LWIG Working Group
Internet-Draft
Intended status: Informational
Expires: July 27, 2012

C. Bormann, Ed.
Universitaet Bremen TZI

January 24, 2012

Guidance for Light-Weight Implementations of the Internet Protocol Suite

draft-bormann-lwig-guidance-01

Abstract

Implementation of Internet protocols on small devices benefits from light-weight implementation techniques, which are often not documented in an accessible way.

This document provides a first outline of and some initial content for the Light-Weight Implementation Guidance document planned by the IETF working group LWIG.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 27, 2012.

Copyright Notice

Table of Contents

1. Introduction

Today's Internet is experienced by users as a set of applications, such as email, instant messaging, and social networks. There are substantial differences in performance between the various end devices with these applications, but in general end devices participating in the Internet today are considered to have relatively high performance.

More and more communications technology is being embedded into our environment. Different types of devices in our buildings, vehicles, equipment and other objects have a need to communicate. It is expected that most of these devices will employ the Internet Protocol suite. The term "Internet of Things" denotes a trend where a large number of devices directly benefit from communication services that use Internet protocols. Many of these devices are not primarily computing devices operated by humans, but exist as components in buildings, vehicles, and the environment. There will be a lot of variation in the computing power, available memory, communications bandwidth, and other capabilities between different types of these devices. With many low-cost, low-power and otherwise constrained devices, it is not always easy to embed all the necessary features.

Historically, there has been a trend to invent special "light-weight" _protocols_ to connect the most constrained devices. However, much of this development can simply run on existing Internet protocols, provided some attention is given to achieving light-weight _implementations_. In some cases the new, constrained environments can indeed benefit from protocol optimizations and additional protocols that help optimize Internet communications and lower the computational requirements. Examples of IETF standardization efforts targeted for these environments include the "IPv6 over Low power WPAN (6LoWPAN)", "Routing Over Low power and Lossy networks (ROLL)", and "Constrained RESTful Environments (CoRE)" working groups. More generally, however, techniques are required to implement both these optimized protocols as well as the other protocols of the Internet protocol suite in a way that makes them applicable to a wider range of devices.

1.1. Objectives

The present document, a product of the IETF Light-Weight Implementation Guidance (LWIG) Working Group, focuses on helping the implementers of the smallest devices. The goal is to be able to build minimal yet interoperable IP-capable devices for the most constrained environments.

Building a small implementation does not have to be hard. Many small

devices use stripped down versions of general purpose operating systems and their TCP/IP stacks. However, there are implementations that go even further in minimization and can exist in as few as a couple of kilobytes of code, as on some devices this level of optimization is necessary. Technical and cost considerations may limit the computing power, battery capacity, available memory, or communications bandwidth that can be provided. To overcome these limitations the implementers have to employ the right hardware and software mechanisms. For instance, certain types of memory management or even fixed memory allocation may be required. It is also useful to understand what is necessary from the point of view of the communications protocols and the application employing them. For instance, a device that only acts as a client or only requires one connection can simplify its TCP implementation considerably.

The purpose of this document is to collect experiences from implementers of IP stacks in constrained devices. The focus is on techniques that have been used in actual implementations and do not impact interoperability with other devices. The techniques shall also not affect conformance to the relevant specifications. We describe implementation techniques for reducing complexity, memory footprint, or power usage.

The topics for this working group will be chosen from Internet protocols that are in wide use today, such as IPv4 and IPv6; UDP and TCP; ICMPv4/v6, MLD/IGMP and ND; DNS and DHCPv4/v6; TLS, DTLS and IPsec; as well as from the optimized protocols that result from the work of the 6LoWPAN, RPL, and CoRE working groups. This document will be helpful for the implementers of new devices or for the implementers of new general-purpose small IP stacks. It is also expected that the document will increase our knowledge of what existing small implementations do, and will help in the further optimization of the existing implementations. In areas where the considerations for small implementations have already been documented in an accessible way, we will refer to those documents instead of duplicating the material here.

Generic hardware design advice and software implementation techniques are outside the scope of this document. Protocol implementation experience, however, is the focus. There is no intention to describe any new protocols or protocol behavior modifications beyond what is already allowed by existing RFCs, because it is important to ensure that different types of devices can work together. For example, implementation techniques relating to security mechanisms are within scope, but mere removal of security functionality from a protocol is rarely an acceptable approach.

1.2. Call for contributions

The present draft of the document is an outline that will grow with the contributions received, which are expressly invited. As this document focuses on experience from existing implementations, this requires implementer input; in particular, participation is required from the implementers of existing small IP stacks. "Small" here is intended to be applicable approximately to what is described in Section 2 -- where it is more important that the technique described is grounded in actual experience than that the experience is actually from a (very) constrained system.

Only a few subsections are fleshed out in this initial draft; additional subsections will quickly be integrated from additional contributors.

1.3. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. As this is an informational document, the [RFC2119] keywords will only be used to underscore requirements where similar key words apply in the context of the specifications the light-weight implementation of which is being discussed.

The term "byte" is used in its now customary sense as a synonym for "octet".

2. Drawing the Landscape

There is not a single kind of constrained, Internet-connected device. To the contrary, the trend is towards much more functional variety of such devices than is customary today in the Internet. This section introduces a number of terms that will be used to locate some of the technique described in the following sections within certain areas of applications.

2.1. Classes of Devices

Despite the overwhelming variety of Internet-connected devices that can be envisioned, it may, be worthwhile to have some succinct terminology for different classes of constrained devices. In this document, the following class designations may be used as rough indications of device capabilities:

Name	data size (e.g., RAM)	code size (e.g., Flash)
Class 1	~ 10 KiB	~ 100 KiB
Class 2	~ 50 KiB	~ 250 KiB

As of the writing of this document, these characteristics correspond to distinguishable sets of commercially available chips and design cores for constrained devices. While it is expected that the boundaries of these classes will move over time, Moore's law tends to be less effective in the embedded space than in personal computing devices: Gains made available by increases in transistor count and density are more likely to be invested in reductions of cost and power requirements than into continual increases in computing power.

