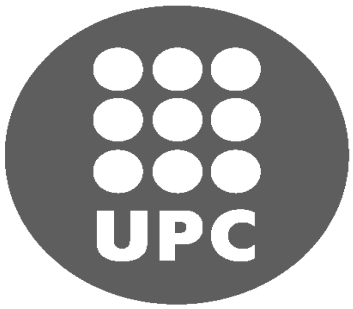

UNIVERSIDAD POLITECNICA DE CATALUNYA

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA DE TELECOMUNICACIO DE BARCELONA

“Implementation and Evaluation of the Enhanced Header Compression (IPHC) according to 6LoWPAN Network”

Directora de Proyecto:

Anna Calveras Auge

Estudiant:

Alessandro Ludovici

Any Acadèmic 2008/2009

Index

1 Overview

1.1 General

The wide diffusion of wireless telecommunication networks and the relative development of new communication protocols, has allowed to create low-cost networks architectures, to redefine the concept of network terminal by giving to it mobility and to apply communication technologies in environments where wired networks could not be applied. In this perspective, Wireless Sensor Networks (WSN) can be intended as the diffusion and the integration of electronic devices with the surrounding natural and man-made environment that communicate each other to share, elaborate and obtain data. Technically speaking, a WSN consists potentially of thousands of tiny, low power motes, each of which will execute concurrent and reactive software applications operating with severe memory and power constraints. Such network, is programmed and developed to perform specific and precise tasks. Among the applications a WSN can have, we found: Environmental and habitat monitoring, Structural monitoring, Traffic control, Industrial control. Because the nature of the applications, the human intervention on the deployed network can be very difficult and not feasible. Consequently, a WSN should be self-healing and fault-tolerant, moreover motes should run for months or even years on battery powers. This reasons results in the low-power and energy saving design that has to be applied when developing any protocol stack for WSN.

The widest used protocol stack for WSN is the IEEE 802.15.4 standard. It defines only the physical

and link layer leaving the definition of upper layers to other protocol stack implementation. This lack of definition has lead to multiples proposal and implementation for layer 3 to 7. Between this proposal is placed 6LoWPAN, a protocol implementation that allow to encapsulate IPv6 datagrams into 802.15.4 packets. This will be the protocol stack that will be used in this work. Fig.1 compare the existing protocols for wireless networks in terms of Data Rate and Power consumption. As we can see, IEEE 802.15.4 has the lowest performance in terms of throughput and energy consumption.

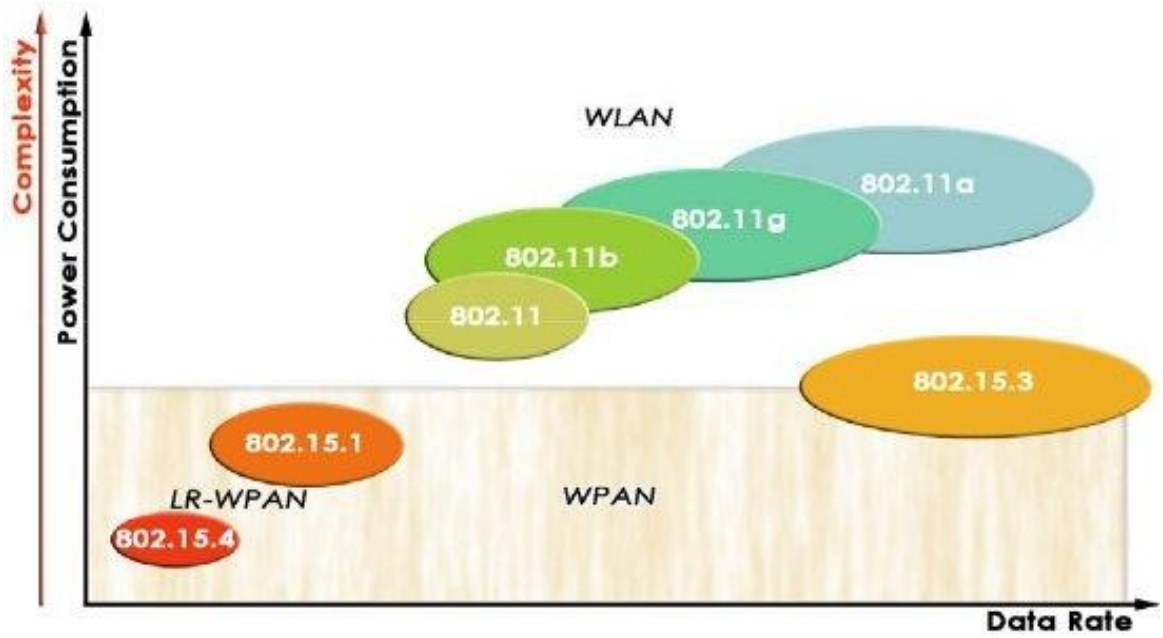

Figure 1: Data Rate and Power consumption comparison for wireless protocols [30]

1.2 Purpose

6LoWPAN defines how to carry Ipv6 packet over IEEE 802.15.4 low power wireless or sensor networks. Limited bandwidth, memory and energy resources require a careful application of Ipv6 in a LoWPAN network. The aim is to develop personal networks, mainly sensor based, that can be integrated to the existing well-know network infrastructure by reusing mature and wide-used technologies. IPv6 has been chosen as network protocol because its characteristics fit to the problematic that characterize LoWPAN environment such as the large number of nodes to address and stateless address auto-configuration. However, an IPv6 header compression algorithm is necessary in order to reduce the overhead and save space in data payload. In fact, the IEEE 802.15.4 standard defines an MTU of 128 bytes that decrease to 102 bytes considering the frame overhead, a further reduction is due to the network and transport protocols frame overhead that, in case of Ipv6 and UDP, allow to carry only 33 bytes for application data. The aim of this work is to describe and

compares the proposed Ipv6 header compression mechanisms for 6LoWPAN environments.

1.3 Document Structure

Before entering in the main issue of this work, that is the IPv6 header compression, the IEEE 802.15.4 and the IPv6 network protocol standards will be introduced respectively in chapters one and two. Such standards will help to have a background detail of the network environment where header compression will be applied, that is 6LoWPAN. Chapter three will be dedicated to 6LoWPAN. There will be showed how IPv6 can run over IEEE 802.15.4, stressing in particular the data frame format composition and the addressing methodology. The header compression will be presented in chapter five, where will be given an overview of the already existing IPv6 header compression algorithm focusing in particular to the ones thought specifically for 6LoWPAN. Chapter six will introduce and illustrate the implementation of the algorithm by exposing its requirements and presenting the hardware and software components used in this project. Finally, the implementation will be evaluated in chapter seven. Chapter 8 will contain the conclusions and the proposal for futures development of this work.

2 IEEE 802.15.4

2.1 General definitions

IEEE 802.15.4[1] standard defines the protocol and interconnection of devices via radio communication in a personal area network (PAN). The standard use the OSI reference model and specifies its two lower layers: the Physical (PHY) and the Medium Access Control (MAC) sublayer of the Data link layer. As upper sublayer is used the IEEE 802.2 Logical Link Control (LLC) that can have access to MAC by means of a Service-Specific Convergence Sublayer (SSCS).

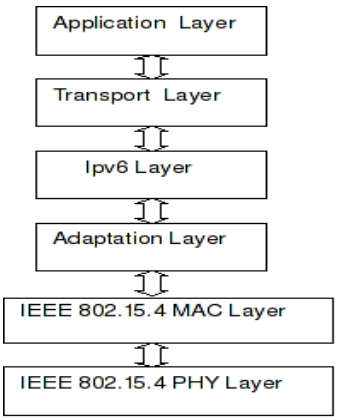

Figure 2: IEEE 802.15.4 protocol stack

The main characteristics of a Low Rate Wireless Personal Area Network (LR-WPAN) defined by this standard are the followings:

- data rates of 250 kb/s, 100 kb/s, 40 kb/s and 20 kb/s
- allocation of IEEE 16-bit short or IEEE 64-bit extended addresses
- use of CSMA-CA
- low power consumption
- Energy detection
- Link quality indication
- 16 channels in the 2450 MHz band, 30 channel in the 915 MHz band and 3 in the 868 MHz.

The basic devices getting involved in such a network are called Reduced Function Device (RFD) and Fully Function Device (FFD). The FFD can have three different roles; a PAN coordinator, a coordinator or a device. An RFD is intended for simple applications where it is not needed to send large amount of data. It can only communicate with an FFD, while an FFD has not limitation in this sense.

The network can assume a peer-to-peer or a star topology. In both is strictly required the presence of a PAN coordinator. Address are assigned during the association phase and can be allocated up to 2^{64} addresses if IEEE 64-bit extended address are used, 2^{16} if IEEE 16-bit short address are used. The communication between single nodes is limited to a Personal Operating Space (POS) that is fixed to 10m. Each PAN have associated a unique identifier (PAN-ID) that permit the devices to communicate within its network using the short-address format.

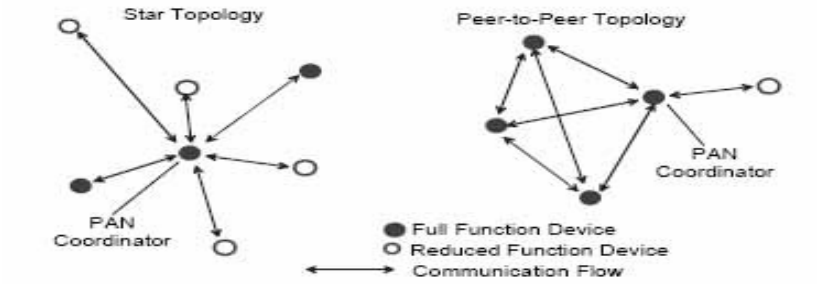

Figure 3: Network topologies [2]

It is important to specify also two other important features of the 802.15.4, that is the self-healing and self-organizing properties. The former means that network is able to detect and recover from faults appearing in either networks nodes or communication links. The latter allow to nodes to be able to detect the presence (at a POS distance as maximum) of other nodes and to organize them into a well defined and structured network. This features are implemented by the association (a node

join the network) and disassociation (a node leave the network) functions present at MAC level.

2.2 Data transfer and data frame

The standard defines three different modalities for data transfer: coordinator to device, device to coordinator or between peer devices. The first two data transaction are differentiated if the PAN is beacon-enabled or nonbeacon-enabled. Peer-to-peer data transfer can use a synchronous or asynchronous transfer model (slotted\unslotted CSMA-CA) and its definition is leaved to the network layer.

Frame structures is defined in four different types:

- beacon frame (synchronize the devices, identify PAN)
- data frame (used for data transfer)
- acknowledgment frame (optional)
- MAC command frame (handle MAC peer entity control transfers)

It is also defined a superframe structure bounded by two beacon frames. They are sent by the coordinator and have a length of 16 slots and can. Optionally, they can be divided into an active or an inactive part giving the option to the coordinator to enter in a power-save mode. The superframe lifetime defines also the contention mechanism.

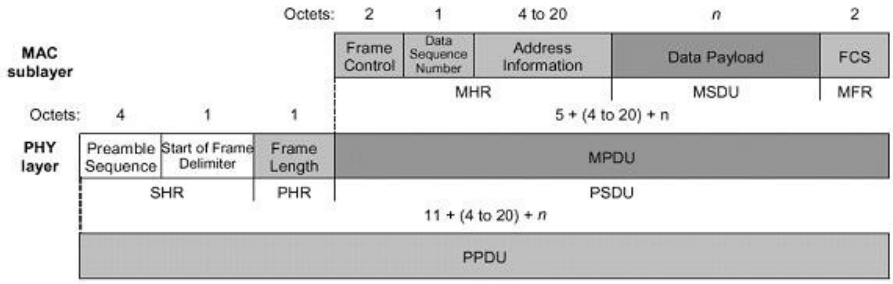

Figure 4: data frame structure[2]

Since the data frame is the one in charge of carry the packets generated at upper layers, we will focus our attention to its structure. It is formed as follow; the upper layer payload when passed to layer 2 is specified as MAC service data unit (MSDU), it is prefixed with a MAC header (MHR) and appended by a MFR composed by a 16-bit Frame Check Sequence (FCS) field. A such formed MAC data frame is specified as MAC Protocol Data Unit (MPDU). Afterwards, it will be passed to the physical layer as Physical Service Data Unit (PSDU). To PSDU will be attached a prefix called synchronization header (SHR) and a physical header (PHR). The come out of this process is the

physical protocol data unit (PPDU). This is the packet ready to be physically transmitted over the 802.15.4 link. In IEEE 802.15.4 definitions, the physical packet size is fixed to 127 bytes where 25 bytes of them are intended for frame overhead. In that way remains 102 bytes as maximum length of MAC service data unit.

3 IPv6

Before illustrating in detail the IPv6 protocol, it is important to remark the changes made respect to IPv4. Differences fall mainly into the address auto-configuration and the increased size of the address length (from 32 to 128 bits). This allows increasing the number on nodes addressable. Furthermore, the header format has been simplified in order to reduce the operational cost and bandwidth consumption derived from its handling. The processing of packet by routers becomes easier than before. They do not have to compute any checksum for layer IP since it is done already in upper layers and do not perform fragmentation. The header size has passed from 20 byte to 40 byte. It has been improved the support for extensions and options headers. Mechanism for congestion control and QoS management has been further improved.

3.1 Header format

The IPv6 header is composed by the following fields:

- Version (4 bit): version of the IP protocol.
- Traffic Class (8 bit): identify and distinguish between different class or priorities of Ipv6 packet.
- Flow Label (20 bit): assigned by the source to a particular flows that require special handling.

- Payload length (16 bit): length of the IPv6 payload.
- Next header (8 bit): specify the header type followings the IPv6 one.
- Hop limit (8 bit): replace the Time to Live of Ipv4. Any time the packet is forwarded this value is decremented by 1. If is equal to zero the packet will be discarded.
- Source address (128 bit).
- Destination address (128 bit).

Such composed header has a size of 40 bytes. In Fig.5 is showed the IPv6 header format.

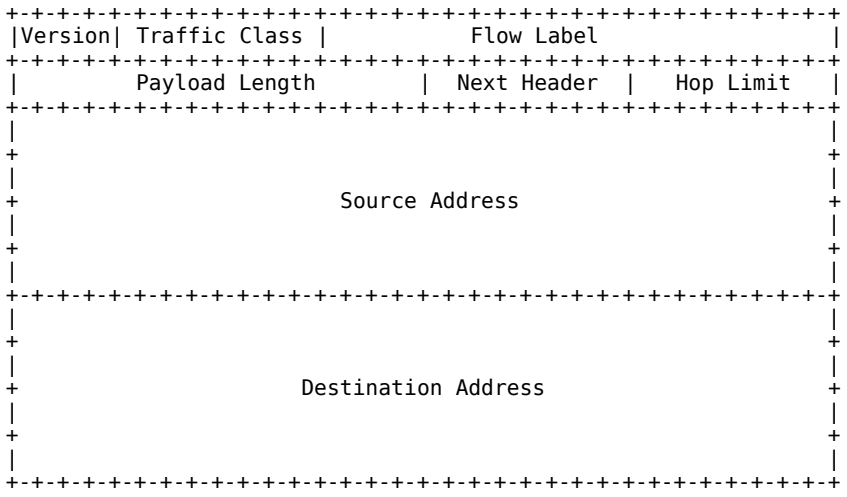

Figure 5: IPv6 header

IPv6 specification requires to every link to have at least 1280 bytes as Maximum Transmission Unit (MTU) [3]. Furthermore, extensions headers are defined as follow: Hop-by-hop Options, Routing, Fragment, Destination Options, Authentication, Encapsulating security payload. For our purposes, the presence of extensions headers should be minimized or better avoided, since they will reduce the number of bytes available to carry the application data payload while enlarging the IPv6 header size

.

3.2 Addressing

The addressing architecture of IPv6 [4] define three type of address:

- Unicast: packet is delivered to the interface having this address.
- Anycast: packet is delivered to one of the set of interfaces having this address.

- Multicast: packet is delivered to all the interface having this address.

The address format is composed by eight fields each of 16 bits. It would appear like "x:x:x:x:x:x:x:x" where the x's are one to four hexadecimal digits of the eight 16-bit pieces of the address. A further look to the address composition permits to distinguish between two logical part [5], each of 64 bit, representing the network prefix and the host address part. The latter is used to identify interfaces on a link and is unique within a subnet prefix. They are derived from the link-layer address (IEEE 64 or 16 bit address). An example of IPv6 address would be:

2001:DB8:0:0:8:800:200C:417A

Between the unicast address is made a further classification:

- Unspecified address:

  It is a all zero address indicating the absence of an address; It is used for host initialization; It is represented as: 0:0:0:0:0:0:0:0

- Loopback Address:

  it has link-local scope and may be used by a source to send a packet to itself; It is represented as: 0:0:0:0:0:0:0:1

- Global Unicast Addresses

  It is formed by the 64 bit interface identifier plus the subnet identifier and a global routing prefix. The format is the following:

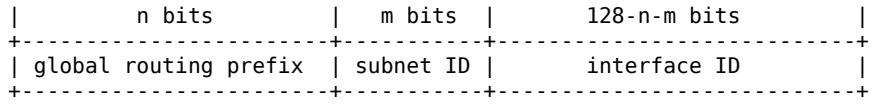

Figure 6: Global Unicast Address format

- Embedded IPv4 addresses:

  It is used to allow the transition and the co-existence of IPv4 with IPv6.

- Link-local address:

  It has been thought to be used for address autoconfiguration, neighbor discovery and when no routers are presents. The format is the following:

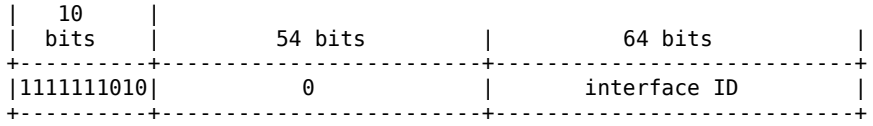

Figure 7: Link-Local Address

- Site-local address:

  It enlarge the scope of link-local address in order to further specify address within a site; It avoid the use of a global prefix for each single site-local address used.

3.3 Stateless auto-configuration

An host can generate its own address combining local available information plus information advertised by routers. The latter provides the subnetwork prefix associated with a link. The former generates the interface identifier. The combination will generate the IPv6 address in the format seen before. The subnetwork prefix is not indispensable to allow to nodes communicate. In fact, nodes on the same link can establish a communication session using its own link-local address.

Addresses have a validity fixed by a lifetime settings and can assume two state: preferred and deprecated. It is also established an algorithm called “duplicate address detection” in order to check the uniqueness of an address on a given link. In the stateless autoconfiguration procedure are used Neighbor discovery messages specified in [20] as “*IPv6 nodes on the same link use Neighbor Discovery to discover each other's presence, to determine each other's link-layer addresses, to find routers, and to maintain reachability information about the paths to active neighbors*”.

In order to assign a link-local address to an interface, it must be verified that the address is not in use or in attempt to use by another interface. For this reason before the assignment of the address, the node send a neighbor solicitation message containing the address demanded. If no answer is received then, the address can be used and assigned to node's interface. At this point the node has IP-connectivity with its neighbor nodes. Then, the configuration is carried by the host itself. It learn the step to follow by a router advertisement message. Here, it is used the duplicate address detection algorithm to verify the uniqueness of the address obtained. An important aspect to underline is that the specification requires that the link where the auto-configuration process has to be applied, must have multicast capabilities. Unfortunately, layer 2 of IEEE 802.15.4 does not have multicast support, although it support broadcast. The characteristics of IPv6 such as neighbor discovery and address

autoconfiguration requires to use multicast address. When running over IEEE 802.15.4 multicast will be substituted with broadcast messages. The point is that having in mind the well-know constraint we should minimize as far as possible the use of broadcast messages. Recently 6LoWPAN Working Group has published a Internet-draft document [21] with the aim to optimize and reduce the use of broadcast messages exchanged for neighbor discovery purpose.

