Design Considerations for Low Power Internet Protocols

Hudson Ayers*, Paul Crews*, Hubert Teo*, Conor McAvity*, Amit Levy†, Philip Levis*
*Stanford University, †Princeton University
{hayers, ptcrews, hteo, cmcavity}@stanford.edu, aalevy@cs.princeton.edu, pal@cs.stanford.edu

***Abstract*—Low-power wireless networks provide IPv6 connectivity through 6LoWPAN, a set of standards to aggressively compress IPv6 packets over small maximum transfer unit (MTU) links such as 802.15.4.**

The entire purpose of IP was to interconnect different networks, but we find that different 6LoWPAN implementations fail to reliably communicate with one another. These failures are due to stacks implementing different subsets of the standard out of concern for code size. We argue that this failure stems from 6LoWPAN's design, not implementation, and is due to applying traditional Internet protocol design principles to low-power networks.

We propose three design principles for Internet protocols on low-power networks, designed to prevent similar failures in the future. These principles are based around the importance of providing flexible tradeoffs between code size and energy efficiency. We apply these principles to 6LoWPAN and show that the modified protocol provides a wide range of implementation strategies while allowing implementations with different strategies to reliably communicate.

***Index Terms*—Interoperability, Embedded Systems, IoT**

I. Introduction

Interoperability has been fundamental to the Internet's success. The Internet Protocol (IP) allows devices with different software and link layers to communicate. IP provides a basic communication substrate for many higher layer protocols and applications. In the decades of the Internet's evolution, we have accumulated and benefited from a great deal of wisdom and guidance in how to design, specify, and implement robust, interoperable protocols.

Over the past decade, the Internet has extended to tens of billions of low-power, embedded systems such as sensor networks and the Internet of Things. Hundreds of proprietary protocols have been replaced by 6LoWPAN, a standardized format for IP networking on low-power wireless link layers such as 802.15.4 [1] and Bluetooth Low Energy [2]. 6LoWPAN was created with the express purpose of bringing interoperable IP networking to low power devices [3]. Many embedded operating systems have adopted 6LoWPAN [4]–[10] and every major protocol suite uses it [11], [12]. In fact, devices today can communicate with the broader Internet.

However, in many cases 6LoWPAN implementations cannot communicate *with each other*. We find that no pairing of the major implementations fully interoperates (Section III). Despite 6LoWPAN's focus on interoperability, two key features of the protocol — range extension via mesh networking of devices, and the convenience of different vendors being able to share a gateway — are largely impossible 10 years later.

Each of the openly available 6LoWPAN stacks, most of which are used in production, implements a subset of the protocol *and* includes compile-time flags to cut out additional compression/decompression options. As a result, two devices might both use 6LoWPAN, yet be unable to exchange certain IP packets because they use required features the other does not implement. This is especially problematic for gateways, which *need* to be able to talk to multiple implementations to enable significant scaling in real world applications.

This paper argues that the failure of 6LoWPAN interoperability stems from applying traditional protocol design principles to low-power networks. Low-power protocols minimize energy consumption via compression. Squeezing every bit out of packet headers, however, requires many different options/operating modes. Principles such as Postel's Law [1] state that an implementation must be able to receive every feature, even thought it only sends some of them. However, code space is tight on many systems. As a result, when an application does not fit, developers cut out portions of the networking stack and stop working with other devices. Put another way, 3kB of unused compression code seems tiny, but when removing it allows an additional 3kB of useful features, developers cut out parts of 6LoWPAN and devices become part of a custom networked system rather than the Internet.

This paper presents three design principles which resolve this tension between interoperability and efficiency. Protocols following these principles get the best of both worlds: resource-limited devices can implement subsets of a protocol to save code space while remaining able to communicate with every other implementation.

Capability spectrum: a low-power protocol specifies a linear spectrum of capabilities. Simpler implementations have fewer capabilities and save less energy, while fuller implementations have strictly more capabilities and are able to save more energy. When two devices differ in capability levels, communication can always fall back to the lower one.

Capability discovery: a low-power protocol provides mechanisms to discover the capability of communicating devices. This discovery can be proactive (advertisements) or reactive (in response to errors).

[1]"Be liberal in what you accept, and conservative in what you send" [13]

2325-2944/20/$31.00 ©2020 IEEE
DOI 10.1109/DCOSS49796.2020.00027

Explicit, finite bounds: a low-power protocol specifies explicit, finite bounds on growth during decompression. Without explicit bounds, buffers must be sized too conservatively. In practice, implementations allocate smaller buffers and silently drop packets they cannot decompress.

This paper is not the first to observe poor 6LoWPAN interoperability [14], but it is the first to identify this root cause. It is the first to define design principles for protocols that allow implementations to reduce code size and still interoperate. This paper examines how these principles could be applied to a low power protocol and evaluates the overhead of doing so. It finds that applying these principles to 6LoWPAN promises interoperability across a wide range of device capabilities, while imposing a code size cost of less than 10%. In particular, capability discovery requires an order of magnitude less code than the code size difference at the extremes of our capability spectrum, with minimal runtime overhead.

II. Background

This paper focuses on protocols for low-power, low-cost network devices, such as sensor networks or IoT devices, that use ultra-low power, low-cost microcontrollers. The marginal cost of additional resources, in terms of price and energy, is significant. Radio communication often dominates the energy budget: each transmitted bit consumes as much energy as thousands of instructions.

