Lasse Jonas Rosenow

Federated Communication for the Lingua Franca: Reactor-µC Runtime

Faculty of Computer Science and Digital Society

Supervising examiner: Prof. Dr. Thomas Schmidt

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Contents

1 Introduction

1.1 Background on Lingua Franca

Lingua Franca [1] is a Coordination Language for building distributed systems.

It guarantees deterministic reactive concurrency, allows specifying timed behaviors, and supports concurrent and distributed execution with the help of a runtime, such as Reactor-C or Reactor-µC.

Based on the selected runtime it transpiles to target programming languages, such as C, C++, Python, Typescript, and Rust.

1.2 Motivation

Reactor-µC is a new runtime implementation for Lingua Franca built for constrained (low in power, Read-only memory (ROM), Random-access memory (RAM), …) devices. It is written in the C programming language and runs on various (IoT) operating systems, such as RIOT [2], as well as on Linux, but it does not yet include an implementation of Lingua Franca Federation. Lingua Franca Federation describes the communication between and not only within runtimes.

Reliable communication between nodes is helpful in in the following use cases.

Drone Swarm Coordination: When multiple drones form a swarm and need to interact with each other, it is helpful that this communication occurs reliably, in-order and within a specified time. Unreliable communication may lead to collisions between drones due to incorrect or outdated information about other drone locations in 3D space. Lingua Franca enables the specification of such constraints and the definition of fallback actions when they are not met to ensure safe operation.

Distributed Health Monitoring Applications: For applications within the medical sector, it can be helpful to have strong guarantees on reliability and determinism. If a medical applications fail because of a deadlock or a lost message it can have dire consequences. Lingua Franca helps to prevent many of these problems out of the box.

1.3 Objective

The focus of the research is to design a minimal abstraction layer within Reactor-µC for federated communication. The abstraction layer is the interface between the Reactor-µC runtime and various implementations for wireless communication. We need this abstraction layer not only to hide the complexity of the federated communication implementation from the runtime, but also as a way for users of Reactor-µC to contribute their own 3rd party implementations of it, giving them more flexibility to make their own choices on implementation details.

Furthermore, the abstraction layer must be designed so that implementations can fulfill all strict requirements of the Lingua Franca language while operating on constrained hardware.

1.3.1 Lingua Franca Requirements

Lingua Franca introduces certain requirements that must be fulfilled in order to maintain its guarantees, such as deterministic execution.

Acknowledged message delivery: Messages between Lingua Franca nodes must be acknowledged. The Lingua Franca application can optionally specify how long timeouts should last and how many retransmissions should occur before the message delivery is considered as failed. The implementation needs to align to what is specified in Lingua Franca while providing reasonable default values. In the case of a timeout, the runtime shall be informed and will handle this case according on how it is specified in the Lingua Franca application.

In-Order message delivery: Messages between Lingua Franca nodes need to be processed in-order, otherwise Lingua Franca can not guarantee to execute reactions in the correct order, which will cause its guarantee for determinism to fail.

1.3.2 Hardware Constraints

Reactor-µC runs on constrained devices, which are limited in power, ROM, RAM, transmission speed and processing speed. The Federated Communications Application Programming Interface (API) and its implementation thus need to be flexible and allow for implementations that maximize performance for the more powerful devices, while also allowing other implementations to prioritize energy efficiency, low memory consumption and other metrics.

Wired Transmission: We intend to support wired communication over Ethernet to connect our federated system to the Internet and over serial for on board communication between multiple processors.

Wireless Transmission: We want to support short- and long-distance wireless communication. For this we plan to use IEEE 802.15.4 for short distance wireless communication and Long Range Wide Area Network (LoRaWAN) for long distance wireless communication.

1.4 Outline

In Section 2 we identify communication protocols that fulfill the strict requirements of Lingua Franca specified in Section 1.3.1 while remaining compatible with the hardware constraints specified in Section 1.3.2. Furthermore, we implement the Federated Communication API for each selected protocol.

In Section 3 we evaluate the memory overhead of each solution in terms of both ROM and RAM usage.

In Section 4 we summarize the results of the evaluation chapter and discuss whether Constrained Application Protocol (CoAP) and Transmission Control Protocol (TCP) and Universal Asynchronous Receiver Transmitter (UART) were correct choices for implementing the Federated Communications API.

Finally, in Section 5 we outline the future work to extend the [CoAP]-based implementation to LoRaWAN (with Static Context Header Compression (SCHC)), and UART. In addition we plan to evaluate message throughput across physical layers.