4 6LoWPAN

In [7] 6LoWPAN is defined as a protocol definition to enable IPv6 packets to be carried on top of Low Power Wireless Personal Area Networks. For LoWPAN are intended networks defined by the IEEE 802.15.2 standard seen before.

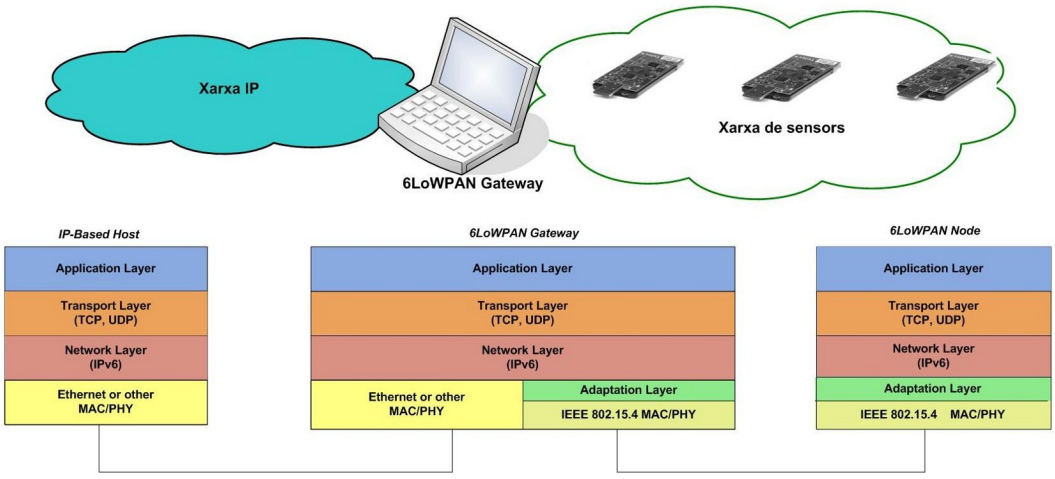

Figure 8: 6LoWPAN architecture

The purpose is to develop personal network, mainly sensor based that can be integrated to the existing well-know network infrastructure by reusing existing and widely used protocols such as IPv6. The version 6 of the IP network protocol has been chosen because it fit for the problematic that characterize a LoWPAN environment such as: the large number of network nodes to address,

the operational environment that very often does not permit manual configuration of nodes that make useful and interesting the address autoconfiguration of IPv6. The use of IP as network layer protocol, permit to reuse mature and wide-used technologies as well as network management tools already in use.

4.1 IPv6 over IEEE 802.15.4: packet encapsulation.

The IPv6 packet has to be carried on 802.15.4 data frames. Exactly the packet has to be encapsulated into the MAC service data unit that as we know has an MTU of 102 bytes. Furthermore, it should be added a link-layer security mechanism (AES-CCM) that imply further overhead that, in the worst case scenario, reduce the MTU to 81 byte. Since the IPv6 has a 40 bytes header and considering also the presence of the transport layer protocol (UDP, TCP), the amount of application data payload that can fit in the MSDU become 33 bytes if using UDP (8 byte header) and 21 using TCP (20 byte header).

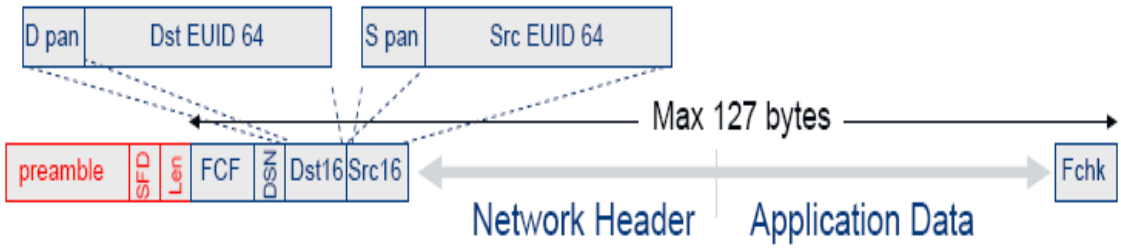

Figure 9: IEEE 802.15.4 frame [9]

Moreover, as we know from the IPv6 specification, the minimum MTU has to be of 1280 byte. This imply the presence of a fragmentation and reassembly adaptation layer at the layer below IP [10]. The presence of this layer will further reduce the amount of bytes for data.

All this problematical issues on data frames composition lead to the need to use an header compression technique. It will increase the application data payload that can be carried in a packet. In [10] is indicated to reuse, with some adaptation, the already existing techniques. According to this specification it will be developed a compression header algorithm suitable for 6LoWPAN environment.

4.1.1 **Adaptation layer**

In [11] is strictly required the presence of an adaptation layer below the network layer in order to, as said before, comply with minimum MTU required by IPv6. The packets are all prefixed by a

LoWPAN encapsulation header, that in its definition in [11] include the presence of one byte length IPv6 Dispatch header and the definition of the followings header fields and their ordering constraints. For dispatch value is intended an octet that specify the type of header that follow the dispatch header. The two leftmost bits are settled to 01 or 00 indicating if there is a 6LoWPAN frame or not. The remaining 6 bits can define up to 64 different dispatch header types. However, only 5 are defined in [11]. In Fig. 10, 11, 12, 13 and 14 are reported the most useful encapsulation format defined in [11]. They initiate the IEEE 802.15.4 MAC protocol data unit (MPDU).

In case of encapsulation of an IPv6 packet we got the following:

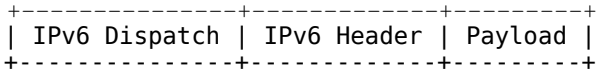

Figure 10: LoWPAN encapsulated IPv6 datagram

In case of compressed IPv6 header:

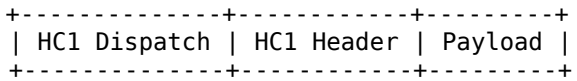

Figure 11: LoWPAN encapsulated IPv6 datagram

If the packet has been fragmented then the header stacks looks like the following:

Figure 12: LoWPAN encapsulated LOWPAN_HC1 compressed IPv6 datagram that requires fragmentation

In the case a is used a mesh addressing, then we include the mesh addressing header on top of the stack. In case of header compression we'll got:

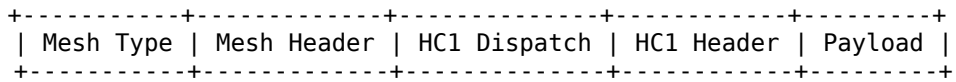

Figure 13: LoWPAN encapsulated LOWPAN_HC1 compressed IPv6 datagram that requires mesh addressing

the mesh addressing type and header will have the following structure:

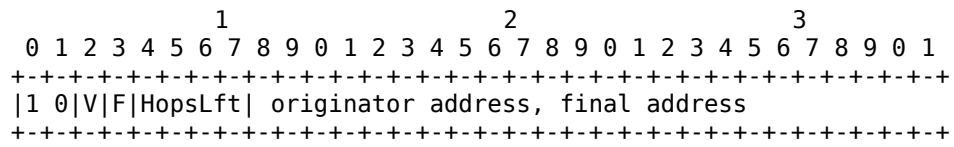

Figure 14: Mesh Addressing type and header

V and F indicates respectively for the originator and final address if they are 16 ('0') or 64 ('1') bit address. Hops Left has the same meaning of the Hops Limit field in the IPv6 header whereas, the originator (source) and final (destination) address are the link-layer address of the two devices involved in the communication.

4.2 Topology

Since we are considering mesh networks, we are implicitly assuming the use of multi-hop routing. But such a routing has to cope with the characteristics of the devices that build up the network. Computational cost and memory usage should be lowered as like as routing overhead. The forwarding operation will be performed by the devices themselves. Each node will act as if it were a router.

In mesh-under, when a node wish to send a packet but there is not a direct link with the destination node, it add the mesh addressing header setting its own link-layer address as originator and the one of the ultimate destination as final address, then it sets the source address in the 802.15.4 header to its own link-layer address, and the address of the next hop as destination address. When a forwarder receive that packet, if it is not the final destination node, then it check its routing table to determine the next hop (setting at that address the destination address field of the 802.15.4 header, and at its own address the source address field) and it reduces the Hops Left field by one (if it is equal to zero it discard the packet). Since the Hops Left field is 4-bit length, we can only have a maximum of 15 hops but in order to accommodate routing for deeper network, the value 0xF ('1111') has been reserved to indicate that an extra byte is included allowing up to 255 hops [7].

The choice of a mesh routing protocol should be done keeping in mind the need to have the smallest possible overhead. Surely it will depend mainly on where and what the 6LoWPAN network will be developed, with this we mean that if the data to collect and send in a packet are composed by few bytes we can have less restriction on the routing protocol.

4.3 Address autoconfiguration

As mentioned before the choice to run IPv6 over IEEE 802.15.4 has been elected also for supporting stateless address autoconfiguration. In the IPv6 chapter has been illustrated how the autoconfiguration works, now we're going to see this functionality applied in the LoWPAN environment.

Following the suggestion given in [21], when a node initialize and wants to join to the network, it has to send a router solicitation message to coordinator. Please note that because of the neighbor discovery procedure of IPv6 [20], the packet should be sent to all-routers IPv6 multicast address. Since 6LoWPAN (to be more precise IEEE 802.15.4) does not support multicast, it will be assumed that the PAN coordinator is also the IPv6 router in order to avoid multicast. When the PAN co-coordinator receives the packet it will unicast the response as Router Advertisement message.

In this environment, the interface identifier is based on the 16 or 64 bit address given to the devices. Since the interface identifier is defined [4] with a length of 64-bit, we got no problem if we are using a 64-bit IEEE extended address but we need to make a proper modification when using short addressing. This modification consists in adding a 48-bit pseudo address to the 16-bit interface identifier to obtain the 64-bit one. The pseudo address formed is formed as follow.

PAN ID (16-bit): zero bits (16): IEEE 16-bit short address

Considering an IEEE 16-bit short address equal to “64” (hex) and PAN ID equal to ”10”(hex) we have the following pseudo address:

00:10:00:00:00:64

How to obtain the 64 bits Ipv6 interface identifier is specified in [4].

5 IP Header Compression

IP Header Compression can be defined as "*the process of compressing excess protocol header before transmitting them on a link and uncompressing them to their original state on reception at the other end of the link*" [12]. Compression is possible since the information carried in a packet flow is redundant. The redundancy may be present when sending packets belonging to a same flow, so the information contained in the IPv6 headers is repeated several times or is already present in other protocol header carried in the packet and it can be consequently inferred from there. Furthermore, the use of header compression implies the transmission of packets smaller than the originals. Consequently, it will drop dramatically the packet loss probability and improve the response time of the network. Header compression makes also a better usage of the available link bandwidth. Considering that communication will be over low-speed link where the available bandwidth is limited, keeping in mind the power constraint that characterize LoWPAN and considering the quite high packet loss in wireless link, arise the necessity of implement an header compression routine to cope with all this limitation.

5.1 Existing techniques

Before considering LoWPAN specifics header compression techniques, a brief explanation of the existing IPv6 header compression standards will be given. Traditionally, header compression is performed over a link between two nodes called compressor and decompressor. It uses the concept of flow context which is a "*collection of information about field values and change patterns of field*

values in the packet header" [12]. Flow context is developed by storing header fields of each packet flow in both compressor and decompressor and maintaining associated to each flow a context identifier. Context is made up by using the first packets of the flow that, for these reasons, should be sent without any form of compression.

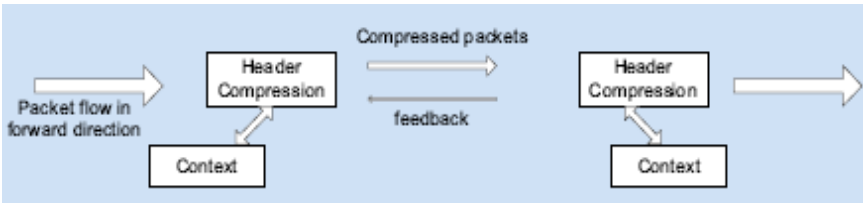

Figure 15: header compression: flow context [12]

IETF specify header compression standards in [13], [14], [15] and [16]. These compression protocols fall into two main categories [12]:

- Internet Protocol Header Compression (IPHC): it is designed for low bit error rate links; It is defined in [14] and [15].
- RObust Header Compression (ROHC): it is designed for wireless links are more suitable for high bit error rate and for links characterized by a long Round Trip Time; it is defined in [16].

5.2 IP header compression for sensor networks

As just mentioned, IP header compression is usually a link compression between two nodes. In a sensor network such hop-by-hop approach has high cost in terms of power consumption especially when routing is done at the network layer. A node, in order to forward a packet, should decompress it and look for routing information in the IPv6 header, then the node should compress the packet and finally forward it. A multi-hop approach would have a better impact on the overall performance since it does not requires to a packet to be decompressed and compressed at each node. However it would introduce further overhead and it would be not useful when the packets belong to a flow requiring congestion control or QoS management. In fact, the latter case requires the decompression of the IPv6 header in order to get the content of traffic and flow label fields.

A Traditional approach might not fit with the constraint of this kind of network. Besides the increased processing operation at each node and the consequent augment of the needed power, there

will be also the problem of the maintenance of the compression states. However it should be valued how a state-full compression affects the performances, they might not be as bad as we can expect. Furthermore, compression context state does not need to be explicitly built. When a device joins a network it shares with the other some state. A compression context state could be constructed reusing those shared state. Giving a further look at the traditional compression protocols [13] [14] [15][16], we note as they work in a time domain in order to develop an appropriate information context. To construct a flow context they have to analyze several packets for each flow (sometimes called as microflow [19]), and in the case of short time flows it become useless. Since in a sensor network can be supposed that most of the packet flows are short time, a compression approach based on time domain should be avoided. In the traditional approach the main goal is the achievement of high compression gain at the cost to have to create overhead to maintain the flow state, for our purpose we could have better results opting out the compression gain as main goal.

5.3 ***IPv6 header compression: IPv6 over IEEE 802.15.4***

In this paragraph are presented the existing IPv6 header compression techniques developed for 6LoWPAN networks. At the time of this writing, only the first presented is part of a RFC [11]. The purpose of this section is mainly to give a description of the state-of-the-art of header compression in 6LoWPAN. No details are given about the header compression of layer 4 protocols since it is out of the scope of this document. Documentation about this header compression is present in [11], [17] and [18].

5.3.1 **LOWPAN_HC1**

The consideration just done about the choice to follow a stateless approach, found a positive feedback from the specification of header compression given in [11] where is assumed "*a very simple and low-context flavour of header compression*". It does not use any context specific to any flow. Regarding the network topology, the header compression should be developed in a mesh network, that should be done starting from a hop-by-hop approach where compressor and decompressor are in "*direct and exclusive communication with each other*"[11]. It is required for a device to be able at least, to send compressed packet via any of its neighbors. A further analysis given in the specification, take in account the limited packet size and suggest to integrate link layer with network layer compression. It is also required the use of bit padding in order to avoid any infringement of the packets boundaries. As said before, 6LoWPAN header compression does not keep any flow state but it "*relies on information pertaining to the entire link*"[11]. So, any header

value that is repeated trough all the flow or that can be inferred from lower layers can be compressed.

Considering the IPv6 header as shown in Fig.5, the common case for 6LoWPAN communications can be listed as:

- IP Version: is 6 for all packets
- Traffic class and flow label: are zero
- Payload length: it can be inferred from layer 2 or from the “datagram size” field in the case we have a fragmented packet.
- Next header: can be UDP, TCP or ICMP, so we can only use 2 bits only.
- Source and Destination address: are link-local (that is, the IPv6 interface identifier can be inferred from source and destination address present in layer 2).

All this field can be compressed down to 1 byte. In [11] is mandatory to not compress the hop limit field that always need to be carried in full. The resulting compressed header would be 2 byte length instead of the 40 bytes of the uncompressed header. Compressed header format for LOWPAN_HC1 is showed in Fig.16.

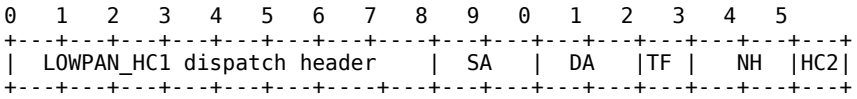

Figure 16: 2 bytes encoding LOWPAN_HC1 format

The HC1 encoding will occupy 1 octet. Here we report the meaning of the fields composing it as specified in [11]:

- IPv6 source address (SA):

  00: PI (Prefix carried in-line), II (Interface Identifier carried in-line)

  01: PI, IC (Interface identifier elided)

  10: PC (Prefix compressed), II

  11: PC, IC

- IPv6 destination address (DA):

  00: PI, II

  01: PI, IC

10: PC , II

11: PC, IC

- Traffic Class and Flow Label (TF)

  0: not compressed; full 8 bits for Traffic Class and 20 bits for Flow Label are sent.

  1: both field are zero.

- Next Header (NH)

  00: not compressed; full 8 bits are sent.

  01: UDP.

  10: ICMP.

  11: TCP.

- HC2 encoding (HC2)

  0: No more header compression bits.

  1: HC1 encoding immediately followed by more header compression bits per HC2 encoding format. Bits 5 and 6 determine which of the possible HC2 encodings apply (e.g., UDP, ICMP, or TCP encodings).

The standard specify also how to encode the UDP header fields defining a HC_UDP format, that permit to have an UDP compressed header of 4 bytes instead of 8 bytes. Fig.17 shows how it would be structured the MAC frame in case of LOWPAN_HC1 is used.

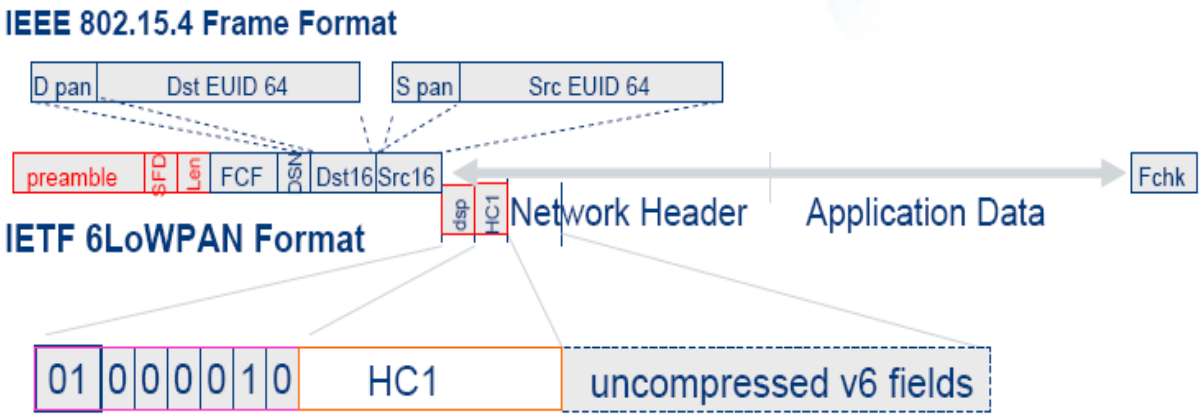

Figure 17: Frame format

5.3.2 **LOWPAN_HC1g**

This compression scheme regards 6LoWPAN communicating with global unicast address. It receives completely the specification given in [11] in the sense that it uses the common case defined for LOWPAN_HC1. LOWPAN_HC1g is not alternative to LOWPAN_HC1 but is complementary to it. It would extend the applicability of compression to different kinds of addresses since LOWPAN_HC1 compression can be only applied to link-local address [18]. Link-local address are commonly used only for configuration protocols or communication between nodes when no router exists. In the case a nodes has assigned a global unicast address it is will be not required carry in-line the 128 bits composing it. This permit to save 32 bytes of link-layer MTU [18]. Global addresses are intended for addressing any node connected to the IPv6 network.