A. Low Power Protocols

6LoWPAN is a set of standards on communicating IPv6 over low-power wireless link layers such as IEEE 802.15.4 and Bluetooth [1], [2], [15]. 6LoWPAN primarily specifies two things: aggressive compression of IPv6 headers and how to fragment/re-assemble packets on link layers whose MTU is smaller than the minimum 1280 bytes required by IPv6. 6LoWPAN also specifies optimized IPv6 Neighbor Discovery. 6LoWPAN is critical to ensuring that IPv6 communication does not exceed the energy budget of low power systems.

The interoperability IP provides is a goal in and of itself, and so many different network stacks, including ZigBee and Thread, have transitioned to supporting IP connectivity with 6LoWPAN. Over the past decade, most major embedded operating systems (OS) have transitioned to using 6LoWPAN [4]–[8], [12]. IP connectivity allows systems to easily incorporate new services and applications, and allows applications to build on all of the existing Internet systems, such as network management and monitoring.

B. Low Power Hardware and Operating Systems

Table I shows a variety of older and more recent low-power platforms. Modern microcontrollers typically have 128-512 kB of code flash. Applications often struggle with these limits on code flash, and rarely leave code space unused. Embedded systems are application specific, and use their limited available resources toward different ends. Despite this, they still rely on a small number of reusable OSes for basic abstractions.

Table I: Flash size varies widely across IoT platforms

IoT Platform	Code (kB)	Year
Tmote Sky	48	2004
Zolertia Z1	92	2013
Atmel RZRaven	128	2007
TI CC2650	128	2015
NXP MKW40Z	160	2015
SAMR21 XPro	256	2014
Nordic NRF52840 DK	512	2018
Arduino Due	512	2012

To support the highly constrained applications and devices for which embedded OSes are used, embedded OSes must be minimal. However, the manner in which they must be minimal varies – some applications require minimal use of radio energy, which can require code size consuming techniques like compression, while others require tiny/low cost MCUs without space for those mechanisms. To support this variety, OSes have compile-time flags to include or exclude parts of the system or networking stack [6]–[8]. Some systems take a more extreme approach, dynamically generating the minimum code to compile from the application itself [4]. Part of this minimalist focus is that until application developers demand certain features, OSes are likely to leave them out entirely (a requirement often specified in contribution guides [7]). These techniques are critical to ensuring a given OS can support a wide range of embedded platforms, and influence the implementation of network protocols.

III. Low-Power IP Today

> *"The Working Group will generate the necessary documents to ensure interoperable implementations of 6LoWPAN networks"*
>
> — 6LoWPAN Working Group Charter [3]

This section gives an overview of 6LoWPAN and its implementations. It finds that each implementation today includes a different subset of the protocol. Source code shows this is due to concerns with code size. Experiences with a new implementation verify these concerns.

A. 6LoWPAN Summary

6LoWPAN [15] [1] [16] defines new header types to compress and structure IPv6 packets over low-power link layers. Because link layers have different communication models and address formats, each link layer has its own specification [1], [2], [17]–[19]. In this paper we focus on 802.15.4, since it was the original driver for 6LoWPAN and its dominant use case.

A standard IPv6 header is 40 bytes, and IPv6 requires that a link layer support 1280 byte packets. Low-power devices, however, have small MTUs (Bluetooth Low Energy, for example is 27 bytes) and often send small data payloads (e.g., 10 bytes). 6LoWPAN therefore provides mechanisms to both fragment and heavily compress IPv6 packets. Because addresses dominate the header, there are context-based and context-free compression schemes for unicast and multicast IPv6 addresses, as well as cases in which other fields within

Table II: 6LoWPAN Interoperability Matrix. Each implementation opts for a feature set that works best for its particular resource requirements.

Feature	Stack					
	Contiki	**Contiki-NG**	**OpenThread**	**Riot**	**Arm Mbed**	**TinyOS**
Uncompressed IPv6	✓	✓		✓	✓	✓
6LoWPAN Fragmentation	✓	✓	✓	✓	✓	✓
1280 byte packets	✓	✓	✓	✓	✓	✓
Dispatch_IPHC header prefix	✓	✓	✓	✓	✓	✓
IPv6 Stateless Address Compression	✓	✓	✓	✓	✓	✓
Stateless multicast address compression	✓	✓	✓	✓	✓	✓
802.15.4 16 bit short address support		✓	✓	✓	✓	✓
IPv6 Address Autoconfiguration	✓	✓	✓	✓	✓	✓
IPv6 Stateful Address Compression	✓	✓	✓	✓	✓	✓
IPv6 Stateful multicast address compression			✓	✓	✓	
IPv6 Traffic Class and Flow label compression	✓	✓	✓	✓	✓	✓
IPv6 NH Compression: Ipv6 (tunneled)			✓		✓	✓
IPv6 NH Compression: UDP	✓	✓	✓	✓	✓	✓
UDP port compression	✓	✓	✓	✓	✓	✓
UDP checksum elision						✓
Compression + headers past first fragment				✓	✓	
Compression of IPv6 Extension Headers		~	~[2]		✓	✓
Mesh Header			✓		✓	~[3]
Broadcast Header						✓
Regular IPv6 ND	✓	✓		✓	✓	~[4]
RFC 6775 6LoWPAN ND				✓	✓	
RFC 7400 Generic Header Compression						

~ = Partial Support

IP and UDP headers may be compressed. In the most extreme case, 6LoWPAN can compress the 40 byte header to 2 bytes; for a 10 byte packet, over Bluetooth Low Energy, this is the difference between a 400% overhead and sub-IP fragmentation or a 20% overhead and no fragmentation.

