interstice
firmware-extensible pervasive embedded networks

Nathan Perry · December 06, 2024

A key decision in the development and deployment of networked embedded devices is networking technology selection. Contemporary solutions are mutually isolated by limited protocol compatibility, requiring application-specific bridges to connect network technologies that were not designed to interoperate.I observe that these bridges can be generically abstracted into modular functional components, comprising a central packet switch performing the networking function, with modular lower-layer drivers which implement access to physical media via a common packet representation. This model is well-established in desktop- and server-class operating systems, but practically absent in embedded firmware.The proposed thesis develops a platform- and runtime-agnostic IP-based networking approach including a firmware packet switch, with particular focus on ease of integration and extension by the end user. Reference integration is planned for both the Arduino and `embassy` firmware families. Case studies based on networks of embedded sensors are proposed for performance evaluation against key metrics.

Proposed thesis for the degree of *Scientiæ Magister*

Department of Media Arts and Sciences
Massachusetts Institute of Technology

Fall 2024

Joseph A. Paradiso, Advisor
Professor of Media Arts and Sciences, MIT Media Lab

Neil Gershenfeld, Reader
Director, Center for Bits and Atoms

Prabal Dutta, Reader
Professor of EECS, University of California, Berkeley

interstice

That which intervenes between one thing and another; especially, a space between things closely set, or between the parts which compose a body. [1]

Contents

[1]https://csa-iot.org/all-solutions/matter/

NOTE: Please see Section 7 (Addendum) for an updated plan reflecting committee feedback.

1. Introduction

The Internet of Things (IoT) was envisioned as a model for pervasive, ubiquitous computing and sensing in devices around us [2]. Everything would be a little data terminal that could communicate; it would make things *smarter*. Your oven would seamlessly communicate with your phone[2], and your phone with the lock on your door. It could remind you if you forgot to lock it, and do so remotely if commanded. Your fridge might have a database of the produce stored inside it, and could automatically notify you when it started to go bad. Your light switches would all talk to each other and let you shut off the kitchen light from your bedroom.

These possibilities are available in some incarnation today, but the user experience of contemporary IoT is flaky, unstable, and fragmented rather than seamless and implicit, as we once dreamt.

Contemporary devices are overwhelmingly implemented as isolated cloud endpoints, rather than participants in a rich local network. They are siloed off from one another if they don't happen to speak the same network protocol and message format, and even when they do, are often still unable to communicate due to vendor differences. There is no singular *lingua franca* in IoT; rather, the space is dominated by vendor-specific interfaces and a proliferation of partially-adopted protocol standards. It is well acknowledged in the literature that communication fragmentation is a problem [3], [4], [5].

The devices we thought would be smart are so-often in fact dumb, delegating their intelligence to servers thousands of miles away. They are highly sensitive to connectivity status and often entirely reliant on an Internet connection to a central controller. The kind of intelligence we hoped for requires a local capability for partial reprogramming (updates to simple rules at a minimum), but this functionality is largely not available. Worse, these devices are prone to bricking completely and permanently if the company that produced them goes out of business or chooses to close the product line.

> Never trust anything that can think for itself if you can't see where it keeps its brain.
>
> – Arthur Weasley [6]

The flakiness and opacity of IoT systems has deeply eroded user trust. *De facto*, they are a different kind of product compared to a kitchen knife or a microwave: they don't "just work", they barely work. They lack interfaces that naturalistically guide humans to an intuitive, physicalized comprehension of their function. What's worse, they block and obscure the function of the underlying device — long delays, bugs, and inconsistent behavior can make an action as simple as turning lights on uncertain. Your analog lightswitch never does that — who would buy a lightswitch with a 1% chance to not switch lights, or that takes five to ten seconds to do so? End users can't modify or fix these devices, and they may stop functioning spontaneously in a way that's entirely out of their power to influence. We hoped

[2]Not that smartphones were conceived at the time.

IoT devices would be magic wands that opened up the world around us, but instead we've been sold limited-use, flaky trinkets peddled by hedge wizards.[3]

This doesn't mean that the vision we strove for is inaccessible, but if we want to get there, we need to be aiming to building magic wands and not flaky trinkets. The key weaknesses I want to extract from this discussion of contemporary IoT are as follows:

1. Deficient application design and validation
2. Lack of local intelligence
3. Connectivity problems

This thesis aims to improve the state of affairs for (3), and in so doing incidentally helps enable (2). My vision for the Internet of Things is one where the word "Internet" is understood to mean a genuine inter-network — i.e. one where the devices on the network actually speak to and through each other. This requires that devices be able to network *robustly* and *implicitly*.

[3]With service contracts the hedge wizards can unilaterally cancel!

2. Problem

The work proposed here is rooted in an observation that standard, well-understood, robust networking approaches can solve problems that are commonly reinvented and re-solved at the application layer. The fragmentation in IoT discussed in Section 1 is substantially a result of backwards- and mutually-incompatible networking standards. While it is in theory possible to provide compatibility between many of these technologies, in practice, to my knowledge, there are no facilities compatible with embedded firmware that close these gaps.

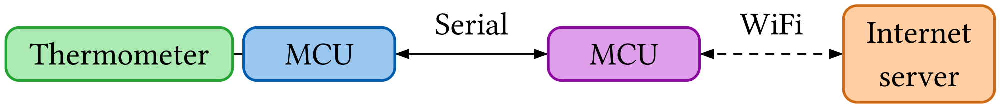

Figure 1: Prototypical application-specific bridge scenario.
We aim to get sensor data gathered on the blue node to the internet server.

Suppose we are developing an embedded project that reads data from a microcontroller over a serial connection and reports it to an Internet server over WiFi. A typical way to design this system would be to write an explicit application-layer bridge between the serial protocol and the WiFi connection — which would decode delimited frames, convert them to a network frame representation, and send the resulting frames over the WiFi interface.

This approach is an implicit manual reimplementation of network bridging — the serial connection can be seen as a candidate network link, and from this perspective, the message conversion (serial to WiFi) is a violation of the end-to-end principle [7]. I.e., if we treat the serial channel as a network link, our node (purple) should not inspect and transform the application-specific traffic.

Application-specific bridges are neither standardized nor portable. Changing the application message format (here: across the serial link) requires firmware changes to both the originating MCU and the bridge. Generally, we expect to see a separate bridge implementation for each pair of network technologies that must communicate, which implies combinatorial blowup — the space of possible application-specific bridges is both very large and very sparse.