The authors of [18] starts form the point that using of a global unicast address would give to 6LoWPAN the almost full capabilities that might receive adopting IPv6, that is end-to-end communication across different PAN and external IP networks. Communication outside the PAN such as monitoring and control applications would receive benefits by compression global unicast address. If link-local address are used, it would be required the presence of an intermediate point such as a translation gateway in order to forward the communication in a external point of the network. With global unicast address such gateway becomes meaningless. LOWPAN_HC1g compression came from the fact that "*To support compression of global unicast address, LOWPAN_HC1g assumes that a PAN is assigned on compressible 64-bit global IP prefix. When either the source or destination address matches the compressible IP prefix, the prefix can be elided*"[18].

The address compression encoding has four possible possibilities:

- Full 128-bit Address: full 128-bit IPv6 address carried in-line.
- Compressed 64-bit Address: 64-bit prefix elided, 64-bit interface identifier are carried in-line; the 64-bit prefix is derived from the 64-bit prefix assigned to the PAN.
- Compressed 16-bit Address: 64-bit prefix elided least significant 16 bits of 64-bit interface identifier in-line; the 64-bit prefix is derived from the 64-bit prefix assigned to the PAN; the upper 48 bits of the interface identifier are assumed to be all zeros.

- Full 128-bit IPv6 address elided: the prefix identifier is derived from the 64-bit global prefix assigned to the PAN; 64-bit interface identifier is derived either from the LoWPAN Mesh Addressing header or from the IEEE 802.15.4 link header; link-layer addresses are used to

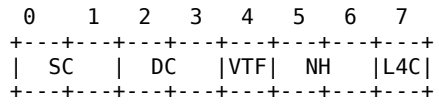

Figure 18: LOWPAN_HC1g Header Compression Encoding

  derive the 64-bit interface identifier is a short address, they describes the lower 16 bits of the interface identifier; the upper 48 bits are assumed to be all zeros.

The encoding format is the following:

- SC: IPv6 source address compression (bits 0 and 1):

  00: full 128-bit address.

  01: compressed 64-bit address.

  10: compressed 16-bit address.

  11: full 128-bit IPv6 address elided.

- DC: IPv6 destination address compression (bits 2 and 3):

  00: full 128-bit address.

  01: compressed 64-bit address.

  10: compressed 16-bit address.

  11: full 128-bit IPv6 address elided.

- VTF: Version, Traffic Class, and Flow Label (bit 4):

  0: full 4 bits for Version, 8 bits for Traffic Class and 20 bits for Flow Label are carried in-line.

  1: Version, Traffic Class, and Flow Label are elided and assumed to be zero.

- NH: Next Header (bits 5 and 6):

  00: Next header field carried in-line.

  01: UDP.

10: ICMP.

11: TCP.

- L4C: Layer 4 Compression (bit 7):

  0: Full layer 4 header is carried in-line; HC2 encoding does not precede the layer 4 header.

  1: Layer 4 header is compressed; HC2 encoding follows the IP header but precedes the layer 4 header; currently, this indicator only supports UDP compression.

5.3.3 **LOWPAN_IPHC**

The third proposal of the 6lowpan work-group [24] has been thought as an improvement of LOWPAN_HC1. It would support a wider range of communication paradigm such as:

- Mesh-under and route-over configurations.
- Communication to nodes internals and externals to the 6LoWPAN network.
- Multicast communication.

The first difference we notice is in the information at which the format relies in order to perform the header compression. In LOWPAN_IPHC is the "*information pertaining to the entire 6LoWPAN network*" [24] while in LOWPAN_HC1 was the information pertaining to an entire link. In the definition of the common case we find important differences. The most relevant is the given possibility to compress the hop limit field. As a matter of fact, if the source node uses a well-known default values for the hop limit fields or it uses a mesh header, it would not be necessary to carry hop limit in-line. Further differences pertain to the address field compression, LOWPAN_IPHC does not specify as LOWPAN _HC1 do, that the common case for a communication is with link-local addresses. The purpose is to extend the use of IPv6 header compression to multicast and global address used in route-over networks where forwarding occurs at IP or when communicating with nodes externals to the 6LoWPAN network. Focusing on the common case definition for address, we have that: "*addresses assigned to 6LoWPAN interfaces will be formed using the link-local prefix or a single routable prefix assigned to the entire 6LoWPAN network; addresses assigned to 6LoWPAN interfaces are formed with an IID derived directly from either the 64-bit extended or 16-bit short IEEE 802.15.4 addresses*" [24].

At the time of this writing LOWPAN_IPHC is at its fourth update. The first version received the

suggestions contained in the “Context-based Header Compression for 6LoWPAN” internet draft [17] and proposed to compress address by using a state-full compression. In [17] was proposed to add a bit in the encoding format to specify whether or not a context-based compression is used, if so an additional byte would be added in order to specify the context used. The first update to LOWPAN_HC1 added two address context (one for source and the other for destination), each one of two bit specifying only the case for link-local address. The third version changed the way to specify the context. By using the Context Identifier Extension (Fig.19), it is possible define up to 16 contexts (eight for source and eight for destination). The extension can be added or not to the encoding format depending on the value of the two bit field DDF (Destination Dependant Field). This additional byte identifies the prefix when it is elided by the address compression, each context specify a prefix. The four rightmost (Destination Address Context) bits indicate the prefix for destination address; the four leftmost (Source Address Context) bits are for the source address.

LOWPAN_IPHC encoding uses 2 octets in a link-local communication and 7 octets when

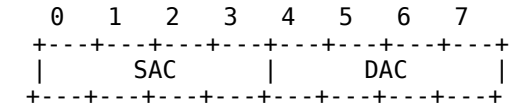

Figure 19: Context Identifier Extension.

communicating over multiple IP hops (2-octet LOWPAN_IPHC, 1-octet Hop Limit, 2-octet Source Address, 2-octet Destination Address).

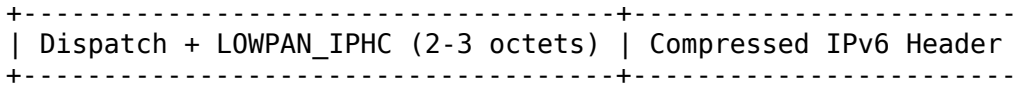

Figure 20: LOWPAN_IPHC Header

The Version field of the IPv6 header is elided as it assumes every time the same value. IPv6 Payload Length field must always be elided and inferred from lower layers using the 6LoWPAN fragmentation header or the IEEE 802.15.4 header.

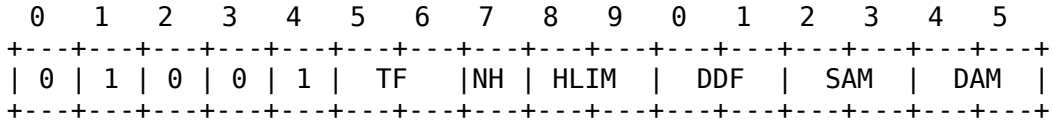

Figure 21: Encoding Format

The encoding format (Fig.21) has the following meaning:

- TF: Traffic Class, and Flow Label (bit 5,6):

  00: 4-bit Pad + Traffic Class + Flow Label (4 bytes).

  01: ECN + 2-bit Pad + Flow Label (3 bytes).

  10: Traffic Class (1 byte).

  11: Version, Traffic Class, and Flow Label are compressed.

- NH: Next Hop (bit 7):

  0: Full 8 bits for Next Hop are carried in-line.

  1: Next Hop is elided and the next header is compressed using LOWPAN_NHC.

- HLIM: Hop Limit (bit 9 and 10):

  00: The Hop Limit field is carried in-line.

  01: The Hop Limit field is compressed and the hop limit is 1.

  10: The Hop Limit field is compressed and the hop limit is 64.

  11: The Hop Limit field is compressed and the hop limit is 255.

- DDF: Destination Dependant Field (bits 11 and 12):

  00: Destination and Source are Global or Unique Local Addresses

  01: Destination and Source are Global or Unique Local Addresses and one additional context octet extends the LOWPAN_IPHC field to disambiguate an elided prefix or address described by the SAM or DAM fields.

  10: Destination and Source are Link Local Addresses.

  11: Destination is a Multicast Address; source is either the unspecified address or a Link Local Address.

- SAM: Source Address Mode (bits 13 and 14):

  00: All 128 bits of Destination Address are carried in-line.

  01: 64-bit Compressed IPv6 address.

  10: 16-bit Compressed IPv6 address.

11: All 128 bits of Destination Address are elided.

- DAM: Destination Address Mode (bits 15 and 16):

  00: All 128 bits of Source Address are carried in-line.

  01: 64-bit Compressed IPv6 address.

  10: 16-bit Compressed IPv6 address.

  11: All 128 bits of Source Address are elided.

Respect to multicast address, they might be compressed down to 24, 16, or 8 bits. LOWPAN_IPHC supports compression of the Solicited-Node Multicast Address (FF02::1:FFXX:XXXX) as well as any IPv6 multicast address where the upper 104 bits of the multicast group identifier are zero (FFXX::XXXX). The encoding format also compressed link-local multicast addresses of the form (FF02::00XX) down to a single byte. The compressed form only carries least-significant bits of the multicast group identifier.

5.4 ***Work Proposal***

As we know we are dealing with resource-limited devices so extreme attention has to be paid when designing the code in order to avoid useless resource consumptions. The main design principles can be listed as follows:

- The application should minimize as much as possible the memory usage of TelosB motes; This will permit to reduce the power consumption, to have fast operation on memory and finally to save space for other possible application on the motes.
- The code should be easily readable and well structured in order to permit further improvement or modifications.

Defined the design principles, there will be illustrated the proposed algorithm. By following the considerations made in the previous chapters, the header compression algorithm should be:

- Stateless.
- Able to work with unicast and multicast address type.
- Suitable for link-local or global address.
- Suitable for mesh-under and route-over configurations.

Among the presented compression algorithms, the one that best adapt to requirements is

LOWPAN_IPHC. At the time of developing the original proposal, LOWPAN_IPHC internet draft was at its first version [24]. The proposed algorithm has been inspired on it and has been thought as an improvement. However, the implemented compression algorithm has not been based on our proposal but on the fourth update of LOWPAN_IPHC [26]. The reasons of this choice will be explained after presenting both algorithm, the proposed and the LOWPAN_IPHC-04. Hereafter we will refer to fourth update of LOWPAN_IPHC as IPHC-04.

5.4.1 Proposal

Considering the IPv6 header as reported in Fig.5, for a 6LoWPAN communication will be assumed as common the following:

- Version is 6; Traffic Class and Flow Labels are 0; these fields can be elided from the header since they will assume every time the same values.
- Payload length can be inferred from lower layers.
- Hop Limit will be set to a well-known value by the source.
- Addresses assigned to 6LoWPAN interfaces will be formed using the link-local prefix or a single routable prefix assigned to the entire 6LoWPAN network.
- Addresses assigned to 6LoWPAN interfaces are formed with an IID derived directly from either the 64-bit extended or 16-bit short IEEE 802.15.4 addresses.

The compressed header will have the format reported in Fig.22:

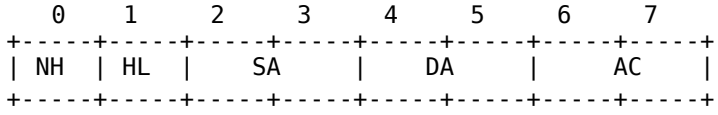

Figure 22: Proposed encoding format

In case of link-local communication, the compressed header will be of 1 byte. When the communication is over multiple IP hops the compressed header will be of 6 bytes (1 byte for the compressed header, 1 byte for Hop limit, 2 bytes for Source Address, 2 bytes for Destination Address).

With respect to the fields composing the encoding format, the meaning is listed as the following:

- NH: Next Hop (bit 0).

0: Full 8 bits for Next Hop are carried in-line.

1: Next Hop is elided and the next header is compressed using LOWPAN_NHC.

- HLIM: Hop Limit (bits 1).

  0: All 8 bits of Hop Limit are carried in-line.

  1: All 8 bits of Hop Limit are elided; if the IPv6 destination address is not assigned to the receiving interface, Hop Limit is assumed to be 64; if the IPv6 destination address is assigned to the receiving interface, Hop Limit is assumed to be 1.

- SA: Source Address (bits 2-3).

  00: All 128 bits of Source Address are carried in-line.

  01: 64-bit Compressed IPv6 address.

  10: 16-bit Compressed IPv6 address.

  11: All 128 bits of Source Address are elided.

- DA: Destination Address (bits 4-5):

  00: All 128 bits of Destination Address are carried in-line.

  01: 64-bit Compressed IPv6 address.

  10: 16-bit Compressed IPv6 address.

  11: All 128 bits of Destination Address are elided.

- AC: Address Context (bits 6-7); it identifies the compression context when the source address is compressed; it can define up to 4 contexts; each context will represent the elided network prefix.

5.4.2 **IPHC-04**

The basics of LOWPAN_IPHC compression have been already reported in 5.3.3 paragraph. Here we will concentrate on changes done in the newest version [26]. Differences fall mainly in the pattern of the encoding format. It has been improved and better clarified the usage and the meaning of the Context Based Compression. The new format is showed in Fig.23:

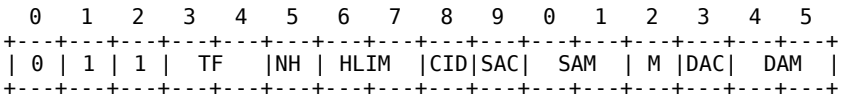

Figure 23: IPHC-04 encoding format

The bits used to specify the dispatch value have been reduced from five to three. In that way, it has been saved space for the specification of address compression, allowing an increase of flexibility and efficiency in the encoding octet. The remaining five bites of the dispatch octet, that is Traffic Flow (TF), Next Header (NH) and Hop Limit (HLIM) fields, keep the same definition reported in 5.3.3. The first bit of the encoding byte indicates the presence of the Context Identifier Extension (CID). If present (CID = 1), it must follow the dispatch and encoding bytes. It is important to stress that the possibility to compress global address would give to a LoWPAN network, full capabilities of the Ipv6 protocol adoption such as end-to-end communication across different PAN and external IP Networks. In fact, by adding the additional CID byte, it can be defined up to 16 compression context (8 for source address and 8 for destination). In that way we might compress global addresses also when communicating with externals IP networks by associating for each different network prefix an index in the CID byte. The CID extension byte format is illustrated in Fig. 24.

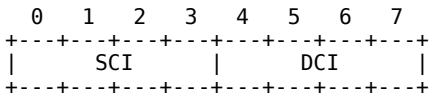

Figure 24: CID octet format

The fields of CID have the following meaning.

- SCI: Source Context Identifier; it identifies the prefix that is used when the IPv6 network prefix of source address is elided.
- DCI: Destination Context Identifier; it identifies the prefix that is used when the network prefix of destination address is elided.

The rest of the encoding pattern specifies the kind of address and how it has been compressed. The meaning and the possible content of each field is listed as follow:

- SAC: Source Address Compression.

  0: Source address compression uses stateless compression.

  1: Source address compression uses state-full, context-based compression.

- SAM: Source Address Mode.

If SAC = 0:

00: 0 bits; address is the unspecified address.

01: 64 bits; first 64-bits of the address are elided; the value of those bits is the link-local prefix padded with zeros; the remaining 64 bits are carried in-line

10: 16 bits; first 112 bits of the address are elided; the value of those bits is the link-local prefix padded with zeros; the remaining 16 bits are carried in-line.

11: 0 bits; address is fully elided; the first 64 bits of the address are elided; the remaining 64 bits are computed from the link-layer address as defined in [11].

If SAC = 1:

00: 128 bits; full address is carried in-line.

01: 64 bits; first 64-bits of the address are elided; the value of those bits is taken from the context and padded with zeros; the remaining 64 bits are carried in-line.

10: 16 bits; first 112 bits of the address are elided; the value of those bits is taken from the context and padded with zeros; the remaining 16 bits are carried in-line.

11: 0 bits; address is fully elided; the first 64 bits are taken from the context; the remaining 64 bits are computed from the link-layer address as defined in [11].

- M: Multicast Compression.

  0: Compressed destination address is not multicast.

  1: Compressed destination address is multicast.

- DAC: Destination Address Compression.

  0: Destination address compression uses stateless compression.

  1: Destination address compression uses state-full, context-based compression.

- DAM: Destination Address Mode.

- If M = 0 and DAC = 0; any elided prefix bits are the link-local prefix padded by zeros. When M = 0 and DAC=1, any elided prefix bits are taken from the context and padded by zeros.

  00: 128 bits; full address is carried in-line.

  01: 64 bits; first 64-bits of the address are elided; the remaining 64 are carried in-line.

10: 16 bits; first 112 bits of the address are elided; the remaining 16 are carried in-line.

11: 0 bits; address is fully elided; the first 64 bits of the address are elided; the remaining 64 bits are computed from the link-layer address as defined in [11].

- If M = 1 and DAC = 0:

  00: 48 bits; address takes the form FFXX::00XX:XXXX:XXXX.

  01: 32 bits; address takes the form FFXX::00XX:XXXX.

  10: 16 bits; address takes the form FF0X::0XXX.

  11: 8 bits.; address takes the form FF02::00XX.

- If M = 1 and DAC = 1:

  00: 128 bits; full address is carried in-line.

  01: 48 bits; address takes the form FFXX::XX[plen]:[prefix]:XXXX:XXXX; the values of plen and prefix are taken from the specified context.

  10: reserved.

  11: reserved.

5.5 Evaluation of header compressions algorithms

Bearing in mind the reason that made necessary the use of IPv6 header compression and the possible scenarios of 6LoWPAN communication, we have chosen to implement the IPHC-04 header compression algorithm instead of the proposed one. Analyzing the possible application fields of 6Lowpan networks, arise the necessity to use traffic engineering tools that can be specified by an optimal use of Flow label and Traffic fields. In the proposal these fields were fully elided with the consequent lost of control over the packet flow. For instance we can not have any mechanism of path selection as it could be to route a flow over the minimum energy consumption path. Furthermore, there would be lost possibilities to reduce packet loss in case of congestion making impossible to guarantee QoS for critical network application. In conclusion, Traffic and Flow Label fields should not to be elided by default but encoded as IPHC-04 specifies. Finally, the Hop Limit field can be incremented of 1 bit assuming in this way the same definition and uses proposed in the IPHC-04 algorithm.

6 Implementation

This section is intended for present and describes the hardware and the protocol stack used to implement the IPv6 header compression. After introducing those components, it will be presented the implementation of IPHC-04.

6.1 Hardware Platform

The hardware platform used for this project is the TelosB motes. This platform has been originally developed at UC Berkeley and is currently produced by the Crossbow Technology Company.

Figure 25: TelosB mote

As defined in [22], "*Crossbow's TelosB mote is an open source platform designed to enable cutting-edge experimentation for the research community. The TelosB bundles all the essentials for lab studies into a single platform including: USB programming capability, an IEEE 802.15.4 radio with integrated antenna, a low-power MCU with extended memory and an optional sensor suite*".

