IP Networks Over Heterogeneous Embedded Serial Links

by

Nathan Perry

B.A., Williams College (2018)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

May 2025

Authored By	Nathan Perry Program in Media Arts and Sciences May 16, 2025
Certified By	Joseph A. Paradiso Alexander W. Dreyfoos (1954) Professor of Media Arts and Sciences Program in Media Arts and Sciences Thesis Supervisor
Accepted By	Tod Machover Muriel R. Cooper Professor of Music and Media Academic Head, Program in Media Arts and Sciences

IP Networks Over Heterogeneous Embedded Serial Links

by

Nathan Perry

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning, on May 16th, 2025,
in partial fulfillment of the requirements
for the degree of Master of Science

Abstract

The Internet Protocol (IP) provides a number of key benefits to networked devices: it serves as a "narrow waist" enabling functional modularity by decoupling lower-layer devices from application behavior, it provides a notion of transitive connectivity and a number of standardized methods to achieve it, and most importantly, it is ubiquitous, enabling almost all networked applications to mutually communicate.

Many embedded microcontrollers cannot take advantage of the benefits of IP because they lack the dedicated networking hardware which is as a practical matter required to interact with nontrivial networks. I observe that multihop point-to-point IP networks can in principle be constructed over the communication media that microcontrollers commonly *do* have, such as UARTs, I^2C, SPI, and CAN bus, but software support is lacking to make this networking approach accessible.

Therefore, this thesis develops and evaluates `interstice`, a platform-independent, open-source software library designed to enable the flexible implementation of modular packet forwarders in userspace. It can be used to interconnect devices and their IP stacks across a variety of conventional and unconventional links. `interstice` exposes a reprogrammable, dynamically-updatable packet-forwarding strategy, enabling forwarder nodes in principle to act as hubs, bridges, full routers, or implement firewalls or NAT, as application requirements and platform constraints permit.

This approach enables benefits for modular, networked systems of microcontrollers which need to talk to the outside world: using IP enables internal microcontrollers to communicate with external devices such as PCs and smartphones without the need for application gateways. Further, to the extent that such networks are runtime-reconfigurable, features of IP such as address assignment, dynamic routing, and link-agnosticity can be incredibly beneficial.

`interstice` is evaluated here primarily against networks of various types of serial links (UART, I^2C, CAN) speaking PPP, selected to demonstrate utility of the approach to connect embedded devices lacking dedicated networking peripherals, and further that link drivers can be specialized to take advantage of the specific characteristics of each link. The approach is showcased in application scenarios including a networked milling machine, and is analyzed for a number of performance metrics.

Thesis Supervisor Joseph A. Paradiso

Title Alexander W. Dreyfoos (1954) Professor in Media Arts and Sciences,
Program in Media Arts and Sciences

IP Networks Over Heterogeneous Embedded Serial Links

by

Nathan Perry

The following people served as readers for this thesis:

Thesis Reader _____

Neil Gershenfeld
Director
Center for Bits and Atoms, MIT

Thesis Reader _____

Prabal Dutta
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

Listing of Figures

1. Introduction

This thesis develops a point-to-point, multihop IP networking approach using PPP links over various physical connections, especially targeted at operation on embedded microcontrollers. Each node in the network runs a packet forwarder, `interstice`, implemented as a software library; these forwarders have the capability to modularly adopt various strategies for learning network routes and making forwarding decisions. I implement flooding and bridging forwarding approaches here and consider an adaptation of the AODV [3] routing protocol.

The system is benchmarked for throughput, latency, and maximum network size. Integrated approaches including a web-based control and a networked milling machine are showcased.

This section provides motivation for this work, initial design considerations, and intended contributions.

1.1. IP

One of the key benefits of the Internet Protocol (IP, both v4 and v6) as standardized and actually deployed in the world is that it is an effective, ubiquitous "narrow waist"; a low-leakage abstraction that cleanly separates lower-layer implementation details of communication channels from the higher-layer semantics of networked applications.

This abstraction is powerful because it enables network technology to evolve separately from the applications that use it — existing technologies can be made faster (e.g. WiFi, Ethernet), new technologies can be introduced, and experimental approaches can be evaluated, all without needing to alter web servers, databases, browsers, and the vast abundance of other networked applications and services that exist. Hardware, firmware, and drivers can be freely upgraded and exchanged, as long as they interface cleanly to the platform's IP network stack.

On the other hand, from the application perspective, IP is a powerful platform abstraction. The networked part of an application is essentially portable to any platform with an IP stack, and the driving hardware is fully abstracted. Writing a web server that only runs on specific network interface hardware is of limited utility compared to one that works on any hardware configuration (presuming appropriate driver support).

This approach to building networking functionality hardware has been commodity for at least 25 years. IP had definitively won out over alternate networking approaches by the year 2000, and so desktop operating systems have built-in IP stacks, driver frameworks, and operating system ABI to support the approach.

1.2. Motivation

This thesis is motivated by my practical experience working on embedded systems. Many systems I encounter in reality are networks-in-disguise, i.e. they deal with many of the characteristic problems of computer networks: addressing, traffic multiplexing, transitive connectivity, framing, routing, naming services, etc., but face practical challenges at design time that obstruct the adoption of standardized networking techniques, instead opting to reinvent a subset of these functionalities in application-specific ways.

Unfortunately, reinventing this wheel typically sacrifices a number of benefits as compared to adopting a ready-made network stack. Most obviously, the "narrow-waist" benefits discussed above disappear entirely: applications are no longer portable, and networking hardware becomes tightly-coupled to application concerns. Baseline network functionality that would have come for free becomes dependent on application scope — true transitive connectivity is rare in these solutions, and applications instead implement ad-hoc, application-specific bridges on an as-needed basis.

Anecdotally, key problems impeding the adoption of networking approaches in these cases include technical unfamiliarity, integration burden, and lack of availability of dedicated networking hardware: adopting e.g. IP is simply seen as having too high of an activation energy to be worthwhile.

This thesis is an attempt to improve this state of affairs by providing a networking approach backed by a software toolkit that facilitates the construction of IP networks out of the (often unplanned) assortment of communication channels available in the embedded systems space. Networking research has for decades been focused on devices with high-speed purpose-built hardware, but has neglected those lacking it, which would nonetheless benefit from even minimal, low-speed network service.

The example systems in Figure 1 showcase a number of applications that could benefit from this kind of approach.

1.3. Programmer's model

The usage model I aim for is along the lines of Listing 1: the intent is to produce an embedded-compatible software library that supports this networking functionality at relatively low overhead for the programmer: a variety of I/O devices can be straightforwardly attached to the network by some mechanism, and most of the networking functionality is abstracted away by the software library, providing a plug-and-play networking experience across a variety of underlying links.

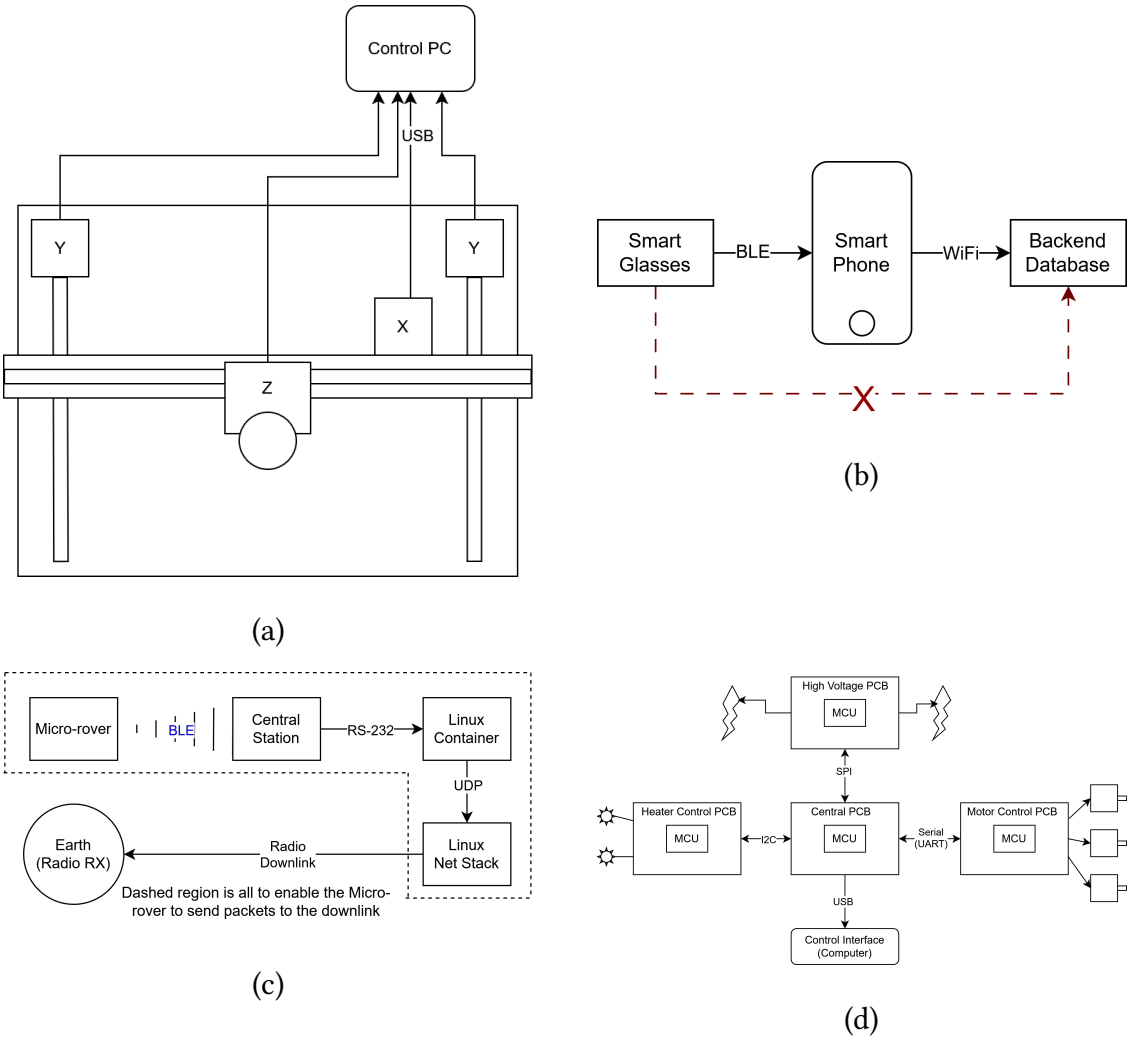

Figure 1: Examples of "implicitly-networked" applications I have worked on that might benefit from an available explicit *networking approach. a) represents the* `scriptorium` *distributed PCB milling machine, which has a number of motors controlled by a host computer over USB. b) captures AirSpec [2] smart glasses, which use a smartphone BLE connection as an intermediary hop to reach backend servers. c) shows the AstroAnt lunar micro-rover, which communicates payload data across a wide range of heterogeneous media in both the mission environment. d) captures a number of features common in hardware products I have previously worked on in industry: it is a PCB-based hardware assembly comprising several subsystems that communicate with and through each other.*

Obviously, this leaves all questions on the table as to how to design a network and software library that looks like this: these are discussed in the subsequent sections.

```
main() {
    net_init()

    net_attach(uart0)
    net_attach(spi0)
    net_attach(wifi)

    sock = net_udp_bind(1234)

    loop {
        sensor_data = read_sensor()
        net_udp_sendto(1.2.3.4:5678, sock, sensor_data)
        delay()
    }
}
```

Listing 1: Pseudocode sketching desired programmer experience.

1.4. Conceptual approach

To outline the problem space, let's consider some properties we'd want a network like this to have. In the interest of flexibility for the user, interfaces would be hotpluggable and the network layer self-bootstrapping, i.e. nodes running the software could be added and removed from the network freely, and would automatically acquire an addresses without user input. The approach would support arbitrary connection topologies, i.e. users should not need to intentionally avoid creating loops — the network's packet forwarding strategy should internalize a mechanism to avoid broadcast storm effects. It would efficiently make use of available links, automatically finding optimal routes between hosts. It would be compatible with existing networking technology outside the system — it should be possible to integrate this networking approach with a user's home or office IP network. Lastly, it should be generalizable and portable — we want assumptions about the link layer to be minimal, and the system to run on a wide variety of embedded systems.

To shape my solution, consider this set of practical observations: embedded IP stacks are commodity; Point-to-Point Protocol (PPP) implementations are plentiful; PPP carries IP traffic and can run over almost any bidirectional, byte-serial link, making it suitable for adaptation to almost any communication interface; and yet despite these tools, there seems to be no popularly-adopted or explored methodology for providing *transitive* connectivity[1] over networks of PPP links, and no library of PPP adaptations for common embedded serial links (outside of UARTs and RS-232).

[1] So emphasized because singleton/leaf PPP links do pop up in practical use in embedded systems occasionally.

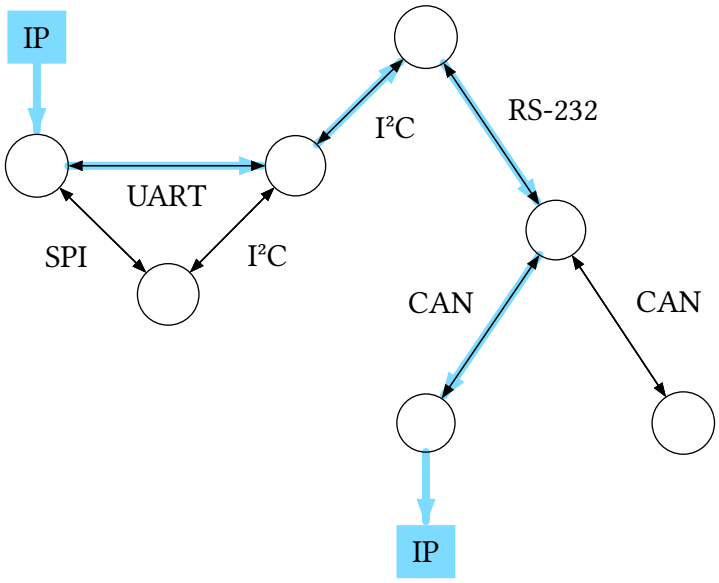

Figure 2: Notional point-to-point networking approach.

The approach I develop here is therefore an IP network of point-to-point links (visualized by example in Figure 2), typically (though not necessarily) communicating over PPP. The primary technical work required to enable the approach is the development of a packet forwarder that can run at each node in order to interconnect the point-to-point links (this is provided as a software library), and a collection of PPP adaptations for various lower-layers. The thesis discusses the design and evaluation of this software.

1.5. Notable alternatives

Here, I address a few alternative approaches to this network of point-to-point links, both to anticipate potential criticism and illustrate the goals of the approach by contrast.

1.5.1. Application gateway architecture

Why build an IP network at all, rather than adopt an application gateway architecture? This is a conventional way to provide IP services to networks of embedded microcontrollers (typically for wireless sensor nets), and has advantages in that it centralizes the implementation costs of the IP stack to the gateway device, enabling remote devices to run lighter-weight communication protocols, which can be specialized according to application requirements.

For example, a system of wireless thermometers as sketched in Figure 3 doesn't require each node to have a full IP stack — we know *a priori* that they just send temperature readings, so it seems like overkill to provide the complete flexibility (and pay the full cost) of an IP implementation on each node. Since they will only ever send one type of message,

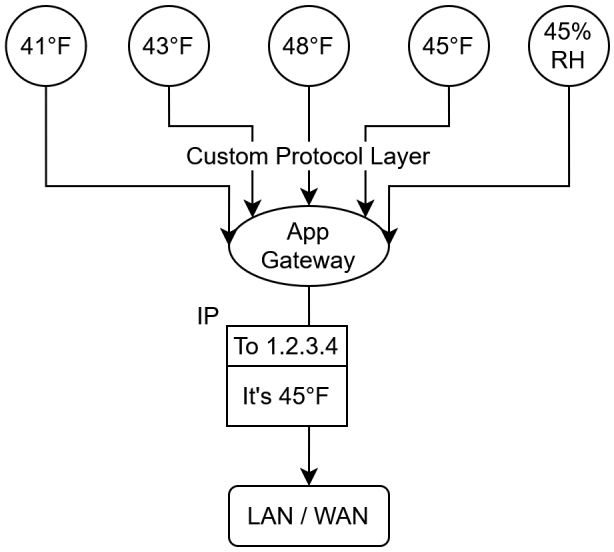

Figure 3: Application gateway-oriented networking architecture.

we can keep them simple and use a more capable application gateway device that listens for the incoming readings and forwards them onto the network layer. The sensors never see the network, but if it's useful for downstream concerns, the gateway could always emulate a separate IP for each device.

While the motivations behind this argument are understandable, and I don't aim to argue that a gateway architecture is a bad idea, the factors making the perspective compelling are that 1) IP stacks are heavy relative to the capabilities of embedded microcontrollers, and 2) implicitly, adopting IP for this kind of system is assumed to be more complex or risky than implementing a custom protocol. But (1) has become less true in recent years, as low-cost, high-capability microcontrollers have become widely available from vendors such as Espressif, Nordic, and STMicroelectronics. And (2) represents a chicken-and-egg problem, as it seems likely that IP appears more risky precisely because it is less popular for this kind of application, meaning that firmware developers are less familiar with it and solutions are less robust and battle-tested. If IP were seen as the standard approach for this kind of problem and it was assumed to "just work" at low integration cost, I think (2) would be a non-issue.

I also want to point out that the solution where each of the thermometers natively speaks IP has meaningful advantages: the gateway is a single point of failure, so if it goes down, the whole network goes down, where in principle, with a distributed point-to-point mesh, the network is naturally fault tolerant for sufficiently-connected topologies. Further, we benefit from the ubiquity of IP: any client (e.g. a smartphone, a laptop) speaking the protocol can connect to the devices directly and address each one individually.

```c
#if LINUX
#include <sys/types.h>
#include <sys/socket.h>
#else
#include "myplatform/socket.h"
#endif

int main() {
    struct sockaddr_in addr;
    int sock = socket(AF_INET, SOCK_DGRAM, 0);

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(1234);

    bind(sock, (struct sockaddr*)&addr, sizeof(addr));

    while (1) {
        addr.sin_addr.s_addr = INADDR_BROADCAST;
        addr.sin_port = htons(5678);

        sendto(sock, "data", strlen("data"), 0, &addr, sizeof(addr));
        sleep(100);
    }
}
```

Listing 2: The C-language Berkeley sockets interface is well-standardized and could be emulated by a custom gateway layer in a source-compatible way, enabling software portability.

1.5.2. Sockets-layer emulation

A possible response to the arguments in the previous section is take a middle ground and say: yes, for some applications, generic network functionality makes sense due to abstraction benefits and portability. However, we might benefit from a subset of these narrow-waist benefits of IP by emulating the *sockets* API specifically, as this is the part of the IP stack that applications will typically interact with directly.

This kind of approach would enable source compatibility with the Berkeley sockets API (see Listing 2), enabling direct portability of networked applications across e.g. desktop and embedded environments, while on the backend the socket API calls would communicate to the gateway device over a slimmed-down link layer, which would be the only member of the network providing a complete IP stack. This kind of approach would presumably provide savings in terms of system resources by not shipping the IP stack everywhere, while also capturing portability benefits.

The question this approach begs is how much is actually being saved by not just using an IP stack, at the point that the remote devices must be IP-aware (in order to address the transport-layer messages), emulate most the semantics of the used transport layers, and support an entirely new, custom link protocol. It seems to me that the relative benefits of using a gateway architecture fall drastically as the approach becomes closer to full IP. This architecture also still retains the single-point-of-failure in the central gateway, which a distributed approach does not have.

