a practical and reusable communication layer for heterogeneous multi-robot systems. Across the implemented platforms, one shared topic contract was sufficient to support teleoperation, retained state publication, structured logs, autonomy control, and media transport, even though the robots themselves differed substantially in compute capability and body plan. Linux-capable robots used a robot-local broker plus narrowly scoped daemon processes to manage exclusive resources, while constrained robots participated directly through the same remote MQTT standard with a reduced feature set. The core architectural result is that interoperability was achieved without requiring every robot to adopt the same robotics middleware stack.

The collected data also clarifies the strengths and limits of the approach. The robot-local broker and daemon ownership model improved modularity, restart behavior, and fault isolation, and the archived traffic confirms that the same MQTT standard is already carrying more than just teleoperation commands. At the same time, the architecture is not intended as a hard real-time fieldbus, the current video path remains weaker than mature media stacks in startup delay and stability, and the current audio path is significantly more bandwidth-hungry than compressed RTC audio (Valin et al., 2012). Even with those limitations, the results support the project's central claim: **for many practical robotics applications, MQTT can serve as a viable foundation for interoperable communication across a heterogeneous robot fleet.**

5.1 Future Work

5.1.1 Improved MQTT Media

The most immediate future work is in media transport. The current MQTT audio path is the clearest bandwidth weakness in the system because it uses an uncompressed packet format. Replacing that path with a compressed codec such as Opus would substantially reduce bandwidth cost and create a fairer comparison against WebRTC-style systems (Valin et al., 2012). The MQTT video path would also benefit from further refinement, especially around startup delay, frame rate stability, and performance under constrained bandwidth. The video has a couple known issues, too: It currently base64 encodes the data before sending it off, which is entirely unnecessary in hindsight, and the delta frame logic incorrectly calculates deltas from the actual camera frame pixel differences instead of the receiver's reconstructed differences, leading

to visual artifacts, since deltas are limited to changes of -128 to +127. For the best improvements, it may even be feasible to route an existing mature media stack over MQTT directly, bypassing the need for custom delta video logic and greatly improving video and audio quality and bandwidth.

5.1.2 Abridged Topic Names and Payloads

A useful future optimization would be support for abridged topic names on bandwidth-constrained links. In MQTT, the full topic string is transmitted with every publish, so a verbose hierarchy such as `{system}/{type}/{id}/outgoing/...` adds nontrivial overhead when messages are small or sent at high frequency (OASIS, 2014; OASIS, 2019). A compact alias scheme, for example shortening frequently used path segments or replacing full topic names with negotiated symbolic identifiers, could reduce per-message bandwidth without changing the logical publish/subscribe model. This would be especially beneficial for constrained radio links, small telemetry packets, and high-rate status traffic, where topic overhead can make up a meaningful fraction of total transmitted bytes. MQTT 5 already introduces Topic Alias for a related purpose, although the implementation described in this report intentionally remains MQTT 3.1.1-compatible (OASIS, 2019). The main tradeoff is reduced human readability, so a practical design would pair abridged transport names with a retained discovery mapping that preserves the full semantic topic structure for tools and operators.

In theory, it should also be possible to combine commands into a single network packet. Some network protocols allow a single message to address multiple channels or topics simultaneously, but unfortunately, MQTT on its own does not. Each PUBLISH packet targets exactly one topic, and multi-recipient delivery is handled only by the broker forwarding that message to matching subscribers. This constraint introduces a tradeoff between efficiency and granularity: tightly coupled updates, such as drive and lighting commands, must either be sent as multiple messages or combined into a single higher-level command on a shared topic. While the latter preserves synchronization and reduces network overhead, it weakens the fine-grained topic structure that makes MQTT flexible. So, future work could also explore extensions or conventions that enable grouped or atomic multi-topic updates while retaining MQTT's lightweight publish/subscribe model.

5.1.3 System-Local MQTT

Our solution incorporates an MQTT broker both in the cloud and local to each robot. As previously described, this allows robot-local systems to share resources that would otherwise belong to a single process. What wasn't discussed in-depth was the extra abstraction gained from keeping some information local to the robot's MQTT broker (e.g., internally-sent drive commands from autonomy scripts). Doing this also saves on bandwidth and decreases network reliance. For systems operating on the same LAN network, an additional system-local MQTT broker (at the LAN level) could have great benefits. This broker would act as the local system's manager for inter-robot communication, with the cloud broker taking a more abstracted role, potentially acting as a multi-system communications manager.