2.1. Suggested solution

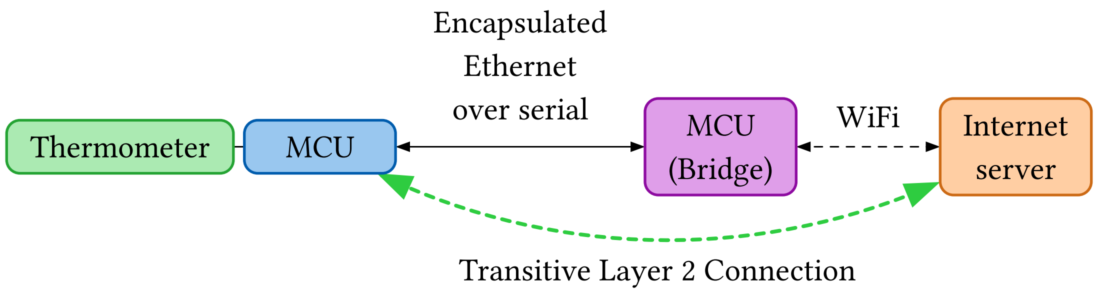
Encapsulated Ethernet over serial
Thermometer
MCU
MCU (Bridge)
WiFi
Internet server
Transitive Layer 2 Connection

Figure 2: Desired: network bridge provides a transparent connection to the remote MCU.

What we want in this situation is instead 1) a way to treat the serial channel as a network link, and 2) a generic way to bridge that link with the existing WiFi network. This way, the remote microcontroller (blue) can incorporate the functionality it requires to service its application needs, and our node (purple) is decoupled from those needs. As it's a bridge, our node automatically forwards traffic between the serial and WiFi interfaces. In a conventional IP network, this function would be performed by a Layer 2 switch — I suggest that implementing such a switch in firmware is a good minimal model for supporting desired functionality, and is the approach I develop in Section 3.

While this discussion has been motivated from the perspective of eliminating application-specific bridges in an Internet-oriented scenario, this is only one perspective through which we can appreciate the benefits of better enabling generic networking approaches.

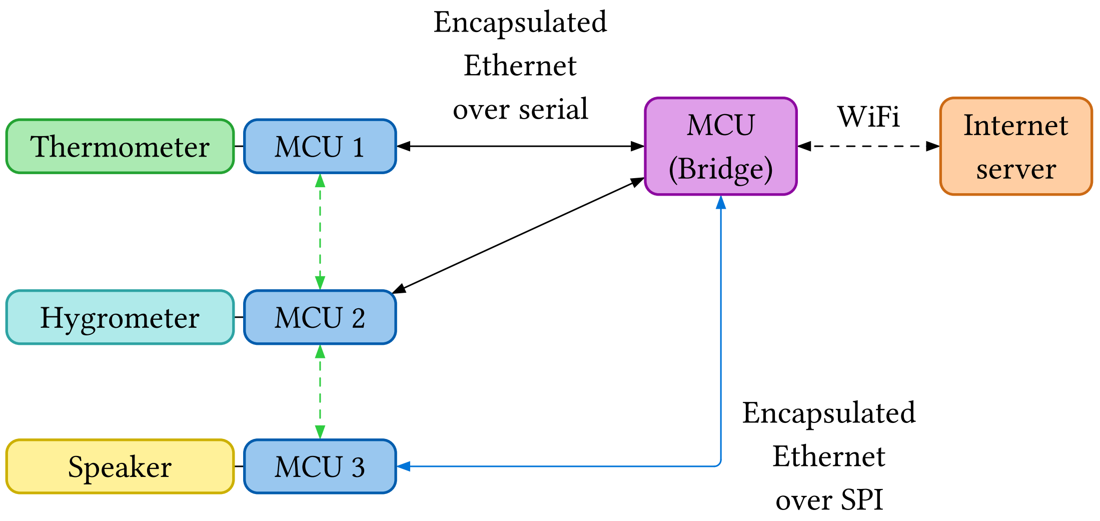

Figure 3: Layer 2 bridge enables networking of multiple devices over different media. Devices can communicate via the bridge to the Internet, and can communicate to each other without traffic leaving the LAN.

Treating unconventional communication channels as network links enables us to build genuine networks over devices that would otherwise be excluded from participation (requiring application-specific bridging otherwise, greatly limiting flexibility and increasing costs of adapting functionality). Furthermore, this approach enables embedded network construction without external infrastructure — the next conceptual step beyond a single bridge device is *all* devices running a bridge:

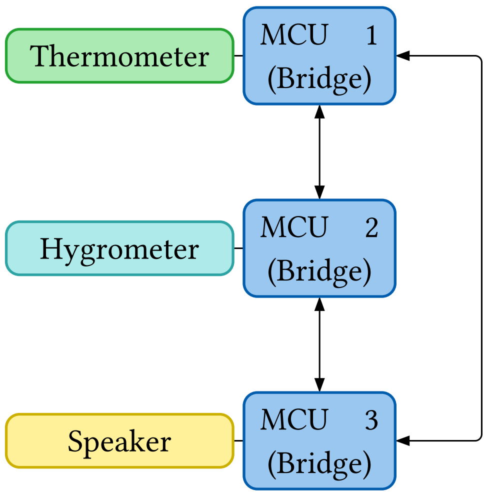

Figure 4: Devices connected in a dense network are able to network between each other, accomplishing application goals without Internet access. We can imagine, for instance, that MCU 3 sounds an audible alarm if environmental parameters are outside an acceptable range.

Especially interesting specializations for this kind of network (beyond IoT generally) include wearable textile networks (such as those that might be built out of devices like [8]), swarms of low-power robots, and in general large heterogeneous wireless sensor nets. This approach can enable networks to self-organize and evolve over time as nodes come on and offline, and application requirements demand new capabilities. No central controller is required, and networks based on standard internet technologies enabled by bridges would experience no lock-in to a particular physical-layer technology (e.g. ZigBee, Thread, ZWave). Individual nodes in these networks would be able to be addressed directly from the network edge, e.g. by a user on a smartphone or laptop; or over the Internet if the device net is connected.

2.2. Related Work

To my knowledge, there do not presently exist tools that support network bridging functionality for microcontroller-based embedded systems, or which make it easy to convert arbitrary communication channels to network links on these platforms. The nearest functional parity exists in mesh network technologies like the following:

Technology	Open	Central Device Optional	Medium-agnostic	MCU Bridge
Bluetooth Mesh	~	~	✗	✗
LoRaWAN	~	✗	✗	✗
Meshtastic	✓	✓	✗	✗
Zigbee	✓	✗	✗	✗
ZWave	✗	~	✗	✗
Thread	~	✓	✗	✗
6TiSCH [9], [10]	✓	~	✗	✗
interstice (This thesis)	✓	✓	✓	✓

Table 1: Comparison of meshing technologies.