1.5.3. Existing embedded wireless networking approaches

I will discuss these in further detail in Section 2, but it bears mentioning that low-power wireless networking for embedded devices has been a major focus of research in the last quarter-century, and there exist a wide range of approaches available on commodity microcontrollers that can support our desired networking semantics. These approaches are readily mobile, relatively robust, and available from a number of vendors.

So why not just use a wireless approach? For one, because the ecosystem is fragmented, and it seems unrealistic presently to expect a convergence onto a singular technology. But secondly, and more to my central point: because not every device has, wants, or will choose to use a wireless radio or indeed any kind of specialized networking peripheral. Brownfield projects are typically stuck with their hardware choices, and meaningful tradeoffs exist in new development that may motivate against the selection of certain integrated wireless functionalities — but despite these considerations, devices may still want or benefit from network connectivity, and as I show through this thesis, it is possible to deliver that functionality over the much more ubiquitous embedded serial links that these devices possess anyway.

1.6. Applications

To ground the concept, I typify below a few applications that I expect would enjoy concrete benefits from the networking approach I describe.

1.6.1. Prototyping networked embedded systems

The examples listed in Section 1.2 are implicit networks of heterogeneous microcontrollers which are either real-world systems I have worked on or models that capture essential features. These systems were developed and deployed relatively quickly for research or development purposes. Due to application constraints, heterogeneity in device platforms and communication approach was at least expedient in all cases (if not completely necessary), and the adoption of a homogeneous link technology was never a design possibility.

I want to underline that these situations all independently developed network-like features, but did not adopt a standard networking approach such as IP. I think (circumstantially) that this is because no one argued for it and it would have been seen as simply too much overhead to integrate all of the functionality from scratch. But I suspect that the situation would have been different if there were an off-the-shelf solution that solved these problems — while my approach may not be energy-optimal or particularly fast, it represents a potentially-powerful alternative for projects such as these with latent networking needs which simply cannot make use of presently-available alternatives.

1.6.2. Transitive link sharing

Consider for instance a smart textile system comprising a number of microcontrollers interconnected in a lattice structure, which perform health monitoring tasks as a distributed network over a user's body. Such a concept ("NETS") is discussed and evaluated for instance by Wicaksono in [4] — he identifies bandwidth constraints and robustness concerns due to a shared link architecture as key concerns for developing such systems.

For one, the network connecting the textile could benefit from an IP-based approach, but more specifically, I observe that it is likely that this network will as a whole use a wireless link for data exfiltration (NETS uses Bluetooth). Presumably, such a system would prefer to power only a single radio in the network at a time for the sake of energy conservation and coordinate some mechanism to send all the sensor data through that one link. While it's possible to implement this functionality with an application-specific gateway approach, transitive connectivity like this is core functionality for a *network* gateway — it seems obvious that adopting networking proper is beneficial to solve this problem.

I point at a smart textile system with a wireless link here as a prototypical example (illustrated in Figure 4); the generalized extension of the concept is some locally-connected set of nodes that all want to communicate to the outside world over a shared link. We would ideally use a general-purpose networking approach in these kinds of cases rather than an application-specific one.

1.6.3. Machine- and board-area-networks

Oftentimes, electronic devices integrate multiple microcontrollers to control discrete subsystems. These microcontrollers are typically connected to one another via a serial-like channel or bus. Several core networking concerns are commonly re-solved in systems like this, as the system must decide how it is going to do framing, addressing, and provide gateway or bridge functionality if data must traverse multiple hops. I'm sure you see the pattern: this is a networking issue, and I suggest that in some cases, it may be useful to

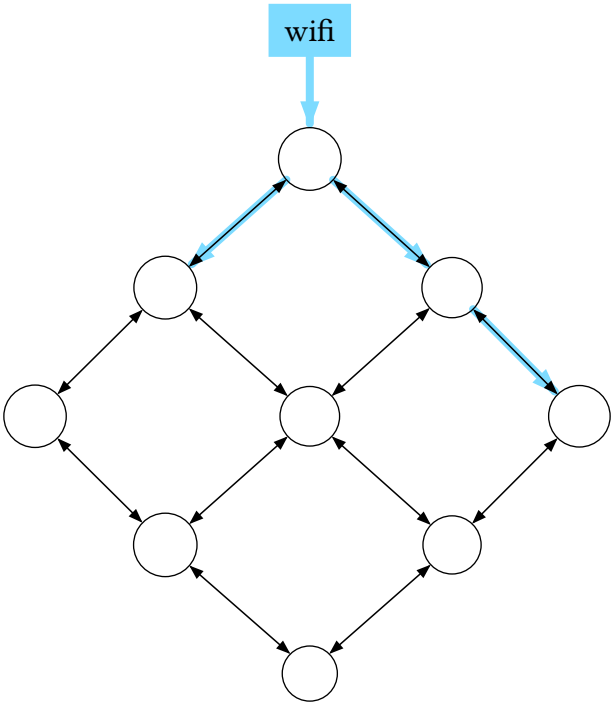

Figure 4: WiFi connection shared transitively through a lattice network.

actually model communicating microcontrollers on PCBs as "board-area" IP networks as represented in Figure 5, or larger assemblies of such PCBs as "machine-area" IP networks.

One feature of this idea that I hope to capture is that there may be classes of solutions that aren't considered simply because the tools that would make them appear practical don't yet exist.

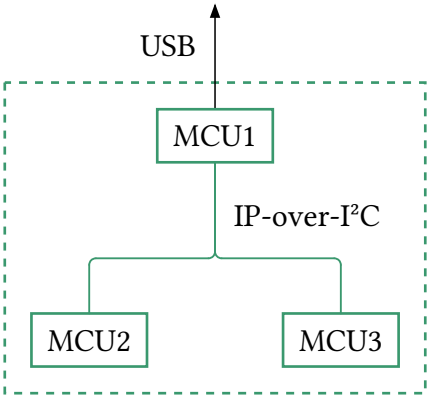

Figure 5: Notional PCB-area IP network.

1.7. Contributions

This thesis aims to make networking functionality easier to adopt and integrate in microcontroller-based embedded systems.

The contributions of the thesis are:

- An open-source software library that generically implements and supports described networking functionality.
- Demonstration systems making use of the library, showcasing key functionalities by example.
- A set of benchmarks that validate system performance.

1.7.1. System overview

The system I will develop here is a network of point-to-point embedded links speaking PPP. In the following sections, I will discuss the design of a system of such links interconnected by packet forwarders implemented in software. These forwarders accept modular forwarding strategies; presently, they support flooding and bridging behavior for IP packets. Adaptation of a distance-vector IP routing protocol is discussed, based on AODV [3].

The system supports I^2C, UART, and CAN bus links, as well as USB CDC-ACM (virtual serial port).

In this work, I use statically-configured node addresses, and the network is treated as an autonomously managed subnet with edge routers. Edge forwarding behaviors (NAT, routes to/from other subnets) are manually configured. A simple address compression scheme is discussed and implemented, targeted at bus links. Name services for the system are optionally provided by mDNS, which is modified to flood requests through the network.

1.7.2. Sections

After this introduction, I provide background on relevant prior art (Section 2). I approach system design by developing a networking approach and software package to support it (Section 3), then evaluate against a series of benchmarks and provide example demonstration systems (Section 4). I provide discussion on the development and implementation process, and remarks about potential future work (Section 5). The thesis is finally summarized and conclusions are provided (Section 6).

2. Background

There are many existing projects in networking generally and embedded networking specifically that merit discussion here.

2.1. Point-to-point networks, switch-based LANs

The networking approach discussed here can be characterized as an arbitrary-topology LAN with a switch[2] at each node. Substantial prior art on this concept exists, most of it from many years ago as networking approaches were less settled than they are today — nowadays LANs are typically arranged around the assumption or emulation of a shared broadcast medium (e.g. Ethernet). In Baran's seminal 1964 article "On Distributed Communication Networks" [5], he performs one of the first characterizations of the benefits of distributed networks of redundant, point-to-point links, a benefit I aim to capture in this work. As Yoo points out in [6], Baran's design relies on each node being a "low-cost, unmanned computer" — while the computers at the time of Baran's conceptualization were certainly different than the microcontrollers I target, the idea remains the same.

Separately, METANET [7] discusses the development of a high-speed packet-switched network that uses the abstraction of a virtual ring to linearize the network. However, as [8] indicates and attempts to correct, this design leaves some links unutilized, as they are not part of the ring. This pattern of link underutilization is a feature I aim to avoid in this implementation, and is discussed later in Section 3.2.

2.2. Microcontroller networking, WSNs

A wide variety of high-quality microcontroller- and embedded-oriented network stacks exist under open-source licenses, from standalone implementations such as lwIP [9] and uIP [10], to more integrated approaches built into Real-Time Operating Systems (RTOSes) such as Zephyr [11] and RIOT [12]. At the far end of this spectrum are WSN- and IoT-specialized solutions such as Contiki [13], which is particularly aimed at adoption of the LR-WPAN stack: IPv6, 802.15.4 radios, RPL-based routing [14], and so on.

All of the listed approaches integrate fully-functional IP stacks, and with the possible exception of uIP have support for configurable packet-forwarding rules if not full MANET routing approaches, enabling them to be used in the ways I describe here: for constructing networks of embedded nodes, potentially across various links and in mobile or usntable topologies. This may beg the question of how this thesis achieves novelty: it is simply that (to my knowledge) there is not substantial published work applying these techniques specifically to wired serial links with a default assumption of point-to-point connections.

[2] Generically: packet forwarder (see my discussion in Section 3.3).

The problems for wireless networks are characteristically opposite: almost all links have limited broadcast semantics, implying a wholly different set of concerns about node mobility, EMI, and the meaning of link scope and subnet extent.

2.3. IoT fragmentation

The communication landscape in the IoT is substantially fragmented, ironically by exactly the techniques designed to connect it. There are many ways that IoT networking has been approached: Thread, WirelessHART, ZigBee, ZWave, 6TiSCH, BLE, and LoRa, to name just some of the more popular ones up through the network layer. Some of these approaches do speak IP (typically IPv6), some of them *can* speak IP in certain situations, and some of them simply cannot. Some of them are based on 802.15.4 standards, some of them are not.

Aly et al. recognize this fragmentary trend as a problem, arguing that:

> It becomes apparent that most of the challenges brought by IoT are that ones related to interoperability concerning multiple layers of the end-to-end protocol stack. [15]

This domain is potentially interesting to our approach as it may be of interest to connect these disparate protocols; it is worth considering approaches undertaken previously to make these different links compatible. Several implementations have been undertaken; Web of Things [16], for instance, adopts a standard message format communicated over HTTP. Matter [17] follows this trend of a standardized message format and data model, but adopts IPv6 as the ubiquitous networking layer. However, neither of these approaches solves the fundamental problem we're aiming at: getting devices on the network when they do not have supported networking peripherals. To my knowledge, no such approach has been taken, so this path appears to be a non-starter.

2.4. Named data networking approaches at the edge

Named data networking [18] and its cousins (or aliases) in the content-centric networking sphere form a family of techniques organized not around networks of *hosts* (i.e. distinct computing machines and devices identified by addresses), but rather networks of *content* identified by *names*; this turns the network's role into one of finding routes for identified information to be delivered to requesters of that information.

The model of browsing the web by URL can be usefully compared: web pages are content identified by a structured name (the URL), and the browser's job is to figure out how to deliver you the data by the name. The difference is that under current networking paradigms this functionality is performed at the application layer; a named data approach would have URLs actually *identified* with the content being requested, rather than dispatched through several layers of indirection — a DNS lookup leading to a TLS connection which connects you to an HTTP server which is really acting as a database frontend. NDN would have it that requesting data by name causes the network to directly deliver you an end-to-end encrypted payload as its core functionality, without all the fuss in the middle.

I bring NDN into the conversation due to the prevalence of related approaches at the edge. The simplest connection to make is to the message broker: a service that provides publish/subscribe semantics by structured topic: this looks a lot like named data, even if it's done at the application layer. Subscribing to a topic is essentially a request for some (future) data by its name.

This brings me to Zenoh [19], a distributed, transtively-connectable pub/sub protocol, which supports multiple modes of operation, including as a message broker, dumb client, and router (forwarder for other brokers). It's interesting to consider here because it doesn't actually specify a particular lower-layer: it runs on anything that can serve as a link (providing byte-serial communication), just as we depend upon PPP to do, and it provides transitive connecitivity. Zenoh can run over serial links, TCP, TLS, UDP: you name it, and is available on both desktop-class platforms and microcontrollers (through zenoh-pico°). The key thing it doesn't do, which I'm concerned with in this work, is provide IP services and compatible API to support application portability: Zenoh applications are built with Zenoh as the target platform.

3. System Design

To briefly re-motivate the design concept I will flesh out below: I suggested in Section 1 that it would be beneficial to be able to build networks of embedded devices over point-to-point links. Networks of this type in general need to support a mechanism whereby traffic can be forwarded by intermediate nodes to reach a destination; this section develops a generic software library package that can abstract over different forwarding strategies and is suitable to run on an embedded microcontroller.

In order to motivate the design of this software, I specify motivating goals for the system and discuss considerations for organizing the networks that will ultimately be created. I then discuss the software design itself, and provide notes on a few extensions to functionality that can further enhance the system.

3.1. Goals and anticipated usage

The networking system I lay out here is designed for operation on small devices (prototypically microcontrollers), whose application needs depend upon or benefit from network communication. I assume that these nodes have *some* mechanism for bidirectional communication, but nothing further than that — no specialized network hardware is required.

The kinds of network activity I aim to support are general-purpose, i.e. not dominated by requirements for high throughput, deterministic latency, or extreme energy conservation. Data payloads I anticipate include sensor readings, asynchronous control plane updates, and user interaction traffic (including loading web pages over HTTP and simple web API interaction), at an overall rate of perhaps several hundred packets per second across any particular link (dependent on link speed), with graceful degradation via dropped packets as links become saturated.

As a usage paradigm, I think of IoT networks deployed in homes and offices, informational industrial monitoring devices such as low-rate cameras and sensors, and other moderate-data-rate embedded sensor systems. The approach should scale to systems of hundreds if not thousands of nodes as system memory and forwarding tables allow.

I aim for the approach to make use of existing standards and approaches where possible: one of the central ideas behind this project is that nearly the entire thing can be built off-the-shelf in order to enjoy broad compatibility with existing network services and applications.

Non-goals include design sacrifices made in the name of optimizing for specialized network workloads, such as high throughput for interactive video playback, hard-realtime

latency bounds for control systems, or extreme energy optimization for long-deployed battery-powered systems. To the extent that these use-cases function naturally, all the better — but the most essential function of the system is to get general networking functionality onto devices that can't access it at all; this trumps optimization concerns for any particular kind of traffic.

That being said, I think it is useful to specify network performance targets in order to provide some anchor to guide design, but they are difficult to specify generically in a system like this, as throughput metrics are necessarily relative to link capabilities: note that a slow 9600 baud UART can transport at most 60 minimal IPv4 headers or 30 minimal IPv6 headers in a second, where by contrast a high-speed 40MHz SPI link could in theory deliver up to 144k 256-byte IPv4 packets per second. All the same, it is useful to provide a goal as a meter for success: I aim for a mean of 80% of nominal physical line rate in full network tests when links are saturated.

3.2. Network organization

For broad compatibility with the world's networking systems, our network is an IP network. Each node in the network has an IP address, and the network-layer traffic is comprised of IP packets. Nodes support both IPv4 (for backwards compatibility and smaller addresses and headers) and IPv6 (for better features and future-proofing): I use "IP" to refer to both protocols simultaneously except where distinctions are relevant.

Because our approach aims to support devices with minimal communication equipment such as UART links, we choose a point-to-point, multihop network representation so that we can treat links uniformly. In order to move data through the network, each node must therefore be a packet forwarder — it must be capable of making decisions about to which point-to-point link(s) it should next forward a packet not bound for its own address. Supported packet forwarding approaches are discussed below in Section 3.3.

Note that this network representation is amenable to multipoint links under the abstraction of a collection of pairwise point-to-point links, with broadcasts filtered by address at each receiver. This concept is illustrated in Figure 6.

The point-to-point link implementation is flexible in principle, as long as it can carry IP. As far as the network is concerned, the link layer can remain unspecified. However, for the benefits discussed in Section 3.2.2, most links in the system use PPP in reality.

3.2.1. Addressing and subnetting

Our network must make some choices about both addressing and determining a subnet border. That is, in general it is assumed that our network may communicate with network

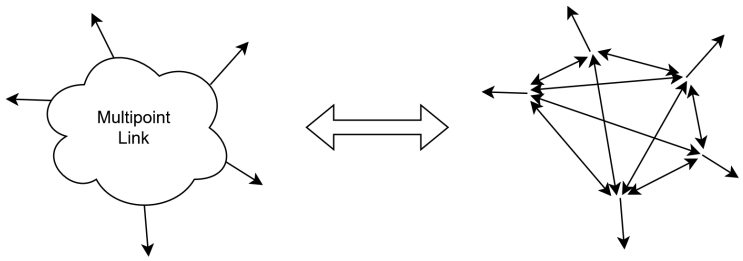

Figure 6: Multipoint link treated as equivalent to a collection of point-to-point links.

nodes that are "off-subnet" — not using our same point-to-point networking approach; we must therefore decide what happens to traffic at the border between our system and foreign ones, and how we should address our own nodes to maintain compatibility.

A situation we might encounter is a device that has both an in-network serial port and an Ethernet port, as in Figure 7. The serial port communicates with our system, but the Ethernet port is connected to a preexisting and separately-managed home LAN with all the usual trappings: we'll treat it as IPv4 for now, so let's say it's running DHCP, has a default gateway router connecting it to the pulic internet, it has its own own subnet prefix, and there are a number of active peer hosts in the subnet.

Ideally, our system would bidirectionally interoperate with this existing LAN, but the question that faces us is how to achieve that; what do we do on this border node in terms of addressing and packet forwarding?

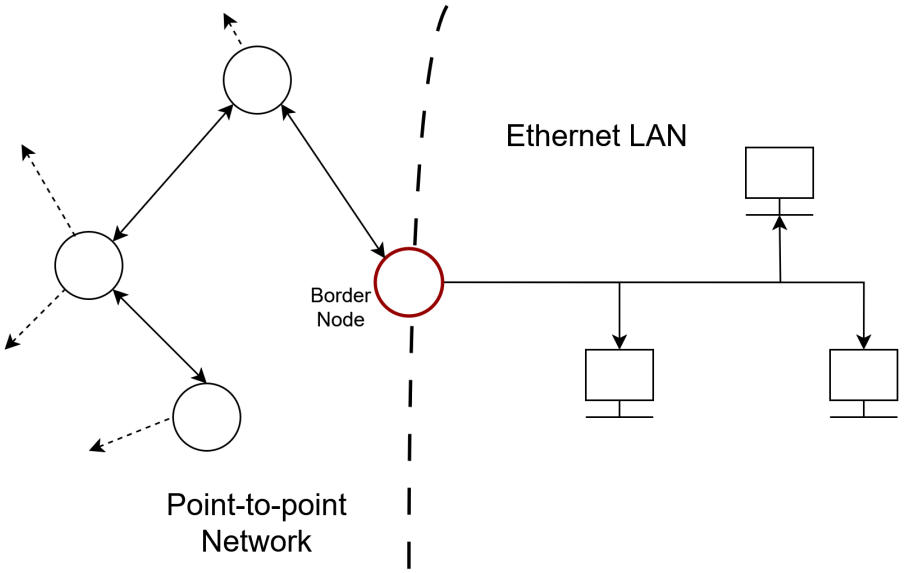

Figure 7: Device on the border between point-to-point network and foreign Ethernet LAN.