B. Feature Fail

A 6LoWPAN receiver has much less flexibility than a sender: it must be able to process any valid compression it receives. Table II shows the receiver features supported by 6 major open-source 6LoWPAN stacks. Some, such as TinyOS, are mostly developed and used in academia. Others, such as ARM Mbed and Nest's OpenThread, are developed and supported commercially. Contiki and Contiki-NG sit somewhere in the middle, having both significant academic and commercial use. Riot is an open-source OS with hundreds of contributors for industrial, academic, and hobbyist use. Two widely used open source stacks excluded from this analysis are LiteOS [9] and Zephyr [10] — they are excluded because LiteOS uses the same 6LoWPAN library (LWIP) used in MBED-OS, and Zephyr simply imports OpenThread.

In almost all cases, each stack's support for features is symmetric for sending and receiving. There are significant mismatches in feature support between stacks. These mismatches lead to deterministic cases when IP communication fails. We verified these failures by modifying existing network applications and testing them on hardware, using Wireshark to verify packets were compressed and formatted as we expected when receivers failed to decode packets. Table III lists the exact software/hardware combinations we used for our code analysis and hardware tests. Every implementation pair fails for some type of packet which can be organically generated by one of the stacks. This result may be surprising when compared to prior work which demonstrated successful interoperability, such as [20]. However, this early success preceded the release of RFC 6282, which increased the complexity (and overhead) of 6LoWPAN.

Table III: Hardware/Software combinations used

Stack	**Commit Hash**	**Device**
Contiki	bc2e445817aa546c	CC2650 LaunchXL
Contiki-NG	7b076e4af14b2259	CC2650 LaunchXL
OpenThread	4e92a737201b2001	Nordic NRF52840
Riot	3cce9e7bd292d264	SAM R21 X-Pro
Arm Mbed	4e92a737201b2001	N/A
TinyOS	4d347c10e9006a92	Atmel SAM4L

In addition to the feature mismatches in Table II, IP communication between different stacks can fail due to disagreement on buffer sizes. This happens when stacks make assumptions about the maximum decompression which is possible from a single link layer frame. The maximum header decompression allowed by the 6LoWPAN specification is technically > 1200 bytes, as it allows packets filled recursively with compressed IPv6 packets. Implementations place much lower limits to avoid a requirement for multiple IPv6 size buffers which would mostly sit empty. Some stacks send packets with more header compression than the limit chosen by others, causing IP communication to fail. For example, Contiki's 38 byte limit is exceeded by any packet with a maximally compressed IP header and any UDP compression. Contiki merely seems to have chosen a 38 byte limit because the limited Contiki stack

[2]Contiki-NG and OpenThread do not support mobility header compression

[3]TinyOS can receive the mesh header, but can't forward or send mesh header packets, preventing TinyOS from participating in route-under networks

[4]TinyOS supports only some of RFC 4861

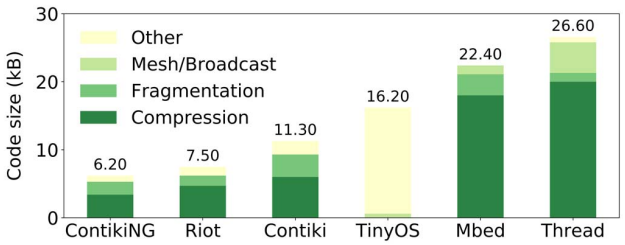

Figure 1. 6LoWPAN stack code size for 6 open source stacks. The code size varies by over a factor of 4, in part due to different feature sets. Compression dominated the code requirements of each stack. TinyOS's whole-program optimization model precluded separating out subcomponents.

will not compress frames by more than that amount.

C. Why?

IP communication can consistently fail in low-power networks, despite the presence of succinct standards (RFC 6282 + RFC 4944 is 52 pages) designed for low-power devices. Worse, this failure is silent: the receiver will simply drop the packet. Examining the documentation and implementation of each stack, code size concerns motivated feature elision. Mbed, Riot, and Contiki even provide compile-time flags to remove additional features for particularly constrained use cases.

Figure 1 shows a break down of the code size of each stack. Compression dominates the code size of 6LoWPAN implementations, and in several cases 6LoWPAN's size is comparable to the whole rest of the IPv6 stack. The Contiki and Contiki-NG implementations are significantly smaller than the others in part because they elide significant and complex features. The ARM Mbed IPv6 stack uses 45kB of flash. This is nearly 1/3 of the available space on a CC2650, just for IPv6: it does not include storage, sensors, the OS kernel, cryptography, higher layer protocols, signal processing, or applications.

Can a careful developer implement a leaner, fully-featured stack? To answer this question, we implemented our own 6LoWPAN stack. Our open-source implementation is written in Rust, for Tock, a secure embedded OS [21].

Our experiences support the comments and documentation of the other stacks. We surpassed the size of the Contiki-NG and Riot 6LoWPAN code before adding support for recursive compression of IPv6 or the mesh and broadcast headers. We noted several aspects of the protocol required surprisingly complex code to properly handle. For example, 6LoWPAN requires IPv6 tunneled inside a compressed packet to compress interior headers as well. This requires the decompression library to support recursive invocation, which increases minimum execution stack sizes and makes tracking buffer offsets during decompression more difficult. Refusing to support tunneled IPv6 packets (e.g., Contiki) greatly simplified the code. Another example: headers in the first 6LoWPAN sub-IP fragment must be compressed, while headers in subsequent fragments must not be compressed. Given that low-power link layers have variable length headers, correctly determining exactly where to fragment and what should be compressed requires complex interactions between layers of the stack. Finally, 6LoWPAN requires support for out-of-order reception of fragments, potentially from different packets. This forced our receiver to store and track state for a collection of received packets, preventing reliance on a single global receive buffer. The exercise of implementing a 6LoWPAN stack from the ground up affirmed that code size concerns encourage feature elision.