It has a 16-bit RISC MCU at 8 MHz and 16 registers. The platform offers 10 kB of RAM, 48 kB of flash memory and 16 kB of EEPROM. Requiring at least 1.8 V, it draws 1.8 mA in the active mode and 5.1 μA in the sleep mode. The RF tranceiver draws 23 mA in receive mode, 21 μA when idle and 1 μA in sleep mode. The MCU has an internal voltage reference and a temperature sensor. Further sensors available on the platform are a visible light sensor (Hamamatsu S1087), a visible to IR light sensor (Hamamatsu S1087-01) and a combined humidity and temperature sensor (Sensirion SHT11). The mote is powered by two AA batteries or it can be plugged into the USB port for programming and communication without the need of batteries.

6.2 Protocol Stack

The software component has been developed on TinyOS 2.1 [23], which is an open-source operating system designed for wireless embedded sensor networks. The implementation of 6LoWPAN functionalities have been based on b6loWPAN protocol stack developed by the Berkeley Wireless Embedded Systems (WEBS) [28]. It has been released as TinyOS contribution. Currently it is at its fourth version and has changed the name to “Berkeley IP implementation for low-power networks” (*blip*). When we start implementing the header compression, b6loWPAN was at first release so we have kept working on this version.

6.2.1 TinyOS

TinyOS [23] [28] is an open-source operating system written in nesC programming language, it is designed for networks embedded systems. The core OS requires only 400 bytes of code and data memory, it has no kernel space and no process management. Memory is allocated statically and there is no memory management or virtual memory. It features a component-based architecture which enables rapid implementations of applications while minimize code size as required by the memory constraints of sensor networks. Thanks to this architecture TinyOS can provide a set of reusable components. An application connects components using a wiring specification independent from the components implementation. The TinyOS concurrency model enables fine-grained power management because the event-driven execution model implemented. There are two kinds of sources of concurrency: tasks and events. Tasks are a postponed computation mechanism. They run to completion and do not compete each other. Components can post tasks. In order to ensure low task execution latency, individual tasks must be short time. When developing task requiring a long time of execution, they should be spread across multiple tasks. Events run to completion, but may preempt the execution of a task or another event. Events signals the completion of a split-phase

operation or the came to fruition of an external event (i.e message reception). The fact that tasks executions are non-pre-emptively, means that TinyOS has no blocking operations. This is a fundamental property for real-time applications especially if installed on devices dislocated in environments where human intervention can not be done frequently. All long latency operations are split-phase, that is the operation request (command) and its completion (event) are separate functions. The packet send command is an example of split-phase operation. A component call the send command to transmit a radio message, when the message is send the communication component signals it with an event. The LED toggling can be an example of a non split-phase operations. A component request the toggling with a command but its realization is not signalled by an event. TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools. All of which can be used as-is or be further refined for a custom application. However, it not include implementation of protocols such as IP, UDP or IEEE 802.15.4

TinyOS started as a collaboration between the University of Berkeley in co-operation with Intel Research, nowadays it is developed by an international consortium, the TinyOS Alliance. At the present time it has been released three versions, 1.1, 2.0 and 2.1. The 2.0 and 2.1 version are not backwards compatible with the 1.1 initial version and 2.1 has some functionalities not compatibles with 2.0. In this project, the 2.1 version has been used since b6loWPAN requires it.

6.2.2 nesC Overview

nesC [28] is a programming language for networked embedded systems. It can be seen as a dialect of C programming language because the reduced expressibility that allows to write code in a safer way. It specifies a concurrency model based on task and events and has data races detection at compile time. To easy detect race conditions, the concurrency model features the distinguish of nesC code into synchronous and asynchronous. The former is defined in [28] as "*code that is reachable from at least one interrupt handler*", the latter as "*code that is only reachable from tasks*". Considering the compiling of nesC programs, they are not compiled "piece-by-piece" but as a whole. The compiler creates a C program as output. Then, it will be compiled by a gcc compiler depending on the platform we are working on (msp430 or avr). In that way we could think to nesC compiler as a pre-processor. Furthermore, nesC can be considered as a "static" language meaning that there is no dynamic memory allocation and no dynamic linking. This allows simplifying and increasing the accuracy of programming analysis and optimization.

Applications written in nesC language are based on interfaces and components. A component

provides and uses interfaces. An interface is a set of commands and events. The former are implemented by the providers while the latter by the users. Events and commands allow having a bidirectional communication between components. Fig.26 shows the send interfaces for IP and UDP protocols.

Considering the send interfaces we see how a split-phase operations is obtained, that is having in the same interface command requests and event response. A component is of two types: module or configuration. A module implements interfaces while a configuration wires modules together. Wiring is provided connecting modules together by their interfaces. Fig. 27 illustrates the relations between components and interfaces.

```
interface IPSend {
 command error_t send(ip_msg_t *msg, uint16_t len);
 event   void    sendDone(ip_msg_t *msg, error_t error);
}
interface UDPSend {
 command error_t send(struct sockaddr *dest,ip_msg_t *msg, uint16_t len);
 event   void    sendDone(ip_msg_t *msg, error_t error);
}
```

Figure 26: IP and UDP send interfaces

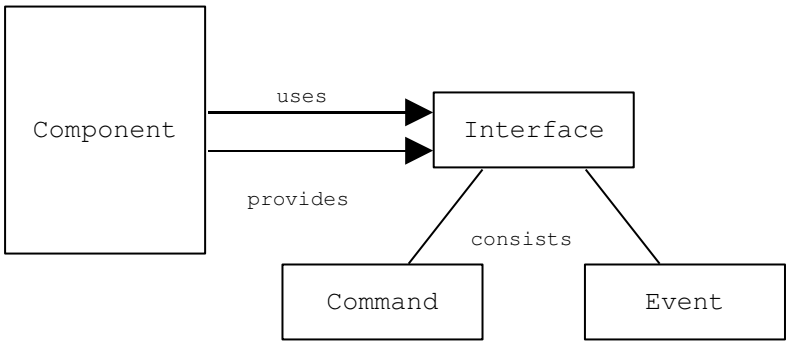

Figure 27: components and interfaces relationship

6.2.3 b6loWPAN

b6LoWPAN [29] adds IPv6 support to TinyOS, it provide address stateless auto-configuration, multi-hop routing, and fragmentation for an MTU of 1280 bytes. Standard internet tools like ping6, nc6, and tracert6 can be used to debug installations using b6lowpan, and applications may use UDP as the transport layer. Moreover, it uses LOWPAN_HC1 header compression and includes IPv6

neighbor discovery, default route selection, point-to-point routing, and network programming support. Packet forwarding is done at network layer meaning that any incoming packet, before to be forwarded, need to be decompressed and compressed again. This would imply that the implementation of header compression will have a hop-by-hop behavior. The implemented Packet fragmentation requires the reconstruction of fragmented packet at each hop, however this characteristic has changed from the second update. b6LoWPAN does not support Mesh Addressing. Fig. 28 shows the architecture of b6LoWPAN.

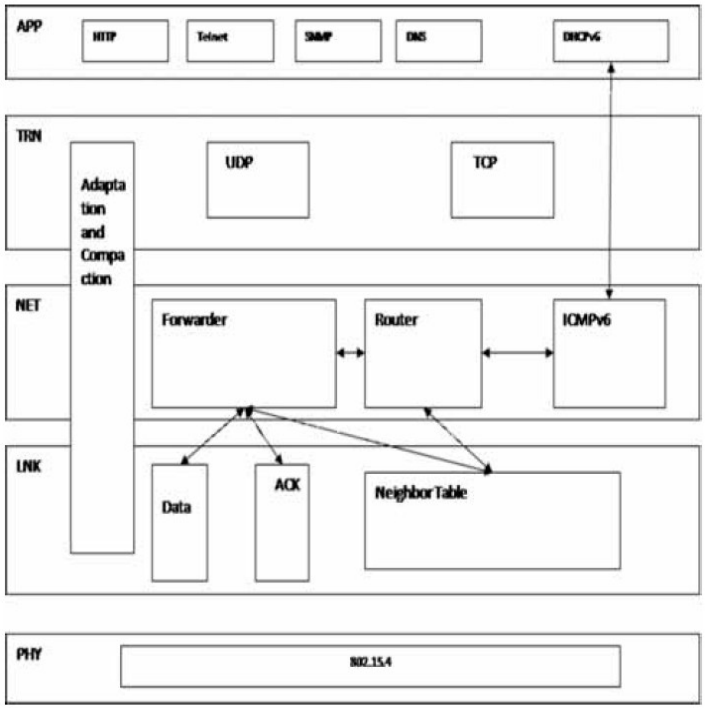

Figure 28: b6loWPAN architecture

b6loWPAN Architecture

The structure of b6loWPAN can be divided into two main parts. The first is written in C programming language and is implemented in the folder `b6lowpan/support.` In the subfolder `sdk/c/lib6lowpan,` are contained the 6lowpan header files defining the data types and structures used in b6lowpan. It also contains the fragmentation, reassembly and IPv6 header compression functions. In `sdk/c/lib6lowpan/tunnel` is implemented a driver and a router. They are needed to add IPv6 over 802.15.4 interface to a Linux environment. The driver uses periodical reports send by the motes in order to maintain the topology within the PAN. The second part of b6loWPAN is written in nesC

programming language and it is implemented in `b6lowpan/tos` folder. The subdirectory `/lib/net/b6lowpan` contain the mote-side code including the implementation of the routing components, ICMP functionalities (i.e router solicitations and requirements), UDP and IP functionalities. Here, there is the root component of b6lowpan that is `IPDispatchP.nc`. It provides the IP send and receive interface and the handling of the incoming or out-coming packets. It is the implementation of the adaptation layer required to run IPv6 over 802.15.4. Finally, in `tos/chips/cc2420` are implemented the radio components and the active message layer. The applications are contained in `b6lowpan/apps` folder.

Sending a packet

The packet send process is initiated by the application calling the `send()` command of the `UDPSend` interface. After forming and filling the UDP header, it will be called the `send()` command of the `IPSend` interface provided by the `IPDispatchP.nc` module. Here, the IPv6 header fields are filled and, as results, we got the complete IPv6 packet ready to be send. The packet is then appended to the `enqueueSend()` and scheduled to the `sendTask()` to send it. The queue is defined as global variable. When the packet is scheduled to be send, in the `sendTask()` it will be called the `getNextFrag()` function implemented in the `lib6lowpanFrag.c` library. That function will add any necessary 6LoWPAN optional headers and will deal with fragmentation if the payload length exceeds the MTU size of 102 bytes. The header compression function, named `packHeaders()`, is called in this function. Header compression and decompression is implemented in the `lib6lowpanIP.c` source file. After the packet processing we will obtain the IEEE 802.15.4 frame that will be passed down the MAC layer by calling the `send()` command implemented in the `IEEE154Send` interface provided in `CC2420MessageP.nc` module. Here it will be called the `send()` interface provided by the TinyOS's `UniqueSendP.nc` module. After TinyOS sends the frame by the `AMSend.send()` , it will be signalled the `sendDone()` event. The sendDone chain goes backwards until it reach the handler function of this event, implemented in `IPDispatchP.nc()` module as `IEEE154Send.sendDone()` event. Here, the first packet in the send queue will be checked, if it was delivered successfully, the sendDone event will be signalled to the upper layers and the packet will be removed from the queue. If the packet has not reached its destination, it will be send again according to the fixed number of send retries. The sequence diagram in Fig. 29 shows the send process in b6loWPAN.

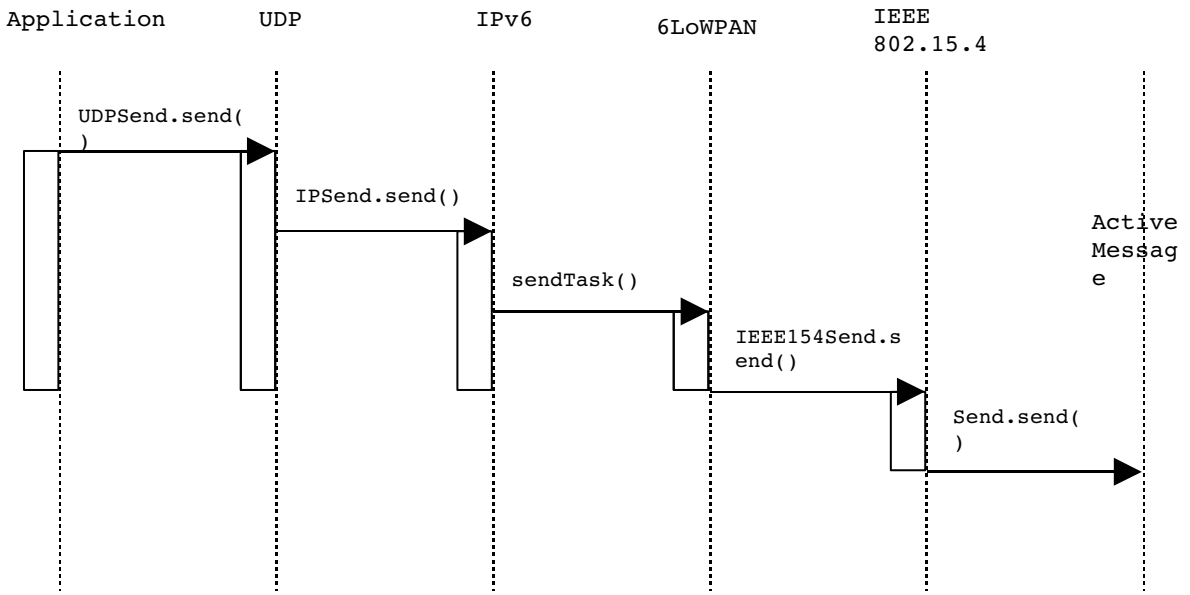

Figure 29: Sending a packet.

Receiving a packet

The receiving process of packets is splitted into different functions, each one handling a different protocol and layer. When a 802.15.4 data frame is received, an event is signalled by the receive interface of TinyOS to the Active Message layer of b6lowpan implemented in `CC2420MessageP.nc`. Then, the message is passed to the 802.15.4 receive interface instance *`IEEE154Receive.receive()`* implemented in `IPDispatchP.nc`. Here, 6LoWPAN dispatch types are checked the in order to determine which optional headers are presents in the packet. These are the mesh addressing, the fragment header and the IPHC compression header. Fragments are reassembled by calling the `handlefrag()` function. The received packet, if compressed, is passed to the compression/decompression routine implemented in `lib6lowpanIP.c` to create the IPv6 packet. Then, the IPv6 packets is passed to a void function called `receive()` that checks if the packet has reached its destination or it need to be forwarded. In case of forwarding, the packet is enqueued and passed to the send handling, if it has to be consumed an event is signalled to `IPReceive.receive()` interface instance. Depending on the value of the next header field of the IPv6 header, the packet should go to the specific transport layer handler. However, b6loWPAN implements only the UDP protocol. The receive events for UDP is implemented in the `UPPP.nc` module. Before signaling the receive event to the UDP interface, the checksum is verified and finally the received UDP datagram is passed to the instance of the `UDPReceive` interface used by the application. The sequence diagram in Fig. 30 shows the receive process in b6loWPAN.

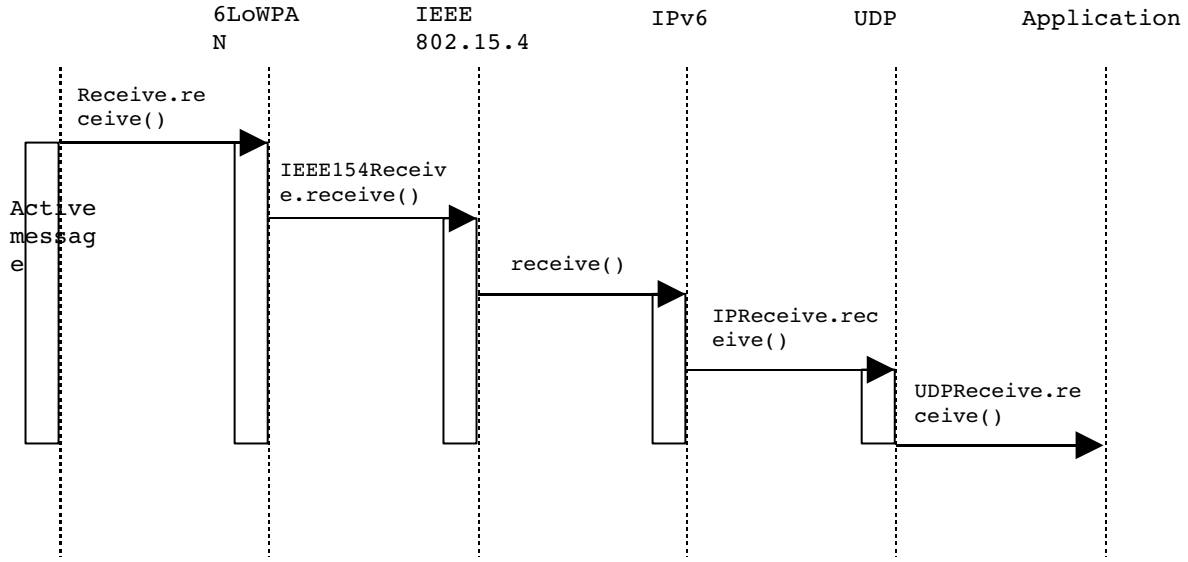

Figure 30: receiving a packet

6.3 lib6lowpanIP.c: The IPv6 header compression implementation

IPv6 header compression has been implemented in a ANSI-C source file called `lib6lowpanIP.c`. Its job is mainly performed by two functions: `packHeaders()`and `unpackHeaders()`. Both are declared in the `lib6lowpanIP.h` header file as shown in Fig.31. Ipv6 header is packed into a buffer pointed by the second argument of the `packHeaders()` function that is `*buf`. After compressing the header, the function returns a pointer referenced where the writing has been stopped. The compressed IPv6 header is passed to the decompression function and it will be unpacked into a buffer pointed by the second argument of `unpackHeaders()`, that is `*dest`. The function will return the length of the byte written to this buffer.

```
uint8_t *packHeaders(ip_msg_t *msg, uint8_t *buf, uint8_t len);

uint8_t *unpackHeaders(packed_lowmsg_t *pkt, uint8_t *dest,
uint16_t len);
```

Figure 31: IPv6 header compression function declaration

`PackHeaders()` is called in the `getNextFrag()` function of the `lib6lowpanFrag.c` source file. As arguments are passed the IPv6 no compressed packet, an array with length equal to the sum of IPv6 and UDP header size and finally the length of the packet. The first two arguments are passed

by reference while the length is the passed by value. `UnpackHeaders()` is called by the `IEEE154Receive` interface provided by the `IPDispatchP.nc` module. As first arguments, it is passed the memory address where the received MAC frame is stored, then a pointer to the IP header and finally the length of the frame payload. The function return a pointer referenced to the position of the first byte of the data payload.

Compressing the IPv6 Header

After initialize the dispatch and the encoding bytes, it is checked if the version of the IP packet correspond to IPv6. If so, the dispatch value relative to the IPHC-04 compression is written into the dispatch byte. Then, it is controlled which kind of destination address is carried in the IPv6 header. This operation is needed in order to decide if the CID extension byte has to be appended or not to the IPHC-04 encoding format. If the address results to be global, then CID is initialized and the corresponding flag is activated. In order to know the IPv6 address we are dealing with, it is used a function called `cmpPfx()`. It compares the current network prefix of the address with the references ones. The network prefixes that can be used in a 6LoWPAN communication are declared in arrays as global variables in `lib6lowpanIP.c`. As we know, they can be link-local, multicast or global prefixes.