3.2.1.1. Challenges for a transparent border

One (strawman) approach might be to proactively detect that we border the foreign LAN and adopt its network configuration — we could have our border node forward DHCP requests into the Ethernet LAN for all nodes in our system, and run proxy ARP (ND) or synthesize MAC addresses for them so they appeared for all intents and purposes to be part of the Ethernet LAN. We would adopt its network prefix and "pretend" to be a normal, well-behaved part of it.

This approach has several downsides: it doesn't deal well with multiple simultaneous borders to multiple (possibly-but-not-necessarily-distinct) foreign LANs, it specializes a significant amount of behavior to Ethernet in particular (implying a requirement for repetitive and ungeneralizable similar implementations for other Layer 2s), it relies on there being enough addresses available in the Ethernet LAN for us (which in general we don't know), and it makes bringup of our network dependent on detection and propagation of this external state — what happens in the case of a network partition (or reconnection)? Presumably, some part of our network comes up assuming that there are no network borders, and so self-configures? Do the self-configured addresses drop once a connection reestablishes? The lack of clear answers to these questions for this "transparent border" approach, predicated on learning external network configurations, hence seems like a total non-starter, with the notable exception of a single unique benefit: it makes us look like part of the same subnet to devices on the Ethernet LAN, meaning that they will have no problem initiating communication into our system.

3.2.1.2. Subnet with border router

Another (mostly preferable) option is to treat our own network as a coherent, self-managed subnet and have devices on the border act as gateway routers between us and foreign LANs. However, this approach runs up against the singular benefit of the last: how do the foreign LAN's hosts actually send traffic into our network? We could install routes in each of its hosts, but this doesn't scale well; conventionally, this would be done by instead running a routing protocol to instead automatically install our routes on the foreign LAN's *router*, which could then handle forwarding traffic to our border nodes. However, support for internal routing protocols varies and authorization may be required in order to update network configuration — in practice, this approach will be hard to operationalize in a manner compatible with automatic bootstrap.

As an aside, it seems that it could solve this problem if IPv6 ND [20] separated the concerns of network bootstrap (which requires determining a *default* router, configuring the network prefix, providing an address-assignment mechanism), and the advertisement of a node as *a* router which can provide routes for a particular private subnet (but is

not a default gateway and does not support bootstrap functionality). Interior routing protocols are the common mechanism for propagating this information, but it might provide a drastically more flexible model for internet connectivity if these semantics were an essential part of the IP model.

We can always fall back to SNAT on the borders, but this creates problems: ingress from foreign subnets becomes impossible in the general case. Specific DNAT approaches (e.g. port forwards and DMZ hosts) can of course be configured, but these require manual intervention, and we're hoping to capture the full benefits of IP, which suggest that the foreign LAN should be able to have full visibility and bidirectional communication with all hosts into our subnet.

For the purposes of this thesis, I leave this problem unresolved, as my evaluation of the point-to-point networking approach does not depend upon having a mechanism like this fully working; I instead statically configure the forwarding rules at the edges. These problems are not specific to my approach, but rather a core challenge of working with IP networks. The fact that it specifically makes sense for me to hang a separate subnet off of an existing LAN simply brings this issue to the forefront.

3.2.2. PPP

PPP, specified in RFC 1661 [21], is a link-layer protocol capable of of carrying network traffic across a variety of underlying physical media, such as serial ports connecting two devices. The primary benefits of PPP as I see them are:

- Traffic multiplexing on the link. A PPP link concurrently carries link control traffic, network layer control traffic, and network data traffic by discriminating on a per-frame 16-bit protocol type field. This enables a single PPP link to carry many different traffic types simultaneously over the same physical channel (including e.g. IPv4, IPv6).
- A generic mechanism for negotiating per-protocol, content-agnostic configuration parameters, conventionally assuming a minimal default parameter set that will produce a functioning link, and allowing peers to opt-in to upgrades that they support.

The PPP RFC [21] does not specify a framing, checksumming or error-correction approach, leaving these concerns open to specialization for the lower layer in use on a given link. Several approaches have been separately specified for different lower layers, including PPP in HDLC-like framing [22], PPP over Ethernet [23] (PPPoE), and PPP over Asynchronous Transfer Mode [24] (PPPoA). I make use of the HDLC-like framing approach for byte-serial connections such as UARTs and TCP overlays, and adapt new

framing variants for lower layers such as I^2C, which already have a framing mechanism at the physical layer.

I pursue PPP as the link layer of choice[3] in this system specifically because its configuration negotiation mechanism provides a useful way to enable opt-in support for experimental features explored in this thesis such as link-specific header compression schemes and selection of forwarding approach, while still enabling interoperation with existing PPP implementations such as `pppd` [27], which would not have support for the experimental options (causing them to automatically be rejected as unknown, turning them off).

This feature of PPP also seemed useful for automatic bootstrapping and implicit subnet boundary detection, whereby network nodes could use the experimental configuration options as discriminants, for instance to enable certain nodes to act as border routers and enable NAT for PPP peers detected to be "out of subnet". This idea is illustrated in Figure 8. These features were not ultimately implemented as part of the thesis, but remain interesting possibilities for future work.

3.3. Forwarding approaches

I consider packet forwarding to refer generically to the process of a device ingesting a packet not bound for its own address and making a decision about what to do with it. This definition is somewhat overloaded for want of a better catch-all: by "packet" I mean

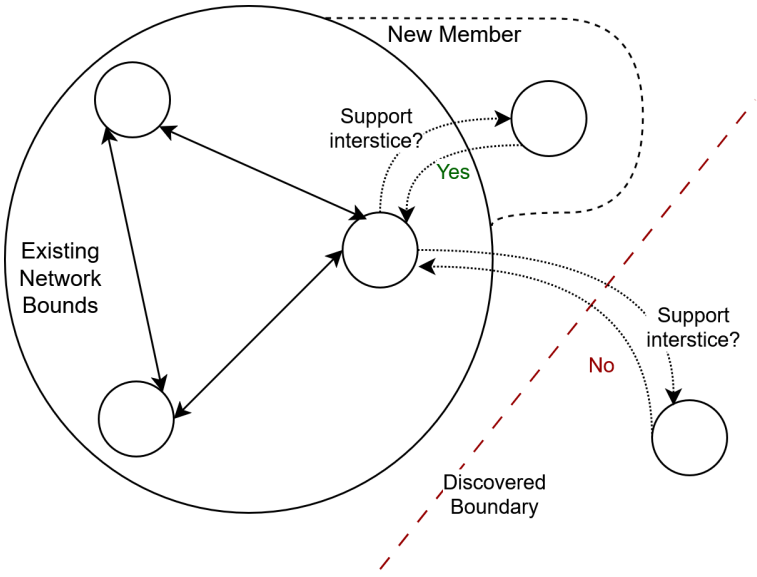

Figure 8: Imagined mechanism for automatic subnet discrimination.

[3] As opposed to e.g. SLIP [25] or COBS [26], both of which are simpler to implement.

interchangeably a network packet or a link layer frame, and by "address" I mean the relevant layer's address, not just an IP address. This is unconventional; these approaches are typically referred to as "switching" at the link layer (because packets are "switched" between links, historically in a very literal electrical sense — think telephones) and generically "packet forwarding" at the network layer, but I intentionally unify them here because they perform the same essential function: a generic forwarder takes in some packet on an *interface* (here typically a PPP link, but generally also including Ethernet, WiFi, etc.), compares header metadata (primarily network address) against some internal state to determine an appropriate action, then sends the packet back out on some *new* set of interfaces in order to move it closer to its destination.

I make this unification because point-to-point links have no meaningful notion of address: there is only the local side and the remote side. All forwarding in our system therefore happens at the network layer and in terms of network-layer addresses. That we know *a priori* that the network is constructed this way fundamentally changes the problem framing compared to typical LANs, which are constructed on the assumption of a shared broadcast domain, such as what Ethernet emulates. This difference in construction means that your home router and computer must perform ARP lookups to resolve IP addresses to link-layer addresses, but after doing so, can assume that Ethernet frames will be faithfully delivered to their destination: devices can pretend that they are sending link frames on a physically contiguous link that all devices in the LAN can see. This is not actually true anymore — modern Ethernet networks are point-to-point, but they provide the illusion of a shared broadcast domain through intelligent use of transparent bridging.

This is all to point out that because our network construction is explicitly one composed of point-to-point links, the problem of packet forwarding is one that we must internalize and solve in order for our network to function, in a sense that is atypical for the network layer in a LAN. As a result, many of the techniques typically relegated to the link layer, such as flooding and transparent bridging, are relevant candidates for our *network*-layer packet forwarder implementation, and are discussed below.

3.3.1. Flooding / Hubs

Perhaps the simplest approach to packet forwarding, flooding forwarders (hubs) broadcast every packet they receive to all other interfaces. This ensures that all nodes in a connected segment of a network graph see every packet (and can locally filter for their own address), but comes at the cost of maximal link utilization and congestion, as every packet entering the network traverses every link at least once.[4]

[4]With the exception of cases where the destination node is at an hourglass point in the network graph.

Special care must be taken in flooded networks not to connect devices in loops, or else the network will experience so-called "broadcast storm" effects, as packets will be forwarded around the loops forever, or until hop limits (a.k.a. TTL) are exhausted. Storms can be avoided by supplying a protocol-level mechanism to suppress packet retransmission in the forwarding implementation (e.g. by providing unique ids to each message, depending on small TTLs, or rejecting duplicates within a specified time window), or by simply passing this burden onto the network user to ensure that the network topology is tree-like / cycle-free.

`interstice` provides a flood forwarder as a dead-simple forwarding strategy implementation that can be used for debugging, in simple network deployments, or as an optimization for highly memory-constrained devices that do not have room to store forwarding tables.

3.3.2. Transparent bridging

Transparent bridging is a forwarding approach that can from one perspective be thought of as an augmented hub that remembers what interface ID last received a packet from a given address. When it needs to make a forwarding decision, it consults the cache[5] retaining this mapping to determine where the incoming packet should be sent. Unicast packets with a "remembered" associated interface ID are sent back on that interface, on the assumption that it is the best next-hop through the network *back* to the original source. When no interface ID is in the cache (or if the packet is a broadcast or multicast) the bridge falls back to flooding behavior. This process is illustrated in Figure 9.

This approach enables passive, on-demand learning of the topology of small-to-medium sized networks with low implementation complexity but good average-case performance in typical network conditions. In particular, for a cycle-free network, there is exactly one route between any two devices, which any bridges on the route will discover as long as there is bidirectional traffic between them; as this route is unique by the tree-like structure of the network, it is also necessarily optimal.

[5]These caches are conventionally referred to as *tables*, as a typical implementation would use a contiguous chunk of memory (or dedicated accelerated hardware storage such as CAM [28]) to store and query the address map. However, in a transparent bridge, the table entries must be considered opportunistic optimizations over packet broadcast, as, if traffic is unidirectional, a bridge may never actually learn the forwarding rule for a given address, and yet still must maintain correct network function. My transparent bridge implementation explicitly acknowledges this and intentionally selects an LRU cache as the data structure responsible for storing the forwarding table. This also provides the benefit of giving an explicit, predictable stance on memory exhaustion behavior: the cache simply drops the least-recently used entry, meaning that the forwarder will begin broadcasting to reach the dropped address (until another packet is received).

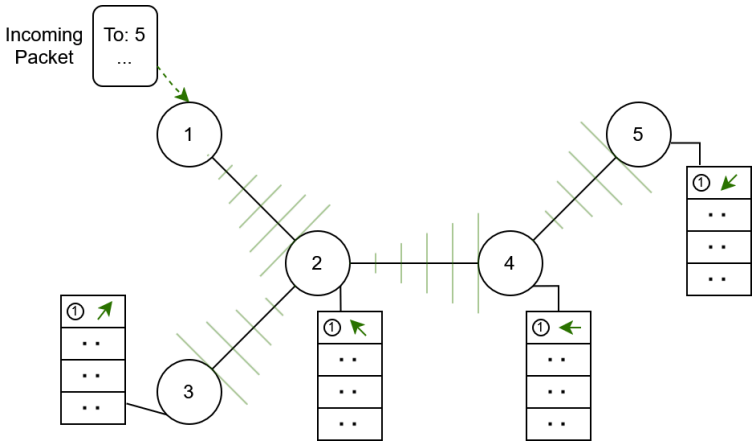

Figure 9: Transparent bridges passively learning forwarding rules.

These properties make transparent bridging an attractive option for this thesis, and indeed, it is the standard selected approach in my evaluations due to its relative implementation and debugging simplicity.

However, the requirement that the network be cycle-free collides with our goals stated in Section 3.1: ideally, we would support a highly-connected mesh topology, but this is impossible under transparent bridging, as cycles in the network graph cause broadcast storms, and the possibility of packets from a single source arriving on different interfaces could cause thrashing effects in the forwarding table. Conventionally, the solution to this problem is to proactively learn the network topology and selectively disable links that create loops (i.e. turn the mesh back into a tree), but doing this eliminates the robustness benefits of redundant network paths and tends to concentrate traffic in central nodes. A full solution to stated design goals cannot rely on transparent bridging.

3.3.3. Routing

Routing approaches actively learn information about network topology and link quality, which is propagated between participating nodes (termed routers). They use this information to determine valid (and ideally cost-optimal) paths through the network ("routes"), which are used to make forwarding decisions. Routing approaches are in general tolerant of arbitrary network topologies, unlike bridges or hubs, because the router actively selects a path using an approach that prefers to minimize cost, which naturally excludes cyclicity.[6]

A wide variety of routing protocols exist for different applications — common approaches used for Internet routing include OSPF [29], BGP [30], and RIP [31]. These approaches

[6]Transient cycles *are* in many cases temporarily possible in routing approaches (typically when nodes are out of sync with respect to the net topology), but they are typically excluded on network convergence.

are unfortunately relatively heavy in implementation and not terribly well-suited to our problem domain, as they optimize for providing routes between *subnets*, where our issue is primarily finding routes between *hosts*. However, development in Mobile Ad-hoc Networks (MANETs) has driven the standardization of routing protocols that consider ad-hoc networks of devices that share many of our constraints. Here we consider Ad-hoc On-Demand Distance Vector Routing (AODV) [3] as a specimen that may suit our needs.

AODV learns routing information both passively and actively: when a forwarding decision must be made, if there is no route for the destination address (i.e. the forwarder doesn't know what interface to forward the packet to), the router broadcasts a "Route Request" (RREQ) into the network containing its address and the destination address. The fact that the transmission is a broadcast means that if the destination host is in the network, it will hear it and subsequently respond (via unicast) with a "Route Reply" (RREP) message. Figure 10 illustrates this process. Unicasting is made possible because devices which forwarded the original RREQ cache the interface ID from which they heard it, so the chain can be traversed backwards. When the RREQ originator receives the RREP, it now knows the best next-hop to reach the destination (the one it received the RREP from), i.e. it has learned the route. Furthermore, all devices *on* the route also necessarily hear the RREP, and have therefore learned their own routes to both the destination and the originator (as they know the next-hop in both directions).

Observe that this process will tend to produce a "good" route because the network itself implicitly computes the path cost by propagating the RREQ — the path through the

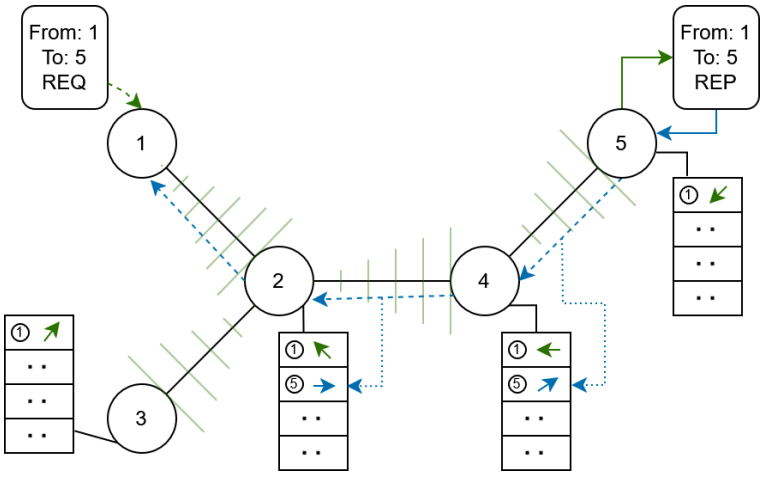

Figure 10: Basic AODV operation. Node 1 doesn't know the path to node 5, and so broadcasts a REQ message. The network propagates it until it reaches node 5, which responds with a unicast REP back along the same path. Intervening nodes learn forwarding rules in both directions.

network that delivers the message first is the one that the destination will use to respond, even if there are actually multiple possible paths, and the RREP will naturally follow the same path back to the RREQ originator. AODV has several other beneficial properties, including relative simplicity of implementation, convergence guarantees, and very low average-case network traffic — it enjoys substantial popularity in the MANET space, being the approach used to route ZigBee, for instance.

For these reasons, I am motivated to adopt it as a routing packet forwarder for `interstice`; however, there is presently no embedded-compatible AODV implementation in Rust (the programming language of choice for this thesis) that I am aware of. For expediency, I instead consider a substantially cut-down, customized design of AODV, which could be specialized to operate over PPP and concerns itself mostly with happy path operation. As transparent bridging is enough to demonstrate network function in most cases, the goal would be to demonstrate that a routed forwarding can be made to work, though a complete, RFC-accurate implementation of AODV in Rust would also be a great candidate for future work.

I did not implement this approach during the thesis, but to explore the concept, my notional, modified version of AODV (uncreatively titled `raodv`: "Rust AODV") models it as a natural extension of a transparent bridging approach. The insight of transparent bridging is essentially the same as that of AODV: broadcasts tell us information about the best paths through the network, information which we can retain as an optimization to avoid the need to broadcast in the future. AODV's enhancements are that it actively requests that the destination send a reply and includes mechanisms to suppress broadcast storms.

In my system, it seems that the most straightforward way to exchange `raodv` routing traffic would be to simply send it over dedicated PPP network data channels. However, we can do one better: having to wait for a routing handshake to occur somewhat complicates the packet forwarder implementation: instead, what if we just acted like a transparent bridge and broadcast our network packet as normal, but piggybacked metadata indicating that this packet is also an RREQ? This could cost a substantial amount of unnecessary link budget in the broadcast (duplication) of the network data, but we should expect that cost to be amortized acceptably over the lifetime of the route. This could create broadcast storms, so let's also attach a unique ID that devices will cache, enabling them to identify and drop duplicates.

And if we assume broadcasts will take a good path, why unicast backwards? Instead, always broadcast an RREP when you get an RREQ.

3.4. Software design

The software implementing the networking approach is implemented in several Rust libraries (an overview of the source code is available in Appendix A). The bulk of the functionality lies in a library called `interstice`, which provides a generic packet forwarding abstraction and the modular packet forwarder strategies discussed in Section 3.3.

This generic forwarder concept is based on the observation discussed in Section 3.3 that packet forwarding can be modeled (Figure 11) as unspecialized to any particular networking approach (or even layer): ingest a packet, compare its address with some internal state, then specify where it should be sent back out.

The `interstice` implementation acknowledges this and provides a general, flexible framework for specifying packet forwarding behavior in a way that is entirely decoupled from the network stack and allows reprogramming or updating during network operations.

The forwarding strategies specified above (hub, transparent bridge, and suggested `raodv` router) can therefore be treated as modules that can be 'plugged in' (Figure 12) to an `interstice` forwarder without regard for the attached links, buffering configuration, or various other parameters of the generic forwarder. This modularity greatly eased development and makes the approach relatively portable.