D. Why Does This Matter?

For any 6LoWPAN implementation, there exists a border router implementation that can connect it to the broader Internet. But this status-quo model of connectivity forces vertical integration and fails to meet the original design goals of 6LoWPAN, for two reasons.

First, a 6LoWPAN gateway can not know how to safely compress packets for different nodes, unless it communicates only with devices produced by the same vendor. As a result, for a coverage area containing devices produced by 5 different vendors, 5 gateways are required. If not for feature mismatches, a single gateway would suffice. Second, the current situation significantly limits the potential for range extension via mesh topologies. Most existing 6LoWPAN meshes rely on "route-over" mesh routing at the network layer, which requires that each node can at least partially decompress and recompress IP headers when forwarding. Mesh-under routing is no better, as implementation of the mesh header is not universal (see table II). Poor interoperability worsens the usability, range, cost, and efficiency of these networks.

IV. Traditional Principles: Not Low-Power

Over the past 45 years, the Internet community has coalesced on a small number of design principles. Connectivity through interoperability is a key premise of the Internet. Principles such as layering and encapsulation support composing protocols in new ways (e.g., tunneling), while the end-to-end principle [22] allows building a robust network out of an enormous and complex collection of unreliable parts. The robustness principle asserts that implementations should make no assumptions on packets they receive: bugs, transmission errors, and even memory corruption can cause a device to receive arbitrarily formatted packets. It also asserts that an implementation must be ready to receive any and all properly formatted packets. This aspect of the principle is often attributed to John Postel as Postel's Law, first written down in the initial specification of IPv4: "In general, an implementation should be conservative in its sending behavior, and liberal in its receiving behavior."

Protocols often have optional features ("MAY, SHOULD, and OPTIONAL" in RFC language). Implicitly, due to Postel's Law, a receiver needs to handle either case. This scenario creates an asymmetry, where sender code can have reduced complexity but receiver code must be large and complex.

We need to think about low-power protocols differently. They need new principles to help guide their design. These

principles need to embrace that there is no "one size fits all" design, while defining how devices choosing different design points interoperate. Flexibility needs to exist not only for senders, but also receivers, without harming interoperability.

V. Three Principles

This section describes three protocol design principles which prevent failures in low-power protocols. These principles are absolutely necessary to ensure interoperable implementations in this space, and should be closely observed.

A. Principle 1: Capability Spectrum

A low power protocol should be deployable/installable on devices which are at the low end of code and RAM resources. Rather than require every device pay the potential energy costs of fewer optimizations, a protocol should support a linear spectrum of device capabilities.

This may seem familiar — the IP [23] and TCP [24] specifications provide optional fields which can be used by endpoints at their leisure; many non-standard HTTP headers will be ignored unless both client and server support them; TLS ciphersuite support is often asymmetrical. But this principle is different. For those examples, no linear spectrum exists — support for any particular capability is generally unrelated to support for any other. Checking for support of any feature requires explicit enumeration of each, making it impossible to effectively compress such options. A non-linear spectrum requires storing feature support for every neighbor in RAM, or re-discovering capabilities on every exchange.

Low power protocols require simpler capability management. A low power protocol should define a capability spectrum with a clear ordering in which especially resource constrained devices can reduce code size or RAM use by eliding features. Such a spectrum makes a protocol usable by extremely low resource devices without forcing more resourceful devices to communicate inefficiently.

This capability spectrum should be a linear scale. For a device to support capability level N, it must also support all lower capability levels. More complex configuration approaches (e.g., a set of independent options) might allow for a particular implementation to be more efficient, picking the features that give the most benefit for the least added complexity. However, this sort of optimization makes interoperability more difficult, as two devices must negotiate each specific feature to use.

B. Principle 2: Capability Discovery

The second principle follows from the first: if different capability levels exist, there should be a mechanism for two devices to determine what level to communicate with.

The capability negotiation we propose here differs from capability discovery mechanisms built for traditional systems, such as IP Path MTU discovery or the Link Layer Discovery Protocol (LLDP). IP Path MTU discovery relies on continual probing until an acceptable value is discovered. LLDP requires regular, detailed capability advertisements at a fixed interval. The energy overhead of network probing or advertising is unacceptable in most low power environments. *Capability discovery in low power networks should require no more than one failure between any two neighbors*, even if this slightly increases the overhead per error. Proactive capability discovery should be built into baseline communication required for tasks like neighbor discovery or route maintenance. Further, assumptions for traditional systems that prohibit storing per-endpoint state do not apply, as nodes store information about link-layer neighbors, *not IP endpoints*. This is needed because low-power networks with route over topologies frequently involve decompression and re-compression at each hop to enable forwarding. Low power nodes have few neighbors, so storing a few bits of state for each is feasible and can significantly reduce the amount of radio energy needed for communication. The code size cost of storing state is small compared to the cost of complex compression mechanisms.

In a low power network with capability discovery, if two devices wish to communicate, they default to the lower of their supported capability levels. E.g. a level 2 and a level 4 device should communicate at level 2. One offshoot of this principle is that it requires implementations have symmetric capabilities for send and receive – no benefits can be realized from an asymmetric implementation.

C. Principle 3: Explicit and Finite Bounds

Protocols must specify explicit and reasonable bounds on recursive or variable features so implementations can bound RAM use. This allow implementations to safely limit their RAM use without silent interoperability failures. This also ensures that capability discovery is sufficient for interoperability.