2 Implementation

2.1 Communication Protocols

For the implementation of federated communication for Lingua Franca, we first needed to decide on top of which network protocols to build on. The objective was to find protocols that work well for Ethernet based transmission, serial based transmission and low power wireless transmission. Based on this we chose a total of three protocols and on top of each, we implemented the Federated Communication API for Reactor-µC. These three protocols fulfill varying use cases and meet the requirements (see Section 1.3.1 and Section 1.3.2) to varying degrees.

Table 1 illustrates the selected protocols and their compliance with the requirements.

We chose TCP for more powerful and less constrained devices that support high bandwidth, high data rate, and reliable communication without power constraints, also known as broadband communication. Furthermore TCP is widely adapted on the Internet and allows us to connect our Lingua Franca nodes to the Internet. This for example can be realised by a single more powerful node, that connects a local edge cloud of nodes to the rest of the Internet via TCP over Ethernet or WLAN. But we cannot use TCP for low-power and lossy networks as its congestion control mechanism does not work well with it, similarly we can also not recommend using TCP over serial. Regarding the requirements of Lingua Franca (see Section 1.3.1), it fulfills both acknowledged message delivery and in-order message delivery. Furthermore, it includes a built-in checksum verification for data integrity.

We chose CoAP over User Datagram Protocol (UDP) for communication over low-power and lossy networks that usually exhibit low bandwidths, low data rates, are usually less reliable, and focus on energy efficiency, such as LoRaWAN or IEEE 802.15.4. Regarding the requirements of Lingua Franca (see Section 1.3.1) CoAP supports acknowledged message delivery natively via its confirmable message type (there is also a non-confirmable message type), however it does not provide built-in in-order message delivery, so we will need to implement an in-order message delivery solution on top of the existing CoAP stack. Furthermore, CoAP messages also contain a checksum in the UDP header[1], which gives us data integrity guarantees.

We chose UART for serial communication. This allows us to quickly communicate between multiple processors on a single board without the overhead of a full Ethernet network stack and hardware, while still profiting from the advantages of wired communication. UART does not fulfill any of the Lingua Franca requirements for federated communication. We need to devise a custom solution for “message acknowledgement”, “in-order message delivery” and “checksum verification”.

[1]The checksum is optional when using IPv4.

Requirement \ Protocol	TCP	CoAP over UDP	UART
Works well on broadband networks	Yes	No	No
Works well on low-power and lossy networks	No	Yes	Yes
Works well for serial transmission	No	No[2]	Yes
Acknowledged message delivery	Yes	Yes	No
In-Order message delivery	Yes	No	No
Checksum verification	Yes	Yes[3]	No

Table 1: Chosen Communication Protocols and their Supported Requirements of the Federated Communication in Lingua Franca

2.2 Architecture

We introduce a new Federated Communication API layer between the Reactor-µC runtime and the various protocols on top of which we transmit messages between different Lingua Franca nodes.

An overview of the architecture is given in Figure 1. The Reactor-µC runtime is located at the top and can send and receive federated messages through the Federated Communication API, which is located below it. The Federated Communication API is simplified for this diagram and only lists the 3 most important functions. (For a complete explanation of the full API, see Section 2.3.) The API specifies the "`is_connected`" function to tell the runtime that it has at least successfully reached the other node and has since then not had any issues reaching it again. How much time may pass since the reachability of the other node has been tested is specific to the implementation of the Federated Communication API. As of now the CoAP based implementation only relies on the successful delivery of the last Lingua Franca message. So the "`is_connected`" status only updates if a transmission is successful or not successful[4]. The TCP implementation exposes the TCP internal connection status. Furthermore, the API specifies a "`register_receive_callback`" function that allows the runtime to react to incoming messages, and finally, there is a "`send_blocking`" function, which allows the runtime to

[2]Sending CoAP over serial is not standardized, but can be achieved using the Serial Line IP Multiplexing (SLIPMUX) draft.
[3]The checksum is optional when using IPv4.
[4]We plan to implement a heartbeat in the future.

send messages to the other node. Three example implementations are given and located below the Federated Communication API in the diagram. The bottom left shows a UART-based implementation that internally uses the RIOT and Zephyr OS APIs. The bottom center shows a CoAP-based implementation that uses the GCoAP library for RIOT and the CoAP client in Zephyr OS. And the bottom right shows an example of a TCP-based implementation, which is built on top of Portable Operating System Interface (POSIX) and uses the POSIX wrappers in RIOT and Zephyr OS[5].