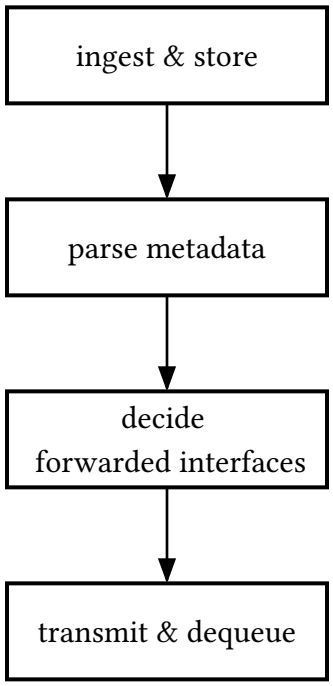

Figure 11: Interstice generic forwarder abstraction.

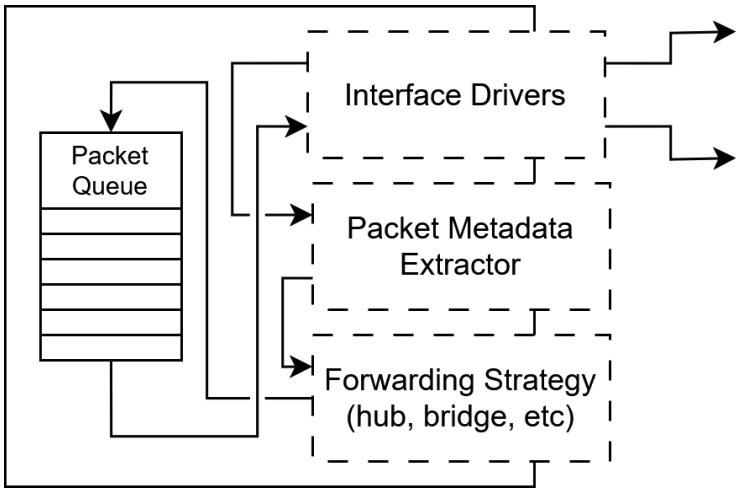

Figure 12: Interstice forwarder modularity.

3.4.1. Generic addressing

`interstice` uses a generic address abstraction based on Rust language interfaces (traits), which present a minimal set of requirements for types to function as addresses. Addresses must provide:

- An opaque byte-string representation.
- Functionality to check if the packet is unicast, broadcast, or multicast.

This minimal interface has proven sufficient to enable generic operation of `interstice` packet forwarders, which are parametric over this abstracted address type. The byte-string representation provides a mechanism for opaque comparison and storage in forwarding tables, while the unicast/broadcast/multicast semantics are required to resolve forwarding behavior in general.

Concretely, this generic address interface is implemented for IPv4, IPv6, and Ethernet addresses — in principle, the forwarder can function with addresses of any of these types. This set of supported address types can be extended by the user if desired.

3.4.2. Packet representation

Packets in the `interstice` system are also abstracted by a language trait. They must support extraction of source address, destination address, and a reference to payload data, as well as a mechanism to check and decrement a hop counter (if present). This functionality is implemented for IPv4, IPv6, and Ethernet frames.

This abstract packet representation has similarly proven sufficient for generic forwarding. Considering the IPv4 header, most metadata other than destination address and TTL are irrelevant to the forwarder; the network-layer semantics (e.g. fragmentation and

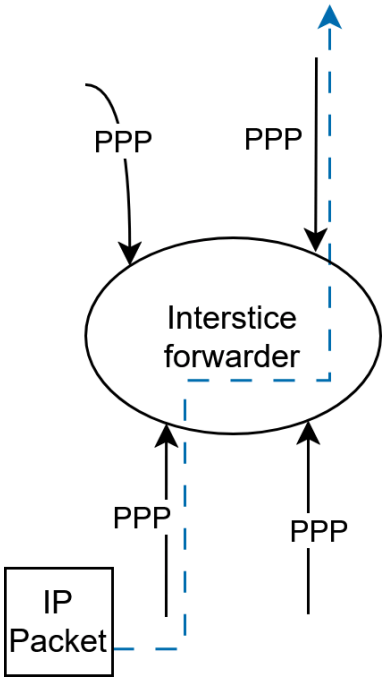

Figure 13: Generic network forwarder can perform packet forwarding without an attached network stack.

reassembly) are implemented by the endpoint IP stacks. Notable exceptions that we might choose to include in the future are QoS via DCSP (which is best-effort anyway, but could be useful to integrate as part of the generic packet representation) and source-routing options, which I simply don't support.

3.4.3. Network stack decoupling

Note that there is nothing in this system description that couples the `interstice` implementation to a network stack — the forwarder can run completely on its own (Figure 13).

This observation led to an organizational shift in connection design for `interstice`: IP stacks are treated as leaf connections (Figure 14) in the overall forwarding topology, and are assumed to have no intrinsic forwarding capabilities — the point-to-point nature of the networking approach is enabled by the forwarders only, which are the only nodes that can attach multiple interfaces. IP stacks will typically not see a PPP link directly — instead, they attach to the forwarders over an in-memory "null modem" link. This leads to the conclusion that packet forwarders are unaddressed entities: the *IP stacks themselves* are the only network members that actually have addresses.

This decoupled approach is beneficial because it disentangles the concerns of complete interpretation of IP packets (including fragmentation/reassembly, option parsing and

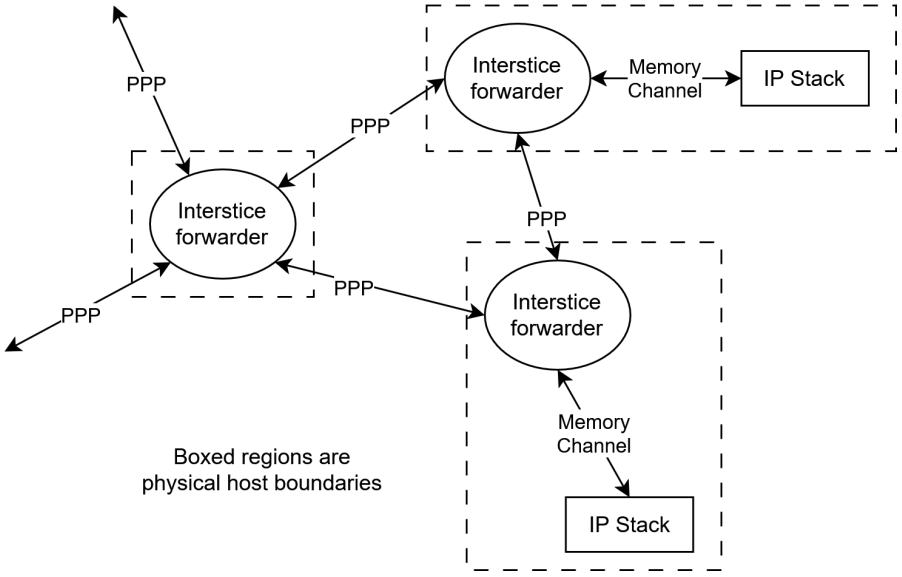

Figure 14: IP stacks as leaf nodes in an `interstice` *network.*

handling, and transport-layer machinery) from the minimal information needed to forward them through the network (destination address), which can simplify implementation. It also increases implementation flexibility (Figure 15), as it eliminates any strict correspondence between forwarder and network stack: conventionally, a network host will have one forwarder (to which all its external interfaces are attached) and one IP stack, but if the host knows that it has exactly one communication interface, it can choose not to run the forwarder at all and instead attach the IP stack directly to the external interface. Or, it can run multiple independent forwarders to support approaches comparable to VLAN. Or if it doesn't need or can't support IP services, but nonetheless wishes to provide transitive connectivity to its peers, it can run only a single forwarder to bridge its external interfaces — this will have a lower cost in system resources through the omission of the IP stack.

There are of course cases where we *want* to establish a firm one-to-one relationship between an IP stack and a packet forwarder, e.g. if the IP stack is exchanging information about network topology using a routing protocol and needs to communicate updates to the forwarder. In this case, the routing client will simply acquire a handle to the forwarder and mutate it to record route updates. The difference compared to an integrated approach is simply that this behavior is opt-in rather than assumed everywhere.

However, it must be noted that routing protocols which identify forwarding nodes by *IP address* will not work in our system, as our forwarders do not have addresses. Distance-vector based approaches such as AODV [3] are commonly formulated in terms of

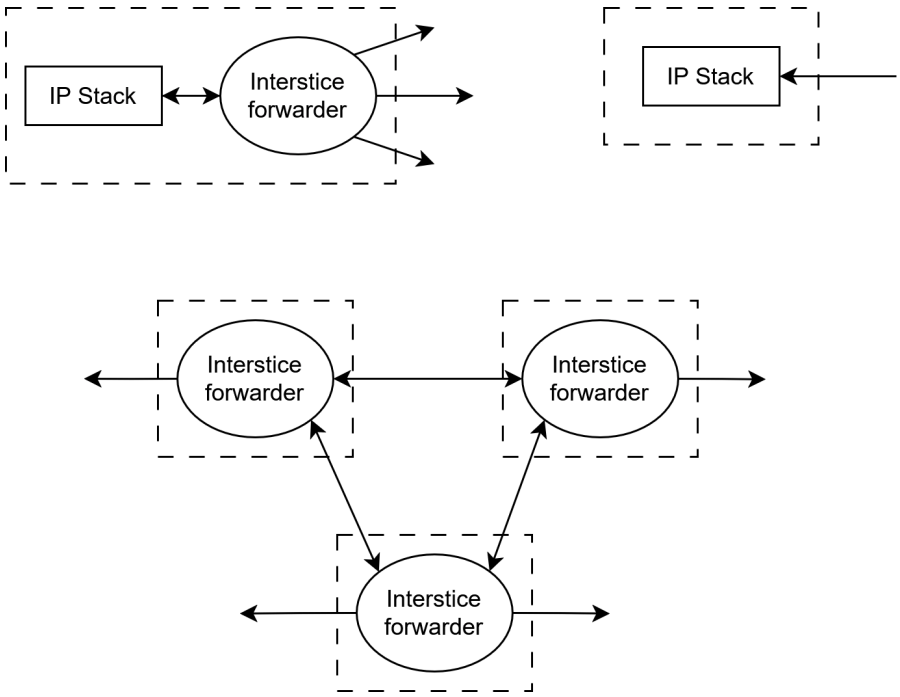

Figure 15: Separation of packet forwarder from network stack enables various connection topologies.

"neighbors" (which are identified by outgoing interface), and so will be more amenable to adaptation, but link-state approaches will of course work as well, so long as forwarder identity can be separated from IP address.

3.5. Lower layers

To demonstrate the flexibility and general applicability of my approach, I develop lower-layer adapters that can interface with `interstice` over various communication channels commonly available in embedded systems. Implementation approaches and considerations are detailed here; performance evaluation is performed and discussed in Section 4.

3.5.1. UART

UART is the paradigmatic default physical layer used in this work, as it is a simple point-to-point serial link that is ubiquitous in embedded systems. It is straightforwardly used to carry PPP traffic using PPPoS [22] framing. UARTs are asynchronous duplex, and have characteristic speed limitations due to the lack of a dedicated clock line, though speeds of on the order of 1Mbps are achievable on links with limited length and minimal exposure to electrical noise.

A notable mention for this category is USB CDC-ACM, a USB device class designed as a generic, virtual serial port. I commonly used PPP over CDC-ACM as the last link to connect my networks to Linux computers running `pppd` [27].

3.5.2. I^2C

I^2C is a very common embedded communication bus sporting 7- or 11-bit addressing, moderate data rates (100kbps is standard, up to 1Mbps), and a master-slave architecture. In the current implementation, the master node is selected statically at compile-time, but in principle it could be possible to negotiate this at runtime, improving flexibility and bootstrap capabilities.

Since I^2C indicates data length intrinsically, we *could* choose to drop the framing specified in [22] and instead assume that each transaction contains at most a single frame. But for the sake of sharing code and potentially improving throughput, I retain the HDLC-like framing in my approach with the addition of a one-byte per-transaction header.

3.5.2.1. Length indicator

The header byte simply indicates n: the number of data the device has buffered for transmission as of the start of the message, with $n = 255$ indicating that at least 255 bytes are ready. The remote end only treats the first n bytes of the message (after the length byte) as valid, ingesting them into its HDLC-like deframer (the rest of the packet may contain arbitrary values). If the bus master sees $n = 255$ or $n >$ len (the actual transaction length), it knows the slave has more data and should try another transaction. $n = 0$ indicates that the remote is not sending any data, which may be used by the slave if it has nothing to send. These semantics imply that an overall read length of 256 bytes (1 header byte + 255 data bytes) is the maximum that should be used.

3.5.2.2. Polling

A limitation of master-slave bus architectures is that slaves cannot initiate communication. While it's possible to use an additional conductor (shared, or one-per-slave) to indicate readiness in the slave $\longrightarrow$ master direction, I don't do that here, and instead adopt a periodic polling approach. After a period of inactivity, the bus master polls each of the slaves for data availability in a round-robin manner.

3.5.2.3. Role-switching approaches

I^2C supports multimaster operation of the bus through collision detection mechanisms, i.e. multiple bus members can concurrently hold a master role. This suggests that I^2C could potentially act as a full multipoint link, but hardware limitations presently make this difficult to achieve, as peer masters cannot typically *listen* on the bus and selectively

switch their role to act as a slave for transmissions destined for their address. Future work might consider attempting to emulate this functionality by actively switching peripheral modes and only occupying the master role while actively transmitting.

3.5.3. CAN

CAN bus is an automotive communication bus based on a differentially-signaled twisted-wire pair. Bus semantics are broadcast, i.e. all nodes hear and may choose to receive all messages. The protocol has many variants, but standard CAN (2.0A) uses 11-bit message identifiers. Notably, identifiers are *not* strictly constrained to addressing semantics, and could instead be used to encode commands or message types. The bus has arbitration mechanisms that ensure that messages with lower identifier values take priority over those with higher values.

In my system, I stipulate that message IDs have addressing semantics; i.e. each node adopts a single ID that it listens for exclusively. At the moment, the set of IDs on testnet CAN links are hard-coded, so discovery and automatic address assignment are not performed, but we could in the future reserve a specific message ID on which on-link devices might periodically announce their presence, which joining devices could monitor in order to determine and claim an unused ID. The current implementation runs basic PPP between each pair of devices on the link with no CAN-specific optimizations. Clearly this approach is inefficient, as it does not take advantage of the bus as a shared medium; the goal of the implementation is primarily to showcase the possibility of using this alternate medium to carry IP.

3.6. Name services

`interstice` provides mDNS [32] support: each node running the software can provide a name for itself, which defaults to `interstice.local` and can be overridden by an environment variable. mDNS resolution requests are forwarded through the network and successfully resolve transitively connected nodes — this is nonstandard behavior, as the RFC specifies that lookups should be link-local. Clearly, given that our links are point-to-point, this would not make sense for our network paradigm, hence the forwarding behavior.

3.7. IP header compression

Header compression is a family of techniques that reduce the amount of information that needs to be transmitted over a given link, typically by inspecting network- and transport-layer packet headers and eliding repetitious information that can be reconstructed by the receiver from context.

This is attractive because it can lead to savings in link budget and hence improved network performance; on a relatively slow medium like a 115200 baud serial port, we can deduce that an upper bound on packet rate (if we sent only empty packets and framing was free) is $115200 \text{ bps} \div 20 \text{ bytes of IPv4 header} = 720 \text{ packets/s}$; if we could omit the 8 bytes of addressing information, that bound would increase to 1200 packets/s, an improvement of 66%. Reality will only approach these numbers if the packets are consistently small, but as [33] observes, some network traffic such as interactive user input *does* produce a stream of latency-sensitive, characteristically-small TCP/IP packets, and regardless we expect the technique to be beneficial overall, even in cases where those benefits might be comparatively marginal.

3.7.1. Stateful approaches

Van Jacobson compression [33] implements header compression for TCP/IP by observing that information commonly repeats *over time* on a given TCP connection: caching the last sent or received packet on a given connection enables us to compare the delta to the next packet and compute a custom header representation that only transmits the changed fields. As the receiver also caches the last packet, it is able to synthesize the original network-layer header before it is handed off to the IP stack. Similar approaches known as IP Header Compression (IPHC [34]) and subsequently Robust Header Compression (ROHC, [35]) generalize the approach to other transport layers, notably for our case including UDP.

While I began an initial implementation of VJ compression as part of the thesis (as there is no existing embedded-compatible implementation in Rust, to my knowledge), implementation proved too involved to complete here, so I do not present a stateful compression approach as part of this work. Future work may complete this partial implementation.

3.7.2. Address compression

Address compression is based on an observation of "spatial" data repetition: it is not uncommon for there to exist a mapping from link layer address (e.g. MAC address) to network address, both of which will conventionally appear physically within the same link-layer frame. In principle, if the address mapping is stipulated to be one-to-one and known *a priori* (such as is possible in IPv6's Stateless Address Autoconfiguration (SLAAC) [36]), the network-layer addresses can be omitted or partially truncated (see Figure 16). This technique is optionally employed in 802.15.4 networks, for instance, as specified in [37].

Notably, address compression can only be applied in situations where a lower-layer address exists, e.g. it is not usable with PPP over UART or RS-232. This makes buses such

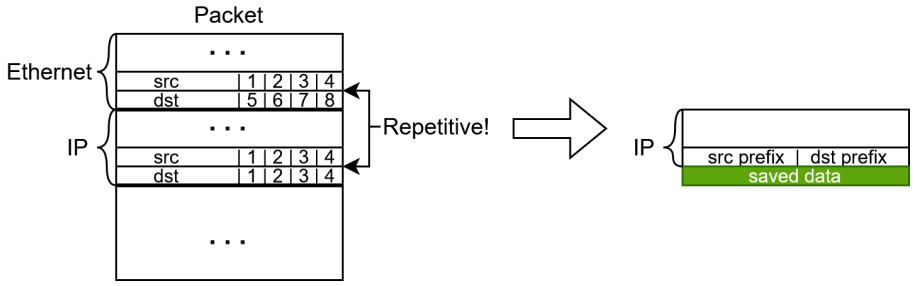

Figure 16: Address compression using link-layer addresses.

as I^2C and CAN interesting as candidates for applying this technique, as they do necessarily make use of addresses which could in principle be used to reconstruct a network level address.

I provide an bus-oriented experimental implementation of address compression in my modifications to `ppproto` (Rust PPP implementation). This change compresses the destination address (normally 4 bytes in IPv4) by optionally replacing it with a single zero octet in the same location in the IPv4 header. If present, this zero (which should never appear in the first octet for a standards-compliant address being transmitted through the network) indicates that the following three bytes of the address were elided, and the receiver should fill in its own address, because the sender is asserting that the packet was unicast to it on-link, and the sender knows the destination's address. This approach requires that the sender have this knowledge, which is achievable in a number of ways. In my implementation, I simply check for a configurable-length prefix match, assuming that the network will be organized such that the shared bus link has its own exclusive prefix.

Compression for source addresses could be performed in almost exactly the same way, but would require that the bus provide a source address (not available for CAN) and a mapping from that bus address into a network address. Future work could implement this approach to improve compression for I^2C links.

3.8. Design summary

In this section, I discussed considerations impacting network and software design and made a number of conclusions.

First, I discussed how the network system should be organized with respect to external networks. I concluded that it would be preferable to treat our network as its own self-contained, self-managed system, rather than trying to integrate addressing and routing with any external LANs it might border.

I then considered the benefits of PPP as a link layer, concluding that it would be useful for traffic multiplexing and per-link configuration exchange, easing the adoption of experimental configuration options. I suggested that the configuration mechanism might be used to automatically detect the boundaries of our system.