However, these technologies do not generally interoperate, at least at a local or device level. While some standards are open, nearly all strictly require a specific radio technology to participate in the network. To make this point clear, despite the fact that many of these technologies use IEEE 802.15.4 frames, there do not exist (to my knowledge) portable means of bridging connections between these media at Layer 2 on an embedded device.

While embedded IP suite network stacks exist in abundance (e.g. lwIP, uIP, smoltcp), they do not ship with Layer 2 bridging functionality, and are often difficult to modify due to tight coupling within firmware platforms.[4]

Net Stack	Type	MCU Compatible	L2 Bridge	*Easily* modified / Extensible
lwIP	standalone	✓	✗	✗
FreeRTOS	RTOS	✓	✗	✗
Contiki [11]	RTOS	✓	✗	✗
Zephyr	RTOS	✓	✗	✗
smoltcp	standalone	✓	✗	~
embassy-net	standalone	✓	✗	~
Linux	full OS	✗	✓	✗
interstice (This thesis)	standalone	✓	✓	✓

Table 2: Comparison of firmware networking stacks.

2.2.1. Matter[5]

Matter is an open application-layer messaging standard for IoT devices, developed by an industry consortium containing members including Google, Apple, and Amazon. It is pertinent to this thesis as its design is motivated by many of the same factors — it is also an attempt to reduce fragmentation in IoT. Matter is not locked to a specific wireless radio type, and is designed for offline, device-to-device connectivity fallback. It attempts to make use of existing standards rather than reinventing the wheel, and appears to be enjoying some success in applying unification pressure in the home IoT market.

However, Matter has major weaknesses as a protocol and system — it is deceptively closed. While its specifications and SDKs are open, participation in a Matter network requires devices to have cryptographic certificates signed by the controlling consortium, which demands substantial up-front, recurring, and per-device fees. For large companies like Google and Apple, these costs are minimal, but they can be onerous for startups, and categorically exclude grassroots device designs by hobbyists and tinkerers. While it is technically true to say that Matter supports multiple wireless technologies, this only includes two: WiFi and Thread (which also imposes per-device licensing costs).

[4]lwIP is commonly integrated in Arduino cores, for instance, but its configuration is typically baked-in, requiring a source code fork to e.g. enable IP forwarding.

[5]https://csa-iot.org/all-solutions/matter/

For our application, these limitations exclude Matter as a solution — it does not provide solutions for improving network access, instead specifying a limited set of network technologies and relying on routing at Layer 3.

2.2.2. Higher-layer compatibility efforts

Many research projects, e.g. [12], [13], [14] has been undertaken to improve and demostrate models for compatibility between devices at higher layers in the network stack. CoAP [15] has been standardized as the IETF as a standard protocol filling a niche similar to HTTP, but better oriented towards embedded devices.

Similarly to Matter, however, these efforts do not improve lower-layer network connectivity between devices themselves, so they do not address our central concern.

2.2.3. Point-to-Point Protocol, Serial Line Internet Protocol

Point-to-Point Protocol (PPP) [16] and Serial Line Internet Protocol (SLIP) [17] are protocols for encapsulating IP traffic over point-to-point links, and are frequently used with serial connections on microcontrollers. It is possible, for instance, to use either of these protocols to provide a network connection to a microcontroller-based device from a desktop computer over a serial connection.

Prima facie, this technology seems like it solves many of our problems. But while encapsulation is useful (and indeed I intend to use PPP for these capabilities; see Section 3.2), the fact that they encapsulate Layer 3 traffic makes them insufficient on their own to support transitive networking. In short, bridging and addressing both remain unsolved problems,[6] where by contrast, operation at Layer 2 provides straightforward answers[7].

[6]How do we form a *network* of PPP links? How do we propagate routing information? How are addresses assigned?
[7]MAC addresses can be derived randomly or from device identity (e.g. serial number), and packet switching is simple.

3. Approach

This thesis develops a platform-agnostic, embedded-friendly firmware packet switch, *interstice_switch*, with a minimal, easy-to-use interface for implementing new Layer 2 drivers and adapting existing ones.

The intent of this project is that embedded devices will run an IP network stack that attaches to the *interstice* switch at Layer 2. The switch is able to accept and bridge additional Layer 2 connections over the microcontroller's peripherals. For instance, the device might attach drivers for its WiFi radio and a UART port to the switch. This would enable it to make IP-based connections over the UART *or* over WiFi, and further, it would be able to share a WiFi connection to neighbor nodes through the UART port. This capability extends transitively, in the sense that other peers running *interstice_switch* that have a transitive connection to our microcontroller could make use of its WiFi connection (and vice versa). While it would be typical for all nodes in a network to run both the switch and IP stack, it is important to note that the design permits either component to be freely omitted if desired (e.g. to limit memory usage — see Figure 5).