2.2. Design Objectives

- Consideration for design or implementation approaches for implementation of IP stacks for constrained devices will be impacted by the RAM usage for these designs. Here the consideration is what is the best approach to minimize overhead.

- In addition, the impact on throughput in terms of IP protocol implementation must take into consideration the methods that minimize overhead but balance performance requirements for the light-weight constrained devices.

- Protocol implementation must consider its impact on CPU utilization. Here guidance will be provided on how to minimize

tasks that require additional CPU execution time.

How does the implementation of the IP stack effect the application both in terms of performance but also of those same attributes and requirements (RAM, CPU usage, etc.) that we are examining for the IP protocol stack?

From performing a synthesis of implementation experiences we will be able to understand and document the benefits and consequences of varied approaches. Scaling code and selected approaches in terms of scaling from, say, a 8-bit micro to a 16-bit micro. Such scaling for the approach will aid in the development of single code base when possible.

2.3. Implementation Styles

Compared to personal computing devices, constrained devices tend to make use of quite different classes of operating systems, if that term is even applicable.

...

- Single-threaded/giant mainloop
- Event-driven vs. threaded/blocking
  - The usual multi-threaded model blocks a thread on primitives such as connect(), accept() or read() until an external event takes place. This model is often thought to consume too much RAM and CPU processing.
  - The event driven model uses a non-blocking approach: E.g., when an application interface sends a message, the routine would return immediately (before the message is sent). A call-back facility notifies the application or calling code when the desired processing is completed. Here the benefit is that no thread context needs to be preserved for long periods of time.
- Single/multiple processing elements
- E.g., separate radio/network processor

Introduce these briefly: Some techniques may be applicable only to some of these styles!

2.4. Roles of nodes

Constrained nodes are by necessity more specialized than general purpose computing devices; they may have a quite specific role. Some implementation techniques may also

- Constrained nodes
- Nodes talking to constrained nodes
- Gateways/Proxies

In all these cases, constrained nodes that are "sleepy" pose additional considerations. (Explain sleepy...) E.g., a node talking to a sleepy node may need to make special arrangements; this is even more true where a gateway or proxy interfaces the general Internet

- Bandwidth/latency considerations

2.5. Overview over the document

The following sections will first go through a number of specific protocol layers, starting from layers of the data plane (link adaptation, network, transport, application), followed by control plane protocol layers (link layer support, network layer and routing, host configuration and lookup services). We then look at security protocols (general cryptography considerations, transport layer security, network layer security, network access control). Finally, we discuss some specific, cross-layer concerns, some "wire-visible", some of concern within a specific implementation. Clearly, many topics could be discussed in more than one place in this structure. The objective is not to have something for each of the potential topics, but to document the most valuable experience that may be available.

3. Data Plane Protocols

3.1. Link Adaptation Layer

6LoWPAN

3.1.1. Fragmentation in a 6LoWPAN Route-Over Configuration

Author: Carsten Bormann

6LoWPAN [RFC4944] is an adaptation layer that maps IPv6 with its minimum MTU of 1280 bytes to IEEE 802.15.4, which has a physical layer MTU of only 127 bytes (some of which are taken by MAC layer and adaptation layer headers). Therefore, the adaptation layer provides a fragmentation and reassembly scheme that can fragment a single IPv6 packet of up to 1280 bytes into multiple adaptation layer fragments of up to 127 bytes each (including MAC and adaptation layer overhead).

In a route-over configuration, implementing this adaptation layer fragmentation scheme straightforwardly means that reassembly and then fragmentation are performed at each forwarding hop. As fragments from several packets may be arriving interleaved with each other, this approach requires buffer space for multiple MTU-size IPv6 packets.

In a mesh-under configuration, adaptation layer fragments can be forwarded independently of each other. It would be preferable if something similar were possible for route-over. Complete independence in forwarding of adaptation layer fragments is not possible for route-over, however, as the layer-3 addresses needed for forwarding are in the initial bytes of the IPv6 header, which is present only in the first fragment of a larger packet.

Instead of performing a full reassembly, implementations may be able to optimize this process by not keeping a full reassembly buffer, but just a runt buffer (called "virtual reassembly buffer" in [WEI]) for each IP packet. This buffer caches only the datagram_tag field (as usual combined with the sender's link layer address, the destination's link layer address and the datagram_size field) and the IPv6 header including the relevant addresses. Initial fragments are then forwarded independently (after header decompression/compression) and create a runt reassembly buffer. Non-initial fragments (which don't require header decompression/compression in 6LoWPAN) are matched against the runt buffers by datagram_tag etc. and forwarded if an IPv6 address is available. (This simple scheme may be complicated a bit if header decompression/compression of the initial fragment causes an overflow of the physical MTU; in this case some

overflow data may need to be stored in the runt buffers to be combined with further fragments or may simply be forwarded as a separate additional fragment.)

If non-initial fragments arrive out of order before the initial fragment, a route-over router may want to keep the contents of the non-initial fragments until the initial fragment is available, which does need some buffer space. If that is not available, a more constrained route-over router may simply discard out-of order non-initial fragments, possibly taking note that there is no point in forwarding any more fragments with the same combination of 6LoWPAN datagram_tag field, L2 addresses and datagram_size.

Runt buffers should time out like full reassembly buffers, and may either keep a map of fragments forwarded or they may simply be removed upon forwarding the final fragment, assuming that no out-of-order fragments will follow.

3.1.1.1. Implementation Considerations for Not-So-Constrained Nodes

[RFC4944] makes no explicit mandates about the order in which fragments should be sent. Because it is heavily favored by the above implementation techniques, it is highly advisable for all implementations to always send adaptation layer fragments in natural order, i.e., starting with the initial fragment, continuing with increasing datagram_offset.