Subsequently, I discussed how the network will forward packets. I suggested that different applications scenarios may call for different approaches and so adopted a modular forwarder design, in which the forwarding strategy could be substituted based on design requirements. I adopted a flooding (hub) forwarder and a transparent bridge, and also discussed the design of a routed approach based on AODV.

Next, I considered the design of the forwarder software. For reasons discussed previously, I adopted a modular design, in which the forwarding strategy could be substituted between various implementations. In order to improve genericity, I abstracted both the address and packet representations used by the software to the minimal interface required for packet forwarding.

To improve flexibility of deployment on constrained devices, I separated the packet forwarder from the IP stack, enabling it to be deployed on its own. I discuss that it is possible to remove IP addresses entirely from the forwarder, and instead relocate them to the IP stacks only, as network endpoints.

I discuss implementation concerns for the lower layer serial links used in this thesis: UART (+ similarly, USB CDC-ACM), I^2C, and CAN. Notable items include that I^2C slave $\rightarrow$ master transmissions include an initial length byte indicating the amount of data the slave has remaining, I^2C masters poll periodically, and CAN uses message ID as a node ID.

Additionally, mDNS was adopted for name service, slightly modified for our network to propagate request multicasts beyond link boundaries.

I also considered header compression approaches, and implemented a simple IP address compression scheme for CAN bus. It makes use of the fact that the zero byte never occurs in the first position of a well-formed transmitted IP address.

4. Evaluation

This project's primary evaluations were benchmarking for performance in several metrics and feasibility and qualitative flexibility demonstration through the implementation of integrated systems. These evaluations are intended to establish the contours of the system's performance (lower- and upper-bounds, suggestive initial trends) under a variety of conditions.

The aim of these evaluations is to validate that this networking approach and the software supporting it meet reasonable standards of performance, which would make their use generally viable and applicable under the design criteria explored in Section 3.

Where possible, the following tests are performed on hardware networks of microcontrollers across electrical links. Some tests, however, were conducted in simulation on PCs, to establish bounding performance trends that were practically prohibitive to realize physically due to scale.

4.1. Testnet configurations

Networks of embedded microcontrollers (Figure 17, Figure 18, Figure 19) were assembled to evaluate the function and performance of the system in tests detailed below. Independent test nets of three to four devices were assembled for each of the link types under consideration.

The size of these networks was limited to this small number of devices due to the time cost of assembling large networks by hand and the number of devices practically available to me for conducting the tests. Future work should consider larger networks of these devices and/or extend the preliminary simulation work discussed in Section 4.3.

4.2. Performance benchmarks

Performance evaluations were performed for each link type to provide a comparison between them, and to enable analysis of which layers might be responsible for slowdowns.

1. An initial test characterizes the performance of the physical layer only — e.g. what is the actual speed my UART is capable of running at, with the consideration of my software in the loop, but no link or network protocols. This is meant to establish any deviation from the *nominal* bitrate for the link: due to overhead and programming particularities, a 115200 baud UART, for instance, might in reality communicate slower than 115.2 kbit/second.

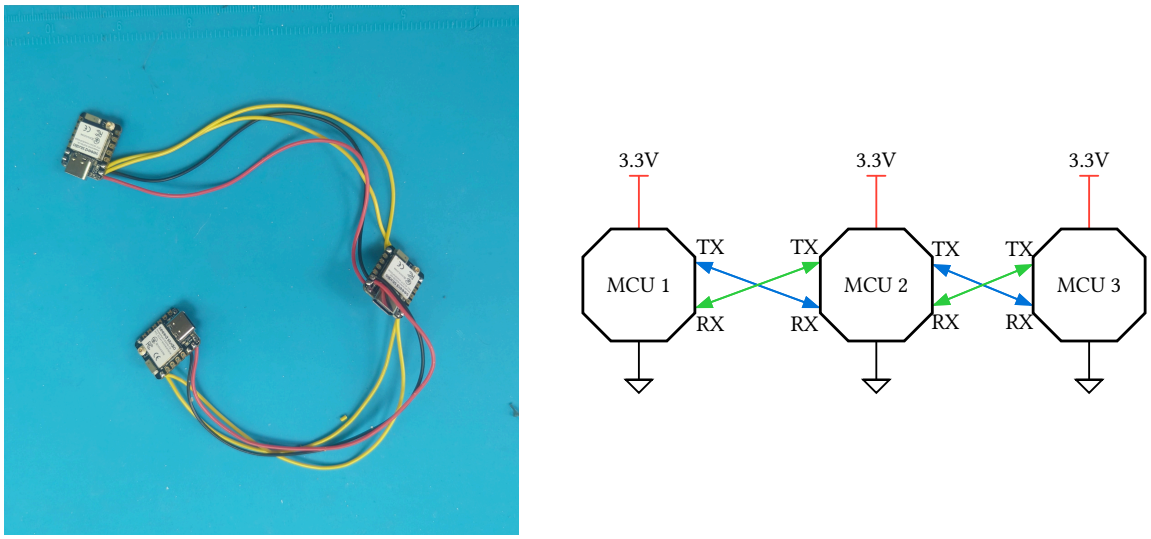

Figure 17: UART test network and topology. ESP32C6 microcontrollers on "Xiao"° boards are connected to their neighbors, forming a point-to-point network.

2. A UDP echo test evaluates the overall bidirectional throughput capabilities of the system with both `interstice` and an IP stack in the loop. These are conducted over a single network hop for the sake of comparison with the physical layer test.

4.2.1. Physical layer

The results of these tests are shown below: Figure 20 evaluates raw physical-layer echo throughput. A transmitter device sent 64-byte payloads to a receiver as fast as possible, and the receiver echoed them back on the same link. The transmitter recorded the number of bytes sent and received over a 10-second window at various configured physical layer rates across each link; the overall throughput per second is indicated in the figure.

As the caption in Figure 20 notes, we see substantially lower roundtrip performance for I^2C and CAN buses compared to UART because they are half-duplex, where UART is full-duplex, meaning that we should expect the total bidirectional throughput to be on the order of half that of UART. This puts the overhead of their byte-stream abstractions in the same ballpark (which shows a remarkably flat 80% utilization).

It bears mentioning that that none of these systems use DMA acceleration due to a lack of peripheral and/or HAL support: UARTs and I^2C are programmed and read using FIFOs, and CAN uses a direct register interface to write and FIFOs to read. This likely imposes a speed cost compared to a theoretical system using DMA bidirectionally with all peripherals.

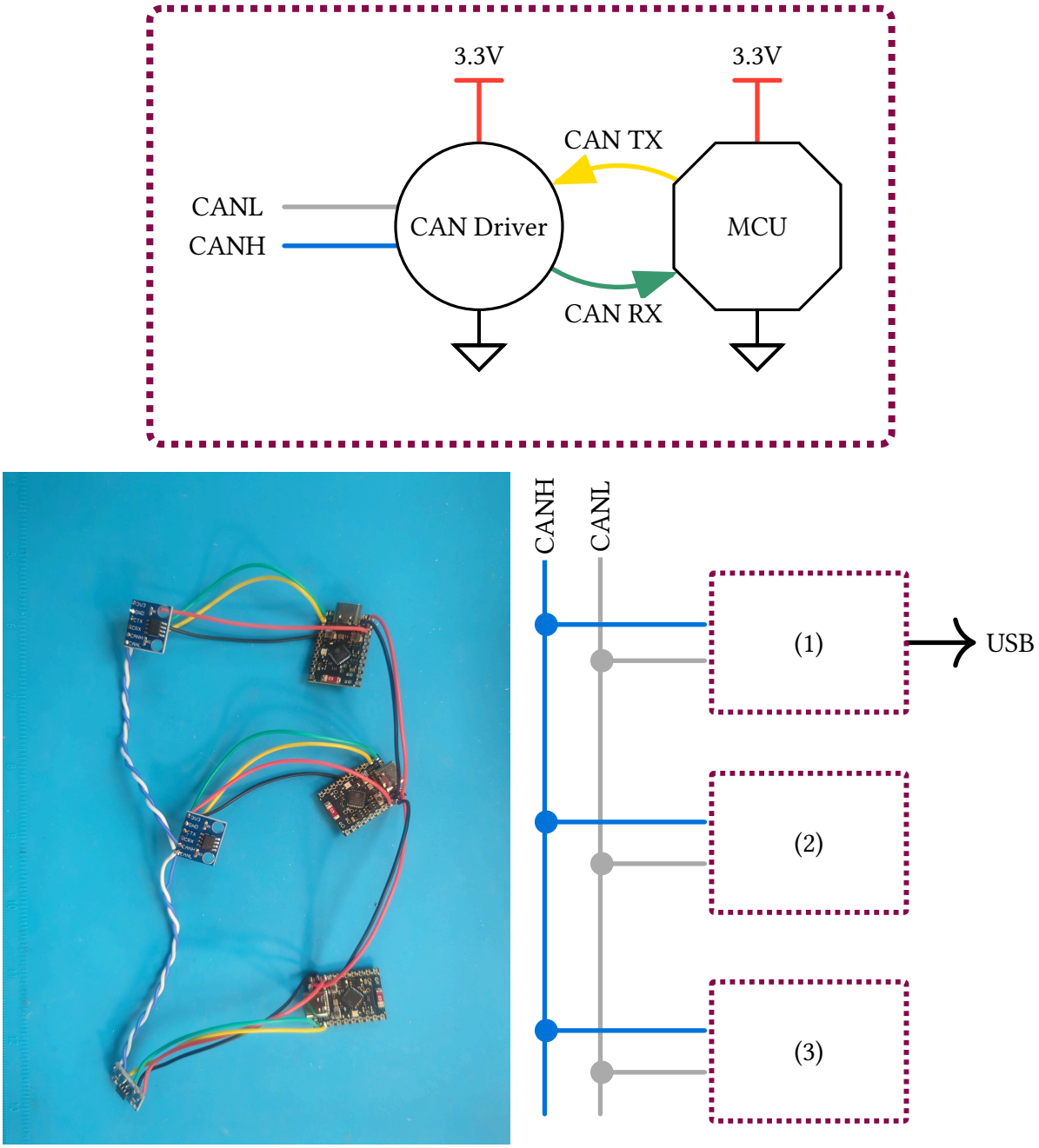

Figure 18: CAN bus test network and topology. ESP32C6 microcontrollers on ESP32C6 "Super Mini" boards° are connected to per-device CAN drivers. The CAN drivers are connected together to form a connected bus, which carries encapsulated IP traffic via `interstice`*. This configuration was tested to support CAN speeds up to 1Mbps, the fastest supported rate for the ESP32C6's CAN peripheral.*

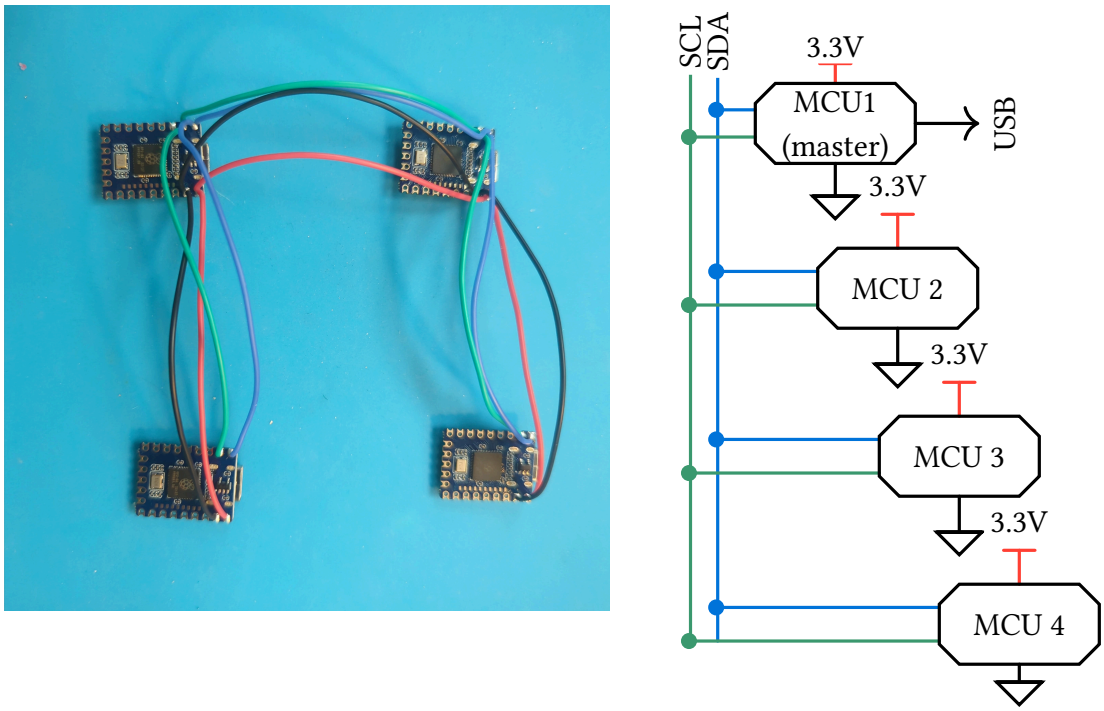

Figure 19: I²C test network and topology. Four RP2040 microcontrollers (on "RP2040 Zero" PCBs°) are connected in a linear topology sharing +5V, ground, and I²C clock and data.

4.2.2. Transport layer

Figure 21 shows the UDP echo test results. This test used the same basic configuration as the physical layer tests: a transmitter device and an echo device. The transmitter sends 64-byte UDP packets to the echo device as fast as it can, and the echo device responds with the same packet.

Notably, though this test also used payloads of 64 bytes, this was the size of the UDP *payload* only. Counting the UDP header (8 bytes), the IPv4 header (20 bytes), and PPP framing and escape sequences (8 bytes), the total transmitted frame sizes were 100 bytes. Figure 22 adjusts the data from Figure 21 by scaling recorded data up by a factor of $\frac{100}{64} \approx 156\%$ to account for these complete transmitted frame sizes. Notably, this indicates that UART at 9600 baud actually exceeded the baseline physical speed tested for this link rate, achieving around 120% utilization relative to baseline. I suspect this to be because the test configuration lucked into a preferable timing situation compared to baseline — the right synchronization primitives were scheduled in the right order: more work is required to establish if this is true.

We can see from these figures that link utilization is near-perfectly efficient at low speeds, but as link speed increases, the higher layers become less capable of taking advantage of

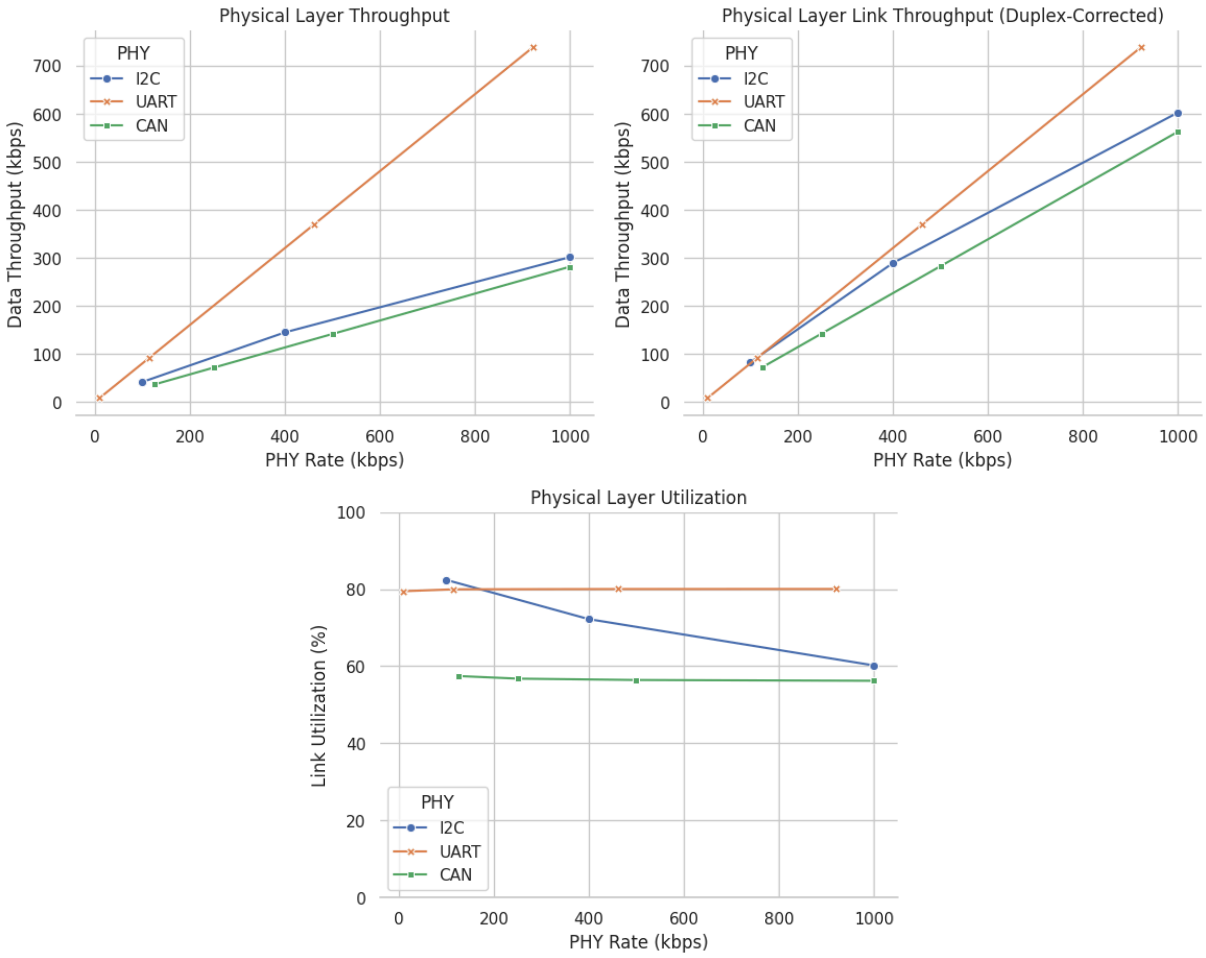

Figure 20: Throughput across tested physical media, running against the software adapters used to model them as abstract point-to-point byte streams (networking and link layers inactive). The top-right figure doubles the measurements for I^2C *and CAN to correct for the fact that they are half-duplex: only one transmitter can be active on a bus at a given time, so effective PHY unidirectional rate is cut in half, or put another way, twice as much effective traffic passes through the bus for a single roundtrip compared to UART. The bottom link utilization graph is calculated based on this corrected data.*

available throughput, dropping off to about 20% when approaching 1Mbit/second. This behavior makes sense at a first glance, as timing requirements to top off and drain data to/from FIFOs become more stringent as data rates get faster, but we would expect the same fall-off to appear in the physical layer tests; since we don't, it is question-begging that this effect only appears here.