The idea of imposing bounds is, on its own, not unique to this space. TCP enforces a finite limit of 40 bytes for TCP options which may be appended to the TCP header, as does IPv4. DHCP allows for the communication of maximum DHCP message sizes. In the space of low power Internet protocols, however, this idea *must be pervasive.* Notably, the original designers of a specification may not know exactly what these values should be. This is not a new problem: TCP congestion control, for example, had to specify initial congestion window values. In this space, bounds should initially be very conservative. Over time, if increasing resources or knowledge suggests they should grow, then future devices will have the onus of using fewer resources to interoperate with earlier ones. The capability spectrum defined in the previous two principles can be helpful in this regard.

VI. A Principled 6LoWPAN

This section proposes how to apply the three principles in the previous section to 6LoWPAN through specific modifications to the protocol. These modifications ensure that two 6LoWPAN devices can communicate even if they choose different code size/energy efficiency tradeoffs. We refer to this modified protocol as Principled 6LoWPAN (P6LoWPAN).

Table IV: Capability Spectrum

Capability	Basic Description / Added Features
Level 0	Uncompressed IPv6 + ability to send ICMP errors • Uncompressed IPv6 • 6LoWPAN Fragmentation (Fragment Header) • 1280 Byte Packets • Stateless decompression of source addresses
Level 1	IPv6 Compression Basics + Stateless Addr Compression • Support for the Dispatch_IPHC Header Prefix • Correctly handle elision of IPv6 length and version • Stateless compression of all unicast addresses • Stateless compression of multicast addresses • Compression + 16 bit link-layer addresses • IPv6 address autoconfiguration
Level 2	Stateful IPv6 Address Compression • Stateful compression of unicast addresses • Stateful compression of multicast addresses
Level 3	IPv6 Traffic Class and Flow Label Compression • Traffic Class compression • Flow Label Compression • Hop Limit Compression
Level 4	Next Header Compression + UDP Port Compression • Handle Tunneled IPv6 correctly • Handle the compression of the UDP Next Header • Correctly handle elision of the UDP length field • Correctly handle the compression of UDP ports • Handle headers past the first fragment, when first fragment compressed.
Level 5 (all routers)	Entire Specification • Support the broadcast header and the mesh header • Support compression of all IPv6 Extension headers

This application of our principles is not intended as a suggestion that these changes be made immediately to 6LoWPAN, as modifying an established protocol is a complex task very different from constructing new protocols. Instead, this application is a tool for evaluating these principles, and an example for how they should be applied.

A. Principle 1: Capability Spectrum

We propose replacing the large collection of "MUST" requirements — the features in Table II—into 6 levels of functionality. These "Capability Levels" are depicted in Table IV.

These levels prioritize features which provide the greatest energy savings per byte of added code size, based off our code size measurements and the number of bits saved by each additional compression mechanism. They allow for a wide range of code size/efficiency tradeoffs.

For example, addresses dominate an uncompressed IPv6 header. Level 0 devices only support compressed source addresses, while level 1 devices support all stateless address compression. In one early design of this spectrum, Level 0 supported only uncompressed packets. However, this raises a problem with ICMP error generation. If a node cannot decompress the source address of a received packet, it cannot send ICMP errors. ICMP errors are required for capability discovery. Stateful compression depends on an out-of-band signal to set up state, such that nodes only send statefully compressed packets to nodes who also support it. Therefore decompressing stateless source addresses is a minumum requirement.

The classes in this scale do not precisely reflect the current feature support of the implementations described in Section III. For example, Contiki supports UDP port compression (level 4) but does not support 802.15.4 short addresses (level 2) or stateful multicast compression (level 3): following this formulation, Contiki only provides level 1 support. If Contiki supported 16-bit addresses, it would provide level 2 support. A concrete spectrum such as the one above gives stack designers a structure and set of guidelines on the order in which to implement features. Based on our experiences developing a 6LoWPAN stack, we believe that if this scale existed as part of the initial specification, implementations would have made an effort to adhere to it.

One additional advantage of this spectrum is that it allows for some future additions to the P6LoWPAN specification without breaking interoperability between new and old implementations. For example, our scale does not include support for Generic Header Compression [25] because none of the open-source stacks we analyzed implement it. Despite this, support for this RFC could easily be added as a new class on this linear scale (as Class 6), and devices supporting it would know to not use it when communicating with lower class implementations.

This spectrum requires that a node store 3 bits of state for each neighbor. Given that low-power nodes often store ten or more bytes for each entry in their link table (link quality estimates, addresses, etc.), this cost is small. 6LoWPAN already assumes that routers are more resourceful devices, P6LoWPAN routers are required to be level 5.

B. Principle 2: Capability Discovery

We propose two mechanisms by which P6LoWPAN performs capability discovery: neighbor discovery (ND) and ICMP. Neighbor discovery [26] is analogous to ARP in IPv4: it allows IPv6 devices to discover the link layer addresses of neighboring addresses as well as local gateways. Devices use neighbor discovery to proactively discover capability levels and ICMP to detect when incompatible features are used. Of the two, only ICMP is required. Neighbor discovery simply allows a pair of differing nodes to avoid an initial ICMP error, and allows for optimization of host-router communication during neighbor discovery.

ICMP: We propose adding a new ICMPv6 message type—P6LoWPAN Class Unsupported—which a device sends in response to receiving 6LoWPAN features it does not understand. This error encodes the device's capability level. A node receiving such an error updates its link table entry with the capability level. In the future, any packets sent to that address use at most the supported level.