The first IPv6 header fields to examinees are Traffic, Flow, and Next header fields. Although the UDP header is present in the packet, the Next header filed value does not specify the presence of UDP as transport layer. Consequently Next Header can not be compressed because of the functioning of b6loWPAN. In fact, the next header fields contains the value referenced to a hop-by-hop extension no-standard header defined only in b6loWPAN, it contains the information needed by the routing protocol to route the packets in the network. It is called Source Routing header and it has a length of 6 byte. The description of the routing mechanism of b6loWPAN is outside the scope of this work. As required in the IPHC-04 specification, the Hop Limit field should be carried in-line if the destination address has a global scope. If it can be compressed, a switch case control structure compares the content of this field against three possible default values. Once Hop Limit is compressed we reach the end of the dispatch byte. Now we need only to compress the source and destination address and, implicitly, to fill the encoding byte and eventually the CID extension byte. First, it is controlled if the CID flag is activated or not, if active the CID bit of the encoding octet is settled to 1. The next field to fill regard the types of the used source address. Here, we have only two possibilities that is link-local or global address. A flag is used to keep track of the source address

type and it will be used when compressing the address. The same thing will be done for the destination address where will be included the possibility to use multicast address. Address compression is implemented in a dedicated function called `packAddress()` that take as arguments the address flag, the buffer we are writing into and the address contained in the IPv6 header. It will return the mask value representing the length of the compressed address. The link-local address are ever compressed down to zero bit. In order to decide how to compress the addresses, when global it is used a function that will control how many zeros are contained in the host part of the IPv6 address. This function will return "1" if the address should be compressed down to 64 bits or "0" if to 16 bits. Before return to the `packHeader()` function, the pointer relative to the buffer will be updated according to the compressed address length. The write operation in memory are carried by the `memcpy()` function declared as `ip_memcpy()`. After to have packed the IPv6 header, the compression routine start to compress the next header if it contain the UDP value.

Decompressing the IPv6 Header

As said before, the compressed headers will be unpacked into a buffer passed by reference to the `unpackAddress()` function. The received MAC frame will be passed to the `getLowpanPayload()` function implemented in the `lib6lowpan.c` source file. It is needed in order to extract the compressed IPv6 header into a buffer. After checking which 6LoWPAN optional headers are presents in the packet, it will return the length of the buffer needed to unpack the compressed IPv6 header. Fig.32 shows the implementation of this function.

```
uint8_t *getLowpanPayload(packed_lowmsg_t *lowmsg){
uint8_t len = 0;
if(lowmsg->headers & LOWMSG_MESH_HDR) len += LOWMSG_MESH_LEN; if(lowmsg-
>headers & LOWMSG_BCAST_HDR) len += LOWMSG_BCAST_LEN; if(lowmsg->headers &
LOWMSG_FRAG1_HDR) len += LOWMSG_FRAG1_LEN;    if(lowmsg->headers &
LOWMSG_FRAGN_HDR) len += LOWMSG_FRAGN_LEN; return lowmsg->data + len;}
```

Figure 32: getLowpanPayload implementation

Once obtained the IPv6 compressed header, it is checked whether or not the CID is present. If so, the pointer to the compressed header is updated and the CID flag is activated. The flag will be used to activate the control structure that will unpack the network prefix for destination and source address. Then, it is examined the value of the Version, Traffic and Flow field. An `ip_memcpy()` function is used to write in the buffer where the IPv6 header will be unpacked, an `ip_memclr()` function is used when the IPv6 fields to write contains all zeros. Following the IPv6 headers structure as reported in Fig.5, the next field to decompress is the Payload length. Since this field has

to be inferred by the MAC layer, it will be filled after have completed the decompression routine for all the other fields. To do this, it is used a pointer that reference the memory location where the Payload length value has to be written. The same idea is used to unpack the Next Header field. However, as said before, b6loWPAN define as Next Header the Source Routing header that is carried in-line without any form of compression. The Hop Limit field is decompressed using a switch case control structure. It takes as statement the value of HLIM field as specified in the IPHC-04 encoding format. With the HLIM fields we finish to unpack the IPv6 header fields compressed in the IPHC-04 dispatch byte. Now we pass to the encoding byte to unpack the source and destination address. The first address part to decompress is the one relative to the network prefix. As said before, a bit flag is used to check the presence of the CID extension octet If active, both source and destination address prefixes take the value of the used global network address, if not they are examined the values of SAC and DAC fields of IPHC-04 encoding byte. The host part of the source and destination address is decompressed by a dedicated function called `decompressAddress()`. It takes five arguments; two of which are passed by value and three by reference. The ones passed by value are the layer 2 address and the value of SAM or DAM fields of the IPHC-04 encoding format depending on if we are dealing with source or destination address. The remaining three arguments are the two buffers used to unpack and IPv6 headers and the assigned network prefix. Depending on the value of the DAM or SAM, a statement of a switch case structure is activated in order to decompress the address. Fig.33 shows the case for a 64 bit compressed address. In case of the address is compressed down to 16 bit, it is called a function named `decompressShortAddress()`. It contains also the routine to decompress multicast address. It returns an int value equal to one if the unpacked address is multicast, zero in the contrary.

After unpack the address we have to fill the IPv6 header fields relative to Payload length. We first control if the packet has been fragment or not. If the packet is part of a fragmented packet, its size is computed by an external function called `getFragDgramSize()`. The final payload length is obtained by subtracting the size o the IPv6 header to the size of the fragmented packet. If the packet results to be not fragmented, the payload length computation is done in three operations. First, length is fixed to the value of the frame length field of the received MAC frame (Fig.4) subtracted to the length of the compressed header. Then, it is added the length of the possible headers (i.e UDP) that can be presents in the packet. Finally, it is subtracted the length of the possible 6LoWPAN extension headers.

7 Implementation

Presently there are not known IPHC-04 public implementations to our best knowledge. The compression routine has been developed focusing on the integration with b6loWPAN protocol stack and reusing functions already provided in it. A performance analysis has been done comparing the implemented IPHC-04 header compression against LOWPAN_HC1 and the case of no IPv6 header compression. From now on we will refers to LOWPAN_HC1 as HC1. The results of the testing are reported and commented in this chapter.

7.1 Performance analysis

As explained in previous chapters, a 6LoWPAN sensor network is characterized by the scarce resources of its operational environment. When deploying such networks it is important to know how the 6LoWPAN limitations will affect the task the network should done in its life-time. For this reason, we found interest in testing such limitations in a real 6LoWPAN sensor network. Consequently, the performance analysis has been done taking into account sensor memory usage, sensor energy consumption, average throughput of packet transmission within the sensor network and average Round-Trip delay time (RTT).

The network topology (Fig.34) used to carry out the tests, is composed by three nodes each one with function listed as follow:

- IPBaseStation: Is the “border router” and acts as an Active Message bridge between the

serial and radio links; it is the destination node.

- Relay Node: it acts as a relay hop within the network.
- Sensor Node: it transmits UDP packets to the IPBaseStation; it is the source node.

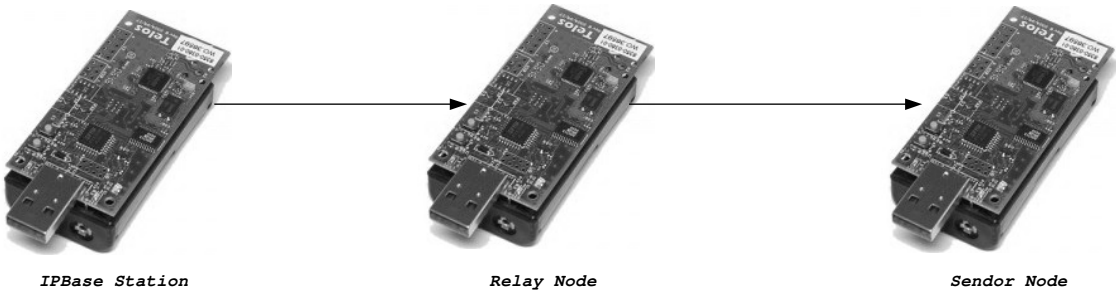

Figure 33: Network Topology

With regard to the methodology of the tests, we briefly explain how each measure has been done. RTT has been measured in a single-hop network topology using the ping6 command included in the b6lowpan. Power consumption analysis has been done over the "relay node" since it is where we made both, the decompression and compression functionalities, apart from forwarding (i.e. each time a packet reaches this node it has to decompress, compress and forward the packet). It has been sampled the energy consumption for different packet sizes. The device used for these measures is the Agilent Technologies DC Power Analyzer N6705A. Finally the measuring of the throughput has been done taking in account various length of the data application payload. For each value, it has been calculated ten values of throughput. The final value represents the mean value of the obtained results.

All the tests have been done on three different cases of compression:

1. IPHC-04
2. HC1.
3. No compression.

Performance analysis has been done on 6LoWPAN communications using global address. In the case of LOWPAN_IPHC it has been compressed down to 16 bits. For IPHC-04 the compressed IPv6 header reaches a size of 31 bytes composed as follows:

- 3 bytes are for dispatch, encoding and CID octets.

- 2 bytes are for Hop Limit and Next Header fields.
- 4 bytes are for the compressed source and destination addresses.
- 14 bytes are for Source Routing.
- 8 bytes are for the UDP header.

HC1 reach a compressed headers length of 58 bytes:

- 2 bytes are for Dispatch and encoding octets.
- 2 are for IPv6 Next Header and Hop Limit fields.
- 32 bytes are for source and destination addresses.
- 8 bytes are for the UDP header.
- 14 for Source Routing.

Finally, 62 bytes compose the overhead for the non compressed headers. The non compressed header carries all the IPv6 header fields in-line except the payload field.

7.2 ***Results***

Fig.35 shows the average throughput (in KB/sec) for the three cases listed above. The application data payload used for the measurements ranges from 5 to 1000 bytes.

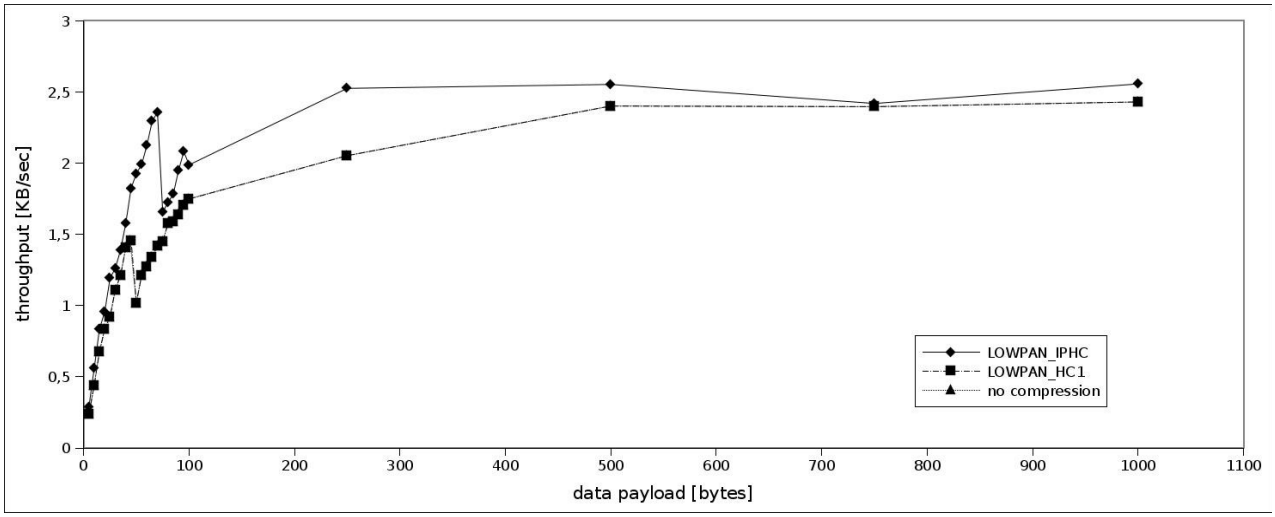

Figure 34: Throughput obtained for IPHC-04, HC1 and no compression.

It can be appreciated how the performances of all the considered cases for IPv6 header compression become similar when increasing the application data payload size. Considering a payload of 500 byte, the use of IPHC-04, with respect to HC1, increase the throughput by a 5,93% while decrease it to 0,93% for 750 bytes. For 1000 bytes the enhancement is of 4,94%. It has to be noticed that, for

large packets, the effect of using one or another header compression algorithm does not have a significant reflection in the performance of 6LoWPAN network. In fact, when dealing with such packets, the ratio between overhead bytes and packet size is low and consequently overhead has a minor impact in the measured throughput. Considering a data payload of 1000 bytes, if we include the overhead we reach 1076 bytes for IPHC-04, 1103 bytes for HC1 and 1107 for no header compression, that is the overhead represent respectively the 7% for IPHC-04, 9,3% for HC1 and 9,6% for no compression. Instead, if we consider an application data payload of 44 bytes (limit for HC1 to not fragment the packet), the ratio grows up to 41,3% for IPHC-04, 56,8% for HC1 and 61,2% for no compression. In this case, the presence of overhead influence strongly the throughput and its reduction will speed up a 6LoWPAN communication. The highest reduction of overhead obtained by applying IPv6 header compression as well as the better throughput performance, are both given by IPHC-04. In fact, taking again into account 44 bytes as data payload, we obtain an improvement of 25% respect to HC1 with IPHC-04. This can be appreciated in the Fig.36, where are shown the average throughput values for an application data payload ranging from 5 to 100 bytes. On the other hand, it is the difference in the packet process time to improve the data rate for IPHC-04 respect to HC1 for large packet size. The average difference of process time between IPHC-04 and HC1 for a application data payload of 1000 bytes, is assessed to be of 20 ms (410 ms for IPHC, 390 for HC1) while for a 44 byes payload it is of 5,4 ms (30,4 ms for IPHC-04 and 25 ms for HC1). IPHC-04 demonstrates to have also a minor packet processing times respect to HC1. Let us consider now Fig.36. When the application data payload exceed the maximum value allowed to not fragment packets (70 bytes for IPHC-04, 44 for HC1 and 40 for no header compression), there is a clear swoop in the throughput trend. This is due to the increased time needed to process the packets. Here, we get the worst throughput values for all the considered header compression algorithms. We remember that b6lowpan does not forward fragments of packets but it reassembles them before to fragment again and finally forward the packet. This behavior get worse the overall performance when dealing with packet fragmentation. In Fig.36, it can bee seen that the IPHC-04 trend reach the maximum distance from HC1 and no compression at 70 byte. Here, IPHC-04 has the maximum increaseament of throughput. It is of 39,77% respect to HC1 and the difference in time to process the packets is of 20 ms.

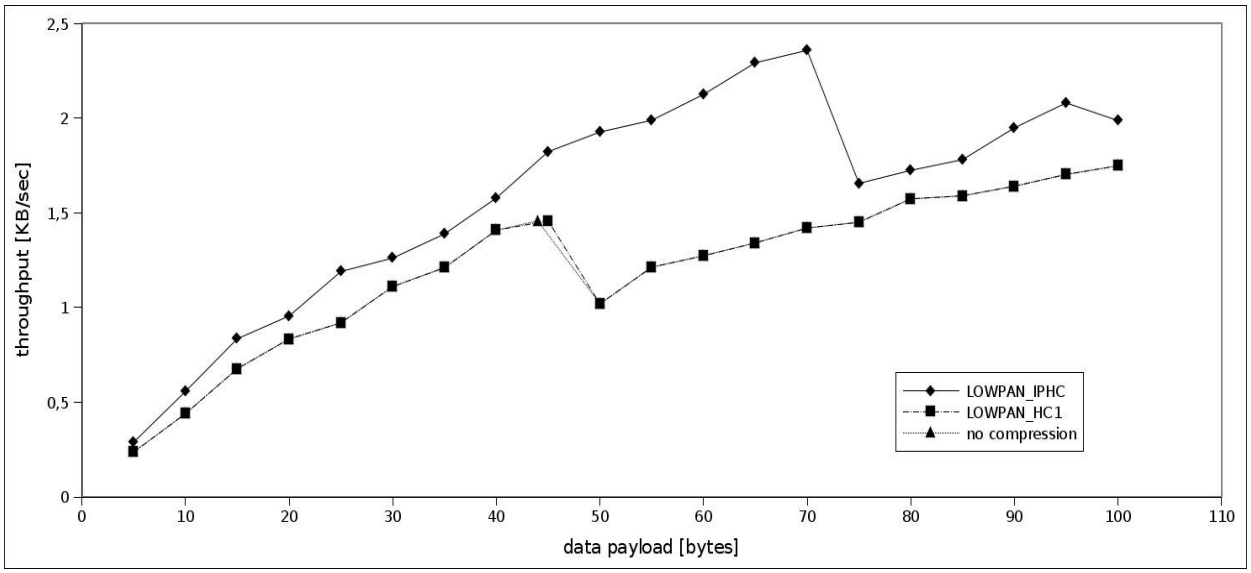

Figure 35: throughput values from 5 bytes to 100.

In terms of energy consumption, we was interested in measure the effects of compressing IPv6 headers without taking into account the data payload. Results are showed in Table 1.

	Consumption (mA)
No compression	19.49
HC1	19.41
IPHC-04	19.27

Table 1: Energy consumption

The sample rate has been fixed to 1 ms for a 10 minute test with Sensor node sending a packet each second and Base Station replying as soon as the packet arrives. IPHC-04 shows a better performance also in this case. Battery consumption is lowered by 0,72 % between IPHC-04 and HC1 and 1,13% between IPHC-04 and no compression case. We made further measurements of energy consumption taking into account the application data payload. The average energy consumption with the relative standard deviation, are reported in Table 2. In Fig.37 are visualized the average energy consumption values. It has been considered only IPHC-04 and HC1 since we was interested to compare only the performance of the two compression algorithms.

Data payload (bytes)	IPHC-04 consumption (mA)	Standard Deviation (mA)	HC1 consumption (mA)	Standard Deviation (mA)
0	19,27	3,51	19,41	3,45
20	19,59	3,50	19,72	3,53
40	19,66	3,48	19,70	3,52
70	19,67	3,42	19,73	3,52
120	19,73	3,52	19,78	3,59
250	19,78	3,59	19,72	3,59
500	19,73	3,48	19,75	3,60
1000	19,73	3,62	19,81	3,50

Table 2: Energy Consumption for IPHC-04 and HC1

As like as for throughput, IPHC-04 outperforms HC1 also in terms of energy consumption. However, it is only a slight improvement. The energy consumptions results are mainly influenced by the fact that the sensors are ever in a non-sleep mode. There is also a strong dependence to the sensor platform in use. As we know, a Telosb motes when receiving has a nominal energy consumption of 23 mA. The better performances of IPHC-04 are due by the reduced time used to process the incomings packets. The trend of the IPHC-04 and HC1 reported in Fig.37, has a likeness with the one obtained for throughput. This could means that, although the unpredictable nature a wireless link generally has, the overall performances of the implemented 6LoWPAN networks seems demonstrates to have a certain stability and repetitiveness.