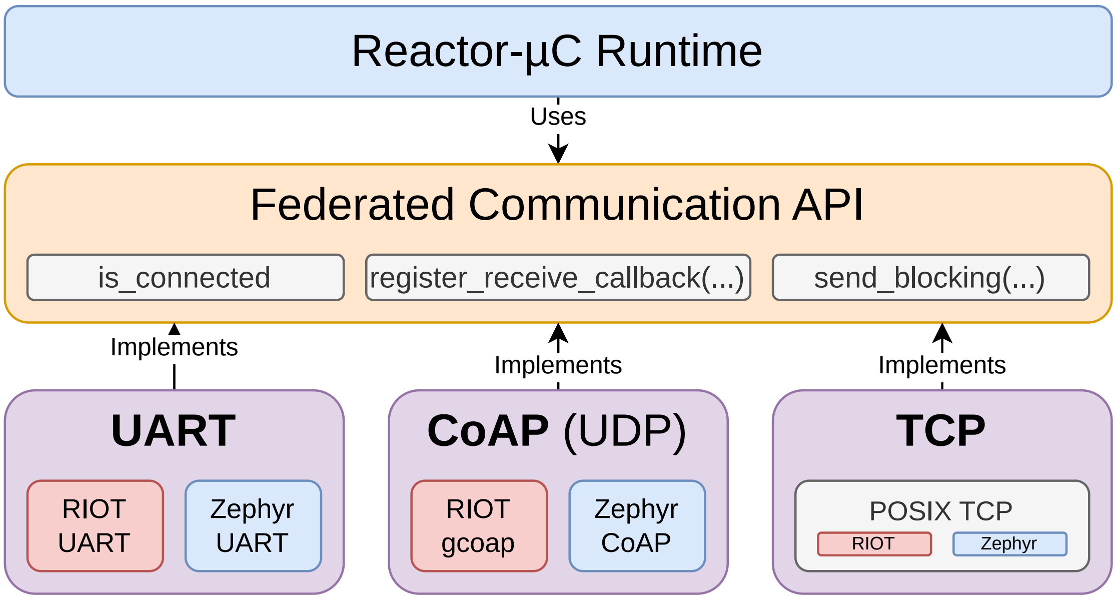

Figure 1: Simplified Architecture of Reactor-µC with the Federated Communication API and a few example implementations.

2.3 Federated Communication API

Listing 1 shows the C header file of the new Federated Communication API.

From lines 1-12, we specify the “FederatedCommunicationState” enum, which is used to track the current state of the “connection” with the other node. Supported states are “uninitialized”, “open”, “connection in progress”, “connection failed”, “connected”, “lost connection”, and “closed”. The relevance of this state is further explained in the state machine in Figure 2.

[5]The POSIX wrappers were chosen so that we do not have to implement the TCP channel for RIOT and Zephyr OS separately.

Lines 11 to 27 specify the “FederatedCommunication” struct, which is the C implementation of the Federated Communication API. Line 13 specifies the “is_connected” function, which indicates whether the node currently has a connection with the other node and can send messages. Line 16 specifies the “open_connection” function, which changes the state of the Federated Communication implementation to “open”. The specific implementation will then internally change the state to “connection in progress”, try to reach the other node, and then change the state to “connected” or “connection failed” depending on the success of the operation. Line 17 specifies the “close_connection” function, which enables the runtime to close the connection with the other node. The specific implementation will set the state to “closed” and inform the other node that it has closed the connection. Lines 20 to 23 specify the “send_blocking” function. This function allows the runtime to send a message to the other node. The send operation in our current design is blocking to guarantee in-order message delivery, regardless of the transport protocol used in the specific implementation[6]. Lines 26 to 33 specify the “register_receive_callback” function, which enables the runtime to register a callback that gets invoked by the specific implementation of the Federated Communication API whenever a new message is received from another node. Line 36 specifies the function “free” which handles the cleanup of storage and other resources when the application is closed.

```
 1 typedef enum {
 2   FEDERATED_COMMUNICATION_STATE_UNINITIALIZED,
 3   FEDERATED_COMMUNICATION_STATE_OPEN,
 4   FEDERATED_COMMUNICATION_STATE_CONNECTION_IN_PROGRESS,
 5   FEDERATED_COMMUNICATION_STATE_CONNECTION_FAILED,
 6   FEDERATED_COMMUNICATION_STATE_CONNECTED,
 7   FEDERATED_COMMUNICATION_STATE_LOST_CONNECTION,
 8   FEDERATED_COMMUNICATION_STATE_CLOSED,
 9 } FederatedCommunicationState;
10
11 struct FederatedCommunication {
12   /* State */
13   bool (*is_connected)(FederatedCommunication *self);
14
15   /* Open / Close Connection */
16   lf_ret_t (*open_connection)(FederatedCommunication *self);
17   void (*close_connection)(FederatedCommunication *self);
18
19   /* Send messages */
20   lf_ret_t (*send_blocking)(
```