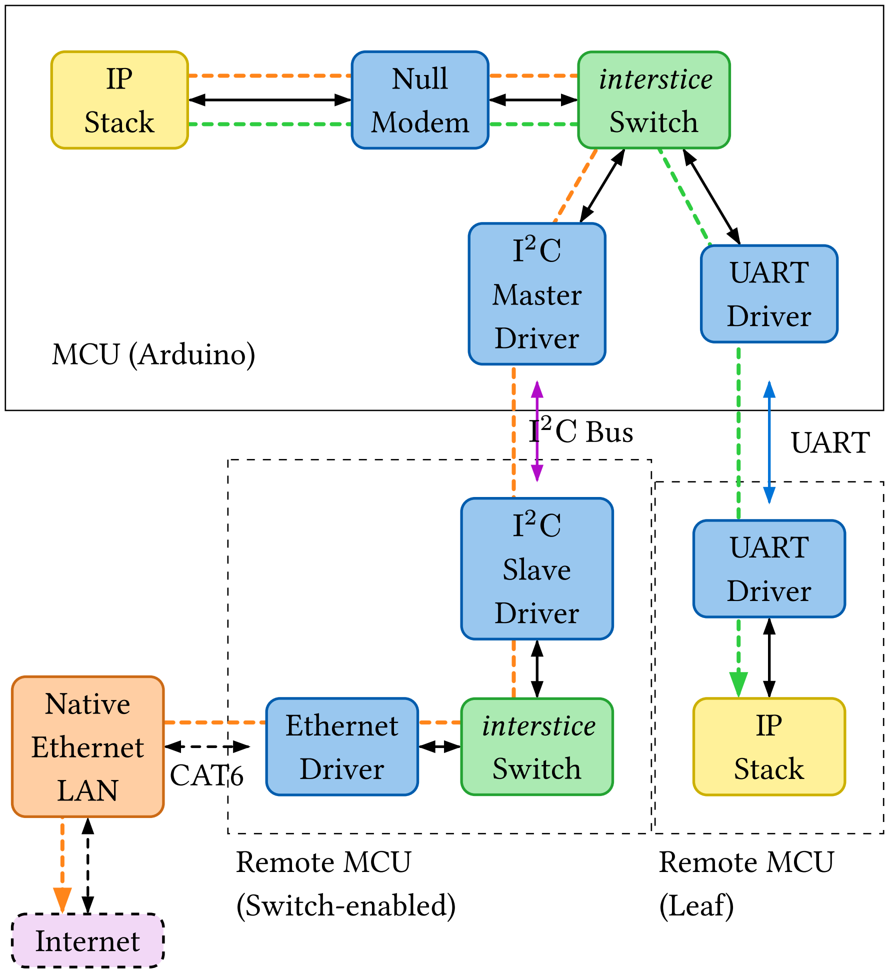

Figure 5: Sample network topology (Layer 2).
Green dashed line indicates transparent local connection from top microcontroller.
The remote device (bottom-right) is not running a switch to conserve memory.
Orange line indicates connection to LAN and Internet from the same microcontroller.

It is an explicit intention of this work to make it accessible for users to write their own Layer 2 drivers over arbitrary PHYs and overlays — I aim to make it easy to cross "last mile" connectivity gaps[8] to ensure it's always possible to get a device on a network, even if the device lacks appropriate dedicated networking hardware (such as using a UART as a network link in the above example). I view bidirectional communication channels as fundamental primitives for networks, and believe that we should always be able to get a device online if it can communicate somehow.

That being said, users should only need to write drivers if their use case is uncommon, so this work also involves developing a collection of reference adapter drivers to address common needs. WiFi and Ethernet drivers will be provided for targeted microcontrollers[9], as well as generic drivers for transporting Ethernet frames directly over UART, I2C, and SPI. A ZigBee or Thread driver will also be implemented, as well as a TAP adapter, minimally including Linux support. Most likely, many more drivers will be built, but this represents a minimal set.

The proposed firmware library is developed in the Rust programming language using the `embassy` [10] family of firmware libraries (centrally `embassy-net`). As a result, the firmware is entirely portable and indeed not restricted to embedded devices; the same packet switch code can be used in standard desktop operating systems for userspace overlay networking. This enables the creation of a natural simulation environment based on construction of in-memory *interstice* switches connected in desired test topologies, whose switching functionality is identical to what runs on embedded microcontrollers.[11]

3.1. Generic switch

The *interstice_switch* firmware library implements a generic packet switch, which is parameterized by an underlying network medium (e.g. Ethernet). The medium is represented by a language interface (`trait` in Rust parlance), whose implementation merely needs to support (de)serialization to/from bytes and retrieval of source and destination address for each frame. Addresses are treated as opaque-but-comparable types, such that they can be entered into and looked up in a MAC table. This abstraction makes the same switch implementation compatible with any packet- or frame-like message representation I've considered; the current implementation has shown initial success with Ethernet frames and is expected to function with 802.15.4 (though this is untested as-yet). In theory, it should also be possible to support Layer 3 switching as well.

The core switching algorithm is conventional: source addresses for incoming packets are stored in a MAC table, and outgoing packets are sent to the port registered for the destination address, if present. Otherwise, the packet is flooded to all ports other than the source. The MAC table is treated as a least-recently-used cache; the oldest entry is flushed if it is full when a new address is discovered.

[8]Of the form e.g.: these devices are right next to each other and both have USB ports — they *should* be able to talk to each other.

[9]Minimally, RP2040, RP2350, ESP32C3, ESP32C6, but I'm aiming to provide broader support by leveraging Arduino as a platform abstraction layer.

[10]https://github.com/embassy-rs/embassy

[11]Early provisional results in the simulation environment have demonstrated successful bidirectional traffic bridging from my home network via TAP device — I have successfully made HTTP requests to the public Internet from userspace via this *interstice* link.

Each "virtual port" in the switch is backed by an implementation of the `embassy_net::Driver` trait, whose essential function is to define how packets are sent and received. Ethernet and WiFi adapters are be implemented in terms of this trait, for instance — as will be UART, SPI, and so on. The `Driver` trait is dynamically dispatched; the ports for a given switch need only share the same underlying medium and maximum transmission unit (MTU). Notably, the `Driver` handles any delimitation, framing, packetization, fragmentation, and/or reassembly, but is neither necessarily Layer 2-aware nor required to adhere to any particular communication specification — custom physical-layer transports[12] are entirely possible.

I am writing two parallel implementations of the switch behavior, one of which supports zero-copy operation based on statically allocated buffers (maximally embedded-friendly), and the other assumes a global heap allocator is available, for faster development iteration and a nicer interface.

3.2. Framing, Point-to-Point Protocol over Serial

PPP [16] is a Layer 2 protocol with useful functionality such as error recovery and link state detection. `embassy-net` provides a Point-to-Point Protocol over Serial (PPPoS) implementation that is generic over any type implementing `io::AsyncRead` and `io::AsyncWrite` types; i.e., any bidirectional communication channel (serial ports, SPI devices, etc.) is relatively straightforward to adapt to use PPP. This capability is very useful, so it will be used for framing in most cases.

As PPP specifies a fixed set of protocols it can encapsulate, and it cannot carry generic payloads, using it this way will involve a method of encapsulating Layer 2 traffic within the encapsulated IP frames and disregarding the addressing information. As a fallback, if this proves too convoluted, I will fall back to using Consistent-Overhead Byte Stuffing (COBS) [19] for framing instead.

3.3. Bus links

Specialized driver support will be provided for embedded buses that have a master/slave architecture (e.g. SPI, I^2C). This architecture requires specific attention because slaves cannot initiate communication in-band, but bidirectional communication will require this in general. So either:

- the master must enumerate and poll slave devices periodically
- slave devices need an out-of-band mechanism to signal data readiness (typically, an IRQ line)
- the bus is homogeneously multimaster, or
- the bus is hybrid multimaster, with some slave-only devices.

A polling implementation for both SPI and I^2C will be developed first as the minimum viable case and extended with IRQ-driven and multimaster support.

