Universitá degli studi di Padova

Dipartimento di Ingegneria dell'informazione

Corso di Laurea Magistrale in Ingeneria delle Telecomunicazioni

Tesi di laurea

IPv6 implementation with 6lowPAN adaptation layer for wirelesss sensor networks: analysis, development and experimentation

Relatore: Prof. Lorenzo Vangelista

Correlatore: Dott. Angelo P. Castellani

Laureando: Giulio Ministeri

Abstract

In questo lavoro di tesi é stato sviluppato uno stack protocollare basato su IPv6 per il sistema operativo TinyOS nell'ambito delle reti di sensori wireless. Lo stack protocollare utilizza come layer di adattamento tra il livello data-link, che segue lo standard IEEE 802.15.4, e il livello di rete, che segue appunto lo standard IPv6, il nuovo standard 6lowPAN. L'architettura generale dello stack permette di cambiare gli standard usati mantenendo inalterata la struttura dei componenti e delle interfacce cosí da rendere il codice riutilizzabile sia per altri scopi che per altri sistemi operativi. Lo sviluppo di un componente che gestisce in maniera autonoma un blocco di memoria RAM, ha permesso di astrarre ulteriormente la gestione dei pacchetti IP rendendola indipendente dal particolare standard implementato.

Nello stack protocollare sviluppato sono state implementate le procedure di compressione e decompressione dell'header IPv6 specificate nel draft hc-15 e le procedure di frammentazione e deframmentazione dei pacchetti IPv6 spiegate nell'RFC 4944.

L'autore.

Contents

Abstract **III**

1 Sensor networks and IPv6 **1**

1.1 Wireless Sensor Networks 1

1.1.1 Sensor Networks and Internet of Things 1

1.1.2 Hardware Platform and TinyOS 3

1.1.3 802.15.4 standard 6

1.2 Internet Protocol version 6 9

2 Related work **19**

2.1 6lowPAN by Matúš Harvan 19

2.2 blip 20

2.3 Contiki and μIPv6 22

2.4 TinyNET 22

3 Implementation **25**

3.1 Introduction 25

3.2 Design principles 26

3.3 Architecture 28

3.4 Memory module 30

3.5 sixlowpan module 32

3.5.1 Compression and decompression processes 40

3.5.2 Fragmentation and defragmentation processes 42
3.6 IPv6 module 45
3.7 UDP module 45

4 Testing and results **47**
4.1 Testing procedures 47
4.2 Send section 48
4.3 Receive section 53
4.4 Blip compatibility 56
4.5 Memory occupation and CPU time analysis 58

5 Conclusion **61**
5.1 Further improvements 61
5.2 Conclusion 62

Bibliography **68**

List of Figures **71**

Chapter 1

Sensor networks and IPv6

Abstract:

In this chaptera brief introduction is given about wireless sensor networks and related internet standards, (IPv6, 6lowPAN, adaptation layer).

1.1 Wireless Sensor Networks

1.1.1 Sensor Networks and Internet of Things

A sensor network is a network where a set of small devices, placed inside the interested environment, keeps under observation some kind of environmental conditions (temperature, light, humidity, position, ...) and communicates these informations to a sink node that collects and store them.
An heavy engineering work have permitted to design of a new generation of devices which are able to consume a relative small amount of energy, but also provide a moderate processing power. These small devices, so called "sensor nodes" or "motes", are usually equipped with sensors to detect and to measure some environmental conditions, and with a radio module to be able to communicate with each other or with the sink node.
Typical deployments of these networks are monitoring and controlling environ-

ment in special situations like wildlife nature (to prevent forests' fire for example), earthquake site, road traffic analysis or in sensitive buildings like bridges or dams.

The network topology is ad-hoc or mesh; nodes can act like server, client or router, they request data to other nodes, answer to a request from another node, or route informations between two nodes that are too far to have a direct communication.

Due to the nature of possible applications, that doesn't permit to easily reach nodes during their operations, these devices have to run for long time on battery power, hence they have to save energy as more as they can. Therefore the characteristics of the nodes are: frequent and long periods on sleep-mode, small radio transmit power (small radio range), relatively slow working clock frequency.

The most important and crucial aspect is the network protocols: a node can live just few days with radio chip always on; hence a reliable and energy-saving communication protocol is necessary to permit to leave switched off the radio chip as more as we can.

Internet of Things

Internet of Things refers to a new concept on how to think about all the physical objects. If we suppose that it is possible to provide an internet connection to every electronic object, we obtain a network made by objects that communicate each other without the human presence. With Internet of Things every object make itself recognizable by the rest of the world, communicate its identity, its assignment and its capabilities; in the same way every object can ask to other things who and where they are, what they can do, and determines if they are useful to perform its work better or if they can extend its capabilities.

It is a communication revolution, every electronic object will interoperate with all other electronic objects in the world. Things will start to work for us and make our lives easier: an alarm clock that rings ealier in the morning if it knows

that it will be road traffic, a fridge that writes the shopping list for us or an house that closes its windows when it starts raining and so on.
An important application that is based on the concept of Internet of Things is the Smart Grid. It concerns the electric distribution network that till nowadays has been being typically unidirectional: a power plant produces and provides energy to factories and private houses. But, with new renewable source energies, everyone can produce and give energy to the community. In this new configuration every single device that consumes or produces energy should be connected to the electric provider network, and it would be possible to control and monitor the energy consumption and production to efficiently capitalize renewable energy.
As the microelectronics research goes on, devices like nodes can be smaller and smaller and with more computing power even keeping a low power consumption, so it is possible to put in every electronic item a small node useful to interoperate with the object and communicate with its neighbor or maybe with every device connected to internet. Therefore we can move the wireless sensor network algorithms, protocols and features to the Internet of Things concept. The issue is to design an efficient and reliable network stack for nodes, to make nodes ready to get in the huge world of internet and internet protocols.

1.1.2 Hardware Platform and TinyOS

Since energy consumption determines sensor node lifetime, nodes tend to have a very limited computational and communication resources. Instead of modern 32-bit or 64-bit CPU with gigabytes of RAM and terabytes of storage memory, they have 8-bit to 16-bit CPU, with few kilobytes of RAM and few tens of kilobytes for program memory. CPUs have 1 to 10 megahertz of clock frequency, and their radio module can send data to a maximum ratio of few hundreds of kilobit per second. As a result, algorithms, protocols and even their implementations need to be vey efficient in terms of resource computation (CPU, ROM, RAM,

energy, bandwidth).

The hardware platform chosen for this project is the TelosB mote. It was originally developed at UC Berkeley and now are produced by the Crossbow Technology company and by Moteiv Corporation, now called Sentilla Corporation. TelosB motes feature a Texas Instruments MSP430 MCU, a 16-bit RISC MCU clocked at 8 MHz. The platform offers 10 kB of RAM, 48kB of program memory and 16 kB of EEPROM to permanently storage essential datas. It draws 1.8 mA in active mode and just 5.1 μA in sleep mode. Its radio chip, a Texas Instruments CC2420, is a low-power RF transceiver compatible with the IEEE 802.15.4 standard, and it can send up to 250 kbps at 2.4 GHz carrier frequency. It provides a 128-byte TX/RX buffer and it draws 18.8 mA to receive and 17.4 mA to send. So it is easy to note that in terms of power, the radio dominates the system.

TinyOS

TinyOS is a lightweight event-driven operating system specifically designed for low-power wireless sensor nodes. The project started as a collaboration between the University of California, Berkeley in co-operation with Intel Research and Crossbow Technology, and has since grown to be an international consortium, the TinyOS Alliance.

TinyOS differs from most other operating system in that its design focuses on ultra low-power operation. It is designed for small, low-power microcontrollers motes; furthermore it has very aggressive systems and mechanisms for saving power by automatically bringing MCU in low-power mode every time it is possible.

TinyOS has a very small footprint, the OS core requires only 400 bytes of program and RAM memory; there is no dynamic memory allocation no memory management and no virtual memory, all memory is allocated statically at com-

pile time. The system provides a set of reusable components which can be combined together. Components implement hardware abstractions of sensors to access to them on an high level interface, a scheduler to handle tasks, hardware interrupts, timers, access to flash memory and radio chip.

In TinyOS blocking operations are avoided, I/O calls or long-latency operations are usually split-phase: rather than block until completion, a function returns immediately and then the caller gets a call back when the function or I/O driver completes its operations. It also provides tasks which are functions that are executed when every other function call have been terminated. Since only one task can be executed at once there is no warry about data races.

TinyOS uses nesC, a dialect of the C programming language. It doesn't count on dynamic memory allocation or linking. This allows the programmers to analyze their programs in terms of memory occupation at compile time resulting in an efficient code optimization. NesC compiler works like a pre-compiler that takes nesC source code and produces a C code. This C code, then, has to be compiled by a C compiler. The structure of a nesC program is relatively simple: there are interfaces that set out what a components can do by declaring a set of commands and events, commands can be called and events must be handled by every component that declares to use that interface. Components realize one or more interface maybe by using other interfaces. Components are of two

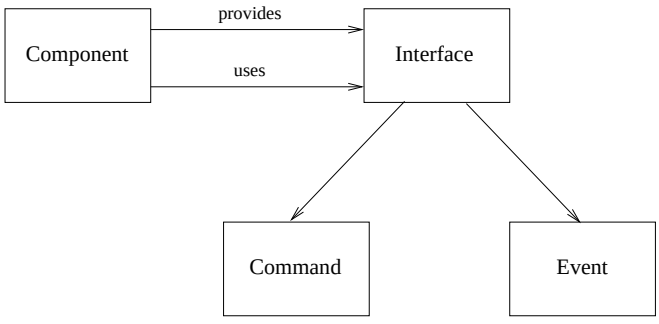


Figure 1.1: nesC program architecture

types: configurations and modules. A module implements interfaces. A configuration connects modules together via their interfaces by providing a wiring specification.

1.1.3 802.15.4 standard

IEEE 802.15.4 standard specifies physical and media access control layers for low-rate and low-power wireless personal area network. Such networks are typically limited to an area of about ten meters width with no infrastructure and limited power availability devices.
It presents a set of network topologies which indicates two types of devices: full-function devices (FFD) and reduced-funcion devices (RFD). RFDs would be simple actuators or sensors like switches or temperature sensors with no large amount of data to send, hence RFDs can be very simple devices. On the other hand, FFDs are smarter than RFDs and can work like Personal Area Network (PAN) coordinator . Therefore FFDs can talk to other FFDs and to RFDs, RFDs can just talk to one FFDs at a time.
In every network cell a PAN coordinator must be present. The smallest network cell is composed by an FFDs acting like PAN coordinator and an RFD connected to the PAN coordinator. This PAN must have a PAN identifier which shall be unique within the radio range. Every PAN can be configured like a star network, where every device must comunicate only with the coordinator; like a cluster-tree network where PAN coordinator uses other FFDs to extend its range and to reach farest devices, RFDs can only partecipate like leave nodes; or like a pure peer-to-peer network, a mesh network, where every FFD can talk to each other, using its neighbor to extend the range, here again RFDs can only talk with the nearest FFD.

Every device shall have unique 64-bit extended IEEE address, set by the manufacturer, that can be used to directly communicate within the PAN. But, a device can use a 16-bit short address that shall be unique just within the PAN.

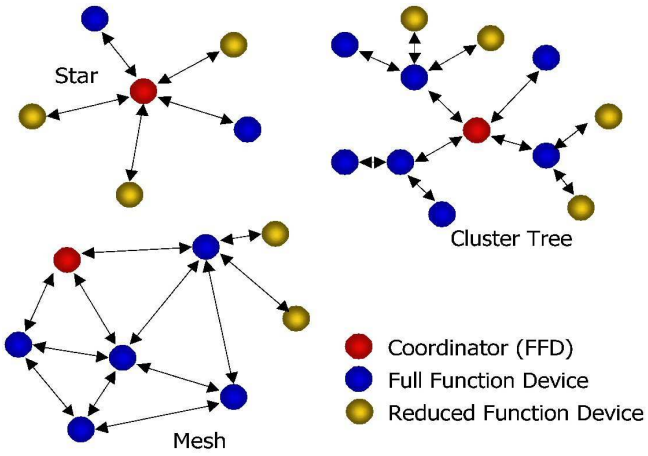


Figure 1.2: 802.15.4 possible network topologies

Therefore a new device that wants to join to a specified PAN needs first to wait for its PAN coordinator which allows the use of a new 16-bit address.
In 802.15.4 standard there are two operation modes: a beacon-enabled mode and non-beacon mode. In beacon-enabled mode, the PAN coordinator periodically sends two beacons to the broadcast address; these two beacons edge a superframe structure.
As it is shown in Figure 1.3, the superframe contains three time sections. The

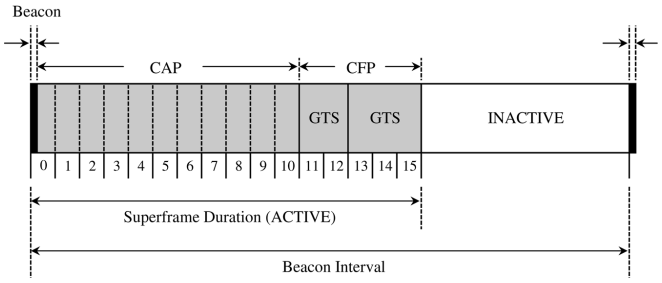


Figure 1.3: 802.15.4 superframe structure in beacon-enabled mode

CAP section (Contention Access Period) is divided in sixteen time slots. The first time slot is reserved for PAN coordinator beacon trasmission, the other

fifteen slots are contended with other devices of the PAN, in a slotted CSMA-CA (Carrier Sense Multiple Access, Collision Avoidance) mechanism. The CFP section (Contention Free Period) is an optional section and it is needed when there are low-latency applications running on devices that need a bandwidth guarantee (GTS means guaranteed time slot). The last section is an inactive section, in this interval of time PAN coordinator usually goes on sleep-mode, and other devices should go too.