[6]Blocking while sending significantly decreases the performance with the advantage of saving memory. An alternative solution is to use receive buffers and reorder the messages on the receiving node, this will result in higher performance but also consumes significantly more memory. See Section 2.3.2 for more details.

```c
    FederatedCommunication *self,
    const FederateMessage *message
  );

  /* Register Incoming Message Callback */
  void (*register_receive_callback)(
    FederatedCommunication *self,
    void (*receive_callback)(
      FederatedConnectionBundle *conn,
      const FederateMessage *message
    ),
    FederatedConnectionBundle *conn
  );

  /* Cleanup Storage / Join Threads */
  void (*free)(FederatedCommunication *self);
};
```

Listing 1: C Header File of the Federated Communication Application Programming Interface (API).

2.3.1 State Machine

Figure 2 visualizes the hierarchical state machine of the Federated Communication API. The state machine consists of 2 hierarchical sub-states: “Running” and “ConnectionOpen”. Everything in the “Running” state happens after the device is turned on and before the device is turned off. The “ConnectionOpen” state contains the whole connection establishment process and can be entered via the open_connection() function and left via the “close_connection()” function.

Detailed Lifecycle: After the device is powered on, the Federated Communication is in the “Uninitialized” state. After the “open_connection” function is called by the runtime, it switches into the “Open” state and the main loop of the specific implementation attempts to establish a connection with the other node and switches to the “ConnectionInProgress” state. If the connection establishment fails, the state machine switches into the “ConnectionFailed” state and retries to connect after a specified amount of time, for which it switches back into the “ConnectionInProgress” state. If the connection is successfully established, the state machine switches into the “Connected” state. If the state machine is in the “Connected” state, the runtime can send messages using the “send_blocking” function. In the case that the send operation fails, the state machine switches from the “Connected” into the “LostConnection” state and tries to reconnect after a specified amount of time, for which it switches back into the “ConnectionInProgress” state. From every state inside the “ConnectionOpen” hierarchical state the state machine can switch into the “Closed” state when the runtime calls the "close_connection" function. Finally, from within each state it is possible to terminate the state machine by powering off the device.