3.4. Asynchronous Token Protocol

Neil Gershenfeld's Asynchronous Token Protocol (ATP)[13] is an autonomously-clocked, point-to-point serial link that can be implemented at high speed on RP2040 Programmable I/O (PIO), field-programma-

[12]Using a custom encoding, or over alternate media — e.g. optical, audio, carrier pigeon [18].

[13]https://academy.cba.mit.edu/classes/networking_communications/ATP/hello.ATP.RP2040.send.py

ble gate array (FPGA), or bitbanged on a microcontroller. A Layer 2 driver implementation supporting minimally the PIO version of this protocol will be provided, along with an evaluation of its effects on network performance.

3.5. Spanning-Tree Protocol

Spanning-Tree Protocol (STP) is a protocol typically implemented by bridges in which they cooperate to share information about network topology. This functionality is used to selectively disable links that would cause loops in the network graph in order to avoid broadcast radiation effects. Since part of the goal of this work is enabling highly-connected networks, and as I intend to engage novice users, this property is desirable to mitigate low-visibility failures and performance degradation. I will be implementing STP support as a configurable mode of operation.

However, it is worth mentioning that STP is a configuration hazard, as there are several versions of the protocol that can mutually conflict (each network must homogeneously adopt a single version for proper function), and some extant network configurations block links that emit STP traffic[14]. It will require some care to present this functionality as an option to novice users — presently, I am inclined to disable the functionality by default and place the responsibility for avoiding loop creation on users by default, but provide clear examples and documentation of situations where it is appropriate to enable it.

3.6. Adoption

It is my view that it is critically important that this work be made available as a usable, open-source tool. In that interest, developing interfaces that are intuitive to novice users is a relevant part of this thesis. I will provide documentation and examples for the use of *interstice*, and undertake a design iteration process with users to ensure the software is accessible for even novices with limited networking experience.

3.6.1. Arduino layer

The Arduino platform has proven to be highly accessible and is now ubiquitous in the firmware space. It is used as much as a tool for novice learners as for development of substantial firmware projects. In the interest of attracting attention and improving adoption of the library, I will provide an Arduino compatibility layer for *interstice*.

This will be accomplished using Rust's C FFI facilities and the `bindgen`[15] and `cbindgen`[16] projects. Precompiled library archives will be provided for common microcontroller ISAs, and Arduino- specific Layer 2 drivers for common embedded peripherals will be provided (e.g. `WiFi.h`, `Ethernet.h`, `Wire.h`, `SPI.h`).

[14]Especially large enterprise networks, such as those found at MIT.
[15]https://github.com/rust-lang/rust-bindgen
[16]https://github.com/mozilla/cbindgen

3.6.2. Documentation & examples

A substantial body of documentation and working examples will be developed to explain and demonstrate *interstice*'s function, especially focused on common use-cases. This documentation will be provided as a website.

3.7. Case study: *ocularium*

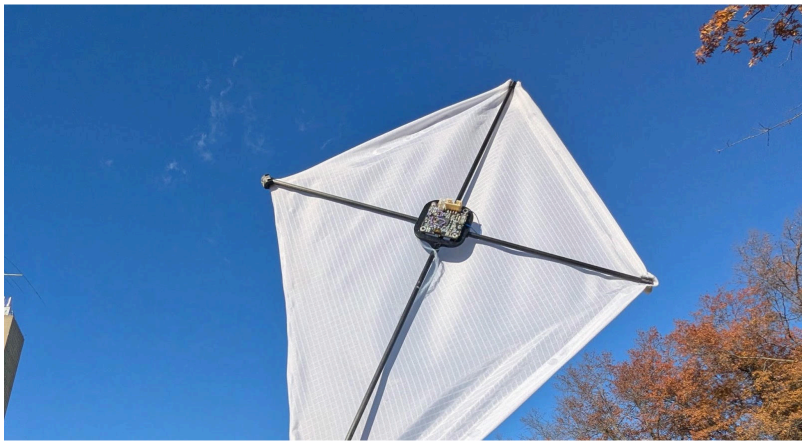

Figure 6: ocularium *node mounted on a kite.*

ocularium is a tiny sensor node I designed to be carried on kites for atmospheric monitoring, with sensors including air quality, temperature, motion, ambient light, and a microphone. It represents an excellent case study for validating *interstice* in the real world. The primary intent of this case study is to exercise and evaluate *interstice*'s ability to provide networking services in less-than-ideal connectivity conditions, with the aim of surfacing any bugs and unfounded assumptions not experienced in laboratory conditions.

ocularium's hardware has already been designed and fabricated, but it will require a revision to add the LoRa radios.