Neighbor discovery: We propose adding an IPv6 ND option that allows a device to communicate its capability class during network association. This option would be included in Router

Solicitations and Neighbor Advertisements, and would allow all devices that obtain link-layer addresses via ND to also know how to send packets which that neighbor can receive. When a node uses ND to resolve an IP address to a link layer address, it learns the supported capability level as well as the link layer address. This option minimizes the energy cost of communicating capabilities. It is worth noting that RFC 7400 already employs a similar method for communicating whether devices implement General Header Compression: adding such an option is clearly viable. [25]

C. Principle 3: Provide Reasonable Bounds

Section III discussed two missing bounds which affect 6LoWPAN interoperability: limits on header decompression and bounds on recursion when decompressing tunneled IPv6.

For P6LoWPAN, we propose that header decompression be bounded to 51 bytes. This bound allows for significant RAM savings in implementations that decompress first fragments into the same buffer in which the fragment was originally held. 51 bytes is a good tradeoff between RAM savings and how frequently we expect such a bound would force packets to be sent uncompressed. A 51 byte limit allows for transmission of a packet containing a maximally compressed IP header (+38 bytes), a maximally compressed UDP header (+6 bytes), and one maximally compressed IPv6 extension header (+7 bytes). This allows saving hundreds of bytes of RAM, without jeopardizing interoperability. Packets requiring more decompression than this are extremely rare, and could be sent uncompressed. How rare? It is only possible to surpass this limit if tunneled IPv6 is used or multiple IPv6 extension headers are present. As of 2014, a real-world study of IPv6 extension header use found that 99% of packets with *multiple* extension headers were dropped in the real Internet [27].

Second, we propose that headers for tunneled IPv6 should not be compressed. The primary motivation for this feature was from the RPL protocol [28], as discussed in Section III-B. However, the fact that RPL must tunnel IPv6 in this way is generally agreed to be a problem and a wart in its design that should be avoided when possible [29]. This change allows implementations to avoid recursive functions to decompress these headers, and instead use simple if/else statements.

VII. Evaluation

This section evaluates the costs of applying our principles to 6LoWPAN. The principles are written such that interoperability comes by construction, and thus interoperability of the modified protocol cannot be directly evaluated without observing implementations written by different stakeholders. But indirect evaluation is possible. Can a reasonable set of capability levels provide a good range of implementation complexity from which a developer can choose? Is the overhead of the proposed mechanisms low enough to make them viable? Are the savings afforded by a linear capability spectrum worth the associated limitations? We find the incremental costs of capability discovery mechanisms is small, adding 172-388 bytes of code in the worst case. We find that the capability spectrum allows meaningful savings in code size and memory usage. Finally, we find capability discovery has a low run-time performance cost when a linear spectrum is used.

A. Implementations

First, we implemented the proposed P6LoWPAN on the Contiki-NG 6LoWPAN stack, modifying it such that a compile-time option determines which features of 6LoWPAN are compiled. We selected Contiki-NG because it has the smallest 6LoWPAN stack of those tested, so any overheads the mechanisms introduce would be most pronounced. Our changes required modifying 500 lines of code relative to the head of the 4.2 release of Contiki-NG. We did not add additional 6LoWPAN features which were absent from the original Contiki-NG 6LoWPAN stack. Our code size numbers therefore represent a conservative lower bound of the total possible savings. All code sizes provided in this section are compiled with the Texas Instruments CC2650 as the target.

We also added ICMP and ND support for capability discovery. The updated stack responds to incompatible 6LoWPAN messages with an ICMP error, and communicates its capability level in Router Solicitation messages using the 6CIO prefix originally defined in RFC 7400 [25]. It stores the capability class of each node in its link table, and compresses IPv6 packets by the maximum amount supported by the destination.

Finally, we implemented a second modified 6LoWPAN stack in Contiki-NG, which does not follow the recommendation of using a linear capability spectrum. In this modified implementation, each node can select any of the 6LoWPAN features it chooses. We refer to this implementation as FLEX-6LoWPAN. For this alternative policy, we isolated 26 features of 6LoWPAN as single bit flags in a 32 bit bitfield. Thus, FLEX-6LoWPAN stores and communicates capabilities using 4 byte objects. FLEX-6LOWPAN also supports the added granularity required to maximally compress outgoing messages intended for a device supporting any specific combination of features. We did not add back in any 6LoWPAN features which the Contiki-NG stack did not originally support. This second implementation required modifying about 300 additional lines of code from the P6LoWPAN implementation.

B. Compile-Time Costs

Table V shows the size of the original Contiki-NG 6LoWPAN stack compiled at each possible capability level. Each capability level adds between 0.25 and 1.05 kB of code, and the spectrum enables implementations to cut the size of the 6LoWPAN stack by up to 45%. The code size cost of adding capability discovery, using the P6LoWPAN implementation with the linear capability spectrum, is shown in table VI. Capability discovery adds 178-388 bytes, a fraction of the size which implementations can save by supporting lower capability levels. The code added for communication varies across capability levels because the number of code paths for ICMP error generation and compression changes.

Table VII presents the compile-time costs of using an arbitrary bitfield instead of a linear capability spectrum by

Table V: 6LoWPAN code size of different capabilities levels in Contiki-NG. The spectrum spans a nearly 100% increase in code size.

Capability	Code Size (kB)	Increase (kB)
Level 0	3.2	-
Level 1	4.2	1.0
Level 2	4.8	0.6
Level 3	5.1	0.3
Level 4	5.6	0.5
Level 5	6.2	0.6

Table VI: The cost of implementing capability discovery in Contiki-NG is on average less than 5% of the total 6LoWPAN size; the maximum size reduction from choosing a lower capability level is 10x the discovery cost.