Our solution does not incorporate system-local MQTT, but could benefit from it. Currently, if a robot A on the same LAN as robot B wishes to communicate with robot B, it must either communicate via the cloud-based MQTT broker or know the local IP address of robot B to publish/subscribe to its broker directly. If *another* MQTT broker is introduced within the LAN network, hosted on something like a Pi, each robot only has to keep track of that one device's local IP address.

A system-local MQTT broker does not have to be its own unit: it can be one of the robots. Both Fred and Goob in our architecture incorporate a Pi-style computer and already host robot-local MQTT brokers, so one of these robots can act as the system-local MQTT broker, too. This still requires all devices on the local network to remember the local IP address of the Pi, or for the router to run a small DNS server to create a hostname which resolves to the local IP address of the Pi. Each of these solutions introduces extra complexity, though, which may outweigh the benefits for some architectures with few systems. For architectures with many systems, this extra complexity is probably worth considering.

References

(ALSA Project, 2026) ALSA Project, "PCM (digital audio) interface." ALSA project - the C library reference. https://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html.

(Atmoko & Yang, 2018) R. A. Atmoko and D. Yang, "Online Monitoring & Controlling Industrial Arm Robot Using MQTT Protocol," 2018 IEEE International Conference on Robotics, Biomimetics, and Intelligent Computational Systems (Robionetics), Bandung, Indonesia, 2018, pp. 12-16, doi: https://doi.org/10.1109/ROBIONETICS.2018.8674672.

(Campos et al., 2021) C. Campos, R. Elvira, J. J. Gomez Rodriguez, J. M. M. Montiel and J. D. Tardos, "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial, and Multimap SLAM," IEEE Transactions on Robotics, 37(6), 2021, pp. 1874-1890, doi: https://doi.org/10.1109/TRO.2021.3075644.

(Das et al., 2025) A. Das, S. Bose and R. K. Jain, "MQTT Protocol Based Real-Time Operation in Mobile Robotic System Using IoT Technique," 2025 IEEE International Conference on Computer, Electronics, Electrical Engineering & their Applications (IC2E3), Srinagar Garhwal, India, 2025, pp. 1-6, doi: https://doi.org/10.1109/IC2E365635.2025.11167565.

(Gielis et al., 2022) Gielis, J., Shankar, A., & Prorok, A. (2022). A Critical Review of Communications in Multi-Robot Systems. IEEE Access, 10, 2693 to 2711. https://doi.org/10.48550/arXiv.2206.09484.

(González Bonifaz et al., 2018) G. B. D. Efraín, V. C. A. del Carmen, E. C. L. Fernando and L. M. D. Cesar, "Implementation of an IoT Architecture based on MQTT for a Multi-Robot System," 2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM), Cuenca, Ecuador, 2018, pp. 1-6, doi: https://doi.org/10.1109/ETCM.2018.8580321.

(Greer et al., 2019) C. Greer, M. J. Burns, D. Wollman and E. Griffor, "Cyber-Physical Systems and Internet of Things," NIST Special Publication 1900-202, 2019, doi: https://doi.org/10.6028/NIST.SP.1900-202.

(GStreamer, 2026) GStreamer, "shmsink" and "shmsrc." GStreamer documentation. https://gstreamer.freedesktop.org/documentation/shm/shmsink.html; https://gstreamer.freedesktop.org/documentation/shm/shmsrc.html.

(Hu et al., 2012) G. Hu, W. P. Tay and Y. Wen, "Cloud robotics: architecture, challenges and applications," IEEE Network, 26(3), 2012, pp. 21-28, doi: https://doi.org/10.1109/MNET.2012.6201212.

(Israel et al., 2020) Israel, D. J., Mauldin, K. D., Roberts, C. J., Mitchell, J. W., Pulkkinen, A. A., Cooper, L. V. D., et al. (2020). LunaNet: A Flexible and Extensible Lunar Exploration Communications and Navigation Infrastructure. IEEE Aerospace Conference. https://ntrs.nasa.gov/api/citations/20200001555/downloads/20200001555.pdf.

(Ji et al., 2021) Q. Ji, C. Dai, C. Hou and X. Li, "Real-time embedded object detection and tracking system in Zynq SoC," EURASIP Journal on Image and Video Processing, 2021, article 21, doi: https://doi.org/10.1186/s13640-021-00561-7.