3.2. Network Layer

IPv4 and IPv6

3.3. Transport Layer

TCP and UDP

Both TCP and UDP employ 16-bit one's-complement checksums to protect against transmission errors. A number of RFCs discuss efficient implementation techniques for computing and updating Internet Checksums [RFC1071] [RFC1141] [RFC1624]. (Updating the Internet Checksum, as opposed to computing it from scratch, may be of interest where a pre-computed packet is provided, e.g., in Flash ROM, and a copy is made in RAM and updated with some current values, or when the actual transmitted packet is composed from pre-defined parts in ROM and new parts in RAM.)

3.4. Application Layer

3.4.1. General considerations about Application Programming Interfaces (APIs)

Author: Carl Williams

Constrained devices are not necessarily in a position to use APIs that would be considered "standard" for less constrained environments (e.g., Berkeley sockets or those defined by POSIX).

When an API implements a protocol, this can be based on proxy methods for remote invocations that underneath rely on the communication protocol. One of the roles of the API can be exactly to hide the detail of the transport protocol.

Changes to the lower layers will be made to implement light-weight stacks so this impacts that implementation and inter-workings with the API. Similar considerations such as RAM, CPU utilization and performance requirements apply to the API and its use of the lower layer resources (i.e., buffers).

Considerations for the proper approach for a developer to request services from an application program need to be explored and documented. Such considerations will allow the progression of a common consistent networking paradigm without inventing a new way of programming these devices.

In addition, such considerations will take into account the inter-working of the API with the protocols. Protocols are more complex to use as they are less direct and take a lot of serializing, de-serializing and dispatching type logic.

So the connection of the API and the protocols on a constrained device becomes even more important to balance the requirements of RAM, CPU and performance.

** Here we will proceed to collect and document ... insert experiences from existing API on constrained devices (TBD) **

3.4.2. Constrained Application Protocol (CoAP)

Author: Olaf Bergmann

The Constrained Application Protocol [I-D.ietf-core-coap] has been designed specifically for machine-to-machine communication in networks with very constrained nodes. Typical application scenarios therefore include building automation and the Internet of Things.

The major design objectives have been set on small protocol overhead, robustness against packet loss, and high latency induced by small bandwidth shares or slow request processing in end nodes. To leverage integration of constrained nodes with the world-wide Internet, the protocol design was led by the architectural style that accounts for the scalability and robustness of the Hypertext Transfer Protocol [RFC2616].

Lightweight implementations benefit from this design in many respects: First, the use of Uniform Resource Identifiers (URIs) for naming resources and the transparent forwarding of their representations in a server-stateless request/response protocol make protocol-translation to HTTP a straightforward task. Second, the set of protocol elements that are inevitable for the core protocol and thus must be implemented on every node has been kept very small to avoid unnecessary accumulation of optional features. Options that -- when present -- are critical for message processing are explicitly marked as such to force immediate rejection of messages with unknown critical options. Third, the syntax of protocol data units is easy to parse and is carefully defined to avoid creation of state in servers where possible.

Although these features enable lightweight implementations of the Constrained Application Protocol, there is still a trade-off between robustness and latency of constrained nodes on one hand and resource demands (such as battery consumption, dynamic memory needs and static code-size) on the other. This section gives some guidance on possible strategies to solve this trade-off for very constrained nodes (Class 1 in Section 2.1). The main focus is on servers as this is deemed the predominant case where CoAP applications are faced with tight resource constraints.

Additional considerations for the implementation of CoAP on tiny sensors are given in [I-D.arkko-core-sleepy-sensors].

3.4.2.1. Message Layer Processing

For constrained nodes of Class 1 or even Class 2, limiting factors for (wireless) network communication usually are RAM size and battery lifetime. Most applications therefore try to avoid dealing with fragmented packets on the network layer and minimize internal buffer space for both transmit and receive operations. One of the most expensive operations hence is the retransmission of messages as it implies additional energy consumption for the (radio) network interface and occupied RAM storage for the send buffer.

Where multi-threading is not an option at all because no full-fledged operating system is present, all operations are triggered by a big

main loop in a send-receive-dispatch cycle. To implement the packet retransmission, CoAP implementations at least need a separate send buffer and a decent notion of time, e.g. as a strictly monotonic increasing tick counter. For platforms that disable clock tick interrupts in sleep states, the application must take into consideration the clock deviation that occurs during sleep (or ensure to remain in idle state until the message has been acknowledged or the maximum number of retransmissions is reached). Since CoAP allows up to four retransmissions with a binary exponential back-off it could take up to 45 seconds until the send operation is complete. Even in idle state, this means substantial energy consumption for low-power nodes. Implementers therefore might choose a two-step strategy: First, do one or two retransmissions and then, in the later phases of back-off, go to sleep until the next retransmission is due. In the meantime, the node could check for new messages including the acknowledgement for any confirmable message to send.

A similar strategy holds for confirmable messages with separate responses. This concept entitles CoAP servers to return an empty acknowledgement to indicate that a confirmable request has been understood and is being processed. Once a proper response has been generate to fulfill the request, it is sent back as a confirmable message as well. The server implementation in this case must be able to map retransmissions of the original request to the ongoing operation and provide the client-selected Token to map between original request and the separate response.

Depending on the number of requests that can be handled in parallel, an implementation might create a stub response filled with any option that has to be copied from the original request to the separate response, especially the Token option. The drawback of this technique is that the server must be prepared to receive retransmissions of the previous (confirmable) request to which a new acknowledgement must be generated. If memory is an issue, a single buffer can be used for both tasks: Only the message type and code must be updated, changing the message id is optional. Once the resource representation is known, it is added as new payload at the end of the stub response. Acknowledgements still can be sent as described before as long as no additional options are required to describe the payload.

3.4.2.2. Message Parsing

Both CoAP clients and servers must construct outgoing CoAP PDUs and parse incoming messages. The basic message header consists of only four octets and thus can be mapped easily to an internal data structure, considering the actual byte order of the host. Once the message is accepted for further processing, the set of options

contained in the received message must be decoded to check for unknown critical options. To avoid multiple passes through the option list, the option parser might maintain a bit-vector where each bit represents an option number that is present in the received request. The delta-encoded option number indicates the number of left-shift operations to apply on a bit mask to set the corresponding bit.