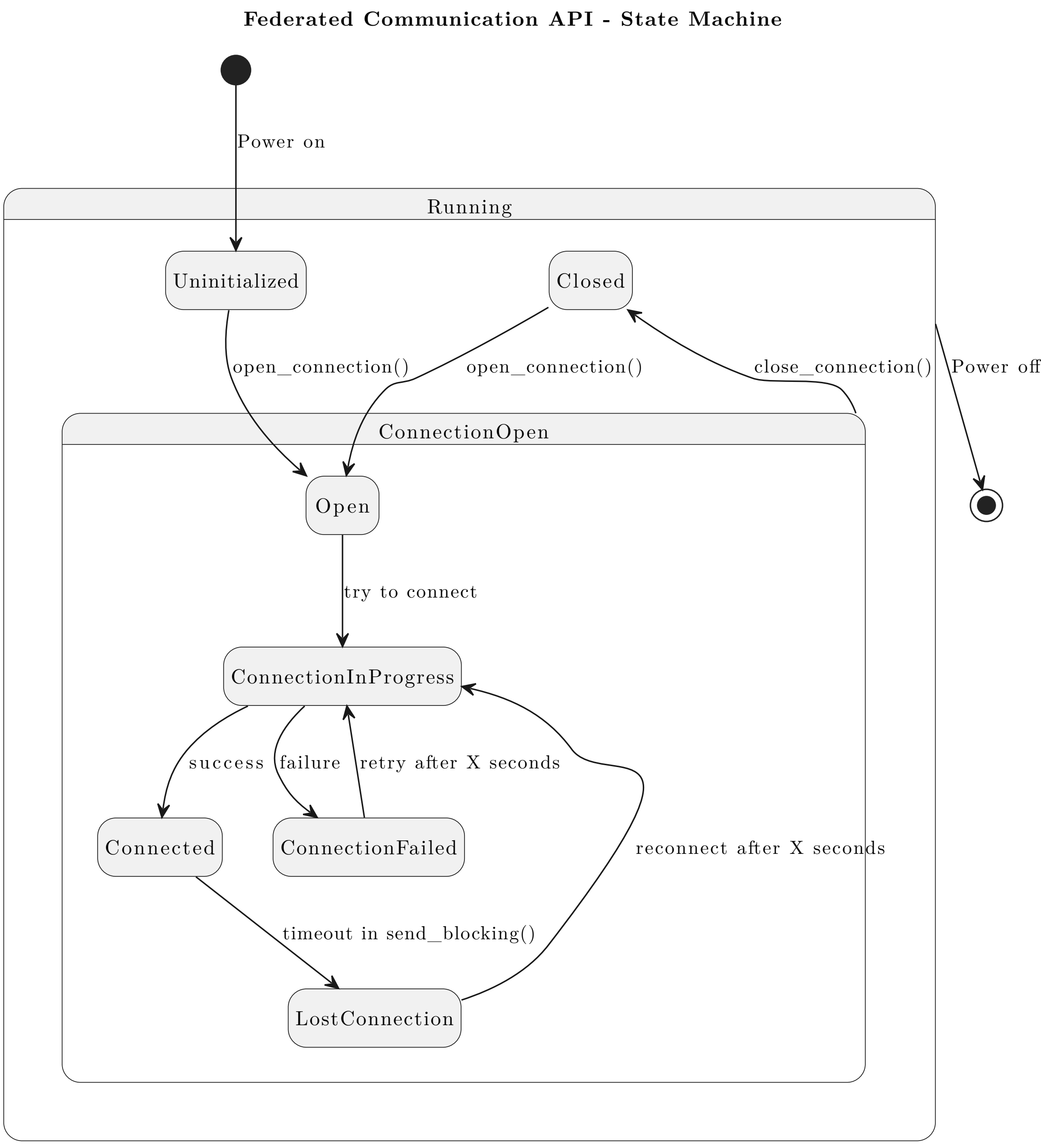

Figure 2: Hierarchical State Machine of the Federated Communication API.

2.3.2 CoAP Implementation

The CoAP implementation uses the RIOT native CoAP stack (GCoAP) and UDP as its transport layer protocol. Internally, each Lingua Franca node acts as both a CoAP server and client, so that they can send and receive messages to and from other nodes. A single node may have multiple CoAP clients to talk to various other nodes, but it only has one CoAP server, which receives messages from all other nodes that may speak to it. We use CoAP resources (“`/connect`”, “`/disconnect`”, “`/message`”) for different operations. The “`/connect`” and “`/disconnect`” endpoints are to establish a connection, which is not part of the CoAP specification. A “connection” in this case means, that a CoAP client issues a request to the CoAP server of another node and the server replies, if it is ready to communicate. In a normal case, it should only be used on the bootstrap to make an initial connection and to check if the other node is reachable. The “`/message`” endpoint is used to send Lingua Franca messages to other nodes. Messages are sent in confirmable mode to fulfill the requirement of acknowledged message delivery. CoAP does not provide in-order message delivery. To work around this we block on the message send operation and keep track of the previously sent message ID. This way, we can ensure that the next message will only be sent after we confirmed that the previous communication is complete. Blocking while sending has the disadvantage of substantially decreasing the performance (we can only send the next message, after we have confirmation, that the current message has arrived). Another way to implement this is by attaching sequence numbers to the sent messages similar to how TCP does it, store the messages in a receive buffer and then put them into the correct order. We can use the message id of the CoAP header for this. The downside of this approach is, that it consumes significantly more memory. So blocking while sending consumes less memory, but decreases throughput. Using sequence numbers and receive buffers instead consumes more memory, but increases performance. In the future we plan to support both use cases. On the receiving side, messages are read asynchronously, and trigger configured callbacks for each resource endpoint. From the callback of the “`/message`” endpoint, we inform the Reactor-µC runtime of the incoming message.

Furthermore, the CoAP-based implementation internally uses a dedicated worker thread that executes the main communication loop. This thread is responsible for establishing and maintaining connections based on the current state (see Section 2.3.1).

2.3.3 TCP Implementation

To establish the initial connection, the TCP implementation divides the Lingua Franca nodes that want to communicate into a client and a server, determined by the “`is_server`” boolean flag. The “server” node binds to a specified address and port and listens for incoming connections using the POSIX “`accept()`” function. The “client” node attempts to connect to a remote server using the POSIX "`connect()`" function. After that, both nodes have an active socket to the other node for communication.

The TCP implementation internally uses a dedicated worker thread that executes the main communication loop. This thread is responsible for establishing and maintaining connections based on the current state (see Section 2.3.1). Furthermore, this thread monitors the socket for incoming data without blocking using the POSIX "`select()`" function, updates the state and calls the Reactor-µC runtime through the registered receive callback when messages arrive.

2.3.4 UART Implementation

The UART implementation of the Federated Communication API is only minimal for now. Currently, messages are only transmitted via UART without acknowledgements, in-order guarantees, or checksums. These capabilities are not part of UART as a protocol, so we need to implement these on top of it. More details about our current plan is described in Section 5.1.