(Kehoe et al., 2015) B. Kehoe, S. Patil, P. Abbeel and K. Goldberg, "A Survey of Research on Cloud Robotics and Automation," IEEE Transactions on Automation Science and Engineering, 12(2), 2015, pp. 398-409, doi: https://doi.org/10.1109/TASE.2014.2376492.

(Kolling et al., 2016) Kolling, A., Walker, P., Chakraborty, N., Sycara, K., & Lewis, M. (2016). Human Interaction With Robot Swarms: A Survey. IEEE Transactions on Human-Machine Systems, 46(1), 9 to 26. https://doi.org/10.1109/THMS.2015.2480801.

(Koodtalang et al., 2022) W. Koodtalang, S. Mangkalajan, T. Supsatien, P. Thammalangka, W. Darawan and T. Sangsuwan, "Monitoring and Controlling the Multiple Mobile Robots in Outdoor Environments Based on IoT," 2022 22nd International Conference on Control, Automation and Systems (ICCAS), Jeju, Korea, Republic of, 2022, pp. 1447-1452, doi: https://doi.org/10.23919/ICCAS55662.2022.10003713.

(Linux Media Documentation, 2026) Linux Media Documentation, "Application Priority." The Linux Kernel documentation. https://docs.kernel.org/userspace-api/media/v4l/common.html#application-priority.

(Linux Media Documentation, 2026) Linux Media Documentation, "Opening and Closing Devices." The Linux Kernel documentation. https://docs.kernel.org/userspace-api/media/v4l/open.html.

(Macenski et al., 2022) S. Macenski, T. Foote, B. Gerkey, C. Lalancette and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," Science Robotics, 7(66), 2022, eabm6074, doi: https://doi.org/10.1126/scirobotics.abm6074.

(Matthee et al., 2024) J. Matthee, K. Uren, G. van Schoor and C. van Daalen, "Predicting the performance of ORB-SLAM3 on embedded platforms," South African Computer Journal, 36(2), 2024, doi: https://doi.org/10.18489/sacj.v36i2/20099.

(Mukhandi et al., 2019) M. Mukhandi, D. Portugal, S. Pereira and M. S. Couceiro, "A novel solution for securing robot communications based on the MQTT protocol and ROS," 2019 IEEE/SICE International Symposium on System Integration (SII), Paris, France, 2019, pp. 608-613, doi: https://doi.org/10.1109/SII.2019.8700390.

(OASIS, 2014) OASIS, "MQTT Version 3.1.1," OASIS Standard, 29 October 2014. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf.

(OASIS, 2019) OASIS, "MQTT Version 5.0," OASIS Standard, 07 March 2019. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.pdf.

(Quigley et al., 2009) M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Y. Ng, "ROS: an open-source Robot Operating System," ICRA Workshop on Open Source Software, Kobe, Japan, 2009. https://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.

(Valin et al., 2012) J.-M. Valin, K. Vos and T. Terriberry, "Definition of the Opus Audio Codec," RFC 6716, IETF, 2012. https://www.rfc-editor.org/rfc/rfc6716.

(Zegarra et al., 2024) Cuadros Zegarra, E., Barrios Aranibar, D., & Cardinale, Y. (2024). IoRT-Based Middleware for Heterogeneous Multi-Robot Systems. Journal of Sensor and Actuator Networks, 13(6), 87. https://doi.org/10.3390/jsan13060087.

Appendix A: MQTT Standard and Repository Material

GitHub Repository

The project's GitHub repository is at https://github.com/acmattson3/pebble. Due to secrets accidentally published to the original repository's commit history, a clean version of the repository was ported to the new location. Commit history was lost, but this also allowed for changing the repository name from the legacy name, PebbleBot, to the current one, Pebble.

MQTT Standard

By Andrew Mattson

Derived from Symergy's MQTT standard by Anand Egan, Andrew Mattson, Kylie Lambries, and Rylan Clavell

1. Scope

This document defines a general MQTT topic and payload convention for heterogeneous component systems. A component may be a robot, satellite, tower, fixed station, operator console, person-carried device, or any other uniquely identified participant in the system.

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in RFC 2119. When those words are not capitalized, they are used in their ordinary English sense rather than as normative terms.

This standard is intended to be reusable beyond the exact setup implemented in this repository. To keep that distinction clear:

- unqualified sections describe the general standard
- any section or table explicitly labeled `Example implementation in this repository` describes the current codebase only
- example topic names, payloads, and schemas are illustrative, not mandatory for future users

The standard topic base is:

```
{system}/{type}/{id}
```

The default full topic form is:

```
{system}/{type}/{id}/{incoming|outgoing}/{metric}
```

Any component-specific topic defined by this standard MUST follow that topic shape. Additional path segments are permitted only inside the `{metric}` portion.