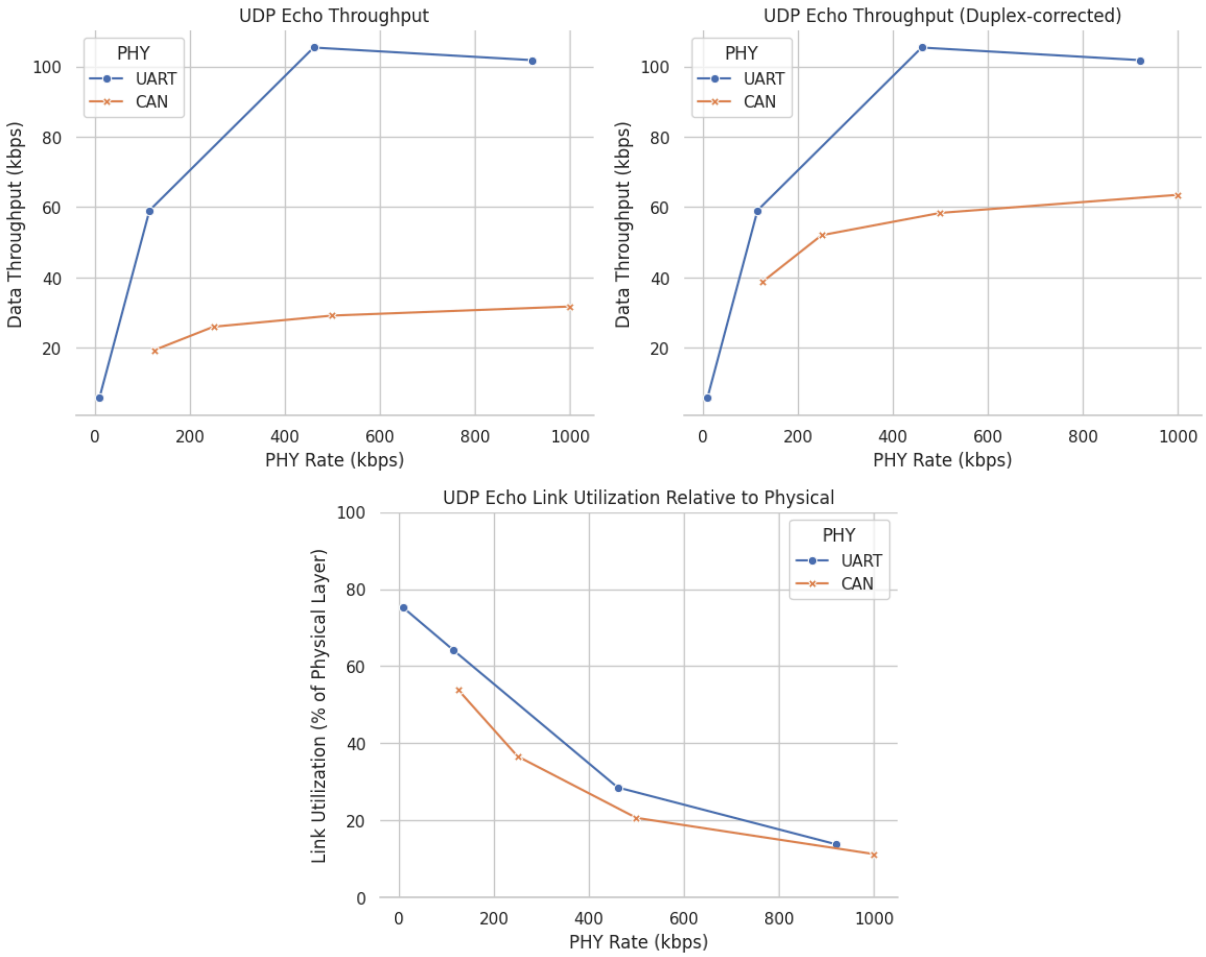

Figure 21: Comparative UDP echo throughput across tested physical layers. As in Figure 20, the top-right figure doubles the CAN measurement to account for the fact that it's half-duplex. The bottom utilization plot is calculated using this corrected data. Previously-unseen bugs appeared in the I^2C implementation when running it at sustained high data rates in combination with PPP, causing it to crash; unfortunately, data for that medium is not currently available.

4.2.3. Diagnosing performance

More work is required to diagnose the reason for throughput fall-off at higher link speeds. I was able to achieve a substantial (more than 2x across-the-board) performance uplift for UART compared to a previous testing iteration by increasing buffer sizes, tweaking the IP stack's socket ingress/egress ordering, and selectively moving certain data processing into elevated-priority interrupt contexts, but capturing full performance at high link rates remains elusive.

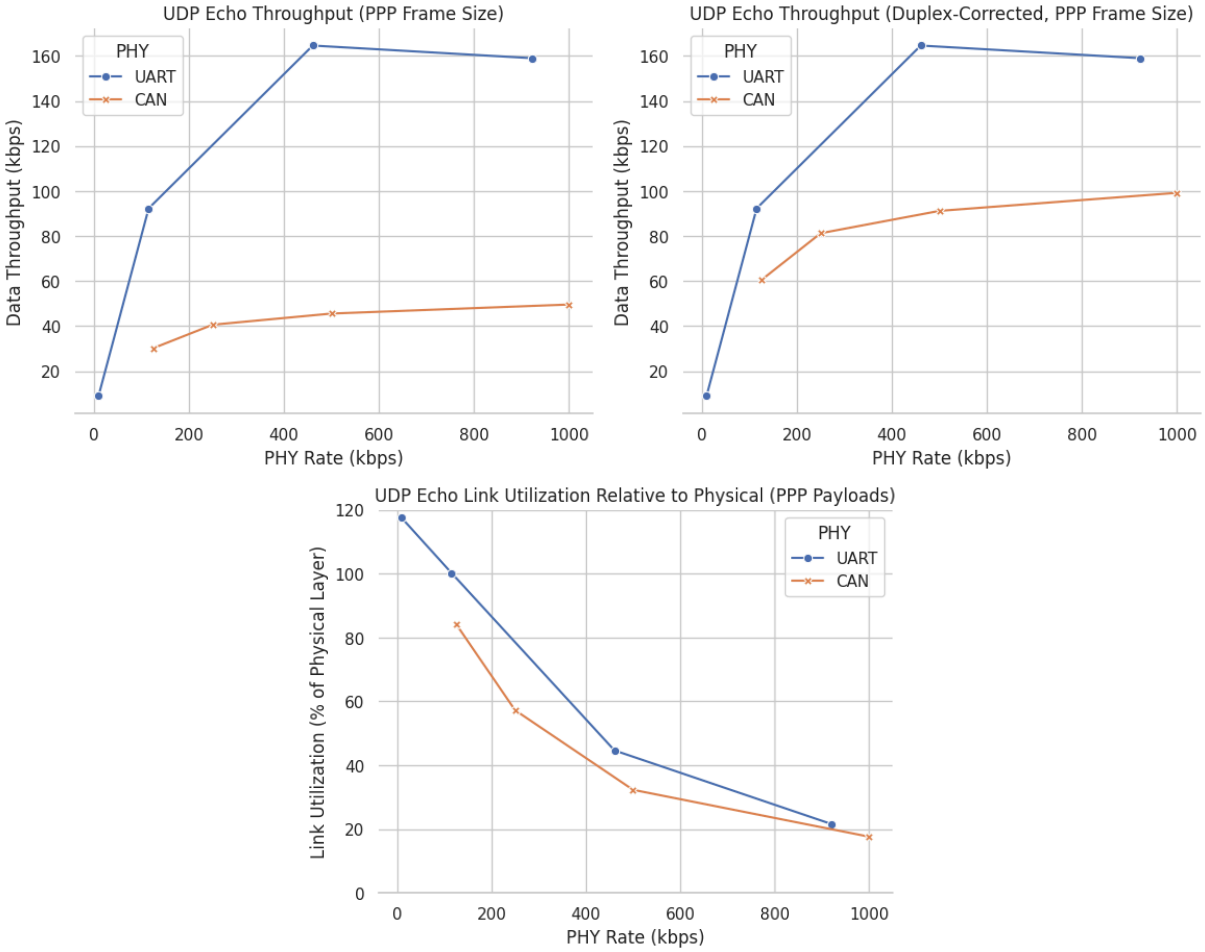

Figure 22: Echo throughput for UDP payloads calculated using complete PPP frame sizes. As in Figure 20, the top-right figure doubles CAN's measurement to account for the fact that it's half-duplex. The bottom utilization plot is calculated using this data.

Symptomatically, in my investigation of UART performance, RX FIFO overflows at the transmitter node (i.e. caused by responses from the echo device) are the proximal cause preventing sending packets faster at higher link rates; when overflow errors occur, the whole 100-byte PPP frame is trashed and becomes wasted bandwidth. These suggest that the transmitter node is not draining the RX FIFO frequently (or consistently) enough.

I hypothesize that this effect is an artifact of imprecise or inconsistent processor timing in filling and draining peripheral I/O blocks, as I suggested earlier. At 960kbps (the fastest UART rate I evaluated), a 100-byte frame is transmitted in $833\mu s$, so $100\mu s$ of FIFO underutilization would mean a 12% decrease in throughput. By contrast, for a link running at 115200 baud, a 100 byte frame is transmitted in 6.9ms, meaning that a $100\mu s$ timing

deviation only creates a throughput impact of 1.4% — it makes sense that performance becomes more sensitive with higher rates.

As UART data handling occurs in an interrupt context, my primary suspect for these slowdowns is overuse of critical sections, which by definition would exclude the UART handler from running. I suggest that we don't see this slowdown effect on the physical layer tests because they make very little use of multitasking (they're not running the IP stack or PPP link drivers).

As a next step, future work should perform a similar performance test on a microcontroller system using DMA for peripheral I/O. This would take the processor out of the I/O loop, enabling us to determine whether that indeed was the bottleneck.

These inferences are grounded somewhat by a simulated test I subsequently performed on a Windows PC, which used the same UDP echo code to achieve a round-trip throughput of 560kbps across an in-memory channel artificially limited to a transmit rate of 960kbps and a simulated FIFO size of 64 bytes. Taken together with the tests from Figure 20 and the presence of FIFO overflow errors, these factors suggest that it may be specifically the interface with the I/O peripheral that is our bottleneck.

4.3. Supported network size

As a first-order upper-bound on maximum network size and for validation of basic system function, random tree-like networks (e.g. Figure 23) of selected sizes were generated and simulated on a Linux laptop equipped with an AMD Ryzen 7 7745HX CPU and 32 GB of DDR5 DRAM.

The system was able to support UDP unicasts between random nodes for networks of sizes up to 5000 nodes using the transparent bridging forwarder, and I have no reason to think that larger networks would not work; these limitations appeared traceable to configurable memory bounds that could be increased.

See the `random_graph` and `random_graph_unicast` files in the source code for this tests.

4.4. Upper-bound forwarding throughput

Tests (Figure 24) were conducted in-memory on a Windows host (Intel i7-12700KF, 64 GB DDR4) to determine maximum throughput across the generic packet forwarder alone, as a presumed upper bound for performance when deployed in embedded systems. The test was single-threaded and configured to run in the `embassy` executor. Code was release-optimized (`-O3`) and logs were disabled. The in-memory I/O channel provided by `embassy_net_null_modem` was used for packet ingress and egress. A mock packet type

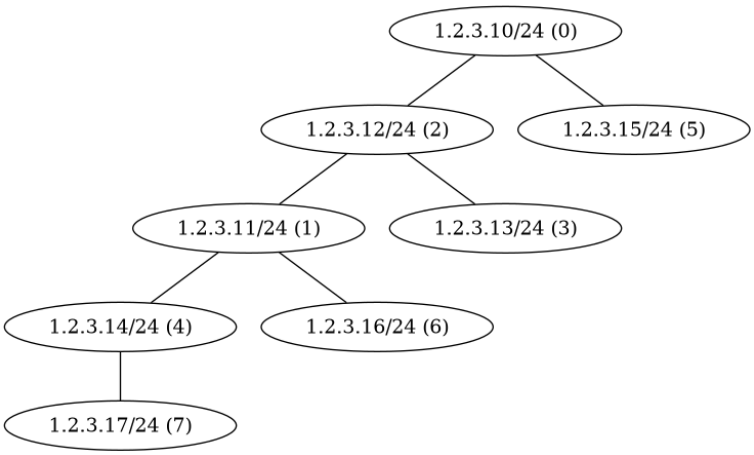

Figure 23: Example randomly-generated net graph for maximum network size evaluation.

consisting of only a data payload of 1024 bytes was used; forwarders treated it as a broadcast. The test can be found in the `throughput_test` file.

The maximum throughput rate measured by this test was 579 MB/s using the hub forwarder and 578 MB/s for the bridging forwarder, which suggests that the forwarders are unlikely to bottleneck network performance.

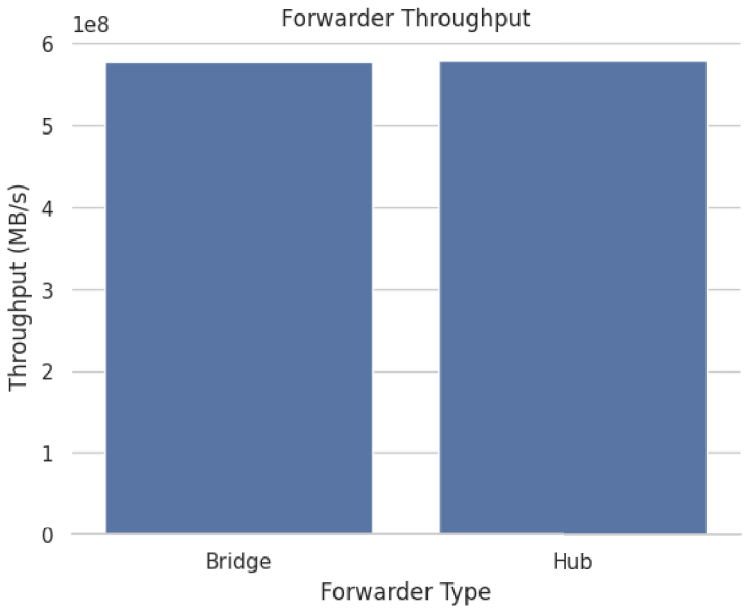

Figure 24: In-memory throughput for `interstice` across each of the packet forwarder implementations.

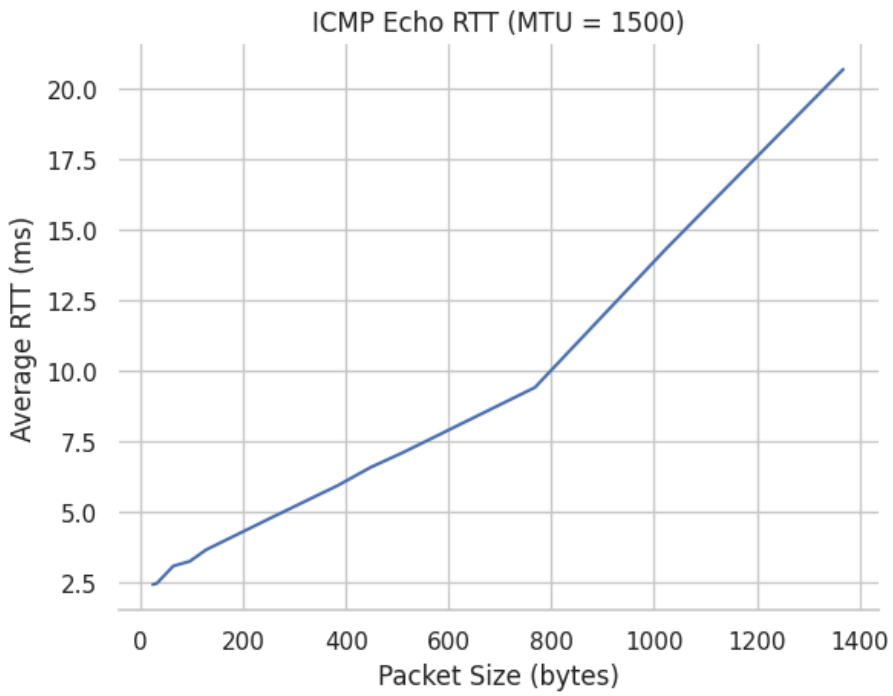

Figure 25: RTT latency as a function of packet size.

4.5. ICMP echo (`ping`) tests

I measured round-trip-time across a number of packet sizes to investigate performance of the approach. Results are summarized in Figure 25.

These tests were conducted between an MCU and a Linux system, across a link making use of a USB CDC-ACM device connected to the Linux kernel IP stack via an instance of `pppd` [27]. The `pppd` instance was configured in modem mode, with `crtcts` set, without authentication, with an MTU of 1500 bytes, and with LCP echoes (used as keepalives) every 1 second. The MCU in use for this test was the ESP32C6; firmware was compiled with release optimizations and logging was completely disabled. The test was performed using the `ping_test` script and firmware in the project repo, which simply instantiates `interstice` with an IP stack responsible for responding to ICMP echo packets. The script allows for packet-size sweeps of fixed-count `ping` invocations.

The noticeable knee in Figure 25 at a packet size of just under 800 bytes was repeatable, and was even more noticeable on a previous test that still had debug logging enabled (which tends to contribute to substantial performance hits, as in some situations the whole packet contents are logged).

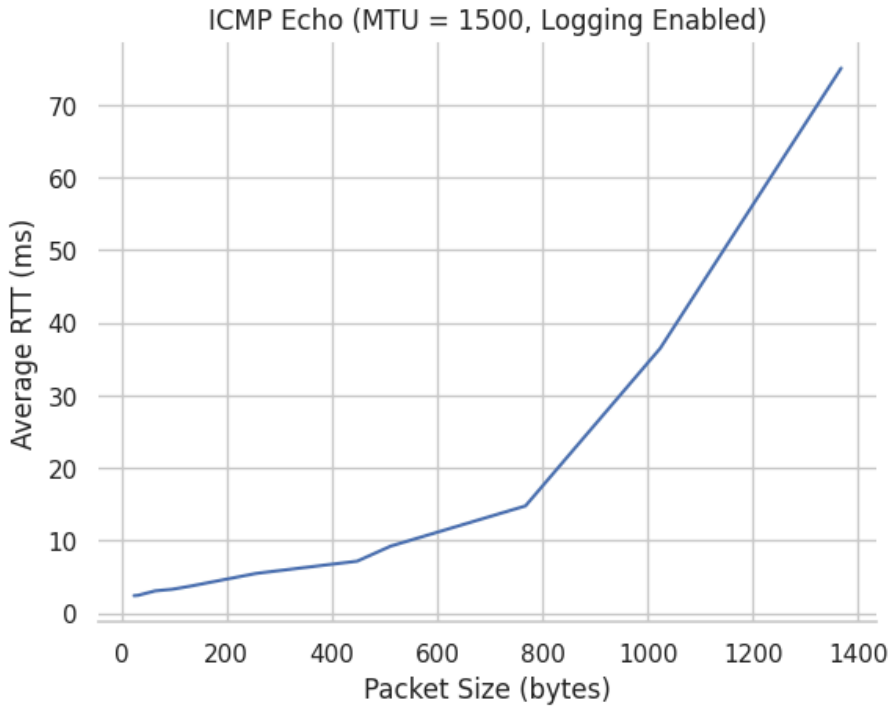

Figure 26: RTT latency as a function of packet size with logging enabled.

Figure 26 shows more clearly the knee at about 800 bytes. I am still uncertain why there is an inflection point here.

4.6. Functional demonstrations

4.6.1. Device-hosted web control page

This demo (Figure 27) shows Web-based device control via device-hosted HTTP servers. Each device hosts its own HTTP server, which is made transitively over the network. The servers presently expose a root webpage and a backend API which enables control of an onboard LED.

This demo showcases the utility of using standard IP for device interfacing: the web page implements autonomous device control in JavaScript (blinking the LED on a timer).

This test presented noticeably sluggish page load times (which only increased with the number of hops to the target device). Wireshark investigations surfaced consistent TCP retransmissions and duplicate acks, as the Linux host loading the webpage would time out. These duplicate packets were not seen on comparatively fast links, such as a page load over the direct CDC-ACM link to a network edge device.