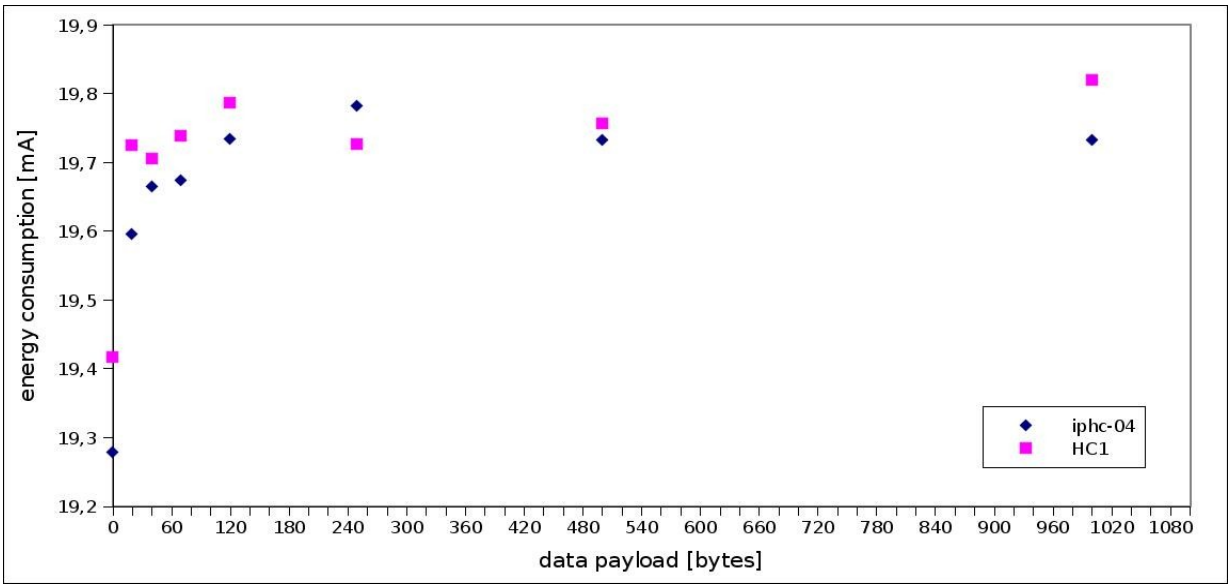

Figure 36: Energy consumption

Table 3 compares the memory usage of the basic *blip* installed function that includes header compression HC1-04 with the one implemented by IPHC-04. IPHC-04 increases by 564 bytes the

occupation of ROM memory. This is a reflection of the increased complexity of the compression algorithm. Mainly, the use of context based compression makes memory performance worse.

	ROM (bytes)	RAM (bytes)
HC1	22020	3421
IPHC-04	22584	3421

Table 3: Memory usage

Finally, Table 4 shows the average round-trip delay time obtained. It can be easily appreciated that IPHC-04 outperforms both no compression and HC1 cases. These results reflects the throughput performance confirming that the space saved using LOWPAN_IPHC and in particular by compressing global address steps up the performance in the data transmission. LOWPAN_IPHC decreases RTT by 51,72% respect to LOWPAN_HC1.The average RTT obtained for LOWPAN_IPHC is comparable to others results found in literature [31].

	Average RTT	Max RTT	Min RTT	Standard Deviation
No compression	171.151 ms	1311.428 ms	87.840 ms	88.397 ms
HC1	164.560 ms	1192.718 ms	81.323 ms	68.654 ms
IPHC-04	79.443 ms	1071.519 ms	63.301 ms	57.741 ms

Table 4: Round Trip Time (RTT) statistics

7.3 Conclusions

The use of IPHC-04 for IPv6 header compression, has demonstrated to enhance the performance of a 6LoWPAN communication. With respect to HC1, IPHC-04 give more flexibility to header compression and permit to reach a major gain in the reduction of the overhead. It seems also to have a better usage of the available link bandwidth. Also the packet processing shows an improvement. As the results obtained for RTT demonstrate, the network has an improved response time when using IPHC-04. Since the increase of the memory usage, we gather that the implementation of IPHC-04 implies an increase in complexity since its state-full approach. However, it can have further worsening if deploying a 6LoWPAN networks that need to communicate with several external IP or generally external PAN networks. In fact, for each external network we would like to communicate, in order to compress the global address, it has to be stored in memory the relative network prefix. The importance to compress global address, is remarked by the enhancement obtained in terms of throughput. In fact, IPHC-04 outperforms the other because the bytes of MAC payload used to carry the compressed headers are halved respect HC1. Mainly, the reduction is due

to the global address compression of IPHC-04. In fact, IPv6 source and destination addresses represent the 35% of the MAC frame data payload, if compressed down to 16 bits the percentage decrease to 3%. In conclusion, the use of one or another compression algorithm depends strongly on how the address fields are compressed more than the compression of other IPv6 fields. However, this remain valid as the packet size remain low. As mentioned before, for large packets it is the impact of the packet processing time to gain importance.

8 Implementation

8.1 Conclusions

With this work, we have presented the header compression mechanisms used to reduce IPv6 headers impact on the performance of 6LoWPAN environments. An implementation has been done and results are presented.

We have introduced the IEEE 802.15.4 standard [1] and explained how IPv6 [3] can be integrated in LoWPAN environments. In particular, we have focused our attention to the problematics arising when deploying a LoWPAN sensor network and how the adoption of IPv6 as network protocol can solve or minimize them. IPv6 allows to the 6LoWPAN networks node to auto-configure their addresses and to be self-healing. Furthermore, it permit to deploy networks with an high number of nodes since the large capabilities that IPv6 has to address nodes.

Then, we have explained and presented the existing IP header compression techniques for LoWPAN network and shown how the traditional approach can not be applied to this environment. An original proposal has been developed but not implemented since its basic ideas reflects the one presents in IPHC-04. It extends and improves our proposal.

Software and hardware components used in the implementation of the header compression has been presented. We have seen the software processing that a packet fulfill when it is send or received. We have presented the ANSI-C source file where the IPv6 header compression is implemented.

Finally, we have reported and commented the result of the tests done on a real implementation of a

6LoWPAN networks. We have compared IPHC-04 with HC1 and a no header compression scheme. The obtained results agree with the expected behavior of IPHC-04 respect to HC1.

The main purpose of IPHC-04 is to offer the performance of a state-full compression in a resource-limited environment such as 6LoWPAN. As we have shown, a state-full compression approach increases the sensor memory usage since it need that sensors store in the memory the state used for compression. However, it outperforms all the other parameters we have taken into account. Moreover, with the refined Traffic and Flow fields compression introduced in IPHC-04, the use of mechanisms of congestion control and QoS management on a 6LoWPAN communication would not affect dramatically the overall performance as it could happen with HC1. This would benefit the application of 6LoWPAN to critical applications (i.e industrial process control, maintenance and surveillance) where there is the need to guarantee the service also in case of network congestion.

Finally, the 6LoWPAN Working Group plans to deprecate HC1 header compression and push IPHC-04 [26] forward to become the new header compression standard for 6LoWPAN.

8.2 Future work

As future work, the implemented IPHC-04 compression routine will be adapted to the latest b6loWPAN version (“blip”) [29]. Moreover, it would be useful to study and test possible enhancements of the header compression definition. A possible enhancement could be in comparing the benefits of using Context based compression only for communication with external IP networks while using a stateless approach for internal communications. This could be done since the network prefix for an IPv6 global address is unique in the PAN boundaries. It would be interesting to test the performances of IPHC-04 by using it in an end-to-end IPv6 header compression instead of hop-by-hop. This could be done by adding to *blip* the needed capabilities to handle 6LoWPAN packets with Mesh Header extension.

To carry out this work, it has been followed the IETF 6LoWPAN working group documentation and discussions. We plan to enter actively and to participate in the just mentioned 6LoWPAN discussion group. The studies and the evaluations made in this projects should be seen as a contribution to a further development and diffusion of 6LoWPAN networks.

Bibliography

[1] IEEE Std 802.15.4™ -2006 (revision of IEEE Std 802.15.4-2003), IEEE Computer Society, September 2006.

[2] Jianliang Zheng, Lee, M.J., "Will IEEE 802.15.4 make ubiquitous networking a reality?: a discussion on a potential low power, low bit rate standard" Communications Magazine, IEEE Volume 42, Issue 6, June 2004 Page(s):140 – 146.

[3] S.Deering, B.Hinden, RFC 2460 – "Internet Protocol Version 6 (IPv6) Specification", IETF Network Working Group, December 1998.

[4] S.Deering, B.Hinden, RFC 4291 – "IPv6 Addressing Architecture", IETF Network Working Group, September 2007.

[5] http://en.wikipedia.org/wiki/IPv6#Addressing

[6] S.Thomson, T.Narten, T.Jinmei, RFC 4862 – "IPv6 Stateless Address Autoconfiguration". IETF Network Working Group, September 2007.

[7] The 6LoWPAN Architecture, G.Mulligan-6LoWPAN working group, IETF Network Working Group, September 2007.

[8] Chang-Yeol Yum et al, "Methods to use 6LoWPAN in IPv4 networks", The 9th International Conference on Advanced Communication Technology, Volume: 2, February 2007, On page(s): 969-972.

[9] D.E.Culler, J.Hui, "6LoWPAN tutorial", Arch Rock corporation.

[10] N. Kushalnagar, G. Montenegro, C. Schumacher, RFC 4919 - "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs):Overview, Assumptions, Problem Statement, and Goal", IETF Network Working Group, August 2007.

[11] N. Kushalnagar, G. Montenegro, D.E.Culler, J.Hui , RFC 4944 - "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", IETF Network Working Group, September 2007.

[12] EFFNET AB, "An Introduction to Ipv6 Header Compression", white paper, February 2004 http://www.effnet.com/sites/effnet/pdf/uk/Whitepaper_Header_Compression.pdf

[13] V.Jacobson, RFC 1144 – "Compressing TCP/IP Headers for Low-Speed Serial Links", IETF Network Working Group, February 1990.

[14] M.Degermark, B.Nordgren, S.Pink, RFC 2507 - "IP Header Compression", IETF Network Working Group, February 1999.

[15] S.Casner, V.Jacobson, RFC 2508 – "Compressing IP/UDP/RTP Headers for Low-Speed Serial

Links", IETF Network Working Group, February 1999.

[16] C.Bormann et al, RFC 3095 - "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed", IETF Network Working Group, July 2001.

[17] C.Bormann, "Context-based Header Compression for 6lowpan", IETF Network Working Group Internet-Draft, July 2008.

[18] J.Hui D.Culler, "Stateless IPv6 Header Compression for Globally Routable Packets in 6LoWPAN Subnetworks", IETF Network Working Group Internet-Draft, June 2007.

[19] C. Westphal, "A User-based Frequency-dependent IP Header Compression Architecture", proceeding of Globecom 2002 Taiwan.

[20] Narten et al., RFC 4861 – "Neighbor Discovery in IPv6," September 2007, IETF Network Working Group, September 2007.

[21] S.Chakrabarti E.Nordmark, "LoWPAN Neighbors Discovery extensions", 6LOWPAN WG Internet-Draft, November 2007.

[22] Crossbow Technology, Inc.TelosBdatasheet,http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf

[23] TinyOS website, http://www.tinyos.net/

[24] J.Hui P. Thubert, "Compression Format for IPv6 Datagrams in 6LoWPAN Networks", 6LOWPAN WG draft-ietf-6lowpan-hc-01, October 2008

[25] J.Hui P. Thubert, "Compression Format for IPv6 Datagrams in 6LoWPAN Networks", 6LOWPAN WG draft-ietf-6lowpan-hc-03, July 2008

[26] J.Hui P. Thubert, "Compression Format for IPv6 Datagrams in 6LoWPAN Networks", 6LOWPAN WG draft-ietf-6lowpan-hc-04, December 2008

[27] "Performance Analysis of IP over IEEE 802.15.4 Radio using 6LoWPAN", http://www.cs.wustl.edu/~jain/cse567-08/ftp/7lowpan.pdf.

[28] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems", In PLDI03, ACM, June 2003.

[29] Berkeley WEBS, "blip", http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip

[30] http://test.laser-group.com/vntech/dmdocuments/RetiSensoriWireless.pdf

[31] J. Hui, D. Culler, "IP is Dead, Long Live IP for Wireless Sensor Networks", Proceedings of the 6th ACM conference on Embedded network sensor systems. Raleigh, NC, USA. Pag. 15 – 28.

2008. ISBN:978-1-59593-990-6.

Illustration Index

Index of Tables

Appendix I

Paper extracted from this work and published by LNCS (Lecture Notes in Computer Science) for the "EUNICE 2009 -The Internet of the Future" international workshop.

Implementation and Evaluation of the Enhanced Header Compression (IPHC) for 6LoWPAN

Alessandro Ludovici[1], Anna Calveras[1], Marisa Catalan[1], Carles Gómez[1], Josep Paradells[1]
[1]Wireless Networks Group (WNG). Universitat Politècnica de Catalunya, C/Jordi Girona, 1-3, Mòdul C3 - Campus Nord. 08034 Barcelona, Spain
anna.calveras@entel.upc.edu

Abstract. 6LoWPAN defines how to carry IPv6 packets over IEEE 802.15.4 low power wireless or sensor networks. Limited bandwidth, memory and energy resources require a careful application of IPv6 in a LoWPAN. The IEEE 802.15.4 standard defines a maximum frame size of 127 bytes that decreases to 102 bytes considering the header overhead. A further reduction is due to the security, network and transport protocols header overhead that, in case of IPv6 and UDP, leave only 33 bytes for application data. A compression algorithm is necessary in order to reduce the overhead and save space in data payload. This paper describes and compares the proposed IPv6 header compression mechanisms for 6LoWPAN environments.

Keywords: 6lowpan, IPv6, header compression, sensor network, IEEE 802.15.4, blip

1 Introduction

6LoWPAN is defined as a protocol to enable IPv6 packets to be carried on top of Low Power Wireless Personal Area Networks (LoWPANs) [1]. LoWPANs are composed of devices compatible with the IEEE 802.15.4 standard.

The aim is to develop personal networks, mainly sensor based, that can be integrated to the existing well-known network infrastructure by reusing mature and wide-used technologies. IPv6 has been chosen as network protocol because its characteristics fit to the problematic that characterizes LoWPAN environments such as the large number of nodes to address and stateless address auto-configuration.

1.1 IEEE 802.15.4

The IEEE 802.15.4 standard [2] defines protocols and interconnections of devices via radio communication in a Low Rate Wireless Personal Area Network (LR-WPAN). It follows the OSI reference model and specifies the physical and the Medium Access Control (MAC) sublayer of the data link layer. The main characteristics of these LR-WPANs include: (1) data rates of 250 kbps, 100 kbps, 40 kbps and 20 kbps; (2) IEEE 16-bit short or 64-bit extended address; (3) Low power consumption.

IEEE 802.15.4 devices are classified into Full Function Devices (FFD) and Reduced Function Devices (RFD). The FFD operates as a PAN coordinator and border router. Two important features of 802.15.4 are its self-healing and self-organizing properties. This means that nodes are able to detect the presence of other nodes and organize themselves in a network, and they can detect and recover from faults.

There exist four different frame types: (1) beacon frame, (2) data frame, (3) acknowledgment frame, (4) MAC command frame. The maximum frame size defined in IEEE 802.15.4 is fixed to 127 bytes, of which 25 bytes are reserved for frame overhead. This leaves 102 bytes for payload.

1.2 6LoWPAN Architecture

In order to transport IPv6 packets over 802.15.4 links it is required, as specified in [3], to provide an adaptation layer below the network layer (Fig.1). It is demanded in order to comply with the minimum MTU required by IPv6 that is fixed to 1280 bytes.

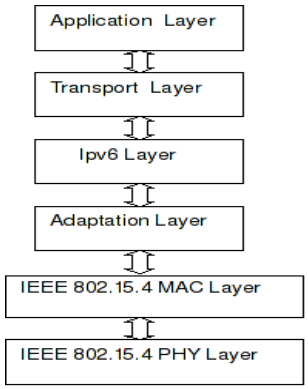

Fig. 1. 6LoWPAN protocol stack

The packet is prefixed by LoWPAN encapsulation headers that, as defined in [3], include the presence of a one byte IPv6 Dispatch header and the definition of the following header fields and their ordering constraints. The two leftmost bits are settled to 01 or 00 indicating if there is a 6LoWPAN frame or not. The remaining 6 bits can define up to 64 different dispatch header types. However, only 5 dispatch header types are defined in [3].

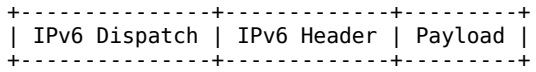

Fig. 2. LoWPAN encapsulated IPv6 datagram

As mentioned before, IPv6 allows stateless address auto-configuration. This property allows hosts to generate their own address combining locally available information together with the one advertised by routers. The host generates the interface identifier while the router provides the subnetwork prefix associated with a

link. The interface identifier is defined with a length of 64 bits [5]. Thus, there is no problem if the PAN uses 64 bits IEEE 802.15.4 extended addresses but a modification is needed when using 16 bit IEEE 802.15.4 short addresses. The modification consists of adding a 48 bits pseudo address to the 16 bits interface identifier in order to obtain the required length of 64 bits. The pseudo address is formed as follows:

PAN ID (16-bit): zero bits (16): IEEE 16-bit short address

Considering an IEEE 16-bit short address equal to "64" (hex) and PAN ID equal to "10" (hex) we obtain the following pseudo address:

00:10:00:00:00:64

2 Related Work on IPv6 Header Compression in LoWPAN

IP Header Compression can be defined as "*the process of compressing excess protocol headers before transmitting them on a link and uncompressing them to their original state on reception at the other end of the link*" [4]. Compression is possible since the information carried in the packet is redundant. The redundancy may be present because we are sending packets belonging to the same flow and so the information contained in the headers is repeated several times, or because it is already present in other protocol headers in the packet.

Traditionally, the header compression is performed over a link between two nodes called compressor and decompressor. Moreover, there is the concept of flow context, which is a "*collection of information about field values and change patterns of field values in the packet header*" [4]. As just mentioned, IP header compression is usually a hop by hop compression. In a sensor network, this compression approach has high cost in terms of power consumption, indeed at each hop the IP header should be decompressed and re-compressed by the devices. Therefore, this approach might not fit with the constraints of 6LoWPAN networks. In addition to the increased processing operation at each node and the consequent increase of the needed power, there is also the problem of the maintenance of the context due to limited memory in sensor devices.

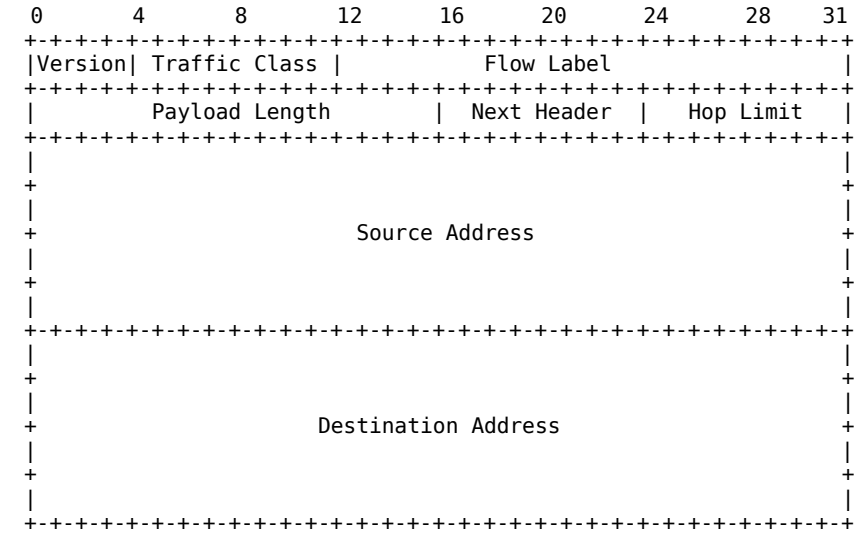

Fig. 1. 40 bytes IPv6 Header

2.1 LOWPAN_HC1

The first specification of IPv6 header compression for LoWPAN has appeared in [3], and it is specified as LOWPAN_HC1. Considering the IPv6 header as shown in Fig.3, the common case for 6LoWPAN communications can be listed as:

- IP Version: it is 6 for all packets
- Traffic class and flow label: they are zero
- Payload length: it can be inferred from layer 2 or from the "datagram_size" field in the case we have a fragmented packet.
- Next header: it can be UDP, TCP or ICMP, so using 2 bits suffices.
- Source and Destination address: they are link-local (that is, the IPv6 interface identifier can be inferred from source and destination address present in layer 2).