Examples:

```
luminsim/robots/astra1/incoming/drive-values
luminsim/satellites/lunanet-sat-1/outgoing/status
luminsim/towers/tower-3/outgoing/power
luminsim/people/operator-1/outgoing/location
```

Important notes:

1. `metric` MAY contain additional path segments, such as `flags/remote-mirror` or `sensors/imu`
2. not every component publishes every topic family
3. future users are free to omit capabilities, video, autonomy, AprilTag, or any other optional topic family

2. MQTT Version

This topic standard is intended to be usable over either MQTT 3.1.1 or MQTT 5, provided the implementation preserves the same publish/subscribe semantics, retained-message behavior, and payload contracts.

Example implementation in this repository

The current implementation uses MQTT 3.1.1. Shared helpers explicitly create clients with `MQTTv311`, the web interface also explicitly uses `MQTTv311`, and the current codebase does not depend on MQTT 5-specific properties.

3. Topic Model

3.1 Component Identity

Each component is identified by:

1. `system`: the namespace or deployment name
2. `type`: the component class, such as `robots`, `satellites`, `towers`, `stations`, `operators`, or `people`

3. `id`: a unique identifier within that component type

The tuple `{system}/{type}/{id}` MUST uniquely identify one component within a deployment.

3.2 Direction Semantics

By convention:

1. `incoming` means data intended to be consumed by that component
2. `outgoing` means data published by that component

This is a logical direction, not necessarily a statement about broker location. For example, an `incoming` topic may be published on a remote broker and then bridged locally.

3.3 Topic Examples

Examples:

1. `{system}/robots/{id}/incoming/drive-values`
2. `{system}/robots/{id}/outgoing/online`
3. `{system}/satellites/{id}/outgoing/status`
4. `{system}/towers/{id}/outgoing/capabilities`
5. `{system}/people/{id}/outgoing/location`

3.4 Wildcards

Examples:

1. `{system}/#` for all live system traffic
2. `{system}/+/+/outgoing/online` for all heartbeat topics
3. `{system}/robots/+/outgoing/#` for all robot-published telemetry
4. `{system}/towers/+/outgoing/status` for status of all towers

4. Message Conventions

4.1 JSON And Binary Payloads

All non-media topics MUST use JSON payloads. High-rate media topics MAY use either JSON or binary payloads.

When binary payloads are used, the encoding SHOULD be documented for that metric family.

4.2 Timestamps

This standard allows two common timestamp forms:

1. `t`: UNIX epoch milliseconds as an integer
2. `timestamp`: UNIX epoch seconds as a float

Implementations SHOULD be internally consistent. Consumers SHOULD tolerate both forms when interacting with mixed-version components.

4.3 QoS Guidance

RECOMMENDED defaults:

1. QoS 1 for commands, retained discovery topics, and low-rate telemetry where delivery matters
2. QoS 0 for high-rate media or telemetry where freshness matters more than guaranteed delivery

Consumers SHOULD tolerate either QoS 0 or 1 when practical.

4.4 Retain Guidance

RECOMMENDED defaults:

1. retained for current-state topics such as `capabilities`, `online`, or other latest-known status topics
2. non-retained for one-shot commands and high-rate streams

One-shot commands MUST NOT be retained unless the consumer explicitly implements stale-command filtering.

4.5 Command Wrappers

For command topics, both of the following patterns are acceptable:

```
{ "x": 0.1, "z": 0.2 }
{ "value": { "x": 0.1, "z": 0.2 } }
```

Similarly, boolean controls may use either a bare boolean or a wrapped form:

```
true
```

```
{ "value": true }
{ "enabled": true }
```

Future users are not required to support every wrapper form, but if a system mixes multiple producers, accepting both direct and wrapped payloads usually improves compatibility.

5. Broker Topologies

5.1 Single-Broker Deployment

The simplest deployment uses one broker for all components. In this case:

Appendix A: components publish `outgoing/...` directly

Appendix B: components subscribe to `incoming/...` directly

Appendix C: retained discovery and current-state topics are resolved by the broker alone

5.2 Multi-Broker Or Bridged Deployment

A deployment may also use multiple brokers, such as:

1. one component-local broker plus one remote broker
2. one site-local broker plus one cloud broker
3. separate brokers for low-rate control and high-rate media

In those cases:

bridge routing SHOULD be explicitly documented

loops MUST be prevented

one-shot command retention SHOULD be handled carefully

implementers SHOULD make it clear which broker is authoritative for each topic family

5.3 Example implementation in this repository

The current repository uses:

1. a component-local broker, usually `127.0.0.1:1883`
2. a remote broker configured in the runtime config

Current bridge behavior:

Table A.1: Current bridge routing behavior by source side and topic class.