In addition, the byte index of every option is added to a sparse list (e.g. a one-dimensional array) for fast retrieval. This particularly enables efficient reduced-function handling of options that might occur more than once such as Uri-Path. In this implementation strategy, the delta is zero for any subsequent path segment, hence the stored byte index for option 9 (Uri-Path) will be overwritten to hold a pointer to the last occurrence of that option, i.e., only the last path component actually matters. (Of course, this requires choosing resource names where the combination of (final Uri-Path component, final Uri-Query component) is server-wide unique.

Note: Where skipping all but the last path segment is not feasible for some reason, resource identification could be ensured by some hash value calculated over the path segments. For each segment encountered, the stored hash value is updated by the current option value. This works if a cheap _perfect hashing_ scheme can be found for the resource names.

Once the option list has been processed at least up to the highest option number that is supported by the application, any known critical option and all elective options can be masked out to determine if any unknown critical option was present. If this is the case, this information can be used to create a 4.02 response accordingly. (Note that the remaining options also must be processed to add further critical options included in the original request.)

3.4.2.3. Storing Used Message IDs

If CoAP is used directly on top of UDP (i.e., in NoSec mode), it needs to cope with the fact that the UDP datagram transport can reorder and duplicate messages. (In contrast to UDP, DTLS has its own duplicate detection.) CoAP has been designed with protocol functionality such that rejection of duplicate messages is always possible. It is at the discretion of the receiver if it actually wants to make use of this functionality. Processing of duplicate messages comes at a cost, but so does the management of the state associated with duplicate rejection. Hence, a receiver may have good reasons to decide not to do the duplicate rejection. If duplicate rejection is indeed necessary, e.g., for non-idempotent requests, it is important to control the amount of state that needs to be stored.

Author: Esko Dijk

CoAP's duplicate rejection functionality can be straightforwardly implemented in a CoAP end-point by storing, for each remote CoAP end-point ("peer") that it communicates with, a list of recently received CoAP Message IDs (MIDs) along with some timing information. A CoAP message from a peer with a MID that is in the list for that peer can simply be discarded.

The timing information in the list can then be used to time out entries that are older than the _expected extent of the re-ordering_, an upper bound for which can be estimated by adding the _potential retransmission window_ ([I-D.ietf-core-coap] section "Reliable Messages") and the time packets can stay alive in the network.

Such a straightforward implementation is suitable in case other CoAP end-points generate random MIDs. However, this storage method may consume substantial RAM in specific cases, such as:

- many clients are making periodic, non-idempotent requests to a single CoAP server;
- one client makes periodic requests to a large number of CoAP servers and/or requests a large number of resources; where servers happen to mostly generate separate CoAP responses (not piggy-backed);

For example, consider the first case where the expected extent of re-ordering is 50 seconds, and N clients are sending periodic POST requests to a single CoAP server during a period of high system activity, each on average sending one client request per second. The server would need 100 * N bytes of RAM to store the MIDs only. This amount of RAM may be significant on a RAM-constrained platform. On a number of platforms, it may be easier to allocate some extra program memory (e.g. Flash or ROM) to the CoAP protocol handler process than to allocate extra RAM. Therefore, one may try to reduce RAM usage of a CoAP implementation at the cost of some additional program memory usage and implementation complexity.

Some CoAP clients generate MID values by a using a Message ID variable [I-D.ietf-core-coap] that is incremented by one each time a new MID needs to be generated. (After the maximum value 65535 it wraps back to 0.) We call this behavior "sequential" MIDs. One approach to reduce RAM use exploits the redundancy in sequential MIDs for a more efficient MID storage in CoAP servers.

Naturally such an approach requires, in order to actually reduce RAM usage in an implementation, that a large part of the peers follow the

sequential MID behavior. To realize this optimization, the authors therefore RECOMMEND that CoAP end-point implementers employ the "sequential MID" scheme if there are no reasons to prefer another scheme, such as randomly generated MID values.

Security considerations might call for a choice for (pseudo)randomized MIDs. Note however that with truly randomly generated MIDs the probability of MID collision is rather high in use cases as mentioned before, following from the Birthday Paradox. For example, in a sequence of 52 randomly drawn 16-bit values the probability of finding at least two identical values is about 2 percent.

From here on we consider efficient storage implementations for MIDs in CoAP end-points, that are optimized to store "sequential" MIDs. Because CoAP messages may be lost or arrive out-of-order, a solution has to take into account that received MIDs of CoAP messages are not actually arriving in a sequential fashion, due to lost or reordered messages. Also a peer might reset and lose its MID counter(s) state. In addition, a peer may have a single Message ID variable used in messages to many CoAP end-points it communicates with, which partly breaks sequentiality from the receiving CoAP end-point's perspective. Finally, some peers might use a randomly generated MID values approach. Due to these specific conditions, existing sliding window bitfield implementations for storing received sequence numbers are typically not directly suitable for efficiently storing MIDs.

Table 1 shows one example for a per-peer MID storage design: a table with a bitfield of a defined length _K_ per entry to store received MIDs (one per bit) that have a value in the range [MID_i + 1 , MID_i + K].

MID base	K-bit bitfield	base time value
MID_0	010010101001	t_0
MID_1	111101110111	t_1
... etc.		

Table 1: A per-peer table for storing MIDs based on MID_i

The presence of a table row with base MID_i (regardless of the bitfield values) indicates that a value MID_i has been received at a time t_i. Subsequently, each bitfield bit k (0...K-1) in a row i corresponds to a received MID value of MID_i + k + 1. If a bit k is

0, it means a message with corresponding MID has not yet been received. A bit 1 indicates such a message has been received already at approximately time t_i. This storage structure allows e.g. with k=64 to store in best case up to 130 MID values using 20 bytes, as opposed to 260 bytes that would be needed for a non-sequential storage scheme.