3 Memory Overhead Evaluation

To confirm that the Reactor-µC runtime is still suitable to run on constrained devices, we evaluate the memory consumption of the new Federated Communication API and its various implementations on RIOT.

To measure ROM and RAM, we use the `make cosy` tool[7], which analyzes the ROM and RAM consumption of the built binary, and starts a web server to visualize its results.

3.1 ROM Overhead

We measured the ROM usage of the CoAP and TCP-based implementation of the Federated Communication API for RIOT using the "`BOARD=adafruit-feather-nrf52840-sense make cosy`" command.

The results are visualized in Table 2 and Figure 3. We can see that in total, the TCP and the CoAP-based implementation both consume not more than 117 kilobytes, TCP needing 116.949 kilobytes and CoAP needing 109.879 kilobytes, which are small enough

[7]https://github.com/RIOT-OS/cosy

to fit on a microcontroller. The Federated Communication API and its implementations together only use 1.689 kilobytes of ROM for the CoAP-based implementation and 3.168 kilobytes of ROM for the TCP-based implementation.

So the increase in ROM usage by adding the new Federated Communication API and its implementation does not substantially increase the ROM consumption of Reactor-µC.

	CoAP	**TCP**
Federated Communication	1.689 KB	3.168 KB
Reactor-µC - Runtime	27.832 KB	28.736 KB
RIOT - Network Stack	34.678 KB	34.472 KB
RIOT - Other	45.680 KB	50.573 KB
Total	109.879 KB	116.949 KB

Table 2: Read-only memory (ROM) Usage of the Federated Communication Application Programming Interface (API) on RIOT.

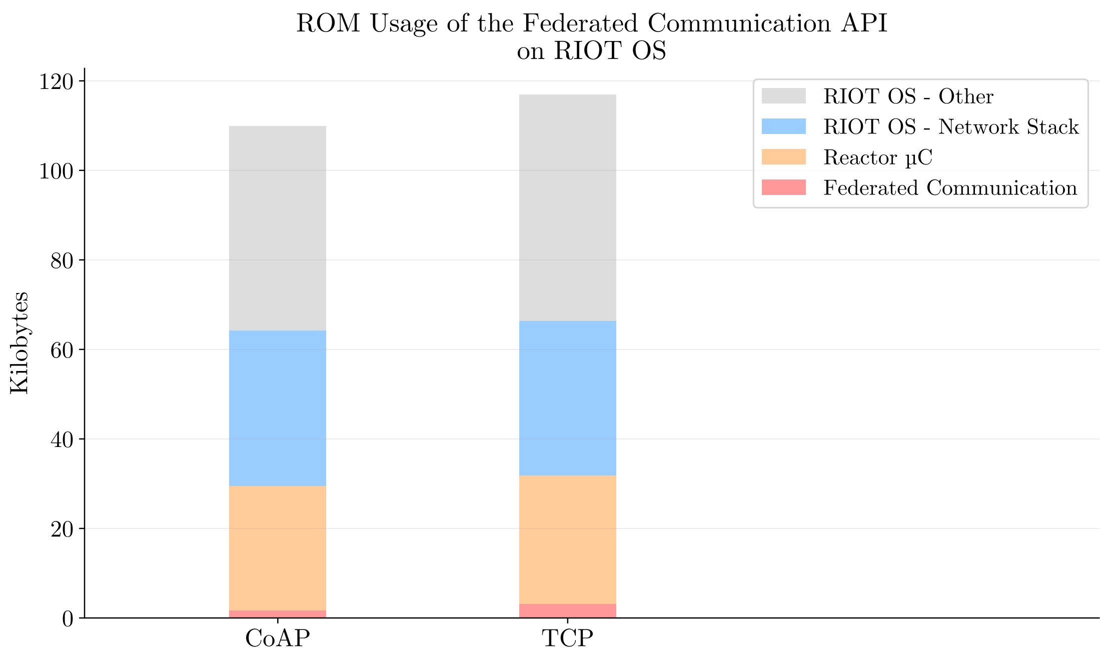

Figure 3: Read-only memory (ROM) Usage of the Federated Communication Application Programming Interface (API) on RIOT Visualized Using a Stacked Bar Chart.

3.2 RAM Overhead

We measured the RAM usage of the CoAP and TCP-based implementation of the Federated Communication API for RIOT using the "`BOARD=adafruit-feather-nrf52840-sense make cosy`" command.