Source side	Topic class	Result
local broker	`{base}/outgoing/#`	always forwarded to remote
remote broker	`{base}/incoming/#`	always forwarded to local, except `incoming/audio-stream`
remote broker	`{base}/outgoing/#`	never forwarded back into local
local broker	`{base}/incoming/flags/#`	always mirrored to remote
local broker	other `{base}/incoming/#`	mirrored only when `incoming/flags/remote-mirror` is `true`
local broker	retained `outgoing/...` topics	cached and replayed to remote after remote reconnect

The current repository also routes some media outside the local broker entirely. That is an example deployment choice, not a requirement of this standard.

6. Recommended Generic Topic Families

These topic families are reusable patterns. They are recommendations, not a complete required set.

Table A.2: Recommended generic MQTT topic families.

Topic family	Direction	Purpose	Typical retain / QoS
`{base}/incoming/<command>`	incoming	control commands for a component	non-retained, QoS 1
`{base}/incoming/flags/<flag>`	incoming	mode toggles or control gating	often retained, QoS 1

{base}/outgoing/online	outgoing	heartbeat / liveness indicator	retained, QoS 1
{base}/outgoing/capabilities	outgoing	retained discovery and control metadata	retained, QoS 1
{base}/outgoing/status	outgoing	summarized current state	often retained, QoS 1
{base}/outgoing/logs	outgoing	structured diagnostics	non-retained, QoS 0 or 1
{base}/outgoing/<telemetry>	outgoing	sensor/state telemetry	retain/QoS depend on metric
{base}/outgoing/<media>	outgoing	video/audio/media streams	non-retained, often QoS 0

Not every system needs all of these. A satellite-only system may never publish `drive-values`; a tower may never publish audio; a person-carried device may publish location and status only.

7. Discovery And Capability Advertisement

If a system publishes retained discovery data, a common envelope shape is:

```
{
  "schema": "<implementation-schema-id>",
  "t": 1710969600123,
  "value": { ... }
}
```

RECOMMENDED characteristics:

1. retained
2. QoS 1
3. published at startup, reconnect, and whenever current capabilities change

4. includes topic names for control and telemetry where those names are configurable

Future users MAY define any schema identifier and any capability structure that fits their system.

If a system publishes a current-state discovery or capability topic intended for late subscribers, that topic MUST be retained.

8. Example Implementation In This Repository

Everything in this section is an example of the current codebase. Future users are not expected to implement every one of these topics or payload families.

8.1 Example Default Topic Set

Here `{base}` means `{system}/{type}/{id}` in the current repository.

Incoming Topics Implemented Here:

Table A.3: Incoming topics implemented in the current repository.

Example topic	Current producer(s)	Current consumer(s)	Notes
`{base}/incoming/drive-values`	web UI, teleop, autonomy scripts	serial MCU bridge, wheel-odometry direction tracker	Optional; only meaningful for drive-capable components.
`{base}/incoming/lights-solid`	web UI	serial MCU bridge	Optional visual-control topic.
`{base}/incoming/lights-flash`	web UI	serial MCU bridge	Optional visual-control topic.

{base}/incoming/front-camera	web UI	MQTT bridge	Optional video start/stop request.
{base}/incoming/audio	web UI	MQTT bridge	Optional audio start/stop request.
{base}/incoming/audio-stream	remote operator audio sender	audio receiver	Optional speaker downlink topic.
{base}/incoming/soundboard-command	web UI	soundboard handler	Optional local playback command topic.
{base}/incoming/autonomy-command	web UI	autonomy manager	Optional autonomy control topic.
{base}/incoming/flags/remote-mirror	web UI, tooling, tests	MQTT bridge	Optional bridge-mode flag.
{base}/incoming/flags/mqtt-audio	web UI	MQTT bridge	Optional media gating flag.
{base}/incoming/flags/mqtt-video	web UI	MQTT bridge	Optional media gating flag.

{base}/incoming/flags/reboot	web UI	MQTT bridge	Optional one-shot reboot control.
{base}/incoming/flags/git-pull	web UI	MQTT bridge	Optional one-shot repo-update control.