The time values t_i are used for removing rows from the table after a preset timeout period, to keep the MID store small in size and enable these MIDs to be safely re-used in future communications. (Note that the table only stores one time value per row, which therefore needs to be updated on receipt of another MID that is stored as a single bit in this row. As a consequence of only storing one time value per row, older MID entries typically time out later than with a simple per-MID time value storage scheme. The end-point therefore needs to ensure that this additional delay before MID entries are removed from the table is much smaller than the time period after which a peer starts to re-use MID values due to wrap-around of a peer's MID variable. One solution is to check that a value t_i in a table row is still recent enough, before using the row and updating the value t_i to current time. If not recent enough, e.g. older than N seconds, a new row with an empty bitfield is created.) [Clearly, these optimizations would benefit if the peer were much more conservative about re-using MIDs than currently required in the protocol specification.]

The optimization described is less efficient for storing randomized MIDs that a CoAP end-point may encounter from certain peers. To solve this, a storage algorithm may start in a simple MID storage mode, first assuming that the peer produces non-sequential MIDs. While storing MIDs, a heuristic is then applied based on monitoring some "hit rate", for example, the number of MIDs received that have a Most Significant Byte equal to that of the previous MID divided by the total number of MIDs received. If the hit rate tends towards 1 over a period of time, the MID store may decide that this particular CoAP end-point uses sequential MIDs and in response improve efficiency by switching its mode to the bitfield based storage.

3.4.3. (Other Application Protocols...)

4. Control Plane Protocols

4.1. Link Layer Support

ARP, ND; 6LoWPAN-ND

4.2. Network Layer

ICMP, ICMPv6, IGMP/MLD

4.3. Routing

RPL, AODV/DYMO, OLSRv2

4.4. Host Configuration and Lookup Services

DNS, DHCPv4, DHCPv6

4.5. Network Management

SNMP, netconf?

4.5.1. SNMP

Author: Brinda M C

This section describes an approach for developing a light-weight SNMP agent for resource constrained devices running the 6LoWPAN/RPL protocol stack. The motivation for the work is driven by two major factors:

- SNMP plays a vital role in monitoring and managing any operational network; 6LoWPAN based WSN is no exception to this.
- There is a need for building a light-weight SNMP agent which consumes less memory and less computational resources.

The following subsections are organized as follows:

- Section 4.5.1.1 provides some background.
- In Section 4.5.1.2, we revisit existing SNMP implementation in the context of memory constrained devices.
- In Section 4.5.1.3, we present our approach for building a memory efficient SNMP agent.

- Using a realistic example, in Section 4.5.1.4, we illustrate how the proposed method can be implemented.

- In Section 4.5.1.5, we explore a few ideas which can further help in improving the memory utilization.

4.5.1.1. Background

Our initial SNMP agent implementation was completely based on Net-SNMP, well-known open-source network monitoring and management software. After porting the agent on to the TelosB mote, we observed that it occupies a text program memory of more than 8 KiB on TinyOS and Contiki OS platforms. (Note that both these platforms already use compiler optimizations to minimize the memory footprint.) 8 KiB is already non-negligible given the 48 KiB program memory limit of TelosB. Added to this, the memory taken up by 6LoWPAN and the related protocol stacks are ever growing, causing serious memory crunch in the resource constrained devices. We reached a situation where we could not build an image on the TinyOS/Contiki OS platforms with our SNMP agent.

We came across SNMPv1 agent implementations elsewhere in the literature which also report similar memory consumption. This motivated us to have a re-look at the existing SNMP agent implementation, and explore the possibility of an alternate implementation using altogether a different approach.

4.5.1.2. Revisiting SNMP implementation for resource constrained devices

If we look at a typical SNMP agent implementation, we can see that much of the memory consuming code is pertaining to ASN.1 related SNMP PDU parsing and SNMP PDU build operations. The SNMP parsing mainly recovers various fields from the incoming PDU, such as the OIDs, whereas the SNMP PDU build is the reverse operation of building the response PDU from the OIDs.

The key observation is that, for a given MIB definition, an OID of interest contained in the incoming SNMP PDU is already available, albeit in an encoded form. This enables identifying the OID from the packet in its "raw" form, simplifying parser operation.

We also can make use of this observation while building the response SNMP PDU. For a given MIB definition, we can think of statically having a pre-composed ASN.1 encoded version of OIDs, and use them while constructing the response SNMP PDU.

4.5.1.3. Proposed approach for building an memory efficient SNMP agent

As noted in the previous section, since an SNMP OID is already _contained_ in the incoming network PDU, we came up with a simple OID signature identification method performed directly on the network PDU through simple memory comparisons and table look-ups. Once the OID has been identified from the packet "in situ", the corresponding per-OID processing is carried out. Through this scheme we completely eliminated expensive SNMP parse operations.

For the SNMP PDU build, we use _pre-encoded_ OID variables which can simply be plugged into the network SNMP response packet directly depending on the request OID. Now that the expensive build operation is taken care, what remains is the construction of the overall SNMP pdu which can be built through simple logic. Through this scheme we completely eliminated expensive SNMP build operations.

Based on these ideas, we have re-architected our original SNMP agent implementation and with our new implementation we were able to bring down its text memory usage all the way down to 4 KiB from the native SNMP agent implementation which occupied 8 KiB.

4.5.1.3.1. Discussion on memory usage

With respect to the memory usage, while we have achieved major reduction in terms of text program memory, which occupies a major chunk of memory, a question might come to mind with regard to the static memory allocation for maintaining the tables. We found that this is not very significant to start with. Through an efficient table representation, we further optimized the memory consumption. We could do so because a typical OID description is mainly dominated by a fixed part of the hierarchy. This enables us to define few static prefixes, each corresponding to a particular hierarchy level of the OID. In the context of 6LoWPAN, it can be expected that the number of hierarchy levels will be small.