The results are visualized in Table 3 and Figure 4. We can see that in total, the TCP and the CoAP-based implementation both consume not more than 43 kilobytes, TCP needing 42.04 kilobytes and CoAP needing 34.768 kilobytes, which are small enough to fit on a microcontroller. The Federated Communication API and its implementations together only use 1.565 kilobytes of RAM for the CoAP-based implementation and 0 kilobytes of RAM for the TCP-based implementation. The TCP-based implementation though, has an increase in RAM usage of the RIOT network stack by around 5.3 kilobytes compared to the CoAP-based implementation. This is caused by the additional need for the POSIX wrappers, which internally allocate RAM for the TCP socket.

Nevertheless, the increase in RAM usage by adding the new Federated Communication API and its implementation does not substantially increase the RAM consumption of Reactor-µC.

	CoAP	**TCP**
Federated Communication	1.565 KB	0 KB
Reactor-µC - Runtime	0 KB	0 KB
RIOT - Network Stack	13.249 KB	18.558 KB
RIOT - Other	19.954 KB	23.482 KB
Total	34.768 KB	42.040 KB

Table 3: Random-access memory (RAM) Usage of the Federated Communication Application Programming Interface (API) on RIOT.

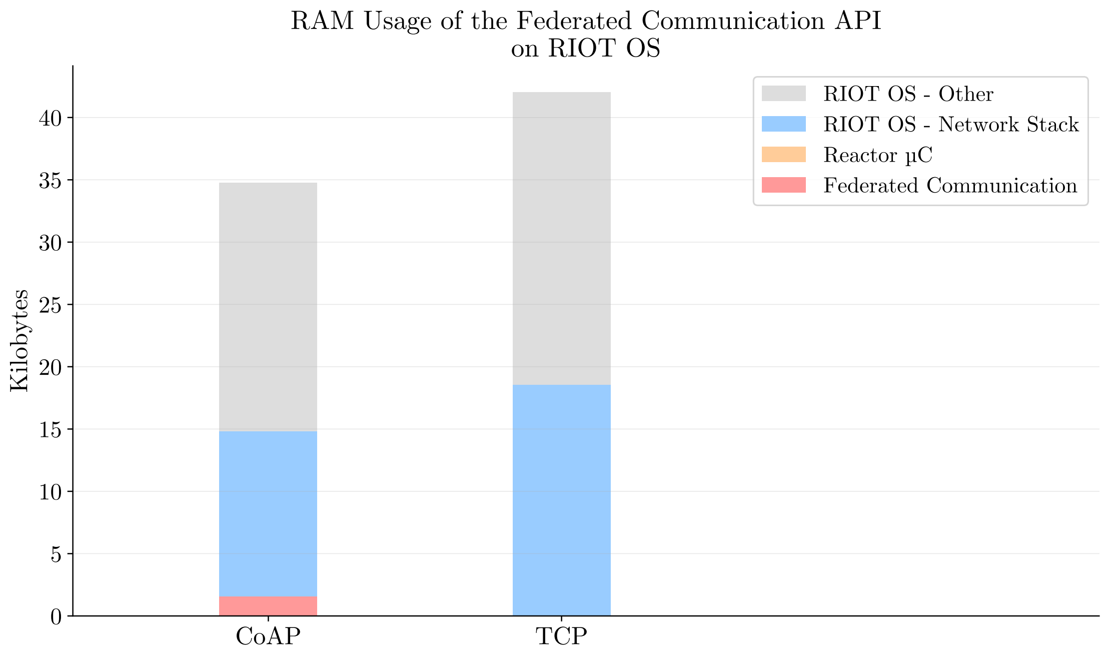

Figure 4: Random-access memory (RAM) Usage of the Federated Communication Application Programming Interface (API) on RIOT Visualized Using a Stacked Bar Chart.

4 Conclusion

Using CoAP for wireless and lossy networks is feasible for Reactor-µC. We were able to send messages between nodes and at least in terms of the memory usage, it is well within the limits that a microcontroller offers.

The TCP implementation also works well, even on RIOT. Its overhead is a little higher than the CoAP-based implementation, but only by a few kilobytes on RIOT.

The message throughput on both wireless and wired connections using TCP and CoAP still needs to be evaluated.

Furthermore, we have realized that using raw UART for implementing the Federated Communication API requires a lot of basic work to get acknowledgments, data integrity and other requirements working. Therefore we decided to use CoAP over UART for now. This of course comes with additional overhead and the need for a full network stack to be included in the binary. For devices that are extremely low in ROM and RAM, it will still make sense to use raw UART, but for now this is out of scope for our use-case.

5 Outlook

5.1 Implement UART over CoAP and LoRaWAN over CoAP

Figure 5 illustrates our new architecture strategic design, which aims to consolidate as many physical layers as possible below the CoAP layer. In particular we now do not have a separate UART implementation of the Federated Communication API, but instead integrate with the CoAP-based implementation to send messages over UART. This new strategy also applies to the not-yet-implemented LoRaWAN-based communication, which will also be handled by the CoAP implementation.