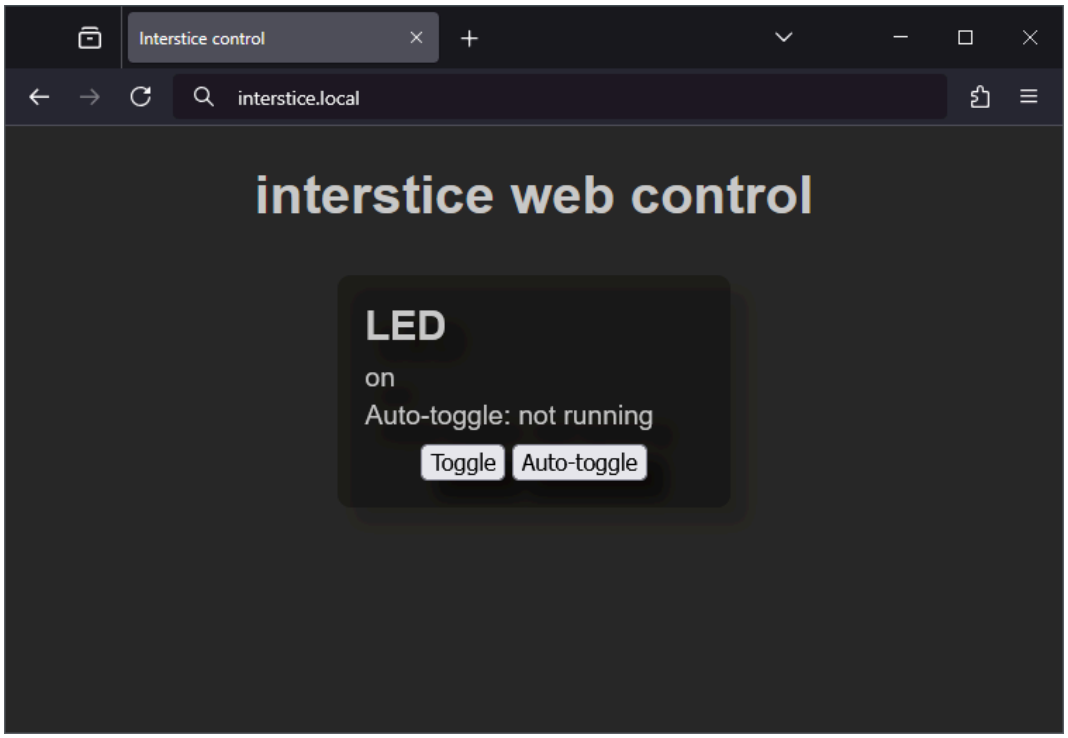

Figure 27: Screenshot of simple device-hosted HTTP control webpage, served through an `interstice` *network, which provides control of device LEDs. The device IP was resolved automatically by the host system using mDNS from the name* `interstice.local`.

4.6.2. Networked machine

`scriptorium` (Figure 28) is a PCB milling machine I previously built, substantially derived from Modular Things [38] and the Clank [39] machine base, which is controlled as a distributed machine: each stepper motor has its own microcontroller driving it. These

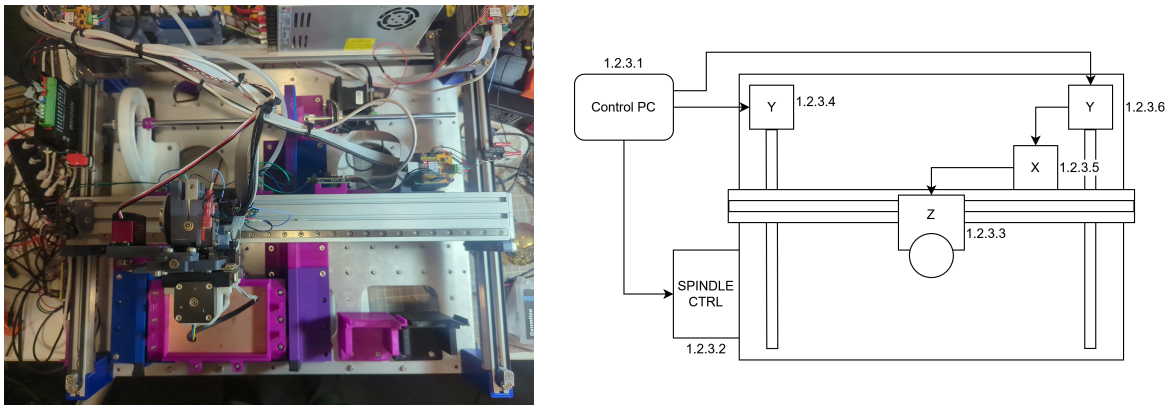

Figure 28: Left: `scriptorium` *PCB milling machine. Right: connection diagram of machine adapted to use* `interstice` *networking for control.*

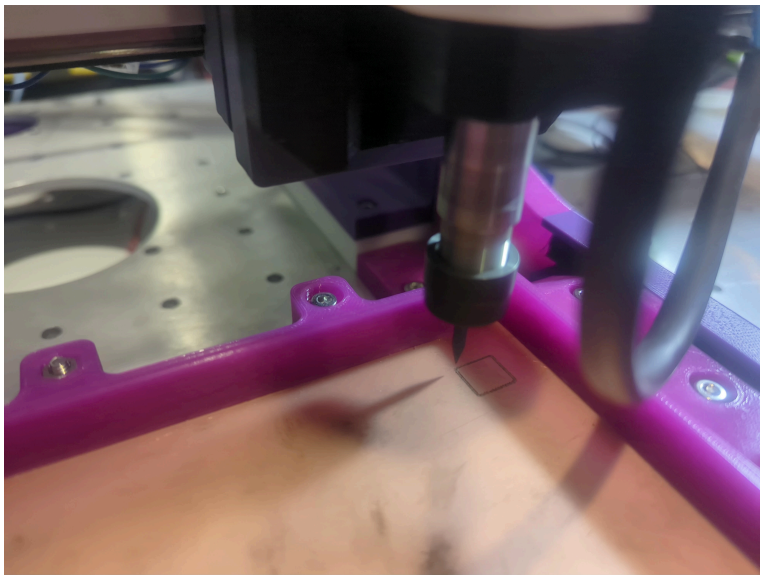

Figure 29: `scriptorium` *milling a PCB blank, controlled over* `interstice` *network.*

microcontrollers each run their own position control loop driving their respective stepper motor. Formerly, the position setpoint was controlled from a host computer using a custom protocol over USB CDC-ACM (virtual serial ports).

I rewrote the motor controller firmware to use `interstice` for communication — each motor accepts commands to update the position setpoint via UDP rather than the former COBS-framed [26] custom serial protocol. As a result, the machine is fully controllable over the network and capable of performing milling operations; the system in operation is shown in Figure 29.

4.6.2.1. Motion Performance

`scriptorium`'s firmware stepper controllers perform microstepping on the half-bridges that drive each of the motor stepper coils. Coil current magnitude is PWM-controlled using dedicated PWM peripherals on the RP2040 MCU at a frequency of 61kHz; a firmware trapezoid generator evaluates motion parameters every 200us (at a rate of 5kHz), and consults a 1024-entry sine LUT for microstepping. The half-bridge gates setting motor direction can be actuated at up to the same 5kHz rate if commands are received this quickly.

It may be useful to compare the approach to a popular integrated stepper driver, Analog Devices' TMC2209 [40]. The PWM frequency range recommended by the driver's datasheet is 20-50kHz, which my system exceeds. The internal step generator can be configured to step at up to $2^{24} \times 0.715 = 12$ MHz using the integrated clock source,

greatly in excess of my maximum supported rate of 5kHz, though my system will simply skip intervening microsteps if motion requirements demand.

Register write commands can be accepted by the TMC2209 at a maximum baud rate of 750,000, assuming the internal clock at 12MHz is used. Standard command packets are 64 bits, so this is a rate of just under 12k packets per second. Extrapolating from Section 4.2.3, our maximum effective data rate through UDP packets over a 960kbaud UART link is on the order of 12kB/s. For a data payload of size 4 bytes (a uint32 target position), we can assume 36 bytes of overhead as observed in Section 4.2.3 for a total packet size of 40 bytes, meaning an effective packet rate of 300/s, just 2.5% of the rate supported by the TMC2209. Of course, these advantages should not be surprising, as the TMC2209 presents an integrated solution.

The key benefit of my system is that the TMC2209 supports 4 stepper addresses on a single network without additional support, while my `interstice`-based solution in principle supports the whole IPv4 or IPv6 address space.

5. Future Work

In this section, I consider possible future extensions to the work completed in this thesis.

5.1. Memory considerations

Memory allocation strategies were an item of significant concern in the implementation, mostly centering around packet buffering. The considerations here include that allocation from the system heap is undesirable on embedded systems, as it can be exhausted and in general will behave nondeterministically, especially in the presence of any other allocating tasks on the system. However, heap-allocation approaches have benefits over the static buffers that my system actually adopts, as allocation efficiency will be better in the average case, supposing non-pathological fragmentation characteristics.

Future work might consider making use of linked-list allocation strategies like lwIP [9], with a mind towards supporting fragmented allocations. This approach seems to represent the best of both worlds, allowing flexible allocation and avoiding allocation failure caused by heap fragmentation. It must be acknowledged that these benefits do come at the cost of programmer ergonomics and runtime efficiency, however, as the list data structure trades off exactly the benefits of a flat homogeneous memory space for its apparent benefits in this use case.

5.2. Foreign Function Interface (FFI)

`interstice` currently ships without a C FFI. This limits its applicability to Rust-based embedded projects and therefore misses out on the opportunity to interoperate and provide utility to the larger population of C and C++ embedded programmers (e.g. in Arduino). Targeting C has not been a development priority, but adaptation of the modular dynamically-dispatched interfaces to FFI is nonetheless technically straightforward and a clear win in terms of usability.

The largest impediment to this implementation is choosing an approach for integrating platform scheduling and synchronization primitives into Rust async, which are a strict requirement for `interstice`. The simplest FFI-compatible asynchronous executor implementation could simply be polled in a hot loop and should provide for portable function, albeit with substantial performance and ergonomics sacrifices. In this minimal case, it would be possible to get away without explicit integration of platform synchronization primitives: the read and write functionality for each attached interface would simply end up being polled in a hot loop. It's expected that this approach would produce a crude but functional stack that could serve the needs of relatively simple embedded systems projects.

Work beyond this minimal integration might consider full `async` support for FreeRTOS and other embedded runtimes using their platform-specific synchronization primitives. `embassy-sync`'s `RawMutex` type can be generically extended to platform mutex types, enabling most of the functionality required to port `interstice`.

5.3. Energy usage

A performance metric of substantial interest on embedded systems is energy usage, as embedded devices commonly run on battery power, so energy spent on communications places a limit on device lifetime between charges (if charging is even possible at all). I did not perform an energy or power evaluation as part of this work, but it would be highly interesting to look at in the future: it seems plausible that the wired links considered here could potentially be lower-power than even minimally-duty-cycled 802.15.4 radios if effective power-optimization techniques are employed — measurements by STMicroelectronics in [41] suggests that mean double-digit microampere operation of low-power UART devices is possible if devices enter sleep mode between transmissions. The kind of niche I suggest might be filled here is illustrated in Figure 30.

In order to achieve these power characteristics, the network system would need to be awake at relatively low duty cycles, as active microcontroller power draw is commonly on the order of multiple milliamps. This requirement presents work for the `interstice` implementation as it exists currently: it would need to be adapted to minimize chatter (presently actively leveraged to detect link state) and resume from deep sleep while maintaining the PPP state machine.

However, on the other side of this energy issue, it bears considering that if application concerns demand high duty cycles on network or microcontroller activity, wired links could be advantageous over wireless technologies, as wireless radios tend to draw a substantial amount of power while turned on. The power characteristics of wired links are on the other hand are likely to be small, as the energy required to transmit a single bit is typically associated with driving a small number of low-capacitance conductors to a minimal voltage.

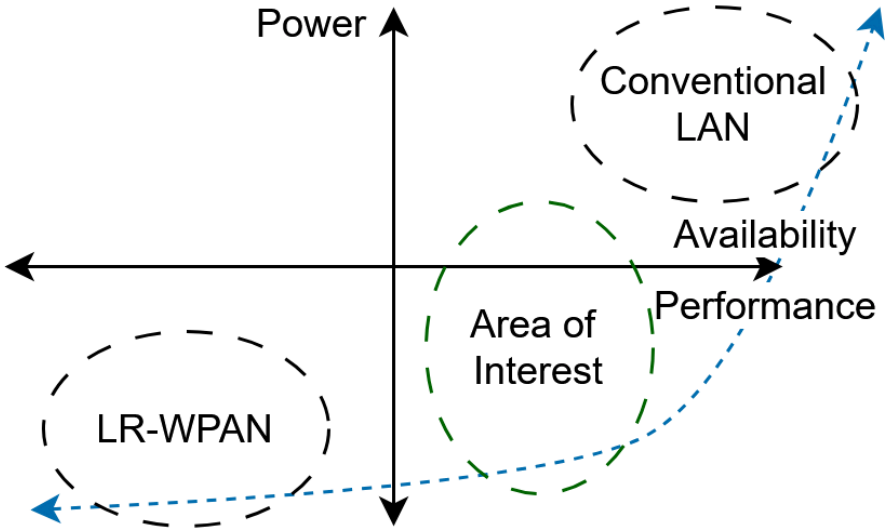

Figure 30: Perceived vacancy in the tradeoff between high-energy, high-speed, reliable communication devices and low-speed, extremely low-power WSN approaches. The dashed line represents a perceived Pareto frontier, the low-right corner of which I suspect could be expanded upon by low-power wired techniques.

6. Conclusion

This thesis has discussed the design and evaluation of an approach for constructing IP networks suitable for deployment on embedded devices across common serial links, providing for transitive connectivity through various packet forwarding approaches. PPP was adopted as a generic link for its wide portability and utility in negotiation of configuration options. The discussed design was evaluated by a number of performance benchmarks and showcased on several integrated systems.

This work successfully demonstrates the feasibility and utility of constructing these multihop, point-to-point networks. The approach showcases nontrivial functionality built on these networks, and has the potential to support previously difficult-to-attain connectivity for devices without dedicated networking peripherals, including brownfield projects, prototypes, and various applications requiring and using implicit networking which is presently difficult to attain. Benchmarks demonstrate that for these use cases, where having network connectivity at all is a substantial boon, performance on at least small networks proves acceptable.

This being said, work remains to advance the project to a state of general readiness. The software remains relatively unoptimized, scoring at times less than 50% of underlying line rate for overall throughput. While having network connectivity at all is better than not, for UARTs these kinds of limitations can put us in the performance territory of fifty-year-old modems, which should be avoided if possible. Compression techniques should also be considered in more depth, as they have the potential to save substantial amounts of bandwidth on these limited links. It remains to complete a proper routing approach, presumably based on AODV or similar, and general challenges nontheless relevant to our approach exist IP in regards to the integration of autonomously-administred private subnets.

My future projects in this domain will take these considerations into account, focus on broadening the library of links with integrations built into the software system, and look at mechanisms for and implications of overlaying such functionality on top of existing networks — a TCP connection is just a byte stream, after all.

A. Project source code overview

This section provides a summary of the source code developed during this project. Anything used in this thesis which has not been upstreamed to another project (upstreams are summarized in Appendix B) is hyperlinked and described here.

In general, source code resources are provided as links to public Git repositories, which can be browsed in a web UI or cloned over HTTPS without authentication. The version of the code used to produce the results reported in the thesis can be found at the tag `npry.sm_thesis.final` in each repo unless noted otherwise (e.g. experimental results earlier in the thesis). An archival copy of the code used in this thesis is also made available on Zonodo°.

Note: see Appendix B for patches that *have* been submitted to upstream.

A.1. Project repositories

A.1.1. `interstice`

Available here°. This is the primary repo for the thesis. If you are interested in building or using any of the functionality presented in this thesis, you only need to download this repo; Rust's `cargo` package manager and build tool (required to build the project) will take care of fetching the rest of the dependencies automatically.

A brief breakdown of the components:

- `interstice` is the root package providing the package forwarder implementation and abstractions, with source code available in `/src`. I plan to publish this in Rust's public crate registry°, but dependent functionality must be upstreamed first (direct VCS dependencies are not allowed, which `interstice` makes liberal use of).
- `embedded` provides functionality specific to the embedded platforms used in testing and evaluation for the thesis (bringup code, peripheral and runtime configuration, etc.) and contains the actual binary programs used for testing (`/embedded/src/bin`).
- `heapless_list` provides sister functionality to the `heapless`° family of crates, which implement versions of typically-heap-allocated data structures such as `Vec` and `String` with `const`-sized inline storage instead. `heapless_list` (this package) provides a singly-linked list type with inline storage and integer- rather than pointer-based node indexing. It is used in a number of places in `interstice` to store data that is expensive to copy but needs to be dynamically reordered — to my knowledge there is no non-allocating linked-list functionality in the Rust ecosystem at the moment. It will be contributed to Rust's public crate registry°.

- `dyn_embassy_net` provides abstractions and wrappers that enable `embassy_net`'s `Driver` type to be dynamically dispatched. This enables the `interstice` packet forwarder to simultaneously hold a heterogeneous collection of link drivers. `dyn_phy` provides the same for `smoltcp`° and its `smoltcp::phy::Driver` trait.
- `embassy_net_null_modem` and `smoltcp_null_modem` provide a virtual "null modem cable" (respectively compatible with `embassy_net` and `smoltcp`), a lower-layer interface that can connect two `interstice` instances together purely in memory. This functionality is how the host-only functionality in Section 4 was tested.

The `scriptorium` (milling machine) firmware was developed in a branch and is available under the `npry.sm_thesis.final_scriptorium` git tag (or also in the Zonondo linked above).

A.1.2. `flip`

Available here°.

Spiritually, this is an extension of `flipperzero-rs`°, the Rust bindings for the Flipper Zero. This code was developed with the intention of using the Flipper as an interactive terminal device that might be used to configure, debug, and visualize network traffic as part of this thesis project. As of the time of writing, work on this code has been suspended for the sake of time.

It implements:

- Safe wrappers around Flipper Zero SDK GUI types.
- Integration with `embassy_executor`, enabling Rust `async` on the Flipper (currently: partial support only). See Section 5.2 for potential application implications.
- `embassy_time_driver` support, providing datetime and alarm support.
- A `critical_section`° implementation for the Flipper Zero.
- Revised access to the Flipper notification service, enabling dynamic construction of notification sequences.
- Initial `embedded_io{,_async}` trait implementations for the Flipper Zero UARTs.

In the future, I intend to polish this work and upstream most of it into `flipperzero-rs`.

A.2. Forks

These are included in the `interstice` repo° under the `/3p` ("third party") directory unless otherwise noted.

A.2.1. `embassy`

An overview of Embassy can be found at Appendix B.2. This section describes notable additional changes yet to be upstreamed.

A.2.2. `embassy-net-ppp`: timeouts, reconnects

`embassy-net-ppp` is a link layer driver that makes use of `ppproto`'s implementation of PPP in HDLC-like framing to provide an `embassy-net` connection. The changes implemented here support timeout and features added to `ppproto` (described below).

A.2.3. `ppproto`

`ppproto` provides a `#![no_std]`, I/O-free implementation of PPP [21] and PPP in HDLC-like framing [22] in Rust. It is owned by the `embassy` authors.