4.5.1.4. Example

This section illustrates the simplicity and practicality of our approach with an example. Let us consider the fragment of a representative MIB definition depicted in Figure 1

```
iso
|
org
|
dod
|
internet
|
mgmt.mib-2
|
lowpanMIB
|
+--lowpanPrimaryStatistics(10)
   |
   +--PrimeStatsEntry(1)
      |
      +-- -R-- INTEGER   lowpanMoteBatteryVoltageP(1)
      +-- -R-- Counter   lowpanFramesReceivedP(2)
      +-- -R-- Counter   lowpanFramesSentP(3)
      +-- -R-- Counter   ipv6ForwardedMsgP(4)
      +-- -R-- Counter   OUTSolicitationP(5)
      +-- -R-- Counter   OUTAdvertisementP(6)
```

Figure 1: A fragment of a MIB hierarchy

4.5.1.4.1. Optimized SNMP Parsing

Let us consider a GET request for the OIDs lowpanMoteBatteryVoltageP and lowpanFramesSentP. Corresponding to these OIDs, a C array dump of the network PDU of SNMP packet with two OIDs in a variable binding would look as in Figure 2.

```
char snmp_get_req_pkt[] = {
    0x30, 0x81, 0x3d, 0x02, 0x01, 0x00, 0x04, 0x06,
    0x70, 0x75, 0x62, 0x6c, 0x69, 0x63, 0xa0, 0x30,
    0x02, 0x04, 0x28, 0x29, 0xe4, 0x5d, 0x02, 0x01,
    0x00, 0x02, 0x01, 0x00, 0x30, 0x22, 0x30, 0x0f,
    0x06, 0x0b, 0x2b, 0x06, 0x01, 0x02, 0x01, 0x83,
    0x90, 0x12, 0x0a, 0x01, 0x01, 0x05, 0x00, 0x30,
    0x0f, 0x06, 0x0b, 0x2b, 0x06, 0x01, 0x02, 0x01,
    0x83, 0x90, 0x12, 0x0a, 0x01, 0x03, 0x05, 0x00 };
```

Figure 2: An SNMP packet, represented in C

Inspecting the above packet, we see that the main components of the PDU are:

1. Version (SNMPv1): [0x02, 0x01, 0x00]

2. Community Name ("public"): [0x04, 0x06, 0x70, 0x75, 0x62, 0x6c, 0x69, 0x63]

3. ASN.1 encoded OIDs for lowpanMoteBatteryVoltageP, and lowpanFramesReceivedP:

   * [0x30, 0x0f, 0x06, 0x0b, 0x2b, 0x06, 0x01, 0x02, 0x01, 0x83, 0x90, 0x12, 0x0a, 0x01, 0x01, 0x05, 0x00]

   * [0x30, 0x0f, 0x06, 0x0b, 0x2b, 0x06, 0x01, 0x02, 0x01, 0x83, 0x90, 0x12, 0x0a, 0x01, 0x03, 0x05, 0x00]

There is a significant overlap between the two OIDs, which can be used to simplify the parsing process. We can, for instance, define one statically initialized array containing elements common between these OIDs. Using this notion of common prefix idea, we can come up with an optimized table and the OID identification then boils down to simple memory comparisons within this table. The optimized table construction will also result in scalability.

4.5.1.4.2. Optimized SNMP Build

Extending the same approach as described above, we can build the GET response by plugging in pre-encoded OIDs into the response packets. So, corresponding to the GET request for the OIDs as given in section 4.1, we can define C arrays containing pre-encoded OIDs which can go into the response packet as in Figure 3.

```
pdu_batt_volt[] = {
    0x30, 0x11, 0x06, 0x0b, 0x2b, 0x06, 0x01, 0x02,
    0x01, 0x83, 0x90, 0x12, 0x0a, 0x01, 0x01, 0x02,
    0x02, 0x00, 0x00 };

pdu_frames_sent[] = {
    0x30, 0x11, 0x06, 0x0b, 0x2b, 0x06, 0x01, 0x02,
    0x01, 0x83, 0x90, 0x12, 0x0a, 0x01, 0x03, 0x41,
    0x02, 0x00, 0x00 };
```

Figure 3: Pre-encoded OIDs

Since the ASN.1 basic encoding rules are in TLV format, the offset within the encoded OID where the value needs to be filled-in can be obtained from the length field.

The table size optimization discussed in the previous section can be applied here, too.

Note: Though we have taken a simple example to illustrate the efficacy of the proposed approach, the ideas presented here can easily be extended to other scenarios as well.

4.5.1.5. Further improvements

A few simple methods can reduce the code size as well as generate computationally inexpensive code. These methods might sound obvious and trivial but are important for constrained devices.

- If possible, avoid using memory consuming data types such as floating point while representing a monitored variable when an equivalent representation of the same that occupies less memory is adequate. For example, while a battery voltage indication could take a fractional value between 0 and 3 V, opt for an 8-bit quantized value.

- Using meta data in the MIB definition instead of absolute numbers can bring down the memory and processing significantly and can improve scalability too especially for a large scale WSN deployments. Using the same example of battery voltage, one might think of an OID which represents fewer levels of the battery voltage signifying high, medium, low, very low.

- While a multi-level hierarchy for MIB definition might improve OID segregation the flip side is that it increases the overall length of the OID and results in extra memory and processing overhead. One may have to make a judicious choice while coming up with the MIB.

4.5.1.6. Conclusion

This subsection proposes a simple SNMP packet processing based approach for building a light-weight SNMP agent. While there is scope for further improvement, we believe that the proposed method can be a reasonably good starting point for resource constrained 6LoWPAN based networks.

5. Security protocols

5.1. Cryptography for Constrained Devices

5.2. Transport Layer Security

TLS, DTLS, ciphersuites, certificates

5.3. Network Layer Security

IPsec, IKEv2, transforms

Advice for a minimal implementation of IKEv2 can be found in [I-D.kivinen-ipsecme-ikev2-minimal].

5.4. Network Access Control

(PANA, EAP, EAP methods)

5.4.1. PANA

Author: Mitsuru Kanda

PANA [RFC5191] provides network access authentication between clients and access networks. The PANA protocol runs between a PANA Client (PaC) and a PANA Authentication Agent (PAA). PANA carries UDP encapsulated EAP [RFC3748] and includes various operational options. From the point of view of minimal implementation, some of these are not necessary for constrained devices. This section describes a minimal PANA implementation for these devices.