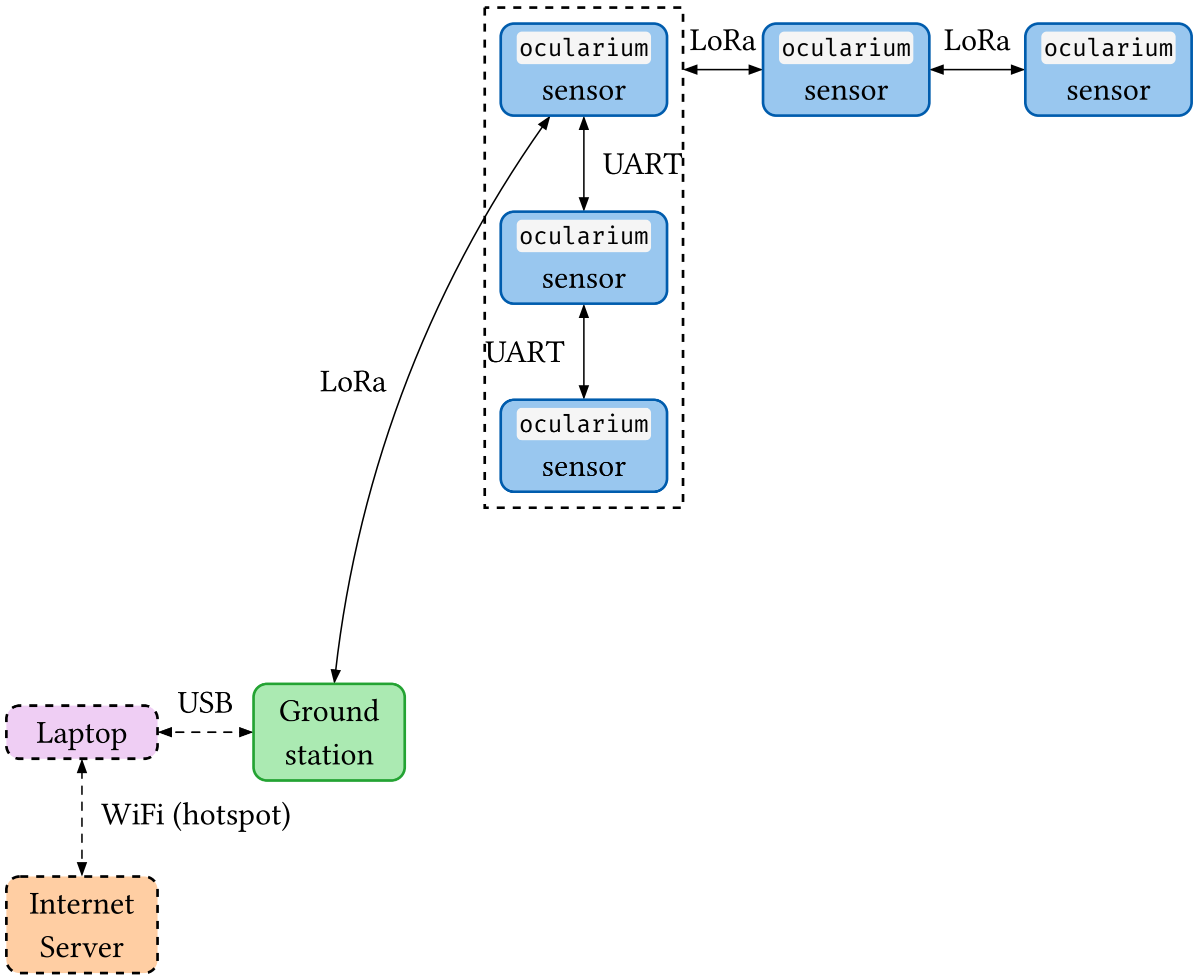

Figure 7: Envisioned ocularium *deployed network.*

I am planning to deploy *ocularium* with *interstice*-enabled firmware on flown kites for several hours across at least two separate occasions. The kites will be linked with LoRa radios, and will store sensor data on onboard SD cards. I will fly at least three kites, one of which will be carrying multiple *ocularium* nodes onboard. This kite will network the sensors onboard over UART links, and only one of the nodes will have a LoRa radio.

Data from the kites will be opportunistically downlinked to a single ground station as the LoRa connections permit. LoRa configuration and physical kite spacing will be tuned to maximize hop-count (to apply some stress on the *interstice* implementation), and only one kite will be in range of the ground station.

Network timing metrics and logs will be recorded for the duration of these flights from the perspective of all network members, to be analyzed as discussed in Section 4. I will use this data to gain an understanding of *interstice*'s performance and robustness, and develop iterative improvements as I conduct additional flights.

ocularium's design is open source — hardware designs and firmware source will be supplied along with the final thesis.

4. Evaluation

There are three key ways in which this work will be evaluated:

1. Functional validation
   - The system must adhere to relevant networking specifications.
   - The system must function in real-world contexts.
2. Performance metrics
   - Pragmatically, a performance floor exists below which the system is not useful (extremely slow throughput, substantial packet loss); I aim to ensure that the system meets a minimum reasonable performance standard by comparison to existing networking technologies.
   - Gathering performance data will enable comparison to existing systems and may guide future work.
3. User accessibility
   - *interstice* is designed to be accessible to novice users; I will gather and report upon user feedback to ensure that the system is both accessible and useful.

4.1. Functional validation

Designing a library to enable compatibility between networking technologies is only useful if the implementation is both theoretically correct and interoperates properly with those technologies in the real world. It is important to gather certain data to ensure that the design complies with these requirements.

4.1.1. End-to-end function

The simplest means of evaluating whether *interstice* functions properly is to bridge it to existing networks and evaluate binary end-to-end function of the attached network stack and application-oriented functionality (e.g. HTTP requests, MQTT-based pub-sub, and so on). Application failure is a straightforward way to gain visibility into function. This evaluation is performed as a natural constituent part of ongoing system development.

The *ocularium* (Section 3.7) case study deployment will depend upon extended correct function of its *interstice*-based firmware in a real-world scenario, and hence also provides an opportunity to evaluate function under more-strenuous conditions.

4.1.2. Integration tests

An automated test suite will be developed for *interstice* that exercises and validates behavior of the implementation in a few key ways. It will run in the simulator environment, and test functionality for networks of various device sizes with multiple topologies:

- Using strictly Layer 2 packets, with no IP stack
- With an IP stack, using Layer 4 communication protocols directly
- Using embedded-relevant application-layer protocols such as CoAP, MQTT, and HTTP

Network topologies to be tested in all cases include:

- Minimal network: two devices only
- Typical DAG-like network topology emulating an office or home LAN, 10 - 100 nodes
- Pathologically long network graph — a long single chain of devices
- Pathologically dense network graph — fully-connected, with many nodes (STP enabled)

The test suite will be supplied as part of the open-source code release.

4.1.3. Packet sniffers

I will leverage third-party packet-sniffers and traffic analyzers such as WireShark[17] and tcpdump[18] to capture complete network frames and serve as an independent cross-validation tool for network behavior and packet formatting. As these tools are free, open-source, and in common use within the networking space, their packet capture formats also represent a useful way to share exemplar network conversations to demonstrate *interstice* functionality; packet captures will be included as artifacts with the final thesis writeup.

4.2. Performance

Below is a table of performance metrics that will be collected.

Metric	Units	Test Condition(s)	Target Value
Throughput (10x burst)	Hz	Loopback	5000
Throughput (steady-state)	Hz	Loopback	1000
Throughput (10x burst)	bit/s	Loopback	5Mbps
Throughput (steady-state)	bit/s	Loopback	1Mbps
Return-trip time	s	Loopback	1ms
Packet loss	%	Loopback	0%
Throughput (10x burst)	Hz	*ocularium*	1000
Throughput (steady-state)	Hz	*ocularium*	100
Throughput (10x burst)	bit/s	*ocularium*	100kbps
Throughput (steady-state)	bit/s	*ocularium*	50kbps
Return-trip time	s	*ocularium*	< 20ms
Packet loss	%	*ocularium*	< 10%
Energy usage	J/bit	*ocularium*	
Flash usage	bytes	*ocularium*	75kB

[17] https://www.wireshark.org/
[18] https://www.tcpdump.org/

Metric	Units	Test Condition(s)	Target Value
RAM usage	bytes	*ocularium*	25kB
CPU usage	%	*ocularium*	10%
Maximum network scale	nodes	simulator	1000

Table 3: Performance metrics of interest.

The loopback condition considers an *interstice* switch node running in memory on a desktop or laptop computer sending UDP packets through a loopback layer 2 *interstice* adapter.