There is difference between data transfers from a device to a coordinator and viceversa. When a device needs to send data to the coordinator it uses slotted CSMA-CA during CAP, instead when PAN coordinator has a message for a device, it indicates in the beacon that data are pending for the device. Then device requests it within CAP, the coordinator replies with data within CAP too, both using CSMA-CA. Hence when a device doesn't have data for the coordinator it can't goes on sleep-mode for a while, but it must periodically wake up and listen to beacon to look if there is any message for it.

On non-beacon mode there isn't any superframe structure and an unslotted CSMA-CA mechanism is used. Beacons are still needed for association processes. Data transfer from coordinator to device still occurs with notification-request-reply procedure. If there is no message for the nodes, the coordinator sends a beacon-data frame with zero-length payload.

The max frame length is 128 bytes, that means a payload of about 110 bytes. 802.15.4 frames are associated with a 16-bit CRC to detect errors. Every frame may be acknowledged with the optional use of acknowledgements. We remark that aknowledgements are sent directly without using CSMA-CA, both in beacon and non-beacon mode.

There are also two optional types of security services that can be chosen. In ACL (access control list) mode, devices maintain a list of devices from which they are willing to receive frames. In secure mode, devices use cryptography services in addition to ACL.

1.2 Internet Protocol version 6

IPv6 protocol is supposed to be the next generation internet addressing standard. It is designed to succeed IPv4. It is quite different from IPv4. The most important difference is the addressing space: from the IPv4 32-bit address, (4×10^9 possible addresses), IPv6 goes to a 128-bit address, that is 3.4×10^{38} possible addresses, 6×10^{23} addresses per square meter on the earth. Obviously this change implies that the header length doubles from 20 to 40 byte. Beyond this, IPv6 simplifies some IPv4 problems and limitations like the concept of Network Address Translation (NAT) that becomes obsolete, the DHCP protocol that, in the 6 version (DHCPv6), becomes more powerful and efficient. Furthermore there are some routing process semplifications.

It is impossible to fully describe all changes in IPv6 so, for the scope of this thesis, only the details inherent to 6LOWPAN implementation will be discussed.

An IPv6 packet can carry 1280 bytes of payload and the header format is shown in figure 1.4.

Version, payload length, next header, hop limit, source and destination address

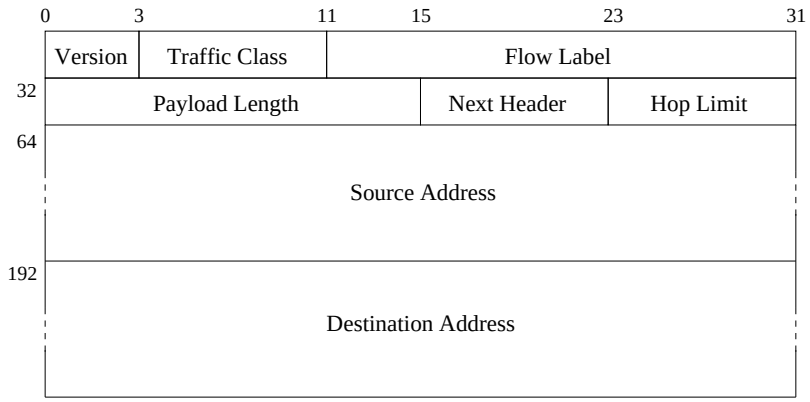


Figure 1.4: Default IPv6 header

fields are the same as in IPv4 header, and their meanings can be simply inferable; traffic class and flow label fields are still on experimental phase, but they will be

Address type	Binary prefix
Unspecified	00...0 (128 bits)
Loopback	00...1 (128 bits)
Multicast	11111111
Link-local unicast	1111111010
Global unicast	everything else

Table 1.1: IPv6 address type identification

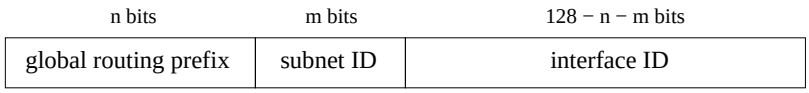
used for QoS and priority queue management. On the contrary some IPv4 fields had been eliminated: checksum is useless, in fact both upper and lower protocols have their error detection mechanisms. Identification and fragment offset fields were elided, this doesn't mean that IPv6 doesn't have fragmentation, but that IPv6 fragmentation is like an option that needs optional header. NextHeader field includes optional headers like source routing, hop-by-hop routing and others, they all have a fixed length with a known pattern, so the length can be calculated, and the payload beginning can be well inferred.

Because of this huge address space extension that IPv6 introduces, address assignement can be rethought in another way.

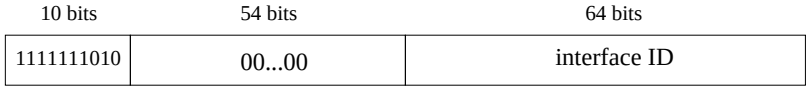
First of all, in IPv6 addresses are assigned to interfaces, and they can be of three types, unicast, multicast or anycast: unicast address indicates one and only one specific interface; multicast address specifies a set of interfaces, a message sent to a multicast address must be delivered to all interfaces of the set; anycast address indicates a set too, but when a message is sent to an anycast address, it can be delivered just to only one interface of the set.

The type of an IPv6 address is identified by its high-order bits as it is shown in table 1.1. Last bits of an IPv6 address represent the interface ID or, for multicast addresses, the group ID. Interface IDs are set up by the device constructor and are permanently stored in the device's memory.

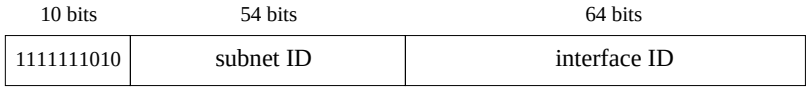
In figure 1.5 some typical IPv6 address structures are shown; figure 1.5b shows



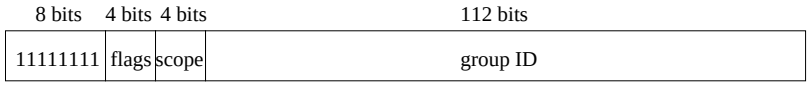
(a) Global unicast address



(b) Link-local unicast address



(c) Site-local address



(d) Multicast address

Figure 1.5: IPv6 address types

the link-local address template; when a device starts up it forms the link-local address without the help of any DHCP server; then using that address as source address it can contact the nearest DHCP server, using UDP, to ask for the global prefix of its subnet to form a global unicast address, and starts to communicate with the whole internet network. The only difference between a link-local address and global address is that routers don't forward link-local packets. Hence subnet masks or NAT services becomes useless.

The IPv6 headers typically is 40 byte long, while IPv6 standard declares that the maximum payload length for a single IPv6 packet can be 1280 bytes, the result is a very efficient division between payload and overhead. But data-link standards typically don't provide messages payload with those dimensions, hence it is often needed to design an adaptation layer to fit IPv6 packets inside data-link messages.

Since in most network topologies, hosts have no energy problems, data-link protocols don't have to save energy or keep under control energy usage, and hence these adaptation layers are quite simple, they just have to provide a way to break into pieces IPv6 packets to let these fragments fit in data-link messages.
In wireless sensor networks, other than packet dimension problems, there are also energy problems to solve, so a more complicated adaptation layer is needed. Moreover another aspect should be noted: in wireless sensor networks, messages carry small amount of data, only configuration informations or small values like temperature or lightness, that can easily fit in a single data-link message, but if hosts use IPv6 protocol they have 40 more bytes to carry in every message, and a noticeable inefficiency appears, especially for what concerns the energy used by radio chip.
6lowPAN working group within the IETF is concerned with the specification for transmitting IPv6 packets over low energy and lossy networks. The group is working on two different documents: header-compression draft and neighbor-discovery draft. Both drafts have arrived at their fifteenth version. The first document describes how to make IPv6 practical on 802.15.4 networks, mechanisms for header compression and for packet fragmentation, and provisions for packet delivery in 802.15.4-based mesh networks. The second draws some changes to IPv6 neighbor discovery process that doesn't suit in low-power, lossy and transitive networks.
In this work, the second draft wasn't been considered, so it will not be commented anymore.

A first raw version of 6lowPAN standard document is RFC 4944 ([1]), in which fragmentation, header compression and mesh dispatching are described.

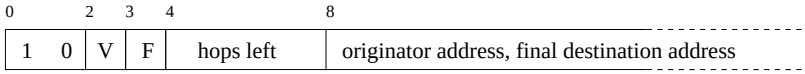


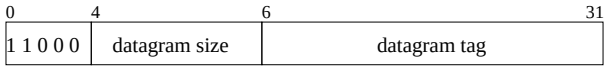
Figure 1.6: Mesh header format

Mesh header

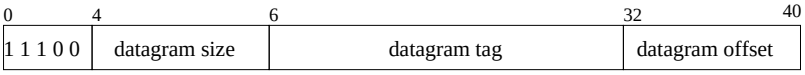
Mesh header has been thought for mesh networks where routing operations is made at data-link level. In that configuration every forwarder host changes the source and destination data-link address fields with respectively its data-link address and the next-hop data-link address. The interesting thing is that every routing operations is made at data-link level, so in order to keep the true source and final destination data-link addresses it is necessary to write them somewhere. Figure 1.6 shows the mesh header format. First two bits compose the pattern to recognize that mesh header is present, V and F flags indicates respectively if the originator or the final destination data-link addresses is written in a 16-bit format or in an IEEE extended 64-bit format. Addresses would follows these flags. With these address informations every forwarder can know who is the originator, who is the final destination and then calculates who is the next hop for that packet to reach the destination host.

Fragmentation process

The fragmentation mechanism uses a fragmentation header that must be present in every fragment of the packet, and some rules on how to break the packet payload and to write the right data in the headers. On figure 1.7 fragmentation headers formats are shown. The first fragment relative to other ones has a different initial pattern value. This difference saves the 1-byte field datagram offset, in fact since the first fragment is recognizable by the pattern, it doesn't need to carry the offset, that is 0. Rules to break the IPv6 packet in two



(a) Fragmentation header for first fragment



(b) Fragmentation header for subsequent fragments

Figure 1.7: fragmentation headers

or more fragments are complicated and have been discussed for long time on 6lowPAN IETF mailing list. First of all, datagram size field must indicate the size of the uncompressed unfragmented IPv6 packet. Then the datagram offset states, by multiple of 8 bytes, the position where to place the fragment payload within the uncompressed unfragmented IPv6 packet. So the IPv6 packet must be broken in parts that are multiple-of-8-byte long, except the last fragment that will contain the remaining bytes. In principle these rules seem to be simple, but in practice there are some problems. In fact the header compressor has to remind the size of the new compressed header and, at the same time, the old size of the uncompressed header just because when the fragmentation module starts breaking the IPv6 packet, it has to take the original size of the IPv6 packet into account to calculate the right size of fragments, even if it will write the compressed version of the IPv6 header.

Anyway this mechanism permits to defragmentation module to instantly allocate a buffer to save and restore the unfragmented IPv6 packet, as it receives the first fragment (chronologically first) of the packet, and, if we are sure that only the first (first by position) fragment will contain compressed headers, it will be able to copy the 802.15.4 frame payload of last (last by position) fragments within the buffer just by watching at the datagram offset of the frame.

Last, there is a tag field that is a random 16-bit field, that must be equal for every fragment of a packet. It is needed to distinguish fragments of different

packets.