Outgoing Topics Implemented Here:

Table A.4: Outgoing topics implemented in the current repository.

Example topic	Current producer(s)	Current consumer(s)	Notes
{base}/outgoing/online	MQTT bridge; some firmware components may publish directly	web UI, replay, analysis	Current heartbeat topic in this repository.
{base}/outgoing/capabilities	launcher	web UI, replay, tooling	Retained capability advertisement.
{base}/outgoing/logs	launcher	web UI, replay, analysis	Structured runtime diagnostics.
{base}/outgoing/touch-sensors	serial MCU bridge when enabled	web UI, analysis	Optional ADC touch telemetry.

{base}/outgoing/charging-status	serial MCU bridge; some MCU firmware	web UI, replay	Optional charging boolean.
{base}/outgoing/charging-level	some MCU firmware	web UI, replay	Optional battery/voltage metric.
{base}/outgoing/front-camera	camera publisher	web UI, replay, receivers	Optional video stream.
{base}/outgoing/audio	audio publisher	web UI, replay, receivers	Optional uplink audio stream.
{base}/outgoing/soundboard-files	soundboard handler	web UI	Optional sound catalog.
{base}/outgoing/soundboard-status	soundboard handler	web UI, replay	Optional sound playback state.
{base}/outgoing/autonomy-files	autonomy manager	web UI	Optional script catalog.
{base}/outgoing/autonomy-status	autonomy manager	web UI, replay	Optional autonomy process state.
{base}/outgoing/wheel-odometry	wheel-odometry script	web UI, replay, follow logic	Optional odometry estimate.

{base}/outgoing/apriltag-locations	AprilTag location script	follow logic, web UI, replay	Optional perception topic.
{base}/outgoing/apriltag-data	AprilTag follow scripts	web UI, replay	Optional target-tag estimate topic.
{base}/outgoing/video-overlays	perception/autonomy publishers	web UI, replay	Optional overlay geometry topic.

8.2 Example Boolean Control Convention Used Here

The current repository's shared boolean parser accepts only:

```
true
{ "value": true }
{ "enabled": true }
```

This currently applies to:

1. incoming/front-camera
2. incoming/audio
3. incoming/flags/remote-mirror
4. incoming/flags/mqtt-audio
5. incoming/flags/mqtt-video
6. incoming/flags/reboot
7. incoming/flags/git-pull

8.3 Example Drive Payload Used Here

Accepted forms:

```
{ "x": 0.2, "z": 0.5 }
{ "value": { "x": 0.2, "z": 0.5 } }
```

Current field meanings:

1. `x`: steering, `-1.0 .. 1.0`
2. `z`: throttle, `-1.0 .. 1.0`
3. `y`: optional auxiliary axis, not used by the serial MCU bridge

The current repository converts this to a differential-drive serial command:

```
left  = clamp(z + x, -1.0, 1.0)
right = clamp(z - x, -1.0, 1.0)
```

Serial example:

```
M 0.700 0.300
```

This is an example actuator mapping, not a general MQTT requirement.

8.4 Example Light Payloads Used Here

Solid LED command:

```
{ "b": 0.1, "g": 0.2, "r": 0.3 }
```

Wrapped form:

```
{ "value": { "b": 0.1, "g": 0.2, "r": 0.3 } }
```

Flash/fade LED command:

```
{ "b": 0.4, "g": 0.5, "r": 0.6, "period": 1.2 }
```

8.5 Example Heartbeat Payload Used Here

Current heartbeat payload:

```
{ "t": 1710969600123 }
```

In the current repository this is published on `outgoing/online`, typically retained with QoS 1.

8.6 Example Capability Payload Used Here

The current repository uses the schema id:

```
pebble-capabilities/v1
```

Example envelope:

```
{
  "schema": "pebble-capabilities/v1",
  "t": 1710969600123,
  "value": {
```

```
      "identity": {
        "system": "pebble",
        "type": "robots",
        "id": "fred"
    },
      "video": {
        "available": true,
        "controls": true,
        "topic": "pebble/robots/fred/outgoing/front-camera",
        "command_topic": "pebble/robots/fred/incoming/front-
camera",
        "flag_topic": "pebble/robots/fred/incoming/flags/mqtt-
video",
        "overlays_topic": "pebble/robots/fred/outgoing/video-
overlays",
        "width": 640,
        "height": 480,
        "fps": 15
    }
   }
 }
```

The current repository may populate the following `value` sections:

1. `identity`
2. `video`
3. `audio`
4. `soundboard`
5. `autonomy`
6. `system.reboot`
7. `system.git_pull`
8. `drive`
9. `lights`

10. `telemetry`

This shape is an example schema, not a requirement for future users.