5.1.1 Implementation Strategy

RIOT already supports sending CoAP packets via SLIPMUX on UART [3].

Sending CoAP messages over LoRaWAN introduces significant overhead due to the redundancy of the CoAP, UDP and IP headers. For example IP addresses are not needed when using LoRaWAN, because it generates its own network layer on the network server. This can be compressed to near-zero overhead using SCHC [4]. The RIOT network stack is technically capable of sending CoAP messages over LoRaWAN, but this requires further investigation.

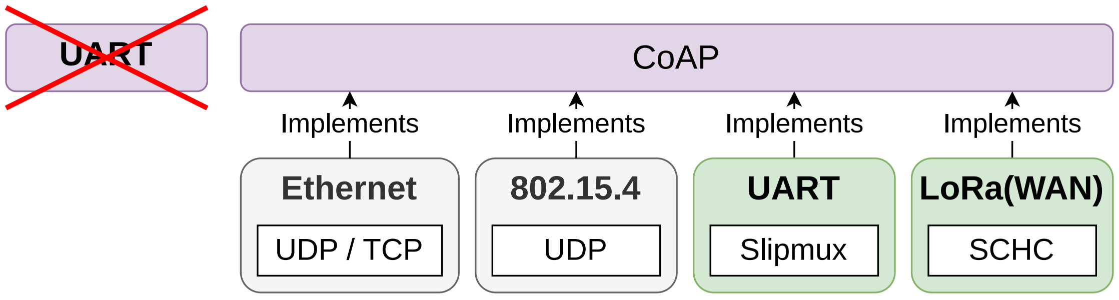

Figure 5: New Architecture with Universal Asynchronous Receiver Transmitter (UART) under Constrained Application Protocol (CoAP) and Long Range Wide Area Network (LoRaWAN) with Static Context Header Compression (SCHC).

5.2 Evaluation of Message Throughput

Evaluating the message throughput of our Reactor-µC implementation on various physical layers, including both wired and wireless ones, is outside the scope of this work. We intend to change this in a future publication once LoRaWAN is supported.

Glossary

***API* – Application Programming Interface**

***POSIX* – Portable Operating System Interface**

***RAM* – Random-access memory**

***ROM* – Read-only memory**

***SCHC* – Static Context Header Compression**

***SLIPMUX* – Serial Line IP Multiplexing**

***TCP* – Transmission Control Protocol**

***UART* – Universal Asynchronous Receiver Transmitter**

***UDP* – User Datagram Protocol**

***CoAP* – Constrained Application Protocol**

IEEE 802.15.4: A technical standard that defines the operation of a low-rate wireless personal area network (LR-WPAN) [5]

Lingua Franca: Lingua Franca is a polyglot coordination language built to enrich mainstream target programming languages (currently C, C++, Python, TypeScript, and Rust) with deterministic reactive concurrency and the ability to specify timed behavior. [1]

***LoRaWAN* – Long Range Wide Area Network**

Reactor-C: A Lingua Franca reactor runtime written in C.

Reactor-µC: A lightweight Lingua Franca reactor runtime targeted at distributed resource-constrained embedded systems.

RIOT: As the Internet of Things (IoT) emerges, compact operating systems (OSs) are required on low-end devices to ease development and portability of IoT applications. RIOT is a prominent free and open source OS in this space. [2]

Bibliography

[1] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, “Toward a Lingua Franca for Deterministic Concurrent Systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, May 2021, doi: 10.1145/3448128.

[2] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, “RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT,” *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, Dec. 2018, [Online]. Available: http://dx.doi.org/10.1109/JIOT.2018.2815038

[3] C. Bormann and T. Kaupat, “Slipmux: Using an UART interface for diagnostics, configuration, and packet transfer,” Internet Engineering Task Force, Internet-Draft draft-bormann-t2trg-slipmux-03, Nov. 2019. [Online]. Available: https://datatracker.ietf.org/doc/draft-bormann-t2trg-slipmux/03/

[4] M. Tiloca, L. Toutain, I. Martínez, and A. Minaburo, “Static Context Header Compression (SCHC) for the Constrained Application Protocol (CoAP),” Internet Engineering Task Force, Internet-Draft draft-ietf-schc-8824-update-05, July 2025. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-schc-8824-update/05/

[5] “IEEE Standard for Low-Rate Wireless Networks,” *IEEE Std 802.15.4-2024 (Revision of IEEE Std 802.15.4-2020)*, no. , pp. 1–967, 2024, doi: 10.1109/IEEESTD.2024.10794632.