All these fields can be compressed to 1 byte. As mentioned in [3], it is mandatory not to compress the hop limit field, which always needs to be carried inline. So the resulting compressed header would have a size of 2 bytes instead of the 40 bytes of the uncompressed header as seen in Fig. 4.

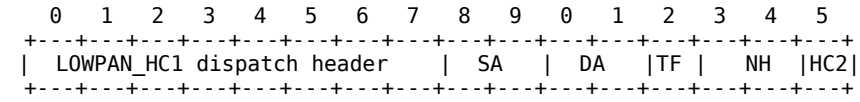

Fig. 2. 2 bytes encoding LOWPAN_HC1 format

LOWPAN_HC1 is only applied to link-local addresses. In consequence, it would not be possible to compress global addresses. The compression of global addresses would save 32 bytes of link-layer MTU. Moreover, a communication with global

addresses would give full capabilities of the IPv6 protocol adoption to a LoWPAN, such as end-to-end communication across different LoWPANs and external IP networks.

To solve this problem, an IETF Internet draft [6], LOWPAN_HC1g, has been published, specifying a method for compressing global addresses. The LOWPAN_HC1g compression came from the fact that "*To support compression of global unicast address, LOWPAN_HC1g assumes that a PAN is assigned on compressible 64-bit global IP prefix. When either the source or destination address matches the compressible IP prefix, it can be elided*" [6]. LOWPAN_HC1g does not substitute LOWPAN_HC1, but it extends its applicability.

The compression of global addresses would be useful to gain bytes in the packet to send user data. In order to achieve this, an alternative header compression scheme has been developed under the name of LOWPAN_IPHC [7]. In this paper we focus and implement this one, which is presented in the following section.

2.1 LOWPAN_IPHC

LOWPAN_IPHC [7] is the third proposed IPv6 header compression scheme. Currently, it is at its fourth update referred as LOWPAN_IPHC-04. Hereafter, LOWPAN_IPHC refers to the fourth update. It has been thought as an improvement of LOWPAN_HC1. In particular, it extends the applicability of header compression to support communication to nodes internal and external to LoWPANs (that is global address), multicast communication and both mesh-under and route-over configurations. Global IPv6 address compression is based on shared states within contexts. In contrast with LOWPAN_HC1, in the proposed LOWPAN_IPHC it is not mandatory to carry inline the hop limit field. A mechanism is specified to compress traffic and flow label in case they are not null fields. LOWPAN_IPHC uses five of the rightmost bits of the dispatch type (bits 3 to 7 in Fig. 5) in order to specify compressed fields of IPv6 header that are not related with the address compression. The dispatch header is followed by the LOWPAN_IPHC header that defines how source and destination addresses are compressed. An additional byte is present when communicating with global address; it is called Context Identifier Extension (CID). The four leftmost bits specify the context for source address. The remaining four rightmost bits specify the context used for destination address. Using context based compression, we could compress up to 16 network prefixes and save 60 bits of payload when communicating with external 6LoWPAN networks.

As reported in [7], LOWPAN_IPHC can compress the IPv6 header down to two octets (the dispatch octet and the LOWPAN_IPHC encoding) with link-local communication as seen in Fig. 5. When routing over multiple IP hops, LOWPAN_IPHC can compress the IPv6 header down to 7 octets (2-octets dispatch/LOWPAN_IPHC, 1-octet Hop Limit, 2-octet Source Address, and 2-octet Destination Address).

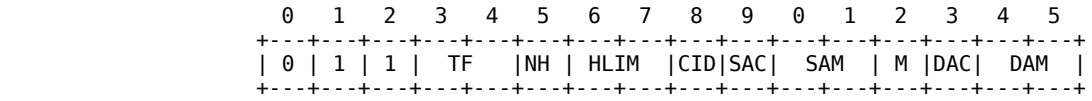

Fig. 1. LOWPAN_IPHC Encoding

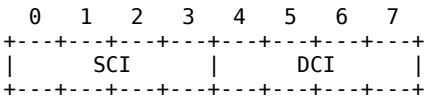

Fig. 2. CID octet

3 Implementation of IPv6 Header Compression over IEEE 802.15.4 Networks

3.1 Protocol Stack

Presently there are not LOWPAN_IPHC public implementations to our best knowledge. Hence, we have developed the compression and decompression routine focusing on the integration with b6lowpan protocol stack, which is presented in the next section, and reusing functions already provided in it.

The software component has been developed on TinyOS 2.1, which is an open-source operating system designed for wireless embedded sensor networks. The implementation of 6LoWPAN functionalities have been developed and implemented by the Berkeley Wireless Embedded Systems (WEBS) [8]. It has been released as TinyOS contribution and initially named b6loWPAN. Currently it is at its fourth version and has changed the name to Berkeley IP implementation for low-power networks (*blip*). When we started implementing the header compression, b6loWPAN was at first release so we have kept working on this version. From now on we will refer to it as *blip.*

It uses LOWPAN_HC1 header compression and includes IPv6 neighbor discovery, default route selection, point-to-point routing and network programming support. Standard tools like ping6, tracert6, and nc6 can be used to interact with and troubleshoot a network of 6loWPAN devices. Pc-side code is written using the standard BSD sockets API (or any other kernel-provided networking interface).

The *blip* implementation of header compression has been substituted by our implementation of LOWPAN_IPHC IPv6 Header compression.

3.2 Hardware Platform

The hardware platform used is the Crossbow's TelosB mote. It is an open source, low-power wireless sensor module. TelosB motes have a 16-bit RISC MCU at 8 MHz and 16 registers. The platform offers 10 kB of RAM, 48kB of flash memory and 16 kB of EEPROM. Requiring at least 1.8 V, it draws 1.8 mA in the active mode and 5.1

Further sensors available on the platform are a visible light sensor (Hamamatsu S1087), a visible to IR light sensor (Hamamatsu S1087-01) and a combined humidity and temperature sensor (Sensirion SHT11).

3.3 Environment and Measurements

A performance analysis has been done taking into account sensor memory usage, sensor energy consumption, average throughput of packet transmission within the sensor network and average Round-Trip delay time. The network topology (Fig. 7) is composed by three nodes:

1. IPBaseStation: it is the "border router" and acts as a bridge between the serial and radio links; it is the destination node.
2. Relay Node: it acts as a relay node.
3. Sensor Node: it transmits UDP packets to the IPBaseStation; it is the source node.

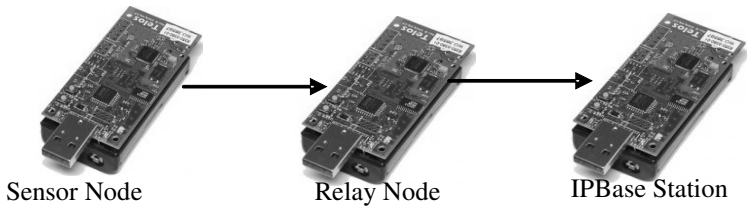

Fig. 1. Network topology

RTT has been measured in a single-hop network topology using the ping6 command included in b6lowpan.

Power consumption analysis has been done at the "relay node" since it is where both, the decompression and compression functionalities were carried out, apart from forwarding (i.e. each time a packet reaches this node it has to decompress, compress and forward the packet). The device used for these measures is the Agilent Technologies DC Power Analyzer N6705A.

All the tests have been done on three different cases of compression: (1) LOWPAN_IPHC, (2) LOWPAN_HC1, (3) No compression.

Performance analysis has been done on communications using global addresses. In the case of LOWPAN_IPHC, the global address has been compressed down to 16 bits.

4. Results

Fig.8 shows the average throughput (in KB/sec) for the three cases listed above. The IP payload ranges from 5 to 70 bytes length. For each payload value, 10 throughput measurements have been done. The final result is the mean value of them. The compressed header reaches a size of 31 bytes for LOWPAN_IPHC, 58 for

LOWPAN_HC1 and 62 for the non-compressed headers. The non-compressed header carries all the IPv6 header fields in-line, except the payload field.

In terms of throughput, LOWPAN_IPHC outperforms the others because the bytes of MAC payload used to carry the compressed headers are halved with respect to LOWPAN_HC1. Throughput increases by 39.77% for 70 bytes of payload, which is the maximum payload admitted by LOWPAN_IPHC without the need of fragmentation. Considering the maximum data payload (44 bytes) allowed by LOWPAN_HC1 without packet fragmentation, we obtain a throughput improvement of 25% with LOWPAN_IPHC compression.

The behavior of LOWPAN_HC1 compared with the no compression case needs a brief explanation. Although the UDP header is present in the packet, it is not declared in the next header field of IPv6. Instead of it, an hop-by-hop extension header named source routing header is specified. It is a non-standard header used in *blip* for source routing. In that way we have to carry in-line 8 bits of next header field. This means that, considering the architecture of the stack, the benefit of using one or another compression algorithm depends strongly on how the address fields are compressed more than the other IPv6 fields.

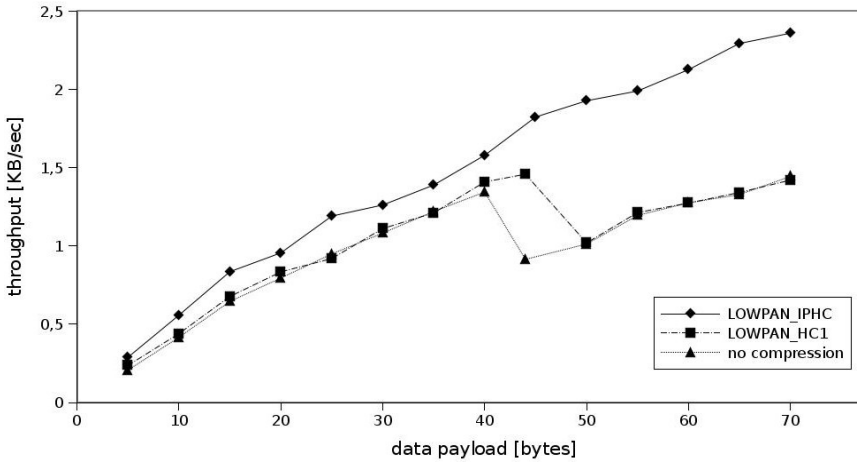

Fig. 1. Throughput obtained for LOWPAN_IPHC, LOWPAN_HC1 and no compression.

In terms of energy consumption, we have focused on the effect of compressing IPv6 headers without taking into account the data application payload. Results are shown in Table 1.

Table 1. Energy Consumption

	Consumption (mA)
No compression	19.49
LOWPAN_HC1	19.41
LOWPAN_IPHC	19.27

The sample rate has been fixed to 1 ms for a 10 minute test with the Sensor node sending a packet each second and the Base Station replying as soon as the packet arrives. LOWPAN_IPHC shows a better performance also in this case. Battery consumption is lowered 0,72 % between LOWPAN_IPHC and LOWPAN_HC1 and 1,13% between LOWPAN_IPHC and non compression case.

Table 2 compares the memory usage of the basic *blip* installed function that includes header compression LOWPAN_HC1 with the one implemented by LOWPAN_IPHC. LOWPAN_IPHC increases by 564 bytes the occupation of ROM memory. This reflects the increased complexity of the compression algorithm. Mainly, the use of context based compression makes memory performance worse.

Table 1. Memory usage

	ROM (bytes)	RAM (bytes)
LOWPAN_HC1	22020	3421
LOWPAN_IPHC	22584	3421

Finally, Table 3 shows the average round-trip delay time obtained from 1000 sent packets. It can be easily appreciated that LOWPAN_IPHC outperforms both no compression and LOWPAN_HC1 cases. These results reflect the throughput performance confirming that the space saved using LOWPAN_IPHC and, in particular, by compressing global addresses steps up the performance in the data transmission. LOWPAN_IPHC decreases RTT by 51.72% respect to LOWPAN_HC1. The average RTT obtained for LOWPAN_IPHC is comparable to others results found in literature [9].

Table 2. Round Trip Time (RTT) statistics

	Average RTT (ms)	Max RTT (ms)	Min RTT (ms)	Standard deviation (ms)
No compression	171.151	1311.428	87.840	88.397
LOWPAN_HC1	164.560	1192.718	81.323	68.654
LOWPAN_IPHC	79.443	1071.519	63.301	57.741

5 Conclusions

In this paper we have presented the header compression mechanisms used to reduce IPv6 headers impact on the performance of 6LoWPAN environments. A first implementation and preliminary results are presented. The obtained results agree with the expected behavior of LOWPAN_IPHC and LOWPAN_HC1.

The main purpose of LOWPAN_IPHC is to offer the performance of a stateful compression in a resource-limited environment such as 6LoWPAN. As we have shown, a stateful compression approach increases the sensor memory usage. However, it outperforms all the other parameters we have taken into account. Moreover, with the refined Traffic and Flow fields compression introduced in

LOWPAN_IPHC, the use of mechanisms of congestion control and QoS management on a 6LoWPAN communication would not affect dramatically the overall performance as it could happen with LOWPAN_HC1. This would benefit the application of 6LoWPAN to critical applications (i.e industrial process control, maintenance and surveillance) where there is the need to guarantee the service also in case of network congestion.

Finally, the 6LoWPAN Working Group plans to deprecate LOWPAN_HC1 header compression and push LOWPAN_IPHC [7] forward to become the new header compression standard for 6LoWPAN.

6 Future Work

As future work, the implemented LOWPAN_IPHC compression routine will be adapted to the latest *blip* version. Moreover, it would be useful to study and test possible enhancements of the header compression definition. We plan to compare the benefits of using Context based compression. This will be tested for global addresses when communication happens inside or outside the network.

Acknowledgments. This work has been supported by I2Cat Foundation, FEDER and the Spanish Government through project TEC2006-04504.

References

1. G.Mulligan, 6LoWPAN Working Group. Proceedings of the 4th workshop on Embedded networked sensors. Cork, Ireland. Pages: 78 – 82. 2007. ISBN:978-1-59593-694-3.
2. IEEE Std 802.15.4™ -2006 (revision of IEEE Std 802.15.4-2003), IEEE Computer Society, September 2006.
3. N. Kushalnagar, G. Montenegro, D. Culler, J. Hui , RFC 4944 "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", IETF Network Working Group, September 2007.
4. EFFNET AB, "An Introduction to IPv6 Header Compression", white paper, February 2004 http://www.effnet.com/sites/effnet/pdf/uk/Whitepaper_Header_Compression.pdf
5. R. Hinden, S. Deering, RFC 4291 - "IPv6 Addressing Architecture", IETF Network Working Group, February 2006
6. J. Hui, D. Culler, "Stateless IPv6 Header Compression for Globally Routable Packets in 6LoWPAN Subnetworks ", 6LOWPAN WG draft-hui-6lowpan-hc1g-00, June 2007
7. J. Hui, P. Thubert, "Compression Format for IPv6 Datagrams in 6LoWPAN Networks", 6LoWPAN WG draft-ietf-6lowpan-hc-04, December 2008
8. Berkeley WEBS, "blip", http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip
9. J. Hui, D. Culler, "IP is Dead, Long Live IP for Wireless Sensor Networks", Proceedings of the 6th ACM conference on Embedded network sensor systems. Raleigh, NC, USA. Pag. 15 – 28. 2008. ISBN:978-1-59593-990-6.

Appendix II

lib6lowpanIP.c : The implementation of IPv6 Header Compression for 6LoWPAN networks.

/*

"Copyright (c) 2008 The Regents of the University of California.

All rights reserved."

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without written agreement is hereby granted, provided that the above copyright notice, the following two paragraphs and the author appear in all copies of this software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITYAND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS."

*/

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "lib6lowpan.h"
#include "lib6lowpanFrag.h"

/*This file presents an interface for parsing IP and UDP headers
in a 6loWPAN packet.
 @author Stephen Dawson-Haggerty <stevedh@cs.berkeley.edu>*/

/* definition of IPv6 network prefixes*/
uint8_t linklocal_prefix [] = {0xfe, 0x80, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00};

uint8_t multicast_prefix [] = {0xff, 0x02, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00};

ip6_addr_t my_address = {0xfe, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00,0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x64};

uint8_t globalPrefix = 0;

/* the function cmpPfx compares the current IPv6 prefix with the
one of the prefixes declared as global variable */
uint8_t cmpPfx(ip6_addr_t a, uint8_t *pfx) {
  return (a[0] == pfx[0] &&
          a[1] == pfx[1] &&
```

```
          a[2] == pfx[2] &&
          a[3] == pfx[3] &&
          a[4] == pfx[4] &&
          a[5] == pfx[5] &&
          a[6] == pfx[6] &&
          a[7] == pfx[7]);
}

/* check if the IPv6 address has zero bit padding */
int ipv6_addr_suffix_is_long(const ip6_addr_t addr) {
  return (!(addr[8] == 0 &&
            addr[9] == 0 &&
            addr[10] == 0 &&
            addr[11] == 0 &&
            addr[12] == 0 &&
            addr[13] == 0));
}

#define ADDRLEN(rv, mask) \

/*getCompressedLen return the length of the compressed fields in
the buffer*/
int getCompressedLen(packed_lowmsg_t *pkt) {
  // min lenght is DISPATCH + ENCODING
  uint8_t encoding, dispatch, CID= 0, len = 2;
  uint8_t *buf = getLowpanPayload(pkt);
```

```
  dispatch = *buf;
  buf += 1;
  encoding = *buf;
/*check the presence of CID extension byte*/
  if ((encoding & LOWPAN_IPHC_CID_MASK) == LOWPAN_IPHC_CID_MASK){
     len +=1;
     buf +=1;
     CID = *buf;
  }
  if ((dispatch & LOWPAN_IPHC_DISPATCH) != LOWPAN_IPHC_DISPATCH)
    return 0;
  if ((dispatch & LOWPAN_IPHC_VTF_MASK) == LOWPAN_IPHC_VTF_INLINE)
    len += 4;
  if ((dispatch & LOWPAN_IPHC_NH_MASK) == LOWPAN_IPHC_NH_INLINE)
    len += 1;
if ((dispatch & LOWPAN_IPHC_HLIM_MASK) == LOWPAN_IPHC_HLIM_INLINE)
    len += 1;
/*check the length of the compressed addresses*/
switch      ((encoding      >>      LOWPAN_IPHC_SC_OFFSET)      &
LOWPAN_IPHC_ADDRFLAGS_MASK) {
  case LOWPAN_IPHC_ADDR_128: len += 16; break;
  case LOWPAN_IPHC_ADDR_64: len += 8; break;
  case LOWPAN_IPHC_ADDR_16: len += 2; break;
  case LOWPAN_IPHC_ADDR_0: len += 0; break;
  }
  switch (encoding & LOWPAN_IPHC_ADDRFLAGS_MASK) {
```