Changes yet to be upstreamed include:

- Supporting `ConfigReq` timeouts in LCP and NCPs (this is required to recover from missed messages or state machine desync, e.g. because one end of the connection restarts)
- General support for connection reestablishment logic from `Dead` state
- Correct handling of `TermReq` messages and generation of `TermAck`s

A.2.3.1. Future work: buffering

The buffering approach used in this package leaves something to be desired — PPPoS frames are decoded into an intermediate buffer, which is read by PPP in one shot. But notably, the parsing semantics of PPP's LCP and NCP protocols are unambiguous and context-free, meaning that no backtracking is required to parse these messages. This suggests that it would be possible to adopt an incremental decoding approach that could eliminate intermediary buffers, which for systems with multiple PPP links contribute not insignificantly to system memory usage. The buffers consume `MTU * n` bytes of memory consumption, which even for a relatively small MTU value (say 256) and number of links (say 4) leads to an additional 1kiB of allocated memory. On an embedded microcontroller, even a larger one, this is not insignificant.

The other side of this issue is that an incremental parsing approach *does* however initially accept and process messages with invalid FCS. Most likely the appropriate action on many links will be to simply reset and restart the relevant option exchange state machine in this case, as internal state may have become misconfigured. This may not be practicable on links that experience regular errors, which should likely retain the buffering behavior.

A.2.4. `ssmarshal`

Original repo here°.

`ssmarshal` is a minimal, non-allocating serialization/deserialization crate suitable for use in resource-constrained environments. It is a transitive dependency of `interstice`, and recently became incompatible (indirectly) because Rust's `Error` type moved from `std` to `core`.

This fork updates `ssmarshal` according to the new organization of `Error`.

I intend to upstream this patch in the future.

A.2.5. `async-channel`

Available here° and here°.

The `portable-atomic`° Rust crate provides atomic integer types and derived functionality for all hardware platforms, even those without atomic intrinsics. It does this by falling back to a `critical_section`° implementation in cases where intrinsics are missing. This enables most dependencies in the transitive dependency closure of atomic integer and pointer types, e.g. `Arc` (Rust's atomically reference counted smart pointer type) to function on any platform that can provide a `critical_section` implementation, albeit at a performance cost.

`async-channel` is an in-memory channel type dependent on `alloc::sync::Arc` rather than `portable_atomic_util::Arc`. This fork adds a feature flag to `async-channel` to switch to `portable_atomic::Arc` for use on embedded platforms.

I intend to upstream this patch in the future.

B. Open-source contributions

The work conducted in this thesis involved a number of contributions to existing open-source projects, which are described below.

Note: see Appendix A for project repos and forks with patches that have not (yet) been upstreamed.

B.1. Rust `liballoc`, `libcore`

One contribution was made to each the `core` and `alloc` libraries of the Rust programming language. These represent the subset of the standard library which operates without OS facilities and is therefore used on embedded microcontrollers.

B.1.1. `const`-ify `String::as_str`, `Vec::as_slice`

Rust has support for `const` expressions, which are evaluated at compile-time, comparable to `constexpr` in C++. These were adopted after the language was released and are gradually being integrated into library functionality on an as-needed basis.

`String` and `Vec` are the heap-allocated string and vector types provided by Rust, comparable respectively to C++'s `std::string` and `std::vector`. They provide methods `String::as_str(&self) -> &str` and `Vec::as_slice(&self) -> &[T]`, which produce slices (references endowed with a length, comparable to e.g. C++ `std::string_view` or Python's `bytes`) into their contents. Previously, these methods were not `const`, despite being `const`-computable; this contribution makes them and several other useful methods (including `len`, `capacity`, `is_empty`) `const` (Listing 3).

As Rust code cannot determine whether it is being evaluated at compile-time, this enables variance over `const`ness for these types.

The feature was implemented here° behind the `const_vec_string_slice` unstable feature flag, is tracked here°, and as of the time of writing is undergoing a stabilization process here°.

```rust
const fn str_is_short(s: &str) -> bool {
    s.len() < 5
}

const EMPTY_STRING: String = String::new();

// Previously not available.
const EMPTY_STRING_IS_SHORT: bool = str_is_short(EMPTY_STRING.as_str());
```

Listing 3: Standard library contribution enables `const`-evaluation of length for `str`, which was previously unavailable.

```
fn get_bytes(ip: &Ipv4Addr) -> &[u8] {
    // Copy address bytes into stack local.
    let bytes: [u8; 4] = ip.octets();

    // Trying to return a reference to function local; invalid reference
in caller
    // scope. Indeed, the Rust compiler outlaws returning a reference to
a local.
    // We want to take a reference into the IpAddr argument instead.
    &bytes
}
```

Listing 4: Illustration of pointer lifetime constraints for IP address types, old approach.

B.1.2. `IPAddr::as_octets`

This contribution provides additional functionality to `core`'s IP address types: `core::net::IpAddr`, `core::net::Ipv4Addr`, and `core::net::Ipv6Addr`. Methods `IpAddr::as_slice(&self) -> &[u8]`, `Ipv4Addr::as_octets(&self) -> &[u8; 4]`, `Ipv6Addr::as_octets(&self) -> &[u8; 16]` were provided, which enable interpretation of these types as network-order array references (or a slice in the case of `IpAddr`).

Previously, these types exposed only `.octets() -> [u8; N]`, which enabled a *copy* of the address contents as bytes onto stack memory (Listing 4). While this was acceptable in many cases, references to stack memory have limited validity, and heap allocation is not always acceptable (indeed, it is actively undesirable in many embedded contexts).

We would prefer to be able to take a reference *directly* into the owning `IpAddr` type, such that this reference is valid exactly as long as the `IpAddr` is, and the owning memory is shared (i.e. additional storage is not required).

This contribution implements this functionality with `as_octets()` functions, enabling the behavior shown in Listing 5.

The feature was proposed here°, implemented here° behind the `ip_as_octets` unstable feature flag, and stabilization is tracked here°.

B.2. `embassy`

The `embassy` project provides a collection of library packages that support writing Rust programs on a variety of embedded microcontrollers with use of Rust's `async` facilities.

```
const fn get_bytes(ip: &Ipv4Addr) -> &[u8] {
    ip.as_octets()
}
```

Listing 5: `const` -ref access for IP addresses in Rust.

```
static CH = embassy_sync::pubsub::Channel::new();

let task_handle = spawn(async move {
    let sub = CH.subscriber();

    loop {
        let val = sub.await;
        println!("{val}");
    }
});

// Fixed by this patch.
CH.publish_immediate(4);

// Task prints '4'
```

Listing 6: Method visibility fix for `embassy_sync::pubsub`.

B.2.1. `embassy_sync::pubsub::Channel` method visibility

`embassy_sync` is a library that provides runtime-agnostic synchronization primitives and data structures designed to be allocation-free and `async`-compatible. One structure exposed by the library is `pubsub::Channel`, which provides an in-memory channel supporting multiple simultaneous publishers and subscribers with broadcast semantics. This change fixes a set of method visibility issues, restoring `pubsub::Channel::publish_immediate`, `pubsub::Channel::capacity`, and `pubsub::Channel::is_full` to public visibility — see Listing 6.

Implementation here°.

B.2.2. `impl futures::Sink for embassy_sync::pubsub::Pub`

See the previous section for an overview of `embassy_sync::pubsub`. The `embassy_sync::pubsub::Channel` type provides a publish-only handle to itself via `Channel::publisher() -> Pub` and a consume-only (subscription) handle via `Channel::subscriber() -> Sub`. The `Sub` handle implements the `futures::Stream` trait, which supports the production of an asynchronous stream of values as they become available within the channel. A dual `futures::Sink` trait exists, supporting asynchronous submission of values to the channel as space becomes available, but it was not implemented for `Pub`. This contribution provides that trait implementation for `embassy_sync::pubsub::Pub`: see Listing 7.

Implementation here°.

```rust
static CH = embassy_sync::pubsub::Channel::new();

let task_handle = spawn(async move {
    let sub = CH.subscriber();

    loop {
        let val = sub.await;
        println!("{val}");
    }
});

let publ = CH.publisher();
let mut stream = futures::stream::repeat(5);

// This method call requires that publ implement `futures::Sink`.
publ.send_all(&mut stream).await;

// Task repeatedly prints '5'
```

Listing 7: `futures::Sink` *implementation for* `embassy_sync::pubsub::Pub`.

B.3. Miscellaneous

B.3.1. `http`: `#![no_std]`

`http`° is the de-facto standard crate in the Rust ecosystem for describing HTTP types. It does not implement server or client functionality, but rather serves as shared location for the definition of types, constants, and utility functionality (status codes, methods, headers, a `Request` type, a `Response` type, and a few other useful facilities that are universal to any participant in the HTTP ecosystem).

Up until now, it has had hard dependencies on Rust's standard library. This contribution makes that dependency more granular and controllable via feature flags – the library can now be included on embedded platforms by selecting the `alloc` feature but not `std`, with little compromise to functionality.

Implementation here° (upstream pending as of writing).

B.3.2. `frunk`: `#![no_std]`

`frunk` is a functional programming toolkit for Rust. This contribution eliminates its dependency on Rust's OS-dependent standard library, making it suitable for use on embedded microcontrollers and other constrained environments.

Implementation here°.

B.3.3. Flipper Zero: GPIO interrupt ordering fix

The Flipper Zero° is an "electronics multitool" based on the STM32WB55 microcontroller. It provides a screen, D-pad, speaker, sub-GHz software-defined radio transceiver, infrared transceiver, Bluetooth radio, OneWire interface, and array of I/O ports exposing discrete GPIOs and processors buses.

This change is a patch to the Flipper Zero firmware which fixes a sequencing issue with its GPIO interrupt handlers. Relevant context is that the firmware accepts interrupt handlers defined by user applications. Previously, the firmware's own handlers would first call the user code, then clear the interrupt status for the given pin *after* the user code had run. This meant that any interrupts that fired within the user code could not be observed, as the interrupt status was always unconditionally cleared afterwards.

This contribution clears interrupt status only before user code runs to resolve this issue.

Proof of concept°, merged fix°.

Bibliography

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

[2] P. Chwalek, S. Zhong, D. Ramsay, N. Perry, and J. Paradiso, "AirSpec: A Smart Glasses Platform, Tailored for Research in the Built Environment," in *Adjunct Proceedings of the 2023 ACM International Joint Conference on Pervasive and Ubiquitous Computing & the 2023 ACM International Symposium on Wearable Computing*, in UbiComp/ISWC '23 Adjunct. New York, NY, USA: Association for Computing Machinery, Oct. 2023, p. 204. doi: 10.1145/3594739.3610796°.

[3] S. R. Das, C. E. Perkins, and E. M. Belding-Royer, "Ad hoc On-Demand Distance Vector (AODV) Routing," Jul. 2003. doi: 10.17487/RFC3561°.

[4] I. Wicaksono, "Textile Macroelectronics: Architecting Sensate and Computational Fabrics across Scales," 2024. Accessed: May 14, 2025. [Online]. Available: https://dspace.mit.edu/handle/1721.1/157711°

[5] P. Baran, "On Distributed Communications Networks," *IEEE Transactions on Communications Systems*, vol. 12, no. 1, pp. 1–9, Mar. 1964, doi: 10.1109/TCOM.1964.1088883°.

[6] C. S. Yoo, "Paul Baran, Network Theory, and the Past, Present, and Future of the Internet," *Colorado Technology Law Journal*, vol. 17, no. 1, pp. 161–186, 2018, Accessed: Apr. 30, 2025. [Online]. Available: https://heinonline.org/HOL/P?h=hein.journals/jtelhtel17&i=175°

[7] Y. Ofek and M. Yung, "METANET: principles of an arbitrary topology LAN," *IEEE/ACM Transactions on Networking*, vol. 3, no. 2, pp. 169–180, Apr. 1995, doi: 10.1109/90.374118°.

[8] B. Yener, Y. Ofek, and M. Yung, "Topological design of loss-free switch-based LANs," in *Proceedings of INFOCOM'95*, Apr. 1995, pp. 88–96. doi: 10.1109/INFCOM.1995.515864°.

[9] "lwIP - A Lightweight TCP/IP stack - Summary [Savannah]." Accessed: Apr. 27, 2025. [Online]. Available: https://savannah.nongnu.org/projects/lwip/°

[10] A. Dunkels, "adamdunkels/uip." Accessed: May 05, 2025. [Online]. Available: https://github.com/adamdunkels/uip°

[11] "Zephyr Project." Accessed: May 05, 2025. [Online]. Available: https://www.zephyrproject.org/°

[12] "RIOT - The friendly Operating System for the Internet of Things." Accessed: May 05, 2025. [Online]. Available: https://www.riot-os.org/°

[13] 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°.

[14] R. Alexander *et al.*, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks." [Online]. Available: https://www.rfc-editor.org/info/rfc6550°

[15] 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°.

[16] "Home - Web of Things (WoT)." Accessed: May 12, 2025. [Online]. Available: https://www.w3.org/WoT/

[17] "What is Matter - Matter." Accessed: May 12, 2025. [Online]. Available: https://handbook.buildwithmatter.com/howitworks/whatismatter/

[18] L. Zhang *et al.*, "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014, doi: 10.1145/2656877.2656887.

[19] A. Corsaro *et al.*, "Zenoh: Unifying Communication, Storage and Computation from the Cloud to the Microcontroller," in *2023 26th Euromicro Conference on Digital System Design (DSD)*, Sep. 2023, pp. 422–428. doi: 10.1109/DSD60849.2023.00065.

[20] W. A. Simpson, T. Narten, E. Nordmark, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)," Sep. 2007. doi: 10.17487/RFC4861.

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

[22] W. A. Simpson, "PPP in HDLC-like Framing." [Online]. Available: https://www.rfc-editor.org/info/rfc1662

[23] D. Carrel, J. Evarts, K. Lidl, L. A. Mamakos, D. Simone, and R. Wheeler, "A Method for Transmitting PPP Over Ethernet (PPPoE)." [Online]. Available: https://www.rfc-editor.org/info/rfc2516

[24] A. G. Malis, D. A. Y. Lin, J. Stephens, G. Gross, and M. Kaycee, "PPP Over AAL5." [Online]. Available: https://www.rfc-editor.org/info/rfc2364

[25] "Nonstandard for transmission of IP datagrams over serial lines: SLIP," Jun. 1988. doi: 10.17487/RFC1055.

[26] 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.

[27] "ppp-project/ppp." Accessed: Apr. 14, 2025. [Online]. Available: https://github.com/ppp-project/ppp

[28] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006, doi: 10.1109/JSSC.2005.864128.

[29] J. Moy, "OSPF Version 2," Apr. 1998. doi: 10.17487/RFC2328.

[30] Y. Rekhter, S. Hares, and T. Li, "A Border Gateway Protocol 4 (BGP-4)," Jan. 2006. doi: 10.17487/RFC4271.

[31] "Routing Information Protocol," Jun. 1988. doi: 10.17487/RFC1058.

[32] S. Cheshire and M. Krochmal, "Multicast DNS," Feb. 2013. doi: 10.17487/RFC6762.

[33] "Compressing TCP/IP Headers for Low-Speed Serial Links." [Online]. Available: https://www.rfc-editor.org/info/rfc1144

[34] M. Degermark, B. Nordgren, and S. Pink, "IP Header Compression," Feb. 1999. doi: 10.17487/RFC2507°.

[35] M. Degermark *et al.*, "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed," Jul. 2001. doi: 10.17487/RFC3095°.

[36] T. Narten, T. Jinmei, and S. Thomson, "IPv6 Stateless Address Autoconfiguration," Sep. 2007. doi: 10.17487/RFC4862°.

[37] P. Thubert and J. Hui, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," Sep. 2011. doi: 10.17487/RFC6282°.

[38] J. R. Read, L. Mcelroy, Q. Bolsee, B. Smith, and N. Gershenfeld, "Modular-things: plug-and-play with virtualized hardware," in *Extended abstracts of the 2023 CHI conference on human factors in computing systems*, 2023, pp. 1–6.

[39] "Clank: Modular CNC." Accessed: May 07, 2025. [Online]. Available: https://clank.tools/°

[40] Analog Devices, "TMC2209_datasheet_rev1.09.pdf." Accessed: May 14, 2025. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/TMC2209_datasheet_rev1.09.pdf°

[41] STMicroelectronics, "How to optimize LPUART power consumption on STM32 MCUs," Oct. 2024. Accessed: May 02, 2025. [Online]. Available: https://www.st.com/resource/en/application_note/dm00151811-minimization-of-power-consumption-using-lpuart-for-stm32-microcontrollers-stmicroelectronics.pdf°

[42] 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.

[43] D. Waitzman, "IP over Avian Carriers with Quality of Service." [Online]. Available: https://www.rfc-editor.org/info/rfc2549°

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

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

[46] R. Monroe, "Standards." Accessed: Oct. 29, 2024. [Online]. Available: https://xkcd.com/927°

[47] 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°.

[48] R. Raji, "Smart networks for control," *IEEE Spectrum*, vol. 31, no. 6, pp. 49–55, Jun. 1994, doi: 10.1109/6.284793°.

[49] "Will fragmentation of standards only hinder the true potential of the IoT industry? | Evothings." Accessed: Oct. 29, 2024. [Online]. Available: https://web.archive.org/web/20210227182106/https://evothings.com/will-fragmentation-of-standards-only-hinder-the-true-potential-of-the-iot-industry/°

[50] N. Pazos, M. Müller, M. Aeberli, and N. Ouerhani, "ConnectOpen - automatic integration of IoT devices," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Dec. 2015, pp. 640–644. doi: 10.1109/WF-IoT.2015.7389129°.

[51] 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°.

[52] G. Rigazzi, F. Chiti, R. Fantacci, and C. Carlini, "Multi-hop D2D networking and resource management scheme for M2M communications over LTE-A systems," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, Aug. 2014, pp. 973–978. doi: 10.1109/IWCMC.2014.6906487°.

[53] 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°.

[54] 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°.

[55] T. Watteyne *et al.*, "OpenWSN: a standards-based low-power wireless development environment," *Transactions on Emerging Telecommunications Technologies*, vol. 23, no. 5, pp. 480–493, 2012, doi: 10.1002/ett.2558°.

[56] 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°.

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

[58] 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°.

[59] 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°

[60] 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°.

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

[62] "Search Results for 'regex' - crates.io: Rust Package Registry." Accessed: Apr. 14, 2025. [Online]. Available: https://crates.io/search?q=regex&sort=downloads°

[63] J. Chroboczek and D. Schinazi, “The Babel Routing Protocol.” [Online]. Available: https://www.rfc-editor.org/info/rfc8966

[64] “Compressing TCP/IP Headers for Low-Speed Serial Links.” [Online]. Available: https://www.rfc-editor.org/info/rfc1144

[65] “embassy-rs/embassy.” Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/embassy-rs/embassy

[66] Espressif, “ESP32-C6 Technical Reference Manual.” Accessed: Apr. 29, 2025. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-c6_technical_reference_manual_en.pdf

[67] S. E. Deering, “ICMP Router Discovery Messages,” Sep. 1991. doi: 10.17487/RFC1256.

[68] M. Schroeder *et al.*, “Autonet: a high-speed, self-configuring local area network using point-to-point links,” *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, pp. 1318–1335, Oct. 1991, doi: 10.1109/49.105178.

[69] G. Baldoni, L. Cominardi, M. Groshev, A. de la Oliva, and A. Corsaro, “Managing the Far-Edge: Are Today's Centralized Solutions a Good Fit?,” *IEEE Consumer Electronics Magazine*, vol. 12, no. 3, pp. 51–61, May 2023, doi: 10.1109/MCE.2021.3082503.

[70] R. T. Braden, “Requirements for Internet Hosts - Communication Layers,” Oct. 1989. doi: 10.17487/RFC1122.

[71] H. Iqbal, M. H. Alizai, Z. A. Uzmi, and O. Landsiedel, “Taming Link-layer Heterogeneity in IoT through Interleaving Multiple Link-Layers over a Single Radio,” in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, in SenSys '17. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1–2. doi: 10.1145/3131672.3136966.