IPv6 header compression

There are several rules to compress an IPv6 header. A first series of rules were explained in RFC4944, but after few months a new document has started to be written, more detailed, with more efficient rules and mechanisms. Such rules are explained on the 6lowPAN draft HC-15 ([2]).
The compressed IPv6 header is signaled by the presence of the LOWPAN-IPHC Dispatch. After a distinctive pattern, there are a series of flags indicating how the original IPv6 header has been compressed and which header fields are carried in-line and immediately follow the dispatch. Figure 1.8 shows the Dispatch.
2 bits TF field refers to Traffic Class and Flow Label fields. 4 combinations

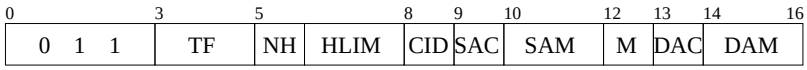


Figure 1.8: LOWPAN IPHC Dispatch

are possible so 4 compression sizes can be used, each combination indicates how many bytes have been compressed and how many bytes are kept from the original IPv6 fields. It is possible to carry fields all in-line (4 bytes) or to carry only one of them, the other is implicitly stated 0, or if they are both 0 it is possible to elide them at all.
1 bit NH field states if the next Header field uses the LOWPAN-NHC compression mechanism or if the IPv6 next header field is carried in-line. Next Header Compression is a particular tecniques that permits to carry some information about transport layer protocol or about IPv6 extension headers within the 1 byte next header field.
2 bits HLIM field compresses the hop limit field; there are some standard hop limit values often used for normal packet transmissions. These three values (1, 64, 255) can be rapresented by a 2 bits value, the fourth combination states that

the hop limit field is carried in-line after the Dispatch.
1 bit CID, SAC and DAC fields deal with the context compression mechanism. It involves a periodically information exchanges by routers to hosts. Document [2] describes how to use these context informations, while document [3] explain how these information should be created and communicated across the network. In practice "context informations" means that hosts should have informations, stored in their memory, about state of the network and other hosts sorrounding them. With these informations shared by every host, it is possible to elide part of, or maybe the whole, internet address of another host. Writing a small code (4 bit) that is used like an index, an host can retrieve in its cache, informations about a neighbor host. This work doesn't deal with this kind of compression.
2 bits SAM and DAM fields states respectively how many bytes of source and destination addresses are written in-line. Possible choices are 128, 64, 16 bits or 0 bit. With a stateless compression, that means no context information available, an address can be compressed if and only if it has a link-local prefix, so first 64 bits can be elided; if the last 64 bits present a particular pattern, it is possible to compress them till 16 or even to 0 bits, that means that the whole address can be calculated by some default known patterns and by address fields of the data-link header.
A special note has to be made about multicast addresses; first of all only destination address can be a multicast address, so the 1 bit M field refers only to destination address and specifies if it is or not a multicast address. If it isn't, destination address compression works like for source address one; if it is, there is another set of patterns to compress it, so if destination multicast address presents one of those patterns it can be compressed to 48, 32 or 16 bits.
Using these compression rules a 40 byte IPv6 header can be compressed to only 3 bytes.

Because of low-power and lossy network behaviours, as transport protocol UDP is often used. In fact UDP doesn't need handshaking operations or ac-

knowledegement mechanisms, so a lot of energy can be saved. The UDP header is much smaller than TCP header, for example, but in [2] a mechanism to compress UDP header too is shown.

As it is explained, if the 1 bit NH value states that LOWPAN-NHC compression

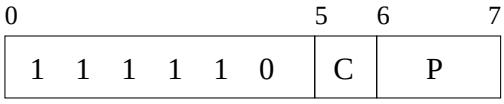


Figure 1.9: LOWPAN NHC header

is used, in the next header field it is possible to carry some information about UDP header. Figure 1.9 shows the 1 byte next header field with UDP protocol informations. First five bits field is a pattern to recognize that informations are about UDP header, C bit states if checksum field is present, while 2 bits P field indicates how source and destination port number fields has been compressed. LOWPAN-NHC compression counts on a set of UDP port numbers that can be fully compressed: if source and destination ports are both in the range that goes from 0xF0B0 to 0xF0BF , they can be compressed in a 1 byte field. If either source or destination port is in the range that goes from Ox0F000 to 0xF0FF it can be compressed in a 1 byte field, the other port number is carried in-line. With this compression mechanism an UDP header can be reduced from a size of 8 bytes to 1 byte.

Last thing to say is about length field of IPv6 and UDP headers: those values can be well inferred either from lower layer or fragmentation header.

Chapter 2

Related work

Abstract:

In this chapter the most important works on 6lowPAN implementation for low-power development platforms are presented. Since it is hard to find information about commercial version of 6lowPAN implementations, for this thesis only academic works were been studied.

2.1 6lowPAN by Matúš Harvan

Matúš Harvan has implemented the very first academic version of 6lowPAN for TinyOS. He shared his work in 2007, and it is based only on RFC4944 [1] since 6lowPAN hc drafts ([2]) ([3]) were not been started to be written yet. So the features of LOWPAN_IPHC compression tecniques are not implemented.
Anyway this implementation is able to manage mesh and fragmentation header, broadcast header (that is an header for link-level broadcast messages) and LOWPAN_HC1 compression mechanism described in [1].
Even if this project still remain a good starting point to study how to build a good implementation of a 6lowPAN module for TinyOS, it has a lot of limitations and gaps. It can't manage two defragmentation processes at same time, so only one fragmented packet at a time can be received. It is impossible to use

another transport protocol since there is no interface to directly access to any IP module or something like that, only UDP datagrams can be sent. Last thing is about the structure of the implementation: when Harvan composed 6lowPAN structures and headers he didn't use packed structure that permit to efficiently store 1-bit flags, but he defined all these structures as traditional structure and then to pick flags he filtered these variables with masks. With packed structures the access to flags becomes easier and more direct, and maybe, since a lot of bit-wise operation to extract 1-bit values becomes useless, some program memory can be saved.

2.2 blip

Blip (Berkely Low-power IP stack) is an implementation in tinyOS of a number of IP-based protocols, that is been being carried on by Berkeley WEBS (wireless embedded systems) group.
Blip first release was on 2008, and now a 2.0 version is available. This last version is based on draft hc-06, so most of the relevant updates from RFC 4944 has been made. WEBS group is working on a new version that will respect last 6lowPAN standard rules, but as written in their website, they are still waiting for a final and approved document, even for neighbor-discovery standard too.
Blip is a very big and well-structured project that supports various interfaces, header files and modules. In fact it doesn't deal only with IPv6 header compression but also with transport layer protocols (UDP and TCP), with neighbor discovery procedures and with routing protocols (next version will have RPL as routing protocol). WEBS group within the first version also have provided a BaseStation application with a configuration script to install an IPv6 network interface on a Linux-based PC to start developing a real IPv6-based sensor network. So Blip is surely a ready-to-go implementation of 6lowPAN for TinyOS.
In this thesis only header compression is dealt with, so all other BLIP parts will

be ignored.

The most considerable thing is how RAM memory is managed while dealing with packets and header compression operations. As said before, in TinyOS RAM memory allocation is static, so variables and buffers are allocated during compilation stage; hence if a big buffer is rarely needed and for the most of the operating time it would be useless, i.e. that amount of RAM memory would stay unused. When dealing with packets of different sizes, this limit is amplificated; in fact even if the most of packets would be few bytes long, it is necessary to allocate an amount of memory to let the biggest possible packet fit in. In blip this problem is solved with a set of functions that manages a huge buffer, called heap, between modules; calling `malloc` and `free` functions every blip modules can ask for some memory and then release it. With this original innovation, it is possible to deal with more than one packet at a time while using less memory than the biggest possible packet size.

Blip also provides more than one transport protocol interface, so it is possible to choose between UDP and TCP, but if both of this protocols are not suitable, it is possible to directly link applications to IPv6 interface, to call `send` and `rcv` (receive) commands, and hence to implement an proprietary transport protocol.

On the contrary blip is very heavy both in terms of program and RAM memory: a small application like `UDPEcho` that just answers to request made to port 7, compiled with blip, weighs about 25 kilobytes in program memory and 5 kilobytes in RAM memory; so in a platform like telosB it remains only 23 kilobytes in program memory and only 5 kilobytes in RAM memory. Generally an application needs not only a transport application but also an application protocol like soap or coap; so less than 23 kilobytes may be not enough for an application protocol and the application too.

2.3 Contiki and μIPv6

Contiki is an open source operating system for memory-constrained networked embedded systems. It is written by Adam Dunkels from the Swedish Institute of Computer Science. Contiki is designed for embedded systems with small amounts of memory. A typical Contiki configuration is 2 kilobytes of RAM and 40 kilobytes of ROM.

Like TinyOS, Contiki adopts an event-driven system to manage memory and threads, but, opposed to tinyOS, it permits to dynamically load and unload programs and services.

One of Contiki's main features is a set of well structured and lightweight network protocol stacks which μIPv6 is surely the leading edge. μIPv6 is the world's smallest certified IPv6 stack, it can runs on IEEE 802.15.4 and Ethernet, its dimension is about 11 kylobytes of program memory and 1.8 kylobytes of RAM memory. Within μIPv6 there is a set of header and function files, so called SICSlowpan, that realizes a 6lowPAN implementation that respects RFC4944 and the second version of 6lowPAN header-compression draft (hc-01). As reported, a full-feature IPv6/6lowPAN Contiky OS image weighs 40 kylobytes of program memory and 10 kylobytes of RAM memory. SICSlowpan is not imlemented like a service but like a set of functions, that are called by MAC when a packet arrives or by the IP service when there is a packet to send. Like 6lowpan by Harvan SICSlowpan defragmentation function can't process more than one packet at a time, so while it is recostructing one packet, all fragments that doesn't belong to that packet will be dropped. Mesh header and link level routing are ignored since Contiki targets the route-over tecnique.

2.4 TinyNET

TinyNET [4] doesn't deal with 6lowPAN, but it is a framework that allows development and a quick integration of network protocols in TinyOS. It is developed

at the University of Padova. The development originated from the fact that very few applications are actually built based on reusable components, since the most widespread approach is to implement ad hoc, monolithic blocks that deliver the required functionalities. The original idea is to create a general structure to support the use of different protocols without totally changing the backbone of TinyOS network stack. With this idea an architecture made of interfaces and configuration files has been implemented, it permits to anyone to implement his own network protocol at any level of protocol stack, without thinking about on how to integrate his protocol in TinyOS network protocol stack.

Since the importance of this thesis is not how 6lowPAN compression functions has been implemented, but how the whole architecture has been thought, and since tinyNET structure design is one of the basic principles of this project, an introduction to this work is in order.

Chapter 3

Implementation

Abstract:

Chapter starts with a general and high-level description, then goes down analyzing interfaces commands and events and some specific and peculiar functions that distinguish this work from other implementations.

3.1 Introduction

Before starting to describe project objects and architecture features, it is necessary to make some remarks.
In this 6lowPAN implementation work the author has had a relevant role at each steps, from the beginning when other tinyOS 6lowPAN implementations must be studied and analyzed to find lacks, through the main steps when the architecture has been designed, and the implementation has been made, even till the end when tests were realized and executed.

3.2 Design principles

The design stage of this sixlowpan implementation were made in collaboration with Eng. Angelo P. Castellani and Eng. Mattia Gheda, who had the lead of starting this new project. They has some specific ideas and objects about how to realize an efficient and versatile network stack and how to structure interfaces and components of this 6lowpan implementation:

- RAM memory centric optimization: create a memory manager component that would manage RAM memory for the whole programs and applications running on a node. Focus on RAM management that compensates that TinyOS gap about static RAM allocation. This module provides two basic functions to deal with RAM memory from applications: `alloc` and `free` functions, and some other functions properly designed for network protocols: `realloc` and `hrealloc`, functions that extends or reduce a buffer respectively on tail or head. These kind of function are very useful when dealing with packets, headers and footers;
  Figure 3.1 shows that this new component's functions and features will be

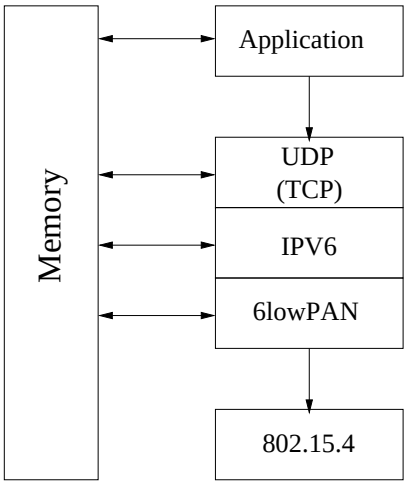


Figure 3.1: Design of memory component architecture

shared by all components of a node maybe by application programs too, not only when they have to send a message but also for their own functions.

- modular standard support: incapsulate as much as possible all protocol dependant procedures and functions to allow future updates or to make functions portable to other platforms or other OSs;

- clear layer design: split layers as much as possible and avoid modules that include more than one protocol to simplify the design operations and code. But at the same time keep reduced the number of function calls that weigh a lot on code dimension. With the right data structures, function calls can be reduced and code can be made light. Hence if the architecture is well designed, it is possible to take advantage of modular functions keeping code light.

- level 2 and level 3 routing support: develop both route-over and mesh-under routing mechanisms to postpone the choice. With route-over, nodes build routing tables with IPv6 addresses and routing protocol works with IPv6 messages, at data-link level only an IPv6 to data-link address translation is needed since route-over provides IPv6 address of next-hop hosts. On the contrary with mesh-under, nodes build routing tables with data-link addresses and routing protocol works with data-link messages, IPv6 module communicates to lower levels the destination's IPv6 addresses ignoring how routing operations are made.

RAM memory manager was designed implemented and tested before this project started, so it will be discussed but not referred to as a component designed during this thesis work.

3.3 Architecture

In figure 3.2 the overall protocol stack architecture with raw rapresentations of different modules is shown.