The minimization objective for this implementation mainly targets PaCs because constrained devices often are installed as network clients, such as sensors, metering devices, etc.

5.4.1.1. PANA AVPs

Each PANA message can carry zero or more AVPs (Attribute-Value Pairs) within its payload. [RFC5191] specifies nine types of AVPs (AUTH, EAP-Payload, Integrity-Algorithm, Key-Id, Nonce, PRF-Algorithm, Result-Code, Session-Lifetime, and Termination-Cause). All of them are required by all minimal implementations. But there are some notes.

Integrity-Algorithm AVP and PRF-Algorithm AVP:

All PANA implementations MUST support AUTH_HMAC_SHA1_160 for PANA message integrity protection and PRF_HMAC_SHA1 for pseudo-random

function (PRF) specified in [RFC5191]. Both of these are based on SHA-1, which therefore needs to be implemented in a minimal implementation.

Nonce AVP:

As the basic hash function is SHA-1, including a nonce of 20 bytes in the Nonce AVP is appropriate ([RFC5191], section 8.5).

5.4.1.2. PANA Phases

A PANA session consists of four phases -- Authentication and authorization phase, Access phase, Re-Authentication phase, and Termination phase.

Authentication and authorization phase:

There are two types of PANA session initiation, PaC-initiated session and PAA-initiated session. The minimal implementation must support PaC-initiated session and does not need to support PAA-initiated session. Because a PaC (a constrained device) which may be a sleeping device, can not receive an unsolicited PANA-Auth-Request message from a PAA (PAA-initiated session).

EAP messages can be carried in PANA-Auth-Request and PANA-Auth-Answer messages. In order to reduce the number of messages, "Piggybacking EAP" is useful. Both the PaC and PAA should include EAP-Payload AVP in each of PANA-Auth-Request and PANA-Auth-Answer messages as much as possible. Figure 4 shows an example "Piggybacking EAP" sequence of the Authentication and authorization phase.

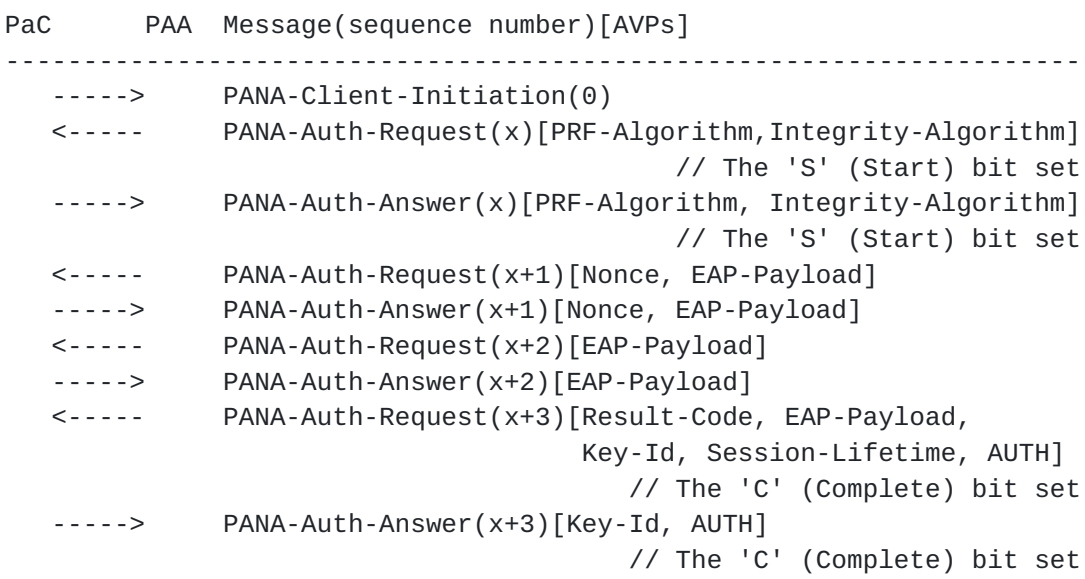

Figure 4: Example sequence of the Authentication and authorization phase for a PaC-initiated session (using "Piggybacking EAP")

Note: It is possible to include an EAP-Payload in both the PANA-Auth-Request and PANA-Auth-Answer messages with the 'S' bit set. But the PAA should not include an EAP-Payload in the PANA-Auth-Request message with the 'S' bit set in order to stay stateless in response to a PANA-Client-Initiation message.

Access phase:

After Authentication and authorization phase completion, the PaC and PAA share a PANA Security Association (SA) and move Access phase. During Access phase, [RFC5191] describes both the PaC and PAA can send a PANA-Notification-Request message with the 'P' (Ping) bit set for the peer's PANA session liveness check (a.k.a "PANA ping"). From the minimal implementation point of view, the PAA should not send a PANA-Notification-Request message with the 'P' (Ping) bit set to initiate PANA ping since the PaC may be sleeping. The PaC does not need to send a PANA-Notification-Request message with the 'P' (Ping) bit set for PANA ping to the PAA periodically and may omit the PANA ping feature itself if the PaC can detect the PANA session failure by other methods, for example, network communication failure. In conclusion, the PaC does not need to implement the periodic liveness check feature sending PANA ping but a PaC that is awake should respond to a incoming PANA-Notification-Request message with the 'P' (Ping) bit set for PANA ping as possible.