The *ocularium* condition considers the deployed *ocularium* case study configuration (Section 3.7), using packet timing data recorded across a single LoRa link between two kites.

Energy consumption will be recorded on an averaged basis by subtracting a baseline measurement with *interstice* disabled (transmitting and receiving the same number of dummy packets) from trials with it enabled and actively switching packets. This metric will be measured in a laboratory setting.

Target values should be interpreted as minimal estimates — system performance worse than the target merits further investigation. I will aim to fix these problems, and report on any surprising results in the thesis writeup. I intend to spend time optimizing regardless, and aim to beat these target metrics by a substantial margin.

Plots of relevant metrics (throughput, RTT, packet loss) as a function of network node count will also be evaluated in the simulation environment to discover practical limitations and guide investigation of the reason for those limits.

4.3. Accessibility

An evaluation of user feedback on the Arduino interface will be performed, summarizing feedback from users and outcomes of the iterative development process. This thesis does not aim to make conclusions about psychology of software or HCI concerns, but it is important to recognize that this evaluation has the pragmatic goal of improving the work, and to that end, I will gather certain metrics to measure its utility and weak points.

I will record quantitative survey measures of difficulty-of-use and utility comparing *interstice*'s Arduino interface to existing networking libraries (`Ethernet.h`, `WiFi.h`) in the Arduino ecosystem. This feedback will be used to guide focus on development iteration.

4.3.1. Methods

I will ask users to perform certain tasks (e.g. make an HTTP request, synchronize time with NTP, send packets to a peer device) using both the standard Arduino libraries and *interstice*. Users will need to succeed at a task using Arduino libraries in order to attempt the same task using *interstice*.

Before performing the tasks, users will be asked to report their level of experience with Arduino, how frequently they write firmware, and their familiarity with networking principles.

After performing the tasks, users will be surveyed for open-ended qualitative commentary and questions on a 3-point scale:

- Comparative difficulty of use (*interstice* is harder to use / equal difficulty / Arduino is harder)
- Comparative utility of documentation (*interstice* was better / equal / Arduino was better)
- Subjective utility (*interstice* is very useful to me / somewhat useful / not useful)
- Frequency of use (Would use *interstice* in most of my Arduino projects / some projects / no projects)

5. Ethical considerations and unintended consequences

Direct ethical concerns with regards to this work are limited — contemporary embedded networking approaches already present data privacy concerns, and part of the utility of this work is to present an alternative to centralized, cloud-based data processing, where data may be collected and sold.

That said, security concerns are present. In enabling more and better network connectivity between devices with limited computation capabilities, we open up additional attack surface that must be secured against malicious actors. Many vulnerabilities have been uncovered in IoT and networked sensor systems that can compromise sensitive information, which is especially concerning in locations with expectations of privacy.

It must be noted however that networking attack vectors are intrinsic to lower-layer unsecured networking protocols such as Ethernet and IP; and this thesis only promulgates wider access to these unsecured network layers. One can imagine attaching an *interstice*-enabled node to a network to be a similar kind of action as plugging a device into an Ethernet-enabled RJ45 port or attaching it to a WiFi network — the same kinds of security considerations should be taken into account, with respect to networking protocols used and trust in other devices on the network.

Novel attack surface exposed by *interstice* is the firmware itself, which mitigates attack to some extent through the use of the Rust programming language, which is memory-safe by default. Published libraries and documentation will, however, include warnings that *interstice* has not undergone a thorough security vetting process.

The thesis writeup will include a section recommending security approaches in embedded networking, including secure protocols such as (Datagram) Transport Layer Security ((D)TLS) and a Public Key Infrastructure (PKI) provisioning process capable of authenticating and revoking device-specific keys. Future work could include the design and implementation of a PKI provisioning server designed to work with *interstice*, or a recommendation for configuring an existing solution.

Time permitting, the *interstice* reference implementation will be subjected to a preliminary fuzzing analysis to test for vulnerabilities.

6. Timeline & Resources

In order to complete this thesis, required resources include: electronics testing and debugging equipment; PCB manufacturing services; 3D printing capabilities (FDM and SLA); and a workstation computer for firmware development, simulation, and evaluation.

Equipment available in the Responsive Environments lab, along with the laptop computer provided to me by the Media Lab, are sufficient to perform this work.

7. Addendum

This section reports an updated plan based on feedback from my committee, which at a high level recommended the following changes:

1. Focus on developing IP over heterogeneous embedded serial links
2. Reframe approach to use Layer 3 routers rather than Layer 2 bridges
3. Consider opportunities to reduce IP overhead (e.g. header compression)
4. Adopt a reference system to drive design and evaluation

My updated approach focuses specifically on bringing IPv6 to heterogeneous embedded serial links, such as UART, RS-232, I^2C, SPI, CAN, and 1-Wire. The basic model is to use PPP or SLIP over typical point-to-point connections, and develop specialized link protocols for buses such as I^2C, which admit special treatment due to their existing addressing and broadcast capabilities. IP header compression approaches will also be developed and evaluated to minimize impact on link budget.

Nodes in the developed networks will support packet forwarding via IP routing techniques, and will as much as possible adopt standard techniques and protocols to support this functionality, in order to enable interoperability with existing networks. However, network bootstrapping protocols (such as DHCP and ND) are organized around the scope of an underlying shared link, and assume that subnet extent is synonymous with link extent. These assumptions disagree with our constraints, given that simple serial links are point-to-point (and yet it is useful to define logical subnets of size larger than 2), so some specialization of this functionality will be necessary, e.g. to support dynamic address assignment.

A reference system will be developed to drive implementation and evaluation of this work, built on top of Jake Read's prior work on "controllerless machines". Jake's systems support machine prototyping and motion control using a standardized library of hardware components, primarily comprising "smart" actuators and communication bridges, which are wired together to form a distributed, networked machine. Jake's network stack is a custom one of his own design; I will substitute an IP stack using the work developed in this thesis and evaluate its performance in this latency-sensitive environment.

7.1. System implementation plan

The reference system will be a machine built using Jake's modules, likely targeting a 2.5-axis CNC milling application that will serve to provide real-world test cases. I plan to implement the network on this system incrementally, starting with a minimal version capturing low-hanging fruit, then adding capabilities on an ongoing basis.

The initial version will only use PPP or SLIP links over RS-232 with static routing tables and node addresses. I will treat this as a baseline to validate system functionality and performance quickly, without initially getting bogged down in the complexities of selecting and implementing the more involved functionality to come later.

Subsequent versions of the system will incrementally develop and integrate dynamic address assignment, dynamic route discovery, border router/default route discovery, specialized Layer 2s (for e.g. I^2C), and header compression.

