IP-based Communication in Wireless Sensor Network

Christian Fuchs
Betreuer: Dr. Alexander Klein
Seminar Sensorknoten: Betrieb, Netze und Anwendungen SS2011
Lehrstuhl Netzarchitekturen und Netzdienste, Lehrstuhl Betriebssysteme und Systemarchitektur
Fakultät für Informatik, Technische Universität München
Email: fuchsch@in.tum.de

ABSTRACT

IP is the standard communication protocol in the Internet. Several attempts have been made to port the TCP/IP to Wireless Sensor Networks, because it would ease data acquisition from external applications. This would make it possible to integrate applications more easily into existing networks. On the other hand, IP was initially designed for non resource constraint systems. This causes a significant decrease in bandwidth mostly because of its large header data. This paper will look first at the 6loWPAN specification and afterwards at two widespread implementations of the IP stack for sensor nodes, uIP and BLIP, and compare their compatibility with other implementations as well as their impact on the bandwidth in an example setup.

Keywords

IP over 802.15.4,WSN,uIP,BLIP,6loWPAN

1. INTRODUCTION

Handling communication efficiently and with low energy consumption is one of the most important tasks, when it comes to the programming of nodes that are to be used in Wireless Sensor Networks(WSNs). Up to date there are different approaches on how to handle communication in such networks, like protocols specifically designed for the use on resource constraint nodes. Despite this there are also urges to use TCP/IP on these nodes to make them directly accessible via the Internet[9][5][14].

Using IP in Wireless Sensor Networks holds many opportunities for reasons of direct communication with WSNs over the Internet as well as remote access to the Sensors' data. On the other hand, most platforms used in WSNs are highly resource constraint in terms of memory as well as computing power. These constraints pose various challenges when implementing IP on these platforms, because IP was initially developed for platforms without these constraints. Therefore, the standard IP-Stack implementations used in Linux or Windows can not be directly adapted for the use in WSNs[7], since they require RAM in order of some megabytes, an amount that is not available on typical sensor motes. Also TCP/IP communication consists of point-to-point flows between arbitrary nodes, whereas most traffic in WSNs is for collecting data from the sensors and sending it to a remote server.

Also most WSNs use IEEE 802.15.4 radio for communication which only offers a limited bandwidth compared to other WLAN technologies. Again, IP was not designed with such constraints in mind leading to IP headers, which seem to be too large to be efficiently used in environments with limited bandwidth. Due to this, the IETF has presented 6loWPAN in [15] which describes techniques to cut down the large overhead caused by using IPv6, to enable its usage in the context of an IEEE 802.15.4 environment.

This leads to different approaches on how to bring TCP/IP to WSNs ranging from the use of a proxy server and smaller versions of the IP protocol to the development of smaller IP stacks [2].

In the following sections this paper will take a closer look at two well known implementations of this standard, uIP which has been created by Adam Dunkels as a part of the Contiki operating system[7] and BLIP which was created for TinyOS [5]. The paper will continue in the following way: Section 2 will give a brief overview of the IEEE 6loWPAN standard for IPv6 in Wireless Sensor Networks. Section 3 and 4 will discuss the uIP and BLIP implementations in greater detail. After this there will be some Case Studies with these protocol stacks in Section 5 and finally section 6 concludes the paper.

2. 6LOWPAN

Since Wireless Sensor Networks consist of mostly battery powered motes, energy efficiency is a very important requirement for the used hard- and software. Therefore, the IEEE 802.15.4 standard for data transmission in *low power wireless personal area networks*(loWPAN) was developed. Because typical packets sent in such a network are quite small and for the reason that most power is consumed by sending packets, this standard proposes a data rate of only 250 kbit/s. While this is perfectly satisfactory for specifically designed protocols, which add only a small overhead to the transmission, it issues a challenge to the use of IP in WSNs. Especially IPv6 would consume most of the available bandwidth just with its headers, if used unmodified. To face this challenge 6loWPAN, an adaptation layer between the link and network layer, was introduced by the IETF in [15]. Its main contribution is the introduction of an alternative header format which supports header compression as well as fragmentation of IPv6 packets into multiple link-level frames [12].