As it is shown every network protocol layer has its own module, so if a right

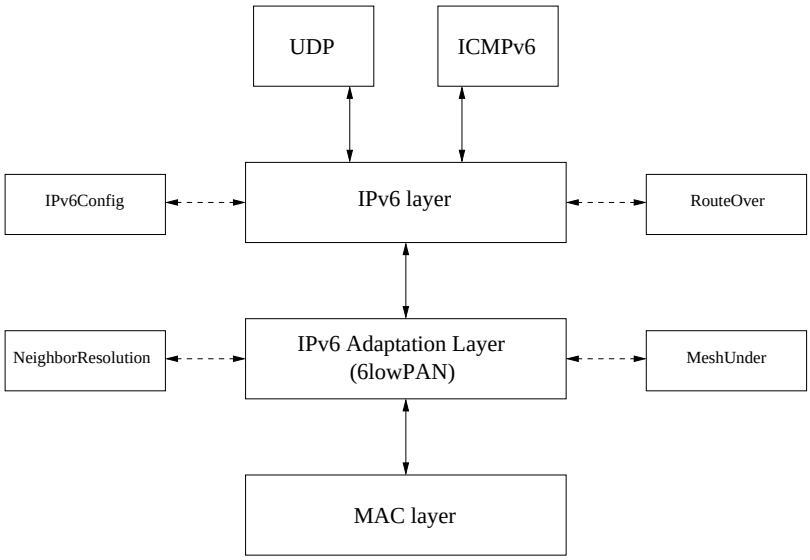


Figure 3.2: An outline about system architecture

set of interfaces and data structures are written with right set of input parameters to each commands and events, it is possible to implement, for example, a different IPv6 adaptation layer for the same MAC layer without modifying any other modules. The ISO-OSI protocol stack principle that says that every layer is indipendent from others is kept. This guideline together with the RAM memory component that stores and shares data between modules, it make possible to implement a generic, reusable and, at the same time, light and efficient structure that stays indipendent from which protocol standard is chosen.

The same reasoning can be made for what concerns routing protocols. A routing protocol has substantially to answer to few questions that may be asked by network or data-link layers about next hop hosts' addresses to reach a given des-

tination. Routing protocols try to find these answers with a messages exchange between nodes. So a routing protocol implementation needs to send and receive messages and have to answer to next-hop questions.

```
interface RouteOver {
  command void getNextHop (slp_ip6_addr_t* addr, slp_ip6_addr_t*
      nextHop);
  command void forwardAddr (slp_ip6_addr_t* addr, slp_ip6_addr_t*
      nextHop);
}
```

Listing 3.1: Route over routing interface

```
interface MeshUnder {
  command ieee154_saddr_t getNextHop (slp_ip6_addr_t* addr);
  command ieee154_saddr_t forwardAddr (ieee154_saddr_t addr);
}
```

Listing 3.2: Mesh Under routing interface

Hence as it can bee seen in figure 3.2 and in listings 3.1 and 3.2, routing interfaces are simple but complete. IPv6 module will be linked to `RouteOver` interface while 6lowPAN module will use `MeshUnder` interface's commands.
Obviously these two modules will never work at the same time, since just one routing module is needed, so if route-over is chosen, mesh-hunder become an useless module and viceversa.
Both MeshUnder and RouteOver interfaces have two commands that, apart the name, appear to be equal; the difference is that with `getNextHop` command routing module must answer with a valid address, but with `forwardAddr` command routing module can answer with a null address if it doesn't want that messages are forwarded.
For messages exchange, RouteOver module will be linked to ICMPv6 module, while MeshUnder module will be directly linked to tinyOS radio drivers.

With this configuration both route-over and mesh-under routing modules become two black boxes, IPv6 and 6lowPAN modules don't need to know anything about them, which routing protocol is used, if route-over or mesh-under is used, what kind of informations are exchanged, it is possible to implement any kind of routing protocol without affecting any other modules.
This aspect is key for this thesis. To build a structure that permits to reuse codes, to implement the newest protocol version without any other thought about the whole structure of the system.
In figure 3.2 there are two more modules not been presented yet: NeighborResolution module provides commands to translate an IPv6 address in a data-link address; IPv6Config module provides all features inherent to host addresses. Every data-link interface can have more than one IPv6 address, normally it has a link-local and a global unicast address, but also it can be registered to one or more multicast addresses; hence IPv6Config maintains and manages a cache with all these addresses.

3.4 Memory module

```
interface Memory {
    command memory_id_t alloc(memory_size_t size);
    command void free(memory_id_t id);
    command void * id2p(memory_id_t id, memory_size_t* size);
    command error_t realloc(memory_id_t id, memory_size_t size);
    command error_t hrealloc(memory_id_t id, memory_size_t size);
}
```

Listing 3.3: memory component interface

As it can be seen in listing 3.3 memory interface provides commands to handle RAM memory: `memory_size_t` and `memory_id_t` are two `uint16_t` data types, the first one is used to define the memory size in bytes, the second one is used to identify an allocated buffer.

When an application has to send a message it calls `alloc` function that returns a valid ID (if such amount of memory is not available it returns 0), then the application calls `id2p` function to take back a pointer to memory space from the ID, to use for writing data. After writing it can pass the ID to the transport protocol for sending. UDP component has to add its own header to the buffer head, so it calls `hrealloc` function that adds an amount of bytes taken as input parameter, if there is no error signals, UDP calls again `id2p` function to begin writing its header; then it passes the packet to lower layer, and so on.
The same procedure is used when a packet is received: the lowest layer that handle the IPv6 packet (`sixlowpan`) asks for a buffer, writes the packet and passes the packet to upper layer, which takes a pointer to the datagram, starts reading its own header and then resizes the buffer by calling `hrealloc` function with a negative value of `size` parameter.
So applications payload, once it is written by the lowest network layer, substantially doesn't move any more till it arrives to the application program that can start reading it and maybe delete it by calling `free` function.
With this tecnique there is no problem about buffer pointers that change values or become obsolete, and there is no need to copy a huge amount of bytes to move data to another buffer.
To implement these features, the memory component uses one huge block of RAM memory statically initialized and few structures made of a modified Pool component, called `SortedPool`. This pool differs from a classic TinyOS pool: sorted pool assigns an ID to every active element, and it is able to get back a pointer to the element by its ID.
In the memory component there are two lists made of those modified Pools, one called `FreeList` and one called `OccList`; these two lists represent blocks of buffer's memory that are respectively free or allocated. At the beginning, OccList is empty and FreeList has one element that represents a block of empty memory as large as the whole RAM memory assigned to this component. When

`alloc` function is called, it checks in the free list if there is an element that represents a block of empty memory bigger or equal than the requested size. If there is, it resizes that element and also add a new element in occupied list. When a buffer has to be cleared, the referred element in the occupied list is removed, and a new element in free list is added, then a check is made: if there are two contiguous elements in free list, they are joined in one element.
`hrealloc` function before doing the same job as `alloc` function, checks if there is an element in free list that represents an amount of free memory that lies before allocated buffer. If there is, it removes the element in the free list and adds that amount of memory to the element that represents the buffer, if there isn't, it works like the `alloc` function, i.e. it looks for a free memory block as large as the sum of the allocated buffer and the `hrealloc` input size parameter, then copies data from old buffer to that just allocated, and frees the old one. `realloc` function works like `hrealloc` function on the buffer tail.

3.5 sixlowpan module

The most significant component that was implemented is called `sixlowpan`; it provides few interfaces and few header files which include the most significant and crucial functions to implement the 6lowpan layer.
In figure 3.3 header file's names and what they realize are shown. Files with `.h` extension define structures and constants for headers, while files with `.c` extension define functions to handle these structures. As it is shown, 6lowPAN headers was splitted in two different header files, one called `RFC4944` where fragmentation and mesh header are handled and the other `HC15` where last 6lowPAN header compression tecniques, described in [2], are implemented.
For better understanding on how these header files are involved in 6lowPAN implementation, some structures and function prototypes are shown.

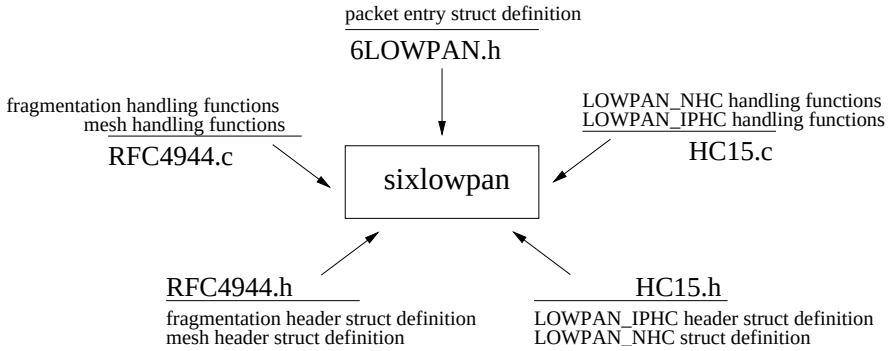


Figure 3.3: An outline about sixlowpan module and header files

```
typedef struct  {
   memory_id_t packet;
   slp_ip6_addr_t nextHop;
   bool checksumElide;
   slp_mesh_opt_t mesh;
   slp_frag_opt_t frag;
   slp_context_t context;
   uint16_t byteLeft;
} slp_packet_entry_t;
```

Listing 3.4: packet entry data structure

```
typedef nx_struct       {
        nx_uint8_t pattern:2;
        nx_uint8_t v:1;
        nx_uint8_t f:1;
        nx_uint8_t hopsLeft:4;
        nx_uint16_t originator;
        nx_uint16_t finalDest;
} slp_mesh_opt_t __attribute__((packed));
```

Listing 3.5: mesh header structure

```
typedef nx_struct       {
        nx_uint16_t pattern:5;
```

```
        nx_uint16_t size:11;
        nx_uint16_t tag;
        nx_uint8_t offset;
} slp_frag_opt_t __attribute__((packed));
```

Listing 3.6: fragmentation header structure

Listing 3.4 shows the crucial structure, used both by sixlowpan send and receive commands, to store fundamental informations to handle IPv6 datagrams with the minimum number of external function calls.

`memory_id_t` and `slp_ip6_addr_t`, are the memory ID where the packet is stored and IPv6 address of the next hop host; `slp_mesh_opt_t` and `slp_frag_opt_t` are respectively mesh and fragmentation headers of the IPv6 packet, stored in proper structures (shown in listings 3.5 and 3.6) of IPv6 packet; `byteLeft` variable is used by receive procedure to keep trace of how many bytes remains to complete the IPv6 packet. `checksumElide` boolean variable indicates if LOWPAN_NHC compression have elided the checksum field. Since to elide checksum field sixlowpan needs the permission from application program, this variable should state the application's order.

```
typedef nx_struct       {
        nx_uint16_t pattern:3;
        nx_uint16_t tf:2;
        nx_uint16_t nh:1;
        nx_uint16_t hlim:2;
        nx_uint16_t cid:1;
        nx_uint16_t sac:1;
        nx_uint16_t sam:2;
        nx_uint16_t m:1;
        nx_uint16_t dac:1;
        nx_uint16_t dam:2;
} slp_hc15_header_t __attribute__((packed));
```

Listing 3.7: LOWPAN_IPHC header dispatch structure

```
typedef nx_struct        {
        nx_uint8_t pattern:5;
        nx_uint8_t c:1;
        nx_uint8_t p:2;
} slp_hc15_udp_nhc_t __attribute__((packed));
```

Listing 3.8: LOWPAN_NHC dispatch structure

In listings 3.7 and 3.8 LOWPAN_IPHC and LOWPAN_NHC header structures are shown. Field and flag's names respect those one assigned in the draft document [2].