Capability	6LoWPAN Code Size (kB)		
	Base	w/Discovery	Increase
Level 0	3.2	3.4	188 bytes
Level 1	4.2	4.4	260 bytes
Level 2	4.8	5.2	388 bytes
Level 3	5.1	5.4	340 bytes
Level 4	5.6	5.9	296 bytes
Level 5	6.2	6.3	172 bytes

comparing our P6LoWPAN implementation with our FLEX-6LoPWAN implementation. The bitfield approach requires 32 bits per neighbor to store capabilities, instead of 3 bits. More importantly, it complicates determining the allowable compression between two nodes, as demonstrated by the code size increase. The important takeaway here is that opting for a less restrictive set of feature combinations mitigates much of the savings provided by implementing capabilities. For example, a FLEX-6LoWPAN device with the equivalent of level 4 capabilities requires more code space than a level 5 P6LoWPAN device – the linear capability spectrum makes a difference. The code size addition for FLEX-6LoWPAN is a conservative lower bound, as we did not need to add checks for handling 6LoWPAN compression features that Contiki-NG does not support.

C. Run-time Performance

1) ND Cost: 6LoWPAN ND communication (RFC 6775) is host-initiated and flows through routers (which must be level 5), and nodes store neighbor capability levels alongside link layer addresses: thus there is no possibility of communication failures due to capability mismatches. Therefore the cost of capability discovery in networks that use IPv6 ND is exclusively that certain ND messages become longer (router solicitations and neighbor advertisements are sent with an added capability option). To put this added cost in perspective, Equation 1 shows the total link-layer payload bytes sent/received for ND by a node in its initial wake-up period[5]. All variables not affected by the use of capability discovery are assigned the minimum possible value for the scenario discussed, so that the overhead of capability discovery represents a worst case.

[5]This equation assumes the configuration described in RFC 6775 as the "Basic Router Solicitation Exchange" – route over topology, 1 6LoWPAN context, 1 on-link prefix, and the host requires address registration.

Table VII: Resource requirements for a 6LoWPAN stack in Contiki-NG using a linear capability spectrum vs. using an arbitrary capability bitfield.

–	Linear Spectrum	Arbitrary Bitfield
6LoWPAN Code Size	5.9 kB	6.5 kB
RAM per neighbor	19 Bytes	22 Bytes

Table VIII: Total ND cost per eq. 1 for each implementation

	RS	NA	C (Total ND Cost)
6LoWPAN	20	24	$168 + 52 * N$
P6LoWPAN	24	28	$172 + 56 * N$
FLEX-6LoWPAN	28	32	$176 + 60 * N$

$$
\begin{aligned}
C = {} & \text{Router Solicitation } \{RS\} + \text{Min. IP Hdr } \{2\} + \\
& \text{Router Advertisement } \{104\} + \text{Min. IP Hdr } \{2\} + \\
& (\text{Neighbor Solicitation } \{24\} + \text{Min. IP Hdr } \{2\}) * N + \\
& (\text{Neighbor Advertisement } \{NA\} + \text{Min. IP Hdr } \{2\}) * N \\
& \text{Address Registration Options in first NS } \{24\} + \\
& + \text{Address Registration Options in first NA } \{16\} \qquad (1)
\end{aligned}
$$

where:

$C =$ Total link-layer payload sent/received for ND

$N =$ # of endpoints requiring address resolution

Table VIII shows the values of RS and NA for each 6LoWPAN implementation, and the resulting total ND cost. Notably, use of an arbitrary bitfield increases the size of the capability option by 4 bytes, making use of existing ND options like the 6CIO option impossible. In both cases the additional bytes added for capability discovery are small compared to the total cost of ND($\leq 8\%$ linear spectrum / $\leq 16\%$ arbitrary bitfield).

2) ICMP Cost: In networks that do not use IPv6 ND the cost of capability discovery is the energy/latency required for one ICMP packet per failure between any two nodes. For P6LoWPAN capability based failures can only happen in one direction, so the size of this link-layer payload is:

$C_{icmp} =$ Compressed IP Header Size $+ 4$

For FLEX-6LoWPAN $C_{icmp} = 48$, because the recipient does not know the capabilities of the sender, and thus must send an uncompressed packet to ensure successful reception of its own capabilities. This example reveals why use of an arbitrary bitfield is so undesirable – the ability to compress headers in ICMP errors can reduce overhead by a factor of 4 or more (in the common case of 8 byte compressed headers).

VIII. Discussion and Conclusion

A new generation of low-power devices face a connectivity dilemma: Internet protocols are not designed for energy efficiency, but compression and other energy saving adaptations takes up precious code space. Device deployments specialized for single-vendor local networks make trade-offs specific to their application requirements. As a result, IP communication

between IP enabled devices fails. This problem is not specific to 6LoWPAN — Iova et. al. recently noted similar issues in the RPL protocol: "RPL has too large of a footprint for resource-constrained devices, and requires all devices in a network to run the same mode of operation, limiting heterogeneity" [30].

Part of the challenge is that some traditional protocol design principles do not apply well to the low-power setting. We present three design principles for low-power protocols that attempt to remedy this. These principles explicitly acknowledge the unique code space/energy tradeoffs of low-power devices.

Looking forward, considering this tension is critical for protocol designers in this ecosystem of diverse hardware capabilities and application tradeoffs. 6LoWPAN is not the only low power Internet protocol — the low power space uses its own routing protocols, address discovery protocols, and application layer protocols [28], [31]. Additional protocols will follow as the space matures. Many of these protocols will be initially developed outside the IETF — Jonathan Hui was a graduate student when he presented the first complete IPv6-based network architecture for sensor nets [32], as was Adam Dunkels when he created Contiki. We present a roadmap for how these principles can reframe the discussion of how to connect the next hundred billion devices to the Internet.