6loWPAN consists of several distinct headers, each serving a different purpose. The headers can be stacked on each other, making it possible to skip the unneeded headers. The dispatch header indicates which network layer protocol is used, like plain IPv6 or 6loWPAN. The fragmentation header is used if a single IPv6 packet is too large to be sent over radio in one step. It contains information on how the packet was fragmented and should be reassembled. And the mesh addressing header serves for layer 2 routing.

2.1 Header Compression

The most important step to efficiently use IPv6 in WSNs is to cut down the transmission overhead caused by the seemingly large IPv6 headers. The standard IPv6 header without any extension headers is 40 bytes large. The largest part of this header are the source and destination addresses, having 16 bytes each. To reduce header sizes either stateless or stateful compression techniques could be used. Using states takes advantage of the fact that certain fields do not change their value in a communication flow between two senders[13]. On the other hand, stateless header compression uses common values of certain fields and information that is stored redundantly across layers to reduce header sizes[15]. Hui and Culler explained in [12] that compression techniques using states are not very effective in loWPANs, since it only pays of in long transmission flows, which typically do not occur in loWPANs. This is the major reason why 6loWPAN uses the stateless header compression format HC1. To denote what kind of header is used for the particular packet, an 1 byte large dispatch header is inserted before the layer-3 header.
One major technique is the use of common values in certain fields[19]. For example, in most packets only TCP,UDP and ICMP are used as upper layer protocols. With this knowledge it is possible to reduce the size of the next header field to only two bits. To manage cases where fields do not carry such a common value, there are reserved values to denote an uncompressed field in the header[12].
To further lessen the header size, 6loWPAN exploits the fact that some information carried in the header can be derived from fields in other layers. For example, the payload length field or the interface identifiers can be determined by the fields in the 802.15.4 header. The only field which must always be carried inline is the 8-bit hops-left field[15], to avoid packets circulating in the network infinitely.
With the exploitation of all these redundancies and common values the IPv6 header can be cut down to 2 bytes only, 1 byte denoting the common values in the original header and 1 byte for the hops-left field. Additionally, an 1 byte large dispatch header is added before the IP header, to denote if the following header is a compressed IP header or native an IPv6 header. If, however, some of the fields cannot be compressed they must be carried inline. For example, when the interface identifier cannot be derived from the layer-2 address, it must be carried inline. In these cases the compressed header would be bigger than 2 bytes. Figure 1 shows the HC1 header with inline IP fields.
By eliding the length field 6loWPAN can also compress UDP to 4 bytes, if source and destination port have a predefined well known value.

2.2 Fragmentation

IPv6 also relies on a Maximum Transmission Unit(MTU) of at least 1280 Bytes, which means that packets smaller than this can be sent without fragmentation. On the other hand, the maximum length of 802.15.4 frames is 127 bytes due to the 7-bit length field in the 802.15.4 frame header. This means there has to be a mechanism for fragmentation on link-layer transparent to higher layers.
As described in [15] 6loWPAN achieves this by splitting up the packets in several frames. Each of these frames has an additional fragmentation header, containing the actual size of the packet, an unique identifier, which is the same for all frames that belong to the same packet, and an offset field, denoting which part of the packet the frame contains. To further reduce the size of the header, the first frame elides the offset field[12]. This results in an additional overhead of 4 Bytes for the first frame respectively 5 bytes for each following frame.

2.3 Routing

Another issue that has to be considered when using IPv6 in a loWPAN, is that sometimes link layer routing is required. Normally, routing would be done on the network layer(layer 3). Since the HC1 format for header compression is optimized for link-local communication[12], it is desirable to do the routing on layer 2.
To solve this issue 6loWPAN has a concept named mesh under. It adds the mesh addressing header, which