Those structures are practically never instantiated, but only pointers of this structure types are instantiated, and then casted to a generic buffer. By this procedure, the access to flags is direct and doesn't need any masks or bitwise operations, and at the same time code is kept simple and more readable.

```
uint8_t fill1stMsg (slp_packet_entry_t* entry, void* messagetPayload
    , uint8_t messagetPayloadLength, void* packetPayload, uint16_t
    packetPayloadLength, void* compHeader, uint8_t compSize, uint8_t
     originSize);
uint8_t fillOtherMsg (slp_packet_entry_t* entry, void*
    messagetPayload, uint8_t messagetPayloadLength, void*
    packetPayload, uint16_t packetPayloadLength);
uint8_t decompress (void* messagetPayload, slp_packet_entry_t* entry
    );
uint8_t fillPayload (slp_packet_entry_t* currentEntry, void*
    packetPayload, void* messagetPayload, uint8_t
    messagetPayloadLength);
```

Listing 3.9: Some of the most important function declarations of RFC4944.c file

In listing 3.9 some functions used to handle mesh and fragmentation headers are shown. `fill1stMsg` function is used to fill an IEEE 802.15.4 message with first IPv6 packet's fragment or even with the whole IPv6 packet if it is small

enough. It receives pointers and sizes of the data-link message payload and the IPv6 packet, it also receives pointer and sizes of the IPv6 header before and after compression, to rightly fill data and to calculate the offset. `fillOtherMsg` is used to write other IPv6 packet fragments in data-link messages, it doesn't need any information about the IPv6 compressed header since in subsequent fragments the IPv6 header would not be present and the offset value is sufficient to calculate which bytes to send are remaining. `fillPayload` function is used in the receive process to write the IPv6 packet payload within the buffer.

```
uint8_t HC15Compress (void* packetPayload, void* buffer, uint8_t*
    originSize, bool useMeshOrNeigh, bool checksumElide);
uint16_t HC15decodeHeader (slp_packet_entry_t* entry, void*
    messagetPayload, uint8_t messagetPayloadLength, void*
    packetPayload, ieee154_saddr_t macSrcAddr, ieee154_saddr_t
    macDestAddr, uint8_t* originSize);
```

Listing 3.10: Some of the most important function declarations of HC15.c file

Listing 3.10 shows the two fundamental functions to compress and decompress the IPv6 header. `HC15Compress` function needs a pointer to IPv6 header begin and a pointer to a buffer where to write compressed header, then it returns sizes of header before and after the compression process. The two parameters `useMeshorNeigh` and `checksumElide` are needed to state if it is possible respectively to elide the last 2 bytes of IPv6 addresses and the UDP checksum field. `HC15decodeHeader` function decompress the IPv6 header by reading it in the first fragment of an IPv6 packet and then writing it in a buffer and, as for compression function, it has to return both compressed and decompressed sizes of IPv6 header. `macSrcAddr` and `macDestAddr` parameters are needed in the case that last 2 bytes of IPv6 addresses have been elided.
The sixlowpan module mainly implements the 6lowPAN standard.

```
interface IPv6Adaptation          {
  command error_t send(memory_id_t pck, slp_ip6_addr_t* nextHop);
  event void sendDone (memory_id_t pck, error_t error);
  event void receive  (memory_id_t pck);
}
```

Listing 3.11: IPV6Adaptation interface

As it is shown in listing 3.11 there is not a 6lowPAN interface, but a more generic interface `IPv6Adaptation`. This solution keeps the system open to future improvments and development also for other possible adaptation layers. It is a quite simple interface with minimal commands and events: `memory_id_t` is the ID of the memory space where IPv6 datagram is located and `slp_ip6_addr_t*` is a pointer to IPv6 address of the nextHop. As already explained, the name nextHop doesn't force IPv6 to provide a true next hop address, in fact if route-over will be used, next hop will be true next hop and NeighborResolution module will translate that address to a data-link address, on the contrary if mesh-under will be used nextHop will be the destination address and the true next hop data-link address will be calculated by mesh-under module.

Since the sixlowpan component is the first component in the network proto-

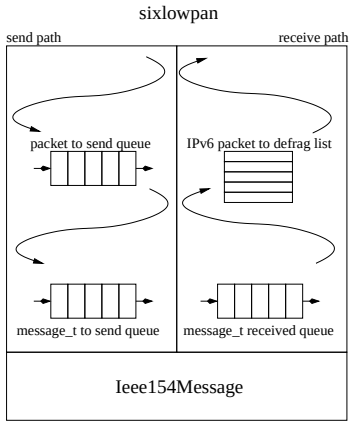


Figure 3.4: sixlowpan queue architecture and paths of send and receive processes

col stack that has to deal with IPv6 packet and data-link message at the same time, and it has also to deal with a packet fragmentation process, some pools and queue are needed. So the memory component become less useful than it is for IPv6 or UDP module. In fact here it is just used to write or read the IPv6 packet, not to add or remove 6lowPAN headers.

Furthermore, sixlowpan module must implement a lot of operations before sending an IPv6 packet, so a non monholitic solution has been chosen. In fact, if all operations would be made in a single step included in the send command, CPU would be pre-empted for too much time. So the operations has been split in three phases computed by tasks. The first phase is implemented by sixlowpan `send` command and consists in instantiating a new packet entry and filling this entry with the most part of the information that can be calculated in that moment, like mesh header or IPv6 next hop address. Last operation is to enqueue the entry in the packet entry queue. The second phase is implemented by a task, called `packetTask`, it makes the most important and long time operation, that is popping first element of packet entry queue, extracting the next fragment to send, if it is the first, then compressing IPv6 header, instantiating a new IEEE 802.15.4 message, writing payload and filling MAC header with the relevant information, and, last, enqueuing the message in the message queue. The third and last phase is made by another task, called `sendTask`, that has simply to pop

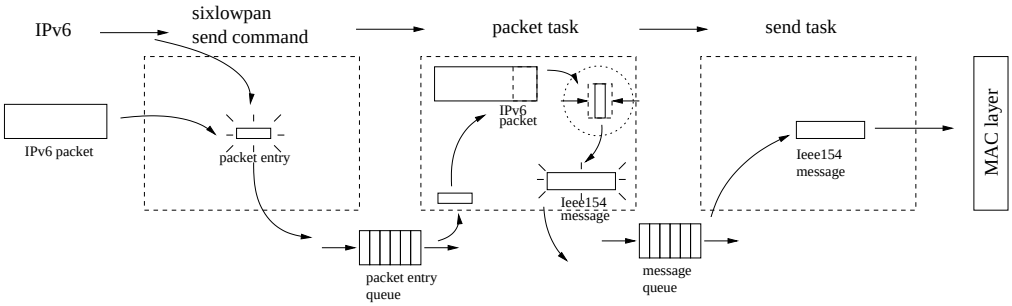


Figure 3.5: sixlowpan sending procedure workflow

first element of message queue, call `send` command of `Ieee154Message` interface and wait for `sendDone` signal to check if message is sent.

As it is shown in figures 3.4 and 3.5 every task manages its own queue. They are posted when an element is enqueued and they don't stop executing till the queue becomes empty. In this way CPU doesn't stay busy in executing one single function but it can be requested by other functions more frequently, making a lighter and prompt system.

The receive procedure is much less complicated, in fact just one task is used. When tinyOS driver signals that a new message is arrived, sixlowpan `receive` calls the event handler, enqueue the message in a receive message queue and then posts the `receiveTask`. The task pops the message, extracts headers, then decompresses IPv6 header if present. If message is a fragment of an IPv6 packet it checks if fragmentation header tag value is already present in the packet-to-defrag list, then if it is, it writes payload in the right place in the buffer, if it is not, it allocates a new buffer, write the payload and puts a new entry in the list. When an IPv6 packet is received and defragmented, it passes the buffer ID to the upper module that handles IPv6 header.

We remark that it is the receiveTask that checks if there is a mesh header and if the message has to be forwarded, it asks to MeshUnder module to provide a next hop address, it changes data-link destination address and then enqueues message in the message-to-send queue.

Sixlowpan module also provides an Ieee154MessageSend and Ieee154Receive interface. Since maybe it is necessary for some applications to directly send and receive non-IPv6 messages, to recognize these kind of messages it is necessary to add on payload head one byte with a known pattern that signals that message is out of IPv6 protocol communications. This job is made by sixlowpan module, and when a non-IPv6 message has to be sent, sixlowpan moves the message payload to empty the first byte and write the NALP (not a lowpan packet) pattern. The same thing must be done when a message is received: sixlowpan

firstly check if NALP pattern is present, if it is, it signals a received message on `Ieee154Receive` interface, otherwise it enqueues the message as explained before.

3.5.1 Compression and decompression processes

Since it is a procedure explained in draft hc-15, compression and decompression functions is performed in HC15.c file.

```
...
slp_ip6_header_t* header = NULL;
slp_UDP_header_t* UDPheader = NULL;
slp_hc15_header_t* HC15Dispatch = NULL;
slp_hc15_udp_nhc_t* nextHeaderCompress = NULL;
...
header = (slp_ip6_header_t*) packetPayload;
packetPayload += sizeof (slp_ip6_header_t);
...
HC15Dispatch = (slp_hc15_header_t*) buffer;
buffer += sizeof(slp_hc15_header_t);
headerSize += sizeof(slp_hc15_header_t);
...
```

Listing 3.12: HC15Compress function's code fragment to show structures use.

As said before, structures like `slp_hc15_header_t` or `slp_hc15_udp_nhc_t` are never instantiated, only the pointer of those structures types are used. In listings 3.12 this use is shown: pointers are initially instantiated with a NULL value; then the pointers to the buffers (`packetPayload` and `buffer`) are casted to be those structure type pointers.
Other code fragments show how compression processes is made, in particular 3.13 refers to the hop limit field, while 3.14 shows source address compression process.

```
switch (header->hopLimit)        {
case 1:
    HC15Dispatch->hlim = SLP_HC15_HLIM_1;
        break;
case 64:
    HC15Dispatch->hlim = SLP_HC15_HLIM_64;
        break;
case 255:
    HC15Dispatch->hlim = SLP_HC15_HLIM_255;
        break;
default :
    HC15Dispatch->hlim = SLP_HC15_HLIM_INLINE;
    memcpy (buffer, &(header->hopLimit), sizeof(header->hopLimit));
    buffer += sizeof(header->hopLimit);
    headerSize += sizeof(header->hopLimit);
        break;
}
```

Listing 3.13: HC15Compress function's hop limit compression process.

```
temp = &(header->source);
HC15Dispatch->sac = SLP_HC15_SAC_STATELESS;

if (memcmp(temp, &SLP_LINKLOCAL_NET_ADDR, sizeof(
    SLP_LINKLOCAL_NET_ADDR))!=0)    {
    HC15Dispatch->sam = SLP_HC15_SAM_128;
    memcpy(buffer, &(header->source), sizeof(header->source));
    buffer += sizeof(header->source);
    headerSize += sizeof(header->source) ;
}    else    {
    temp += sizeof(SLP_LINKLOCAL_NET_ADDR);
    if (memcmp(temp, &SLP_EUI64_SHORT_ADDR, sizeof(
        SLP_EUI64_SHORT_ADDR))!=0)    {
        HC15Dispatch->sam = SLP_HC15_SAM_64;
        memcpy(buffer, temp, sizeof(header->source) - sizeof(
            SLP_LINKLOCAL_NET_ADDR));
```

```
        buffer += sizeof(header->source) - sizeof(
            SLP_LINKLOCAL_NET_ADDR);
        headerSize += sizeof(header->source) - sizeof(
            SLP_LINKLOCAL_NET_ADDR);
    }   else    {
        if (useMeshOrNeigh)
            HC15Dispatch->sam = SLP_HC15_SAM_0;
        else    {
            HC15Dispatch->sam = SLP_HC15_SAM_16;
            temp += sizeof(SLP_EUI64_SHORT_ADDR);
            set_16t(buffer, get_16t(temp));
            buffer += sizeof(header->source) - sizeof(
                SLP_LINKLOCAL_NET_ADDR) - sizeof(SLP_EUI64_SHORT_ADDR);
            headerSize += sizeof(header->source) - sizeof(
                SLP_LINKLOCAL_NET_ADDR) - sizeof(SLP_EUI64_SHORT_ADDR);
        }
    }
}
```

Listing 3.14: HC15Compress function's source address compression process

Just to better understand: `set_16t` function is needed to solve well known problems about TinyOS' c compiler for MSP430 MCU that causes some troubles when copying 2 bytes fields.
Decompression function works in the same manner, buffer pointers are casted to be specified structure pointers and then, by simply reading LOWPAN_IPHC flags, IPv6 header is rebuilt.

3.5.2 Fragmentation and defragmentation processes

Fragmentation and defragmentation functions are implemented in RFC4944.c file. The mechanism counts on more than two functions since it involves data-link payload filling and extracting.

```
if ( packetPayloadLength - originSize + compSize <=
    messagetPayloadLength)    {
  ...    // IPv6 packet can fit in a single message_t
}   else   {
   ...  // writing fragment header
   if (compSize < messagetPayloadLength)    {
      ...// fill first fragment with IPv6 compressed header and
          first fragment of payload
   }   else   {
      // condition not considered
   }
}
```

Listing 3.15: fill1stMsg function's code fragment

As explained, fragment header offset value must be calculated taking care of uncompressed IPv6 header, so `fill1stMsg` function, that deals with filling data-link message's payload with first fragment, receives both size values of the IPv6 header, before and after compression. This function has firstly to check if the whole IPv6 compressed packet can fit in a single message, if not then starts with fragment operations. In listing 3.15 this checks are shown, in particular the first `if` statement is made to check if 6lowPAN packet can fit in a single data-link message, this control is made by picking IPv6 packet size value `packetPayloadLength`, subtracting IPv6 header size value `originSize` (it may takes care of compressed UDP header) and then adding 6lowPAN header value `compSize`. The second `if` statement is made to check if 6lowPAN header can fit in a single data-link message. This statement must be always true, since, as written, the false case is not handled.

```
...    //copy fragmentation header
messagetPayloadLength -= sizeof(entry->frag);
packetPayloadLength -= entry->frag.offset*8;
if (messagetPayloadLength < packetPayloadLength)    {
```

```
messagetPayloadLength = (uint8_t) (messagetPayloadLength / 8);
memcpy (messagetPayload, packetPayload, messagetPayloadLength*8);
...
entry->frag.offset += messagetPayloadLength;
}   else   {
...   /copy last bytes of packet payload
}
```

Listing 3.16: fillOtherMsg function's code fragment

In listing 3.16 `fillOtherMsg` function code is shown: after writing the fragmentation header in the 802.15.4 message payload, `messagetPayloadLength` and `packetPayloadLength` values are calculated, first one by subtracting the size of fragmentation header, second one by subtracting the offset value which is stated as an 8 multiplier. Then if IPv6 packet payload left over is still larger than 802.15.4 message payload, the available space in the data-link message is divided by 8 and rounded to obtain a minimum common multiplier of 8, which will be the amount of bytes (multiplied by 8) of the IPv6 packet that will be written in the message. On the contrary if IPv6 packet bytes left over are less than the data-link message payload, they are directly written, and since the IPv6 packet is sent at all, no more offset value has to be calculated.