8.7 Example Logs Payload Used Here

Current structured log payload:

```
{
  "t": 1710969600123,
  "level": "INFO",
  "service": "mqtt_bridge",
  "message": "Remote MQTT connected",
  "logger": "root",
  "pid": 4321
}
```

8.8 Example Video Stream Used Here

The current repository's example video topic is:

`{base}/outgoing/front-camera`

Payload:

```
{
  "id": 123,
  "keyframe": true,
  "data": "<base64(zlib(jpeg_bytes))>",
  "timestamp": 1710969600.123
}
```

Current encoding rules:

1. keyframes contain full JPEG frames
2. delta frames encode `(current - previous) + 128` before JPEG + zlib + base64
3. consumers reconstruct deltas by subtracting `128` and adding to the previous reference frame

This is a repository-specific transport choice. Future users may replace it with another video payload format or omit video entirely.

8.9 Example Audio Stream Used Here

The current repository uses:

Appendix B: `{base}/outgoing/audio` for outbound component audio

Appendix C: `{base}/incoming/audio-stream` for inbound playback audio

Both use the same binary packet format:

1. big-endian header struct: `!4sBBBBHHIQ`
2. `magic`: `PBAT`
3. `version`: `1`
4. `codec`: `1` (`pcm_s16le`)
5. `channels`: channel count
6. `flags`: currently `0`
7. `rate`: sample rate in Hz
8. `frame_samples`: samples per channel in the packet
9. `seq`: monotonic packet sequence
10. `timestamp_ms`: UNIX epoch milliseconds
11. followed by interleaved PCM16LE payload bytes

Current default operating profile in this repository:

1. `16000` Hz
2. mono or stereo depending on component config
3. `20 ms` frames
4. QoS `0`

This is an example binary media format, not a requirement for future users.

8.10 Example Soundboard Topics Used Here

Current accepted command forms:

```
{ "action": "play", "file": "intro.wav" }
{ "action": "stop" }
{ "enabled": true, "file": "warn/alarm.wav" }
{ "value": { "action": "play", "file": "intro.wav" } }
false
```

Example files payload:

```
{
  "files": ["intro.wav", "warn/alarm.wav"],
  "controls": true,
  "timestamp": 1710969600.123
}
```

Example status payload:

```
{
  "playing": true,
  "file": "intro.wav",
  "error": null,
  "controls": true,
  "timestamp": 1710969600.123
}
```

Future users do not need to implement a soundboard topic family.

8.11 Example Autonomy Topics Used Here

Current accepted command forms:

```
{ "action": "start", "file": "apriltag-odom-follow", "config": { "tag_id": 13 } }
{ "action": "stop" }
{ "enabled": true, "file": "wheel-odometry", "config": { "use_tune_state": false } }
{ "value": { "enabled": true, "file": "apriltag-follow" } }
```

Example files payload:

```
{
  "files": [
   {
      "file": "apriltag-odom-follow",
      "label": "apriltag-odom-follow",
      "configs": [
```

```
          { "key": "tag_id", "label": "Tag ID", "type": "int",
"default": 13 }
     ]
     }
    ],
    "controls": true,
    "timestamp": 1710969600.123
  }
```

Example status payload:

```
  {
    "running": true,
    "file": "apriltag-odom-follow",
    "pid": 4321,
    "dependencies": [
     { "file": "apriltag-locations", "pid": 4310, "running":
true },
     { "file": "wheel-odometry", "pid": 4311, "running": true }
    ],
    "error": null,
    "config": { "tag_id": 13 },
    "controls": true,
    "timestamp": 1710969600.123
  }
```

This is an example process-control family, not a required part of the general standard.

8.12 Example Odometry And Perception Topics Used Here

These are optional example telemetry families currently used in this repository.