7.2. Optimization efforts

In an embedded context, IP stacks are relatively heavy to implement in terms of code size, memory usage, and link budget; hence, opportunities to optimize these quantities are attractive.

7.2.1. Header compression

IP header compression is a technique to avoid retransmitting repeated information by encoding only differences between packets headers, where possible. These techniques are well-standardized for PPP and SLIP, but may not be present in existing embedded libraries. I will ensure that these functions are enabled, implement them where missing, and evaluate their impact on system performance.

7.2.2. Bus links

Buses such as I^2C and CAN present their own opportunities for optimization of IP traffic, as they have existing addressing, broadcast, and length indication semantics. Much of the IP header could be omitted by nodes communicating via these protocols, as long as an addressing scheme is chosen with a bijection between IP addresses and bus addresses — in this case, the IP address fields can be completely stripped from the header. Length fields can also be eliminated (including potentially at Layer 4), as the bus transaction indicates its own length, suggesting that the combination of these approaches could eliminate 34 out of the 40 bytes of the IPv6 header, without the requirement to retain per-flow state.

For a machine application especially, individual messages may be small, so the reduction of this overhead could represent a substantial reduction in traffic.

7.3. Routing

In the general case, information about network topology within our system is unknown *a priori*, but it is required to perform packet forwarding. The function of discovering and propagating network topology information is conventionally performed by an Interior Gateway Protocol (IGP), such as the Routing Information Protocol (RIP), Open Shortest Path First (OSPF) protocol, or Interior Border Gateway Protocol (iBGP).

However, certain challenges are present in porting an existing IGP to embedded systems, most notably, that the memory required to produce an optimal route is linear in the size of the network, but memory is a constrained quantity on embedded systems, so it is desirable to minimize its usage. While prior art has shown that it is possible to reduce memory overhead, doing so typically comes at the cost of route optimality. In the setting of a distributed machine, that tradeoff is expensive, as latency is a critical performance quantity. This component of the work will therefore focus on empirically evaluating suitability of different routing approaches in the context of the reference system and adopting one that makes favorable tradeoffs.

7.4. Network bootstrap

Nodes in our system must be able to adopt dynamically-assigned IP addresses and discover border routers. However, conventional network bootstrapping approaches that would typically perform these functions (DHCP, IPv6 neighbor discovery) depend on a shared link with broadcast semantics, which we lack, as links are assumed point-to-point in general.

This component of the work will evaluate how best to perform this function — I will investigate simple propagation of information from Router Advertisement messages, as well as stateless, default-assignment of Unique Local Addresses (ULAs) and communication of bootstrapping information at higher protocol layers.

References

[1] "interstice." GNU Collaborative International Dictionary of English. Accessed: Oct. 28, 2024. [Online]. Available: https://gcide.gnu.org.ua/?q=interstice&define=1

[2] M. Weiser, "The computer for the 21st century," *ACM SIGMOBILE mobile computing and communications review*, vol. 3, no. 3, pp. 3–11, 1999.

[3] M. Aly, F. Khomh, Y.-G. Guéhéneuc, H. Washizaki, and S. Yacout, "Is Fragmentation a Threat to the Success of the Internet of Things?," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 472–487, Feb. 2019, doi: 10.1109/JIOT.2018.2863180.

[4] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in Internet of Things: Taxonomies and Open Challenges," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, Jun. 2019, doi: 10.1007/s11036-018-1089-9.

[5] M. A. Jamshed, K. Ali, Q. H. Abbasi, M. A. Imran, and M. Ur-Rehman, "Challenges, Applications, and Future of Wireless Sensors in Internet of Things: A Review," *IEEE Sensors Journal*, vol. 22, no. 6, pp. 5482–5494, Mar. 2022, doi: 10.1109/JSEN.2022.3148128.

[6] J. Rowling, "Harry Potter and the Chamber of Secrets," Scholastic Press, 1999, p. 329.

[7] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[8] A. Dementyev, H.-L. (. Kao, and J. A. Paradiso, "SensorTape: Modular and Programmable 3D-Aware Dense Sensor Network on a Tape," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, in UIST '15. Charlotte, NC, USA: Association for Computing Machinery, 2015, pp. 649–658. doi: 10.1145/2807442.2807507.

[9] D. Dujovne, T. Watteyne, X. Vilajosana, and P. Thubert, "6TiSCH: deterministic IP-enabled industrial internet (of things)," *IEEE Communications Magazine*, vol. 52, no. 12, pp. 36–41, Dec. 2014, doi: 10.1109/MCOM.2014.6979984.

[10] X. Vilajosana, K. Pister, and T. Watteyne, "Minimal IPv6 over the TSCH Mode of IEEE 802.15.4e (6TiSCH) Configuration." [Online]. Available: https://www.rfc-editor.org/info/rfc8180

[11] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, Nov. 2004, pp. 455–462. doi: 10.1109/LCN.2004.38.

[12] S. Russell and J. A. Paradiso, "Hypermedia APIs for sensor data: a pragmatic approach to the web of things," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, in MOBIQUITOUS '14. London, United Kingdom: ICST (Institute for Computer Sciences, Social-Informatics, Telecommunications Engineering), 2014, pp. 30–39. doi: 10.4108/icst.mobiquitous.2014.258072.

[13] M. Kojima and S. Sakazawa, "Overcoming IoT Fragmentation Using a Web of Things Framework "CHIRIMEN" for Raspberry Pi 3," in *2021 IEEE 10th Global Conference on Consumer Electronics (GCCE)*, 2021, pp. 381–382. doi: 10.1109/GCCE53005.2021.9622090.

[14] R. Fantacci, T. Pecorella, R. Viti, and C. Carlini, "Short paper: Overcoming IoT fragmentation through standard gateway architecture," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Mar. 2014, pp. 181–182. doi: 10.1109/WF-IoT.2014.6803149.

[15] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)." [Online]. Available: https://www.rfc-editor.org/info/rfc7252

[16] W. A. Simpson, "The Point-to-Point Protocol (PPP)." [Online]. Available: https://www.rfc-editor.org/info/rfc1661

[17] “Nonstandard for transmission of IP datagrams over serial lines: SLIP.” [Online]. Available: https://www.rfc-editor.org/info/rfc1055

[18] D. Waitzman, “Standard for the transmission of IP datagrams on avian carriers.” [Online]. Available: https://www.rfc-editor.org/info/rfc1149

[19] S. Cheshire and M. Baker, “Consistent overhead byte stuffing,” *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 159–172, 1999, doi: 10.1109/90.769765.