Functions to defragment an IPv6 packet are so simple that no code samples are needed.

`decompress` function simply checks if the fragmentation header is present on data-link message head, then if there is, it copies the header in the `entry->frag` structure.

`fillPayload` function receives pointers to the data-link message and to the IPv6 packet buffer, it calculates the offset by reading its value in the `entry->frag` structure and then it copies the right amount of bytes in the buffer. In the case of first message which contains 6lowPAN header, receiveTask, before calling `fillPayload` function, moves the IPv6 packet buffer's pointer to the first byte

after the decompressed IPv6 header, so `fillPayload` function, that reads a 0 in the offset field, doesn't notice that it is writing not in the real first buffer's byte, which would a wrong position, but in the first byte after the IPv6 decompressed header.

3.6 IPv6 module

IPv6 component is very simple, it doesn't have any tasks, send command and receive event handler do their job at once.
Send command doesn't provide any extension header features, so it has to add 40 bytes of the default IPv6 header to the packet buffer, then it fills IPv6 fields and calls send command of the sixlowpan component.
Same thing is done by receive event handler that can't recognize any extension headers and it just reads the default IPv6 header, removes IPv6 header by using `hrealloc` Memory's function, and redirects receive signals to right transport protocols.

3.7 UDP module

As the IPv6 module, the UDP functions rely on the memory module to add and remove UDP header from application packets. `send` and `receive` processes are entirely held in single functions and they simply handle the classic UDP header. UDP module is linked to IPv6 module by its IANA next Header number, which is 17. On the other hand, applications can be linked to UDP module by the source port number, since UDP doesn't deal with LOWPAN_NHC compression mechanism and hence it doesn't know anything about port patterns for compression, there is no formally restrictions when choosing port numbers. Applications and application programmers, will decide what port number to use and they will take care about squeezable port numbers.

Chapter 4

Testing and results

Abstract:

In this chapter, test results are shown. Since neither neighbor discovery nor dhcp client modules were implemented or developed, test procedures only deal with point-to-point communications, IPv6 addresses have been statically assigned to nodes, and routing modules only return default values. For the same reason, even mesh header using was not tested.
Hence principally these trial programs aim to put this sixlowpan implementation under stress situations, both in compression and decompression, fragmentation and defragmentation procedures, to find the saturation points of the send and receive functions.

4.1 Testing procedures

All test programs work over UDP protocol with ports and addresses values set to permit to fully compress headers. Principally three types of test have been made: one to test the send functions, one to test the receive functions and the last to make this implementation compatible with the most important tinyOS 6lowPAN implementation i.e. blip.
Throughput performances aimed by this implementation are all compared with

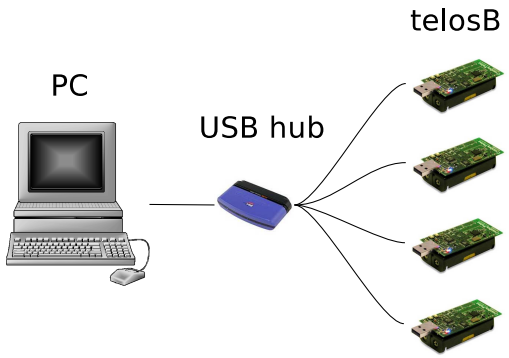


Figure 4.1: Network configuration during tests

those aimed by tinyOS CC2420 radio drivers, so it will be possible to weigh this sixlowpan implementation on the whole network protocol stack of tinyOS.

4.2 Send section

Send test programs aim to find the maximum bitrate that send functions can substain. Normally to find the maximum throughput of a network protocol, an application send a message and wait for the send-done signal before sending another one; the throughput value is calculated by counting the number of messsages sent every second. But, since sixlowpan module makes use of few pools, queues and tasks, to real stress the send functions this kind of procedures would not be enough.

So test programs are designed in a way that pools and queues are filled as much as possible and hence tasks never stop executing themselves: the application requests to the UDP module to the send a packet every specific time interval without waiting for send-done signals, throughput is calculated by taking note of the time interval when a lot of error messages are returned by send command. Since some parameters, like pools dimension, must be set at compile time, to study the best configuration and to find the best performance a complete set of parameters was used, where all parameters combinations are included.

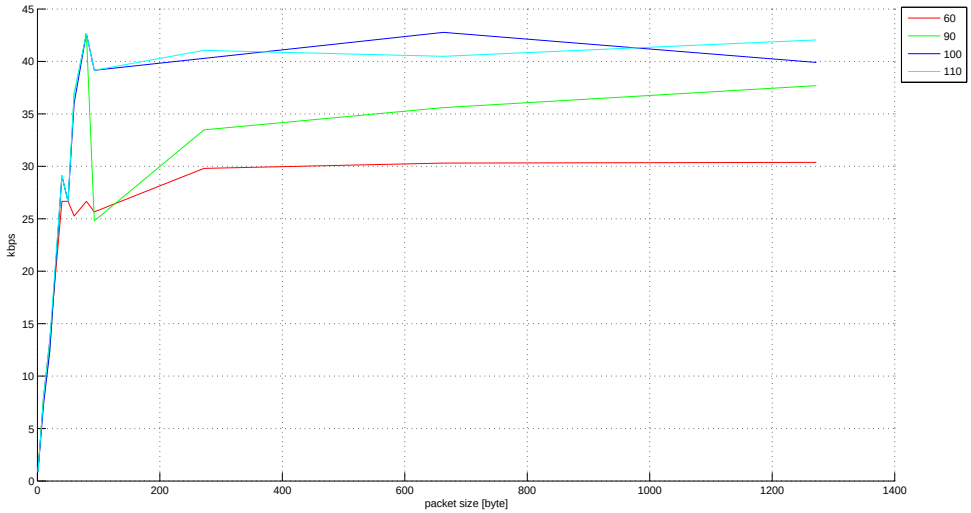


Figure 4.2: Throughput in function of UDP packet size for different IEEE802.15.4 frame dimensions

To make these tests automatic few scripts have been written in bash language, these scripts simply compile and program node with a specific set of configuration parameters, then by using well known, tinyOS java programs, they intercepet fail messages printed on serial port by nodes. If the number of received fails is more than a specific value they consider the test as finished and start another one with different configuration parameters combination. Nodes are programmed with a test program that, as said before, requests to send an UDP packet at a specific time interval counted by a timer. Every few minutes test application reduces the time interval and send the new rate time on serial port. In this way after the script completes all possible tests, in a log file all results are available.
Apart from pools dimension the IEEE802.15.4 frame size too was changed during the tests. This parameter affects very much performances, in fact if the data-link payload size is reduced, more fragments would be needed to send a packet. In figure 4.2 this kind of influence is shown, different color lines represent different

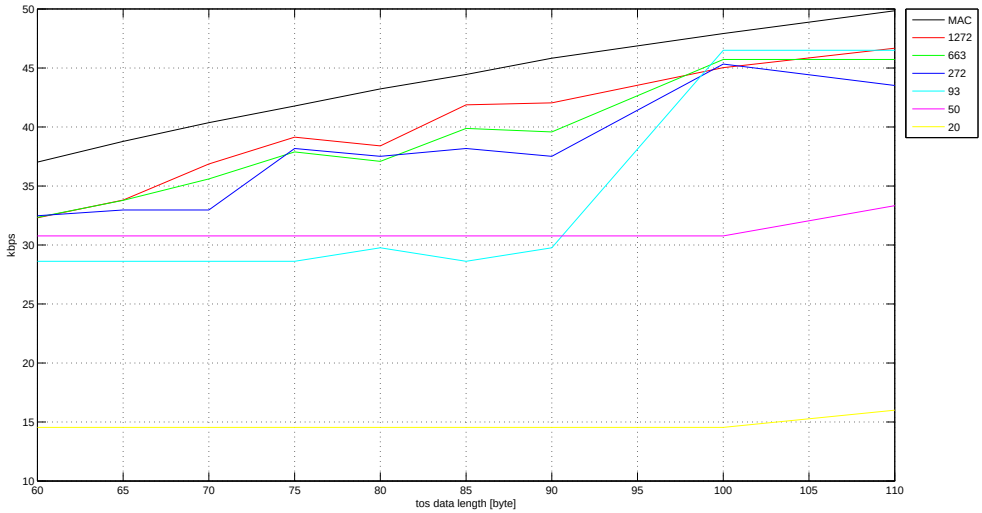


Figure 4.3: Throughput in function of IEEE802.15.4 frame size for different UDP packet size

IEEE802.15.4 message sizes. After a transitional range of packet sizes, throughputs achieve a stable value; for a specific packet size, different IEEE802.15.4 frame sizes change the number of fragments per packet.

Figure 4.3 shows another behavior of this implementation compared with that of CC2420 tinyOS drivers. The black line states the maximum throughput that can be reached by tinyOS CC2420 drivers, while other lines indicate throughput values for different UDP packet sizes. It is possible to see that the maximum throughput value reached by both this implementation and tinyOS drivers are far from the maximum bitrate value supported by CC2420 radio chip, that is 250 kbps. On the contrary, the reduction caused by sixlowpan is small, and hence it doesn't make throughput performances so much worst.

In figure 4.4 another kind of influence is shown. In this chart throughputs are calculated for different values of sixlowpan module's pools size while UDP packet size and data-link frame size stay fixed. Throughput trends is substantially sta-

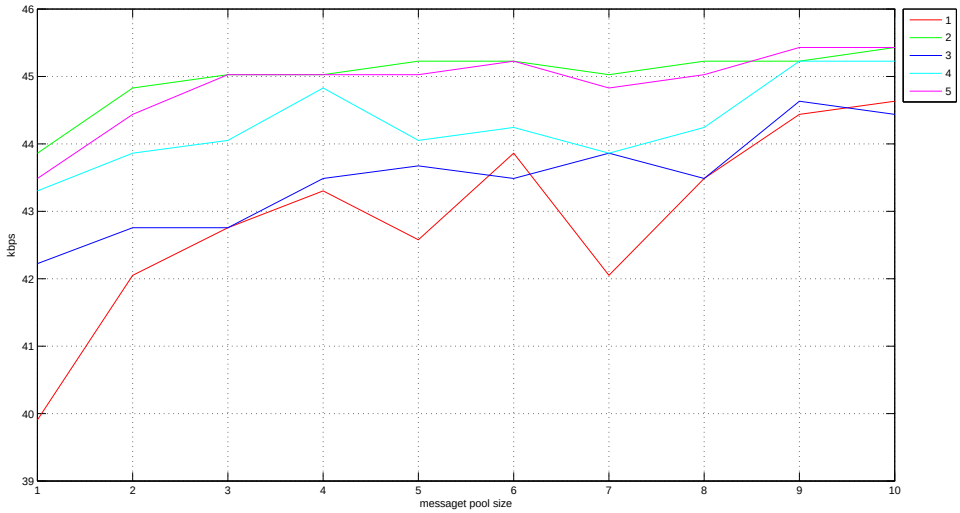


Figure 4.4: Throughput in function of message_t pool size for different packet entry pool dimensions

ble in the range from 40 to 45 kbps, and maybe the small variation does not dipend on the parameter change but on radio interferences and timer accuracy. If a queue dimension is set to one (or even two), the queue practically doesn't exist any more, so all advantages that come from the tasks based system disappear. Hence for very small pool dimensions the throughput goes down. On the contrary for bigger pool dimensions, throughput differences are less visible. This happens because of the test application architecture: since only one application requests to send a packet at a time, the number of packet entries that can be stored at the same time doesn't matter, in fact just one is needed. For future applicatons this parameter should be set taking into account the number of applications that could request to send an IPv6 packet.

The same behaviours appears if the data-link message queue is too small: the tasks have to stop themselves because the queue is always full, moreover a too big queue is useless if the mean number of fragments in which an IPv6 packet

is split, is less than the queue size. Moreover, since the most of the time the CPU is waiting for tinyOS radio driver to send phisically messages (as it will be shown next), having a big data-link message pool is useless, in fact after a while, tasks have to stop themselves to wait for the data-link message queue emptying. So even this parameter too, should be set to a suitable value that considers the mean dimension of IPv6 packets that are sent and hence the mean number of fragments in which a packet is divided.

To better understand the send speedness reduction of this implementation, an-

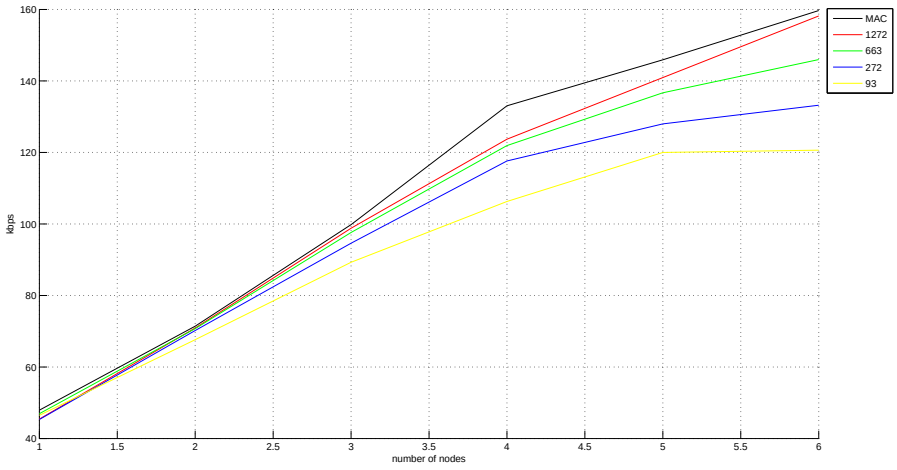


Figure 4.5: Equivalent throughput for different UDP packet sizes in function of the number of sender nodes that sends in the same radio channel, compared with tinyOS radio driver

other series of tests has been made. Their objective was to saturate the radio channel. This purpose was reached by programming more than one nodes with the same application used to test the throughput, set to send messages over the same radio channel. In this way, the presence of more than one nodes, balances the slowness of tinyOS radio drivers and hence a bigger equivalent throughput can reached.