Example `wheel-odometry` payload:

```
  {
    "value": {
      "x_mm": 0.0,
      "y_mm": 0.0,
```

```
    "heading_rad": 0.0,
    "heading_deg": 0.0,
    "left_mm": 0.0,
    "right_mm": 0.0,
    "distance_mm": 0.0,
    "left_crossings": 0,
    "right_crossings": 0,
    "unknown_direction_events": 0,
    "left_step_total": 0.0,
    "right_step_total": 0.0,
    "left_mm_per_step": 0.0,
    "right_mm_per_step": 0.0,
    "left_extrema_events": 0,
    "right_extrema_events": 0,
    "left_rejected_sign_flips": 0,
    "right_rejected_sign_flips": 0,
    "left_stall_events": 0,
    "right_stall_events": 0,
    "sample_seq": 0,
    "publish_seq": 0
  },
  "unit": "mm",
  "source": "wheel-odometry",
  "timestamp": 1710969600.123
}
```


Example `apriltag-locations` payload:


```
{
  "source": "apriltag-locations",
  "detections": [
   {
      "marker_id": 13,
```

```
        "corners_norm": [[0.1, 0.2], [0.2, 0.2], [0.2, 0.3],
[0.1, 0.3]],
        "center_norm": [0.15, 0.25]
     }
    ],
    "frame_width": 640,
    "frame_height": 480,
    "timestamp": 1710969600.123
  }
```

Example `apriltag-data` payload:

```
  {
    "detected": true,
    "marker_id": 13,
    "corners_norm": [[0.1, 0.2], [0.2, 0.2], [0.2, 0.3], [0.1,
0.3]],
    "distance_m": 0.74,
    "angle_rad": -0.11,
    "timestamp": 1710969600.123
  }
```

Example `video-overlays` payload:

```
  {
    "source": "apriltag-locations",
    "shapes": [
      [[0.1, 0.2], [0.2, 0.2], [0.2, 0.3], [0.1, 0.3]]
    ],
    "labels": [
     { "marker_id": 13, "center_norm": [0.15, 0.25] }
    ],
    "frame_width": 640,
    "frame_height": 480,
    "timestamp": 1710969600.123
```

}

Future users do not need to adopt any of these perception-specific metrics.

9. Example Compatibility Topics Still Recognized By This Repository

These are examples of older or compatibility topics still recognized by current tools here:

Table A.5: Compatibility topics still recognized by the current repository.

Example topic	Current status
`{base}/outgoing/heartbeat`	Legacy synonym accepted alongside `outgoing/online`.
`{base}/outgoing/status`	Legacy generic state topic still accepted by the web UI.
`{base}/outgoing/charging-level`	Optional battery/voltage topic still consumed by the web UI and replay tools.

10. Example Topic Override Points In This Repository

These are current config-driven topic override points in this repository. They are implementation details, not part of the general standard itself.

Table A.6: Config-driven topic override points in the current repository.

Config location	Affects
`services.launcher.capabilities.topic`	`outgoing/capabilities`
`services.launcher.logs.topic`	`outgoing/logs`
`services.mqtt_bridge.topics.remote_mirror`	`incoming/flags/remote-mirror`
`services.mqtt_bridge.topics.mqtt_audio`	`incoming/flags/mqtt-audio`

services.mqtt_bridge.topics.mqtt_video	incoming/flags/mqtt-video
services.mqtt_bridge.topics.reboot	incoming/flags/reboot
services.mqtt_bridge.topics.git_pull	incoming/flags/git-pull
services.mqtt_bridge.topics.audio_control	incoming/audio
services.mqtt_bridge.topics.video_control	incoming/front-camera
services.mqtt_bridge.heartbeat.topic	outgoing/online
services.mqtt_bridge.media.video_publisher.topic	outgoing/front-camera
services.mqtt_bridge.media.video_publisher.overlays_topic	example overlay topic field in capabilities
services.mqtt_bridge.media.audio_publisher.topic	outgoing/audio
services.mqtt_bridge.media.audio_receiver.topic	incoming/audio-stream
services.serial_mcu_bridge.topics.*	drive, lights, touch, charging topics
services.soundboard_handler.topics.*	soundboard command/files/status topics
services.autonomy_manager.topics.*	autonomy command/files/status topics
autonomy CLI args such as --topic, --locations-topic, --overlays-topic, --odom-topic, --tags-topic	script-specific published / subscribed topics

11. Implementation Notes

1. Future users SHOULD treat the topic families in sections 6 and 8 as modular. A valid implementation may use only a small subset.

2. If a deployment uses retained current-state topics, it SHOULD republish them at startup, reconnect, and on meaningful state changes.
3. If topic names are configurable, publishing those effective topic names in a retained capabilities or discovery topic SHOULD be preferred because it improves interoperability.
4. If a system bridges brokers, loop prevention and retained-message replay SHOULD be documented explicitly.