Re-Authentication phase:

Before PANA session lifetime expiration, the PaC and PAA MUST re-

negotiate to keep the PANA session. This means that the PaC and PAA enter Re-Authentication phase. Also in the Authentication and authorization phase, there are two types of re-authentication. The minimal implementation must support PaC-initiated re-authentication and does not need to support PAA-initiated re-authentication (again because the PaC may be a sleeping device). "Piggybacking EAP" is also useful here and should be used as well. Figure 5 shows an example "Piggybacking EAP" sequence of the Re-Authentication phase.

```
PaC      PAA  Message(sequence number)[AVPs]
-----------------------------------------------------------------------
  ----->     PANA-Notification-Request(q)[AUTH]
                          // The 'A' (re-Authentication) bit set
  <-----     PANA-Notification-Answer(q)[AUTH]
                          // The 'A' (re-Authentication) bit set
  <-----     PANA-Auth-Request(p)[EAP-Payload, Nonce, AUTH]
  ----->     PANA-Auth-Answer(p)[EAP-Payload, Nonce, AUTH]
  <-----     PANA-Auth-Request(p+1)[EAP-Payload, AUTH]
  ----->     PANA-Auth-Answer(p+1)[EAP-Payload, AUTH]
  <-----     PANA-Auth-Request(p+2)[Result-Code, EAP-Payload,
                                    Key-Id, Session-Lifetime, AUTH]
                                    // The 'C' (Complete) bit set
  ----->     PANA-Auth-Answer(p+2)[Key-Id, AUTH]
                                    // The 'C' (Complete) bit set
```

Figure 5: Example sequence of the Re-Authentication phase for a PaC-initiated session (using "Piggybacking EAP")

Termination Phase:

The PaC and PAA should not send a PANA-Termination-Request message except for explicitly terminating a PANA session within the lifetime. Both the PaC and PAA know their own PANA session lifetime expiration. This means the PaC and PAA should not send a PANA-Termination-Request message when the PANA session lifetime expired because of reducing message processing cost.

5.4.1.3. PANA session state parameters

All PANA implementations internally keep PANA session state information for each peer. At least, all minimal implementations need to keep PANA session state parameters below (in the second column storage sizes are given in bytes):

State Parameter	Size	Comment
PANA Phase Information	1	Used for recording the current PANA phase.
PANA Session Identifier	4	
PaC's IP address and UDP port number	6 or 18	IP Address length (4 bytes for IPv4 and 16 bytes for IPv6) plus 2 bytes for UDP port number.
PAA's IP address and UDP port number	6 or 18	IP Address length (4 bytes for IPv4 and 16 bytes for IPv6) plus 2 bytes for UDP port number.
Outgoing message sequence number	4	Next outgoing request message sequence number.
Incoming message sequence number	4	Next expected incoming request message sequence number.
A copy of the last sent message payload	variable	Necessary to be able to retransmit the message (unless it can be reconstructed on the fly).
Retransmission interval	4	
PANA Session lifetime	4	
PaC nonce	20	Generated by PaC and carried in the Nonce AVP.
PAA nonce	20	Generated by PAA and carried in the Nonce AVP.
EAP MSK Identifier	4	
EAP MSK value	*)	Generated by EAP method and used for generating PANA_AUTH_KEY.
PANA_AUTH_KEY	20	Necessary for PANA message protection.

PANA PRF algorithm number	4	Used for generating PANA_AUTH_KEY.
PANA Integrity algorithm number	4	Necessary for PANA message protection.

*) (Storage size depends on the key derivation algorithm.)

Note: EAP parameters except for MSK have not been listed here. These EAP parameters are not used by PANA and depend on what EAP method you choose.

6. Wire-Visible Constraints

- Checksum
- MTU
- Fragmentation and reassembly
- Options -- implications of leaving some out
- Simplified TCP optimized for LLNs
- Out-of-order packets

7. Wire-Invisible Constraints

- Buffering
- Memory management
- Timers
- Energy efficiency
- API
- Data structures
- Table sizes (somewhat wire-visible)
- Improved error handling due to resource overconsumption

8. IANA Considerations

This document makes no requirements on IANA. (This section to be removed by RFC editor.)

9. Security Considerations

(TBD.)

10. Acknowledgements

Much of the text of the introduction is taken from the charter of the LWIG working group and the invitation to the IAB workshop on Interconnecting Smart Objects with the Internet. Thanks to the numerous contributors. Angelo Castellani provided comments that led to improved text.

10.1. Contributors

The RFC guidelines no longer allow RFCs to be published with a large number of authors. As there are many authors that have contributed to the sections of this document, their names are listed in the individual section headings as well as alphabetically listed with their affiliations below.

Name	Affiliation
Brinda M C	Indian Institute of Science
Carl Williams	MCSR Labs
Carsten Bormann	Universitaet Bremen TZI
Esko Dijk	Philips Research
Mitsuru Kanda	Toshiba
Olaf Bergmann	Universitaet Bremen TZI
...	...

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, September 2007.

11.2. Informative References

[I-D.arkko-core-sleepy-sensors] Arkko, J., Rissanen, H., Loreto, S., Turanyi, Z., and O. Novo, "Implementing Tiny COAP Sensors", draft-arkko-core-sleepy-sensors-01 (work in progress), July 2011.

[I-D.ietf-core-coap] Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-08 (work in progress), October 2011.

[I-D.kivinen-ipsecme-ikev2-minimal] Kivinen, T., "Minimal IKEv2", draft-kivinen-ipsecme-ikev2-minimal-00 (work in progress), February 2011.

[RFC1071] Braden, R., Borman, D., Partridge, C., and W. Plummer, "Computing the Internet checksum", RFC 1071, September 1988.

[RFC1141] Mallory, T. and A. Kullberg, "Incremental updating of the Internet checksum", RFC 1141, January 1990.

[RFC1624] Rijsinghani, A., "Computation of the Internet Checksum via Incremental Update", RFC 1624, May 1994.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3748] Aboba, B., Blunk, L., Vollbrecht, J., Carlson, J., and H. Levkowetz, "Extensible Authentication Protocol (EAP)", RFC 3748, June 2004.

[RFC5191] Forsberg, D., Ohba, Y., Patil, B., Tschofenig, H., and A.

Yegin, "Protocol for Carrying Authentication for Network Access (PANA)", RFC 5191, May 2008.

[WEI] Shelby, Z. and C. Bormann, "6LoWPAN: the Wireless Embedded Internet", ISBN 9780470747995, 2009.

Author's Address

Carsten Bormann (editor)
Universitaet Bremen TZI
Postfach 330440
Bremen D-28359
Germany

Phone: +49-421-218-63921
Fax: +49-421-218-7000
Email: cabo@tzi.org