```
  case LOWPAN_IPHC_ADDR_128: len += 16; break;
  case LOWPAN_IPHC_ADDR_64: len += 8; break;
  case LOWPAN_IPHC_ADDR_16: len += 2; break;
  case LOWPAN_IPHC_ADDR_0: len += 0; break;
  }
  if ((encoding & LOWPAN_IPHC_NH_MASK) != LOWPAN_IPHC_NH_INLINE) {
// figure out how long the next header encoding is
      uint8_t *nh = buf + len;
    if ((*nh & LOWPAN_UDP_DISPATCH) == LOWPAN_UDP_DISPATCH) {
// we are at a udp packet
      len += 1; // add LOWPAN_HCNH
      uint8_t udp_enc = *nh;
    if    ((udp_enc    &    LOWPAN_UDP_S_MASK)    &&    (udp_enc    &
LOWPAN_UDP_D_MASK))
        len += 1;
   else   if   ((udp_enc   &   LOWPAN_UDP_S_MASK)   ||   (udp_enc   &
LOWPAN_UDP_D_MASK))
        len += 3;
      else
        len += 4;
      if ((udp_enc & LOWPAN_UDP_C_MASK) == 0)
        len += 2;
    }
  }
  return len;
}
// decompress addresses compressed down to 16 bits
```

```
int decompressShortAddress(uint8_t encoding, uint8_t *s_addr, uint8_t *dest, uint8_t *prefix) {
  if ((*s_addr & LOWPAN_IPHC_SHORT_MASK) == 0) {
// simplest case, just use the appropriate prefix
    ip_memcpy(dest, prefix, 8);
    ip_memclr(dest + 8, 8);
    dest[14] = (*s_addr) & ~LOWPAN_IPHC_SHORT_MASK;
    dest[15] = *(s_addr + 1);
    return 0;
  }
/*otherwise we either have an invalid compression, or else it's a multicast address*/
  ip_memcpy(dest, prefix, 8);
  ip_memclr(dest + 8, 8);
  switch (*s_addr & LOWPAN_IPHC_SHORT_LONG_MASK) {
  case LOWPAN_IPHC_HC1_MCAST:
    dest[14] = (*s_addr) & ~LOWPAN_IPHC_SHORT_LONG_MASK;
    dest[15] = *(s_addr + 1);
    break;
  case LOWPAN_IPHC_HC_MCAST:
     dest[1] = ((*s_addr) & LOWPAN_HC_MCAST_SCOPE_MASK) >> LOWPAN_HC_MCAST_SCOPE_OFFSET;
/*we'll just direct map the bottom 9 bits in for the moment, since HC doesn't specify anything that would break this. In the future, a more complicated mapping is likely.*/
    dest[14] = (*s_addr) & 0x1;
    dest[15] = *(s_addr + 1);
```

```
    break;
  default:
    return 1;
  }
  return 0;
}

// function to unpack compressed addresses
int decompressAddress(uint8_t encoding, uint16_t src, uint8_t
addr_flags, uint8_t **buf, uint8_t *dest , uint8_t *prefix) {
  uint16_t tmp;
  int rc = 0;
  switch (addr_flags) {
  case LOWPAN_IPHC_ADDR_128:
    ip_memcpy(dest, *buf, 16);
    *buf += 16;
    break;
  case LOWPAN_IPHC_ADDR_64:
    ip_memcpy(dest, prefix, 8);
    ip_memcpy(dest + 8, *buf, 8);
    *buf += 8;
    break;
  case LOWPAN_IPHC_ADDR_16:
    rc = decompressShortAddress(encoding, *buf, dest, prefix);
    *buf += 2;
    break;
```

```
  case LOWPAN_IPHC_ADDR_0:
    ip_memcpy(dest, prefix, 8);
    ip_memclr(dest + 8, 6);
    tmp = hton16(src);
    ip_memcpy(dest + 14, (uint8_t *)&tmp, 2);
    break;
  }
  return rc;
}

/*Unpacks all headers, including any compressed transport headers
if there is a compression scheme defined for them.
  @pkt  - the wrapped struct pointing to the compressed headers
   @dest - buffer to unpack the headers into
   @len  - the len of 'dest'
   @return the number of bytes written to dest, or zero if
   decompression failed. Should be >= sizeof(struct ip6_hdr)*/
uint8_t  *unpackHeaders(packed_lowmsg_t  *pkt,  uint8_t  *dest,
uint16_t len) {
// declare dispatch, encoding and CID bytes
  uint8_t dispatch, encoding, CID;
  uint8_t hop;
  int flag_CID = 0;
  CID = 0;
  uint16_t size, extra_header_length = 0;
  uint8_t *buf = (uint8_t *)getLowpanPayload(pkt);
// pointers to fields  we may come back to fill in later
```

```
  uint8_t *plen, *prot_len, *nxt_hdr;
  uint8_t *prefix_src, *prefix_dst;
  prefix_src = NULL;
  prefix_dst = NULL;
/*a buffer we can write addresses prefixes and suffexes into. now
we don't need to check sizes until we get to next headers*/
  if (buf == NULL || len < sizeof(struct ip6_hdr)) return NULL;
  len -= sizeof(struct ip6_hdr);
  dispatch = *buf; buf++;
  encoding = *buf; buf++;
//check CID flag
  if ((encoding & LOWPAN_IPHC_CID_MASK) == LOWPAN_IPHC_CID_MASK){
     CID = *buf;
     buf++;
     flag_CID = 1;
  }
  if ((dispatch & LOWPAN_IPHC_VTF_MASK) == LOWPAN_IPHC_VTF_INLINE)
{
//copy the inline 4 bytes of fields
    ip_memcpy(dest, buf, 4);
    buf += 4;
  } else {
/*clear the traffic class and flow label fields, and write the
version*/
    ip_memclr(dest, 4);
    *dest = IPV6_VERSION << 4;
  }
```

```
  dest += 4;
  plen = dest;
  prot_len = dest;
// payload length field requires some computation...
  dest += 2;
  if ((dispatch & LOWPAN_IPHC_NH_MASK) == LOWPAN_IPHC_NH_INLINE) {
    *dest = *buf;
    buf++;
  }
  nxt_hdr = dest;
  dest += 1;
/*otherwise, decompress IPNH compression once we reach the end of
the packed data.*/
  hop = (dispatch & LOWPAN_IPHC_HLIM_MASK);
  switch (hop){
  case LOWPAN_IPHC_HLIM_INLINE: *dest = *buf;
                    buf++;
                    break;
  case LOWPAN_IPHC_HLIM_MASK_1: *dest = 0x1;
                    break;
  case LOWPAN_IPHC_HLIM_MASK_64: *dest = 0x40;
                    break;
  case LOWPAN_IPHC_HLIM_MASK_255: *dest = 0xff;
                    break;
}
  dest += 1;
```

```
// time to decompress the address. check prefix here
  if (flag_CID == 1){
     if ((CID & LOWPAN_IPHC_SCI_MASK) == LOWPAN_IPHC_SCI_MASK)
        prefix_src = my_address;
     if ((CID & LOWPAN_IPHC_DCI_MASK) == LOWPAN_IPHC_DCI_MASK)
        prefix_dst = my_address;
  } else {
// check SAC
  if ((encoding & LOWPAN_IPHC_SAC_MASK) != LOWPAN_IPHC_SAC_MASK){
        prefix_src = linklocal_prefix;
        } else {
        prefix_src = my_address;
        }
//check DAC
  if ((encoding & LOWPAN_IPHC_DAC_MASK) != LOWPAN_IPHC_DAC_MASK){
   if ((encoding & LOWPAN_IPHC_M_MASK) != LOWPAN_IPHC_M_MASK){
      prefix_dst = linklocal_prefix;
   } else {
      prefix_dst = multicast_prefix;
           }
  } else {
      prefix_dst = my_address;
         }
  }
//address decompression functions
decompressAddress(encoding, pkt->src, ((encoding >>
```

```
LOWPAN_IPHC_SC_OFFSET) & LOWPAN_IPHC_ADDRFLAGS_MASK), &buf, dest,
prefix_src);
  dest +=16;
decompressAddress(encoding, pkt->dst, (encoding &
LOWPAN_IPHC_ADDRFLAGS_MASK), &buf, dest, prefix_dst);
  dest += 16;
/*dest points at the start of the source address IP header
field.we're done with the IP headers; time to decompress any
compressed header headers which follow...  We need to re-check
that there's enough buffer to do this.*/

  if ((dispatch & LOWPAN_IPHC_NH_MASK) != LOWPAN_IPHC_NH_INLINE) {
    // time to decode some next header fields
    // we ought to be pointing at the HCNH encoding byte now.
    if ((*buf & LOWPAN_UDP_DISPATCH) == LOWPAN_UDP_DISPATCH) {
      if (len < sizeof(struct udp_hdr)) return NULL;
      len -= sizeof(struct udp_hdr);
      struct udp_hdr *udp = (struct udp_hdr *)dest;
      uint8_t udp_enc = *buf;
      uint8_t dst_shift = 4;
      extra_header_length = sizeof(struct udp_hdr);
      buf += 1;
// UDP
      *nxt_hdr = IANA_UDP;
      if (udp_enc & LOWPAN_UDP_S_MASK) {
        // recover from 4 bit packing
        udp->srcport = hton16((*buf >> 4) + LOWPAN_UDP_PORT_BASE);
```

```
        dst_shift = 0;
      } else {
        ip_memcpy((uint8_t *)&udp->srcport, buf, 2);
        buf += 2;
      }
      if (udp_enc & LOWPAN_UDP_D_MASK) {
          udp->dstport = hton16((((*buf >> dst_shift)) & 0x0f) +
LOWPAN_UDP_PORT_BASE);
        buf += 1;
      } else {
        if (dst_shift == 0) buf += 1;
        ip_memcpy((uint8_t *)&udp->dstport, buf, 2);
        buf += 2;
      }
      if (udp_enc & LOWPAN_UDP_C_MASK) {
/* we elided the checksum and so are supposed to recompute it
here.however, we won't do this */
      } else {
        ip_memcpy((uint8_t *)&udp->chksum, buf, 2);
        buf += 2;
      }
/* still must fill in the length field, but we must first
recompute the IP length, which we couldn't do until now */
      prot_len = (uint8_t *)&udp->len;
      dest += sizeof(struct udp_hdr);
    } else {
/* otherwise who knows what's here... it's an error because the
```

```
NH bit said we were inline but when we got here, we didn't
recognize the NH encoding*/
      return NULL;
    }
  }
/* we can go back and figure out the payload length now that we
know how long the compressed headers were */
  if (hasFrag1Header(pkt) || hasFragNHeader(pkt)) {
    getFragDgramSize(pkt, &size);
    size -= sizeof(struct ip6_hdr);
  } else {
// it's a one fragment packet
    size = pkt->len - (buf - pkt->data);
    size += extra_header_length;
    size -= getLowpanPayload(pkt) - pkt->data;
  }
  *plen = size >> 8;
  *(plen + 1) = size & 0xff;
/*finally fill in the udp length field that we finally can
recompute*/
  switch (*nxt_hdr) {
  case IANA_UDP:
    *prot_len = size >> 8;
    *(prot_len + 1) = size & 0xff;
  }
  return buf;
}
```

```
/* packs addr into *buf, and updates the pointer relative to the
length * that was needed. returns the bit flags indicating which
length was used */
uint8_t packAddress(uint8_t flag, uint8_t **buf, ip6_addr_t addr)
{
     if(flag == 0x0 || flag == 0x3){
//link local address compression
     return LOWPAN_IPHC_ADDR_0;
   } else if (flag == 0x1 || flag == 0x2){
//global address compression
     if (ipv6_addr_suffix_is_long(addr)){
       ip_memcpy(*buf, &addr[8], 8);
       *buf +=8;
      return LOWPAN_IPHC_ADDR_64;
    } else {
//global to 16 bit
       ip_memcpy(*buf, &addr[14], 2);
       *buf +=2;
      return LOWPAN_IPHC_ADDR_16;
     }
   } else if (flag == 0x4){
    **buf = LOWPAN_IPHC_HC_MCAST; // set the encoding bits
    **buf |= (addr[1] << LOWPAN_HC_MCAST_SCOPE_OFFSET); // scope
// direct mapped group id
    **buf |= addr[14] & 0x1;
    *(*buf + 1) = addr[15];
```

```
    *buf += 2;
     return LOWPAN_IPHC_ADDR_16;
   }   else {
     ip_memcpy(*buf, addr, 16);
     return LOWPAN_IPHC_ADDR_128;
     }
}
/*pack the headers of msg into the buffer pointed to by buf
returns a pointer to where we stopped writing */
uint8_t *packHeaders(ip_msg_t *msg, uint8_t *buf, uint8_t len) {
  uint8_t *dispatch, *encoding, *CID, addr_enc;
  uint8_t hop;
  struct ip6_hdr *hdr = &msg->hdr; //point to IPv6 header
  int flag_CID = 0;
  int flag_UDP = 0;
  uint8_t flag_addr = 0;
  dispatch = buf;
  *dispatch = 0;
  buf += 1;
  encoding = buf;
  buf += 1;
  *encoding = 0;
//add dispatch type
  if (hdr->vlfc[0]==(IPV6_VERSION << 4))
      *dispatch |= LOWPAN_IPHC_DISPATCH;
  else
```

```
      return 0; //no IPv6 Heade
//check if CID is needed
  if(cmpPfx(hdr->dst_addr, my_address)){
       flag_CID = 1;
       CID = buf;
       *CID = 0;
       buf +=1;
      }else{
      CID = NULL;
      }
  //check traffic and flow fields
  if (hdr->vlfc[1] == 0 &&
      hdr->vlfc[2] == 0 &&
      hdr->vlfc[3] == 0)
      *dispatch |= LOWPAN_IPHC_VTF_MASK;
  else{
      ip_memcpy(buf, &hdr->vlfc, 4);
      buf +=4;
    }

//check NH field...compress if UDP
  if(hdr->nxt_hdr == IANA_UDP){
     *dispatch |= LOWPAN_IPHC_NH_MASK;
     flag_UDP = 1;
  }else{
     *buf = hdr->nxt_hdr;
```

```
     buf +=1;
  }
//hop limit field
  hop = hdr->hlim;
   if(cmpPfx(hdr->dst_addr, linklocal_prefix) || !(globalPrefix &&
cmpPfx(hdr->dst_addr, my_address))){
   switch(hop){
   case 0x1: *dispatch |= LOWPAN_IPHC_HLIM_MASK_1;
             break;
   case 0x40: *dispatch |= LOWPAN_IPHC_HLIM_MASK_64;
             break;
   case 0xff: *dispatch |= LOWPAN_IPHC_HLIM_MASK_255;
             break;
   default: *buf = hdr->hlim;
             buf += 1;
             break;
             }
    }else{
     *buf = hdr->hlim;
      buf += 1;
   }
// dispatch octect is done. Now pass to IPHC encoding octect
  if (flag_CID == 1)
  *encoding |= LOWPAN_IPHC_CID_MASK;
// SAC
  if (cmpPfx(hdr->src_addr, linklocal_prefix)){
```

```
     flag_addr = 0x0;
  } else if (globalPrefix && cmpPfx(hdr->src_addr, my_address)){
    *encoding |= LOWPAN_IPHC_SAC_MASK;
    flag_addr = 0x1;
    *CID |= LOWPAN_IPHC_SCI_MASK;
  } else {
//unspecified address
  }
  addr_enc = packAddress(flag_addr, &buf, hdr->src_addr);
  *encoding |= addr_enc << LOWPAN_IPHC_SC_OFFSET;
//DAC
  if (globalPrefix && cmpPfx(hdr->dst_addr, my_address)) {
     flag_addr = 0x2;
     *encoding |= LOWPAN_IPHC_DAC_MASK;
     *CID |= LOWPAN_IPHC_DCI_MASK;
  } else if (cmpPfx(hdr->dst_addr, linklocal_prefix)) {
     flag_addr = 0x3;
  } else if (hdr->dst_addr[0] == 0xff){
    *encoding |= LOWPAN_IPHC_M_MASK;
    flag_addr = 0x4;
  }
  addr_enc = packAddress(flag_addr, &buf, hdr->dst_addr);
  *encoding |= addr_enc;

// now come the compressions for special next header values.
  if (flag_UDP == 1) {
```

```
    struct udp_hdr *udp = (struct udp_hdr *)msg->data;
    uint8_t *udp_enc = buf;
    uint8_t *cmpr_port = NULL;
    // do the LOWPAN_UDP coding
    *udp_enc = LOWPAN_UDP_DISPATCH;
    buf += 1;
    if ((ntoh16(udp->srcport) & LOWPAN_UDP_PORT_BASE_MASK) ==
        LOWPAN_UDP_PORT_BASE) {
      //printf("compr to 4b\n");
      cmpr_port = buf;
 *cmpr_port = (ntoh16(udp->srcport) & ~LOWPAN_UDP_PORT_BASE_MASK)
<< 4;
      *udp_enc |= LOWPAN_UDP_S_MASK;
      buf += 1;
    } else {
      ip_memcpy(buf, (uint8_t *)&udp->srcport, 2);
      buf += 2;
    }
    if ((ntoh16(udp->dstport) & LOWPAN_UDP_PORT_BASE_MASK) ==
        LOWPAN_UDP_PORT_BASE) {
      if (cmpr_port == NULL) {
/* the source port must not have been compressed, so  allocate a
new byte for this guy */
*buf = ((ntoh16(udp->dstport) & ~LOWPAN_UDP_PORT_BASE_MASK) << 4);
        buf += 1;
      } else {
/* already in the middle of a byte for the port compression,so
```

```
fill in the rest of the byte */
    *cmpr_port    =    *cmpr_port    |    ((ntoh16(udp->dstport)    &
~LOWPAN_UDP_PORT_BASE_MASK));
      }
      *udp_enc |= LOWPAN_UDP_D_MASK;
    } else {
      ip_memcpy(buf, (uint8_t *)&udp->dstport, 2);
      buf += 2;
    }

// we never elide the checksum
    ip_memcpy(buf, (uint8_t *)&udp->chksum, 2);
    buf += 2;
  }
// I think we're done here...
  return buf;
}

/*indicates how much of the packet after the IP header we will
pack */
int packs_header(ip_msg_t *msg) {
  switch (msg->hdr.nxt_hdr) {
  case IANA_UDP:
    return sizeof(struct udp_hdr);
  default:
    return 0;
  }
```

```
}
#define CHAR_VAL(X)  (((X) >= '0' && (X) <= '9') ? ((X) - '0') :
((X) - 'A'))
void inet6_aton(char *addr, ip6_addr_t dest) {
  uint16_t cur = 0;
  char *p = addr;
  uint8_t block = 0, shift = 0;
  if (addr == NULL || dest == NULL) return;
  ip_memclr(dest, 16);
  // go to upper case
  while (*p != '\0') {
    if (*p >= 'a' && *p <= 'z')
      *p -= 'a' - 'A' - 10;
    p ++;
  }
  p = addr;
  // first fill in from the front
  while (*p != '\0') {
    if (*p != ':') {
      cur <<= 4;
      cur |= CHAR_VAL(*p);
    } else {
      ((uint16_t *)dest)[block++] = hton16(cur);
      cur = 0;
    }
    p++;
```

```
    if (*p == '\0')
      return;
    if (*(p - 1) == ':' && *p == ':') {
      break;
    }
  }
/* we must have hit a "::" which means we need to start filling in
from the end.*/
  block = 7;
  cur = 0;
  while (*p != '\0') p++;
  p--;
// now pointing at the end of the address string
  while (p > addr) {
    if (*p != ':') {
      cur |= (CHAR_VAL(*p) << shift);
      shift += 4;
    } else {
      ((uint16_t *)dest)[block--] = hton16(cur);
      cur = 0; shift = 0;
    }
    p --;
    if (*(p + 1) == ':' && *p == ':') break;
  }
}
```