ACKNOWLEDGMENTS

We thank the anonymous IEEE DCOSS reviewers for their helpful reviews, as well as the IETF 6lo working group, which provided early feedback on this work. We also acknowledge the support of the Intel/NSF CPS Security grant No. 1505728 (End-to-End Security for the Internet of Things), NSF CPS grant No. 1931750 (Secure Smart Machining), the Stanford Secure Internet of Things Project, and the Stanford System X Alliance. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this mate- rial are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of the NSF or the U.S. Government.

REFERENCES

[1] J. Hui and P. Thubert, "Compression format for ipv6 datagrams over ieee 802.15.4-based networks," Internet Requests for Comments, RFC Editor, RFC 6282, September 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6282.txt

[2] J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, Z. Shelby, and C. Gomez, "Ipv6 over bluetooth(r) low energy," Internet Requests for Comments, RFC Editor, RFC 7668, October 2015.

[3] T. Lemon, "Ipv6 over low power wpan wg charter," IETF WG Charter, IETF, Tech. Rep., March 2005. [Online]. Available: https://datatracker.ietf.org/doc/charter-ietf-6lowpan/

[4] T. Alliance, "Tinyos," https://github.com/tinyos/tinyos-main, 2018.

[5] F. Berlin, "Riot os," https://github.com/RIOT-OS/RIOT, 2018.

[6] ARM, "Arm mbed os," https://github.com/ARMmbed/mbed-os, 2018.

[7] S. Duquennoy, "Contiki-ng," https://github.com/contiki-ng, 2018.

[8] A. Dunkels, "Contiki os," https://github.com/contiki-os/contiki, 2018.

[9] Huawei, 2019. [Online]. Available: https://github.com/LiteOS/LiteOS

[10] Z. Project, 2019. [Online]. Available: https://github.com/zephyrproject-rtos/zephyr

[11] Z. Alliance, "Zigbee 3.0," https://www.zigbee.org/, 2018.

[12] Nest, "Openthread," https://github.com/openthread/openthread, 2018.

[13] R. Braden, "Requirements for internet hosts - communication layers," Internet Requests for Comments, RFC Editor, STD 3, October 1989. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1122.txt

[14] X. Huang, "Techical interoperability 6lowpan-coap report from interop event," http://www.probe-it.eu/wp-content/uploads/2013/09/Technical-Interoperability-6LoWPAN-CoAP-Report.pdf, 2013, accessed: 2018-02-01.

[15] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of ipv6 packets over ieee 802.15.4 networks," Internet Requests for Comments, RFC Editor, RFC 4944, September 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4944.txt

[16] Z. Shelby, S. Chakrabarti, E. Nordmark, and C. Bormann, "Neighbor discovery optimization for ipv6 over low-power wireless personal area networks (6lowpans)," Internet Requests for Comments, RFC Editor, RFC 6775, November 2012.

[17] A. Brandt and J. Buron, "Transmission of ipv6 packets over itu-t g.9959 networks," Internet Requests for Comments, RFC Editor, RFC 7428, February 2015.

[18] P. Mariager, J. Petersen, Z. Shelby, M. V. de Logt, and D. Barthel, "Transmission of ipv6 packets over digital enhanced cordless telecommunications (dect) ultra low energy (ule)," Internet Requests for Comments, RFC Editor, RFC 8105, May 2017.

[19] K. Lynn, J. Martocci, C. Neilson, and S. Donaldson, "Transmission of ipv6 over master-slave/token-passing (ms/tp) networks," Internet Requests for Comments, RFC Editor, RFC 8163, May 2017.

[20] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-Haggerty, A. Terzis, A. Dunkels, and D. Culler, "Contikirpl and tinyrpl: Happy together," in *Workshop on Extending the Internet to Low Power and Lossy Networks (IP+ SN)*, vol. 570, 2011.

[21] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 234–251. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132786

[22] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357402

[23] J. Postel, "Internet protocol," Internet Requests for Comments, RFC Editor, STD 5, September 1981. [Online]. Available: http://www.rfc-editor.org/rfc/rfc791.txt

[24] ——, "Transmission control protocol," Internet Requests for Comments, RFC Editor, STD 7, September 1981. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[25] C. Bormann, "6lowpan-ghc: Generic header compression for ipv6 over low-power wireless personal area networks (6lowpans)," Internet Requests for Comments, RFC Editor, RFC 7400, November 2014.

[26] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, "Neighbor discovery for ip version 6 (ipv6)," Internet Requests for Comments, RFC Editor, RFC 4861, September 2007. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4861.txt

[27] J. Linkova and F. Gont, "Ipv6 extension headers in the real world," Jul 2014. [Online]. Available: http://www.iepg.org/2014-07-20-ietf90/iepg-ietf90-ipv6-ehs-in-the-real-world-v2.0.pdf

[28] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "Rpl: Ipv6 routing protocol for low-power and lossy networks," Internet Requests for Comments, RFC Editor, RFC 6550, March 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6550.txt

[29] J. Hui, "Ip-in-ip (why, when, and overhead)," https://mailarchive.ietf.org/arch/browse/roll, July 2010, accessed: 2018-04-08.

[30] O. Iova, P. Picco, T. Istomin, and C. Kiraly, "Rpl: The routing standard for the internet of things... or is it?" *IEEE Communications Magazine*, vol. 54, no. 12, pp. 16–22, 2016.

[31] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," Internet Requests for Comments, RFC Editor, RFC 7252, June 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7252.txt

[32] J. W. Hui and D. E. Culler, "Ip is dead, long live ip for wireless sensor networks," in *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460415