Figure 4.5 shows results of channel saturation tests. The black line shows equivalent throughput reached by tiniyOS radio driver, while other lines show throughput reached by UDP protocol for different packet sizes. Like in figures above, this 6lowPAN implementation reduces the sending speedness, but follows the trend made by tinyOS radio driver.

4.3 Receive section

A first series of tests was made: while one or more nodes periodically send packets, another node, acting as a receiver, receives packets and checks if it lose some packets by comparing an inner progressive counter with the one written in messages, if they are not equal it means that some packets were lost and a fail message is signaled. But this kind of tests provides bad results both at data-link and UDP level: a receiveing rate of 1 to 5 kbps. Hence this test algorithm was quickly abandoned.

A second type of receiver tests was made: one or more node send messages,

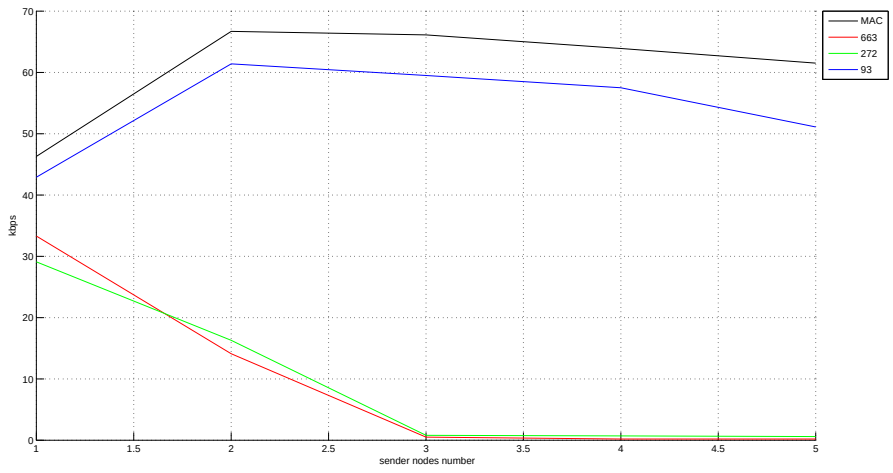


Figure 4.6: Receive rate for different UDP packet sizes, compared with tinyOS drivers in functions of sender nodes number

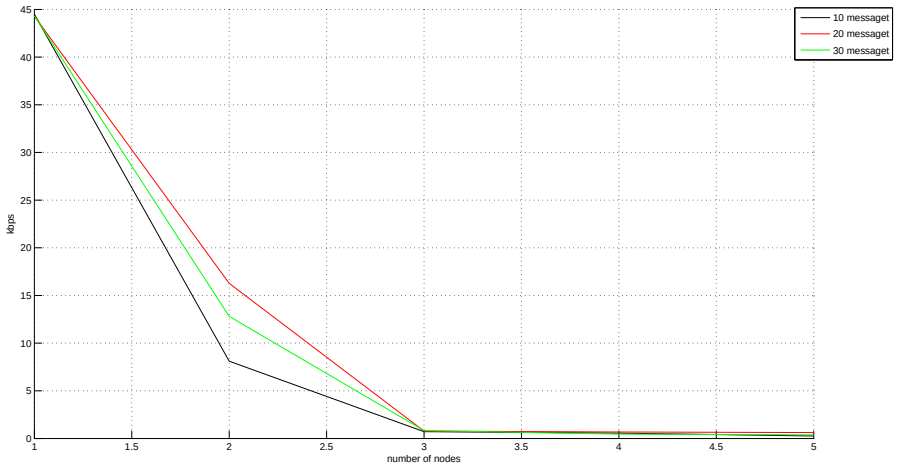


Figure 4.7: Receive rate for different data-link message pool sizes

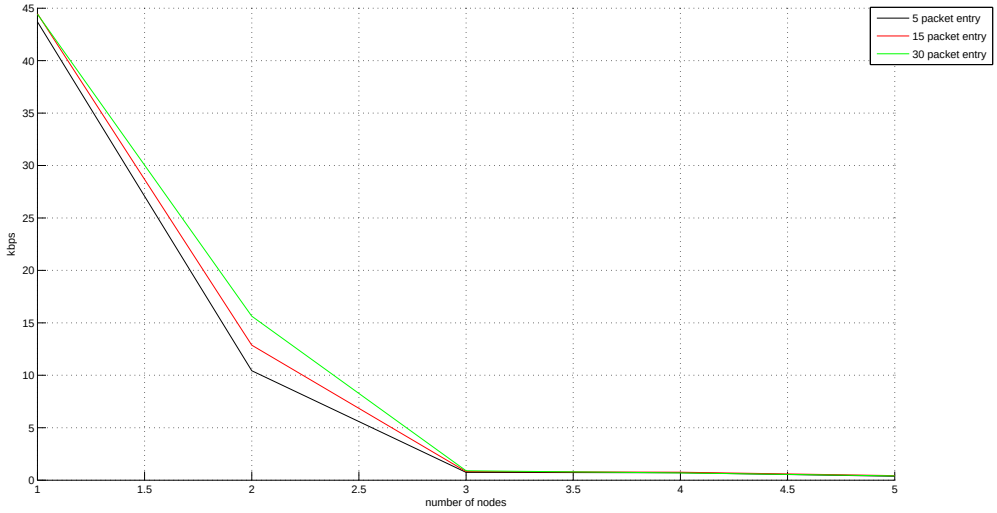


Figure 4.8: Receive rate packet for different packet entry pool sizes

while the receiver counts how many messages it can receive every minute. This receiver results, made both on data-link and UDP level, were more encouraging,

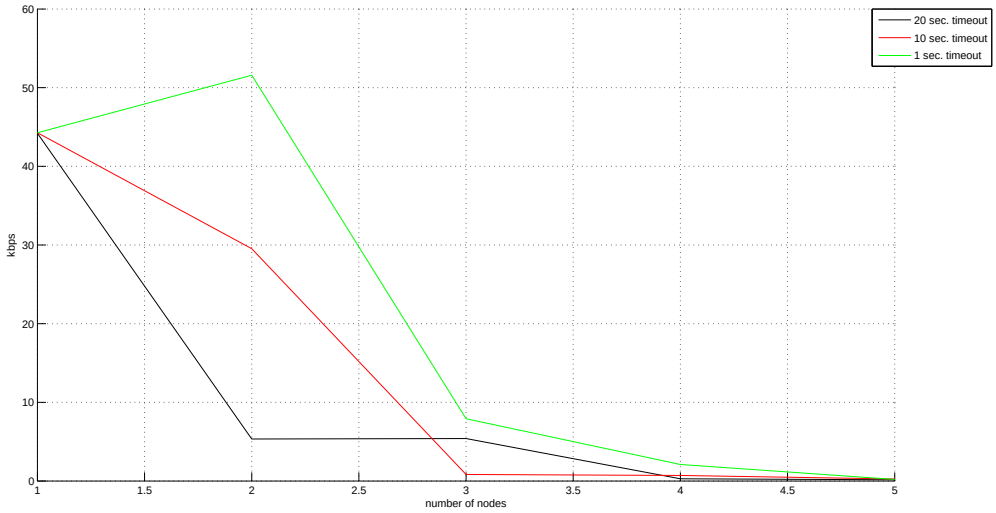


Figure 4.9: Receive rate for different packet entry timeout interval

and they are shown in figure 4.6. Since throughputs for some UDP packet sizes go down as the number of nodes grows, it seems that sixlowpan stack doesn't work. However this is not completely true, UDP packets are divided into more than one data-link messages, and if one of those messages is lost, all other fragments have been dropped after few seconds, and hence they are not included in the number of received packets. In addition, since one packet entry structure is busy on waiting for the last fragment (that is lost), other UDP packet fragments may are dropped because no other packet entry structures are available, so packet entry pool dimension heavily affects receive rate. Moreover, even data-link message queue dimension affects the rate: receiver node is not able to process received messages at the required speed, so if more than one sender node sends a big UDP packet, receiver node has to enqueue all those fragments, but, since the pool dimension could be smaller, it is pretty sure that some fragments would be dropped. Another parameter that affects receive rate is the RAM memory buffer size assigned to memory component: if memory buffer

assigned to memory component is reduced, there is less space to allocate buffers and hence less available space to reconstruct IPv6 packets.
This kind of influence on pools dimension and timeout interval is shown in figure 4.7, 4.8 and 4.9. Some sample tests were made, for different data-link message pool dimensions, and also for two different packet entry timeout intervals. In figure 4.8 the affects of packet entry pool dimension is shown.
Unlike send section, tuning operations to maximize the receive rate appear to be complicated, apparently all consistent hypothesis that could be made about pool dimensions seem to be true in reality only for big changes of parameter values, but memory availability avoid any kind of tests to proof these rules. In fact an indirect consequence of this parameter changes is that if one pool is set to a big value, because of limited RAM memory availability, it is necessary to reduce other pool dimensions, in particular, since RAM memory assigned to memory component draws the biggest part of available RAM memory, it must be heavily reduced. So when a test to check if a bigger pool dimension may cause better performaces is made, results could be heavily affected by other pool dimensions reduction. However tests made on receiver section were very hard and maybe do not reflect a typical operating situation where nodes rarely send big packets at the same time. So even if charts shows a low raceive rate, this 6lowPAN implementation would not have any receive problems if used in a normal environment.

4.4 Blip compatibility

As said before, unfortunately blip project is stopped to sixth version of the draft [2], and further the actual blip version is not completely supported: there isn't any test application that works with it, and the application that realizes the basestation is not ready. Anyway an attempt to let this two implementations communicate each other has been made.

`UdpEcho` application was modified to suit the newer blip version and also some changes for what concern IPv6 address assignement has been made. On the other side, in this 6lowPAN implementation some modification are needed to downgrade the addressing compression mechanism from fifteenth to sixth version of the draft.

Finally a small system has been made: a node running UDP Echo application with blip, answers to another node that sends an ICMPv6 echo request by using this 6lowPAN implementation. Ping tests has been running for few years without any problems or fails.

With this compatibility result a comparing chart has been drawn, to compare send throughput of this two 6lowPAN implementations.

Since blip UDP interface doesn't provide a `sendDone` signal, test procedure that

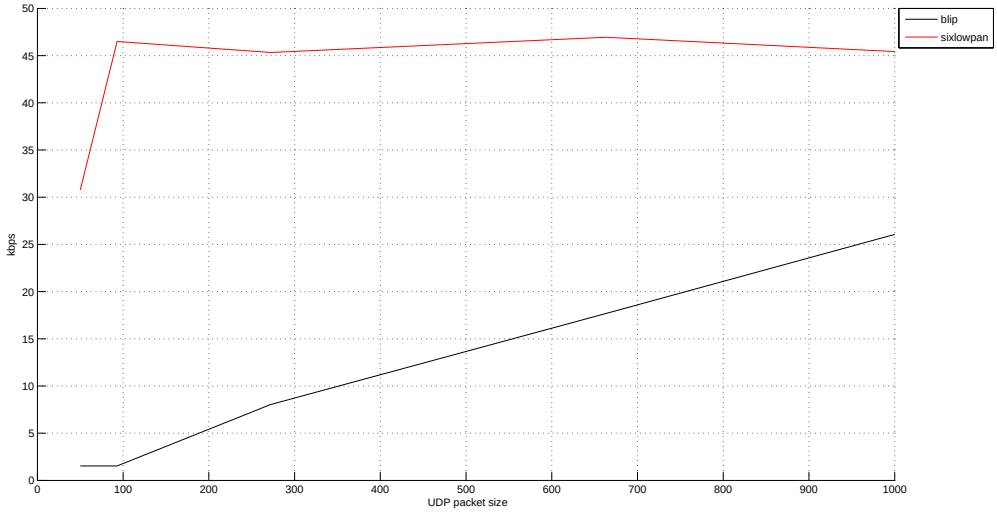


Figure 4.10: A comparison between sixlowpan and blip stack, throughput for different UDP packet sizes

was used to test blip has been the same used to test send section of this 6lowPAN implementation: a timer that marks the rate of UDP packet sending.

Figure 4.10 shows that blip can reach good values of throughput just for big packets, this means that the rate, number of packets per seconds substantially doesn't change when packets grows, and hence blip spends always the same time to send a packet, no matter how big it is.

4.5 Memory occupation and CPU time analysis

Low power network means also low memory platforms, so after the analysis of the performances it is necessary to study and analyze the memory usage of this implementation. The most heavy module is obviously sixlowpan and its header files.

Sixlowpan module itself occupy 8522 bytes of program memory and 543 bytes

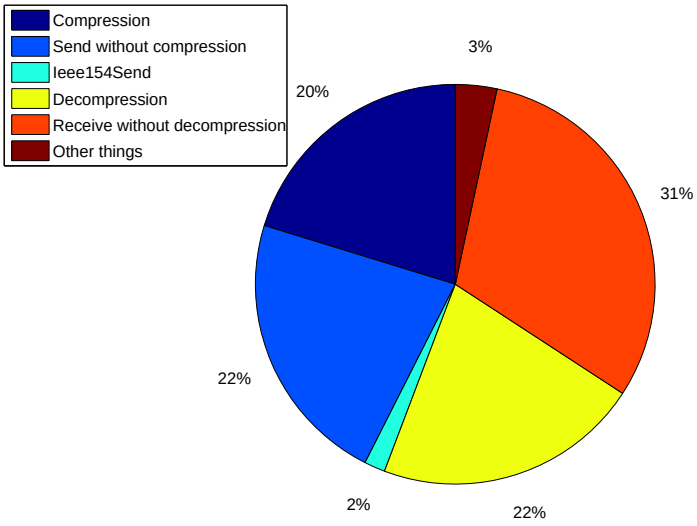


Figure 4.11: Program memory occupation of sixlowpan module

of RAM memory. Those amount of RAM memory is principally used by queues pools and function calls, since only pointers are istantiated during 6lowPAN procedures (compression, fragmentation ...).

Figure 4.11 shows how program memory is shared out among various sixlowpan sections. Compression and decompression functions occupy the same amount of memory, while receive tasks are heavier than send ones.

IPv6 module occupies about one kilobyte of program memory and only few bytes of RAM memory. IPv6 send section occupies about 70 % of its ROM memory and 30 % the receive section. Anyway those values might be wrong since IPv6 module is not complete at all.

UDP module is even much smaller, it occupies just three hundreds of byte, principally used by send function.

Both IPv6 and UDP module substantially don't use RAM memory, this advantage come from the using of the memory module that permits to only use pointers and to save structure instantiations.

After memory occupation it is possible to see CPU time using of various processes execution. This chart is obtained by keeping tracking of CPU time when processes start and when they finish.

Obviously this measure can't be accurate since the function that saves and calculates CPU times use itself the CPU and so the measure is affected by its. Anyway it can give an idea of how much tinyOS radio drivers use the CPU time. More than 80 % of the time, CPU is busy on executing radio drivers' functions and procedures. Sixlowpan module spend a lot of time not in compression or fragmentation functions but in functions calling to take elements by queues or to set data-link header.

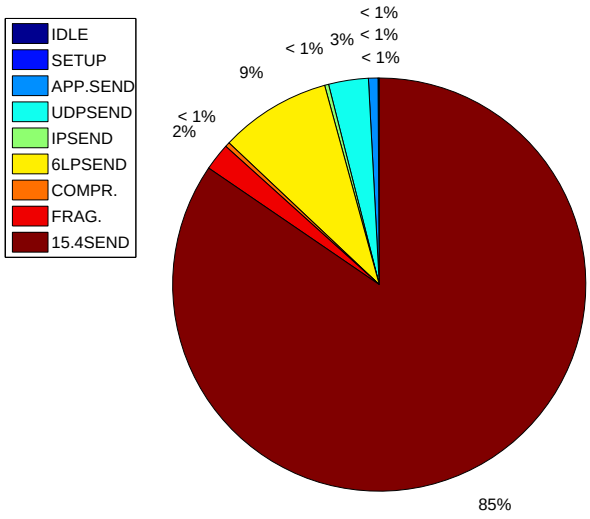


Figure 4.12: CPU usage of network modules

Chapter 5

Conclusion

Abstract:

In this chapter few final considerations about this project are made. Pros and cons are analyzed. To understand if this work will be useful for future implementations of high-level network applications or even if this architecture could be reused to implement low-level network protocols. Missing parts are signaled and finally some remarks to explain where performances can be improved maybe with few changes on algorithms.

5.1 Further improvements

A lot of parts are still missing from this implementation and hence this implementation is not ready to start working in a network.
Anyway there are some code parts that could be changed to enhance performances both for memory usage and energy consumption sides.
Memory component passed a series of tests that has proved that the component as it is, is almost stable. But it occupies quite some RAM memory as a side effect. This means that other than the used buffer there are few structures that use a lot of RAM and program memory to manage buffers among applications. Moreover this memory component is quite stable if buffers are allocated just to

send a message and then they are released, no tests have been made to check how this component behaves when an amount of memory are constantly used and allocated while the rest of RAM are used by many applications. No defragmentation tasks are implemented, so it is possible that after a while some kind of fragmentation problems could rise.
Compression and decompression functions are written without any kind of provisions for stateful compression so code dimensions surely will grow as context-base compression would be implemented.
Moreover compression function simply reads IPv6 header and starts to compress. This procedure could be improved if IPv6 module passes to sixlowpan module some informations on which IPv6 addresses has been used, if it can pass a boolean value to let sixlowpan knows if link-local or global unicast addresses are written in IPv6 header, compression function could save a lot of computation time and program memory space.
Unfortunatelly, the decompression function, that already now is bigger than the compression one, can't be improved, in decompression phase, since sixlowpan module doesn't know anything about IPv6 header of a packet, so every possible compression combination must be handled.

5.2 Conclusion

It's too early to tell if this implementation could be useful to develop applications easier and faster than now, but surely the ideas that stand behind this project are quite good to change the network stack structure of tinyOS.
Static allocation of RAM memory is a good thing to develop programs and module that runs on memory constrained platforms, but it suffers when dealing with packets and more than one network layer.
Similarly, developing one huge stand-alone component that implements all network layers needed for applications, is useful to save program memory, to make

an efficient module that doesn't waste RAM memory or computing time, but when standards are updated or maybe only some rules are modified it's very complicated to handle those complex programs to make the changes. Hence even if some program or RAM memory are wasted, it is better to separate standards in different modules to permit in an easier way to change parts of code or to implement other standards, even later.

Conclusion

Abstract:

In questo capitolo vengono fatte alcune considerazioni finali su questo progetto. Vengono analizzati i pro e i contro per capire se questo lavoro potrá essere utile per future implementazioni di appplicazioni di rete ad alto livello, o magari se l'architettura potrá essere riutilizzata per implementare protocolli di rete a basso livello. Vengono segnalate le parti mancanti e, per finire, vengono fatte alcune note per spiegare dove modificare l'implementazione per aumentare le performance dello stack.

Ulteriori miglioramenti

Molte parti sono ancora mancanti, perció questa implementazione non é pronta per poter funzionare all'interno di una rete.
In ogni caso esistono alcune parti di programma che potrebbero essere modificate per migliorare le prestazioni sia in termini di memoria usata sia in termini di energia consumata.
Il componente memory ha subito numerosi test ed é stato provato che allo stato attuale esso é pressocché stabile. Forse peró vi é un eccessivo spreco di memoria RAM come effetto collaterale, infatti oltre al buffer allocato, ci sono alcune strutture dati, utili a gestire i buffer tra le applicazioni, che occupano troppa memoria RAM e memoria di programma . Inoltre, questo componente é stabile se i buffer

vengono allocati solo per inviare messaggi per poi essere liberati; nessun test é stato fatto per verificare il comportamento del componente nella situazione in cui un buffer é allocato permanentemente mentre il resto della memoria viene usata dalle applicazioni. Nessuna procedura di deframmentazione, infatti, é stata implementata, perció é possibile, che dopo un certo di periodo di funzionamento, possa insorgere un problema di deframmentazione della memoria.
Le funzioni di compressione e decompression sono scritte senza nessun tipo di predisposizione alla compressione di tipo stateful, quindi con molta probabilitá la dimensione del codice aumentare, non appena la compressione a contesto verrá aggiunta.
La funzione di compressione semplicemente legge l'intestazione IPv6 e comprime il piú possible secondo le regole; questa procedure potrebbe essere migliorata se il modulo IPv6 passasse al modulo sixlowpan qualche informazione circa il tipo di indirizzo IPv6 usato. Con un valore booleano, ad esempio, IPv6 potrebbe informare sixlowpan se gli indirizzi sono in formato link-local o global, facendo cosí risparmiare al modulo sixlowpan tempo e memoria programma.
Sfortunatamente, la funzione di decompressione, che giá allo stato attuale occupa piú che quella di compressione, non puó essere resa piú leggera: in fase di decompressione, il modulo sixlowpan non conosce nulla sul tipo di compressione usata, perció deve essere in grado di interpretare qualunque tipo di intestazione 6lowPAN.

Conclusione

É troppo presto per dire se questa implementazione potrá essere utile per sviluppare applicazioni in maniera piú semplice e veloce rispetto ad adesso, ma di sicuro le idee che stanno alla base di questo progetto saranno utili alla riformulazione della struttura dello stack protocollare di tinyOS.
L'allocazione statica della memoria RAM Ã¨ utile per lo sviluppo di programmi

e moduli impiegati su piattaforme con vincoli sulla memoria, ma é deleterio quando si ha a che fare con pacchetti e piú di un layer protocollare.
In maniera analoga, sviluppare un unico grande componente che implementa tutti i livelli di rete necessari alle applicazioni, é necessario se bisogna risparmiare memoria, o per realizzare un modulo efficiente che non sprechi memoria RAM o tempo di calcolo; ma quando gli standard vengono aggiornati o magari soltanto alcune direttive vengono modificate, diventa molto complicato maneggiare questi componenti per implementare i cambiamenti. Perció, anche a costo di sprecare un po' di memoria RAM e programma, é meglio separare gli standard in diversi moduli per permettere, di fare piccoli cambiamenti, o addiritttura cambiare l'intero standard, in maniera piú semplice e facile, anche se ci'ødovesse avvenire in un secondo momento.

Bibliography

[1] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of ipv6 packets over ieee 802.15.4 networks," RFC 4944, Network Working Group, Sept. 2007.

[2] J. Hui and P. Thubert, "Compression format for ipv6 datagrams in low power and lossy networks," internet-draft 15, Network Working Group, Aug. 2011.

[3] Z. Shelby, S. Chakrabarti, and E. Nordmark, "Neighbor discovery optimization for low-power and lossy networks," internet-draft 15, Network Working Group, June 2011.

[4] A. P. Castellani, P. Casari, and M. Zorzi, "Tinynet-a tiny network framework for tinyos: description, implementation, and experimentation," *Wireless Communications and mobile computing*, vol. 10, pp. 101–114, 2010.

[5] R. Hinden and S. Deering, "Internet protocol version 6 specification," RFC 2460, Network Working Group, Dec. 1998.

[6] R. Hinden and S. Deering, "Ip version 6 addressing architecture," RFC 4291, Network Working Group, Feb. 2006.

[7] J. Postel, "User datagram protocol," RFC 768, Aug. 1980.

[8] A. Conta and S. Deering, "Internet control message protocol for the internet protocol version 6 specification," RFC 2463, Network Working Group, Dec. 1998.

[9] R. Braden, D. Borman, and C. Partridge, "Computing the internet checksum," RFC 1071, Network Working Group, Sept. 1998.

[10] M. Harvan, "Connecting wireless sensor networks to the internet - a 6lowpan implementation for tinyos 2.0," Master's thesis, Jacobs University Bremen, may 2007.

[11] P. Levis and D. Gay, *TinyOS programming.* Cambridge univeristy press, 2009.

[12] P. Levis, *TEP 111: message_t.* Core working group, http://www.tinyos.net/tinyos-2.x/doc/txt/tep111.txt.

[13] J. Hui, P. Levis, and D. Moss, *TEP 125: TinyOS 802.15.4 Frames.* Core working group, http://www.tinyos.net/tinyos-2.x/doc/txt/tep125.txt.

[14] D. Moss, J. Hui, P. Levis, , and J. I. Choi, *TEP 126: CC2420 radio stack.* Core working group, http://www.tinyos.net/tinyos-2.x/doc/txt/tep126.txt.

[15] "Tinyos web page." http://www.tinyos.net.

[16] "Blip web page." http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip.

[17] "Contiki os web page." http://www.sics.se/contiki.

List of Figures

1.1 nesC program architecture 5
1.2 802.15.4 network topology 7
1.3 802.15.4 superframe structure 7
1.4 Standard ipv6 header 9
1.5 IPv6 address types 11
1.6 Mesh header 13
1.7 fragmentation headers 14
1.8 lowpan iphc 15
1.9 lowpan nhc 17

3.1 memory architecture 26
3.2 general system architecture 28
3.3 6lowpan header files 33
3.4 6lowpan architecture 37
3.5 6lowpan task 38

4.1 test configuration 48
4.2 packet and TOS graph 49
4.3 packet and TOS + radio driver graph 50
4.4 messaget and packet pools 51
4.5 radio channel saturation test 52
4.6 receive results 1 53

4.7 receive results 2 54
4.8 receive results 3 54
4.9 receive results 4 55
4.10 blip comparison 57
4.11 Memory 6lowpan 58
4.12 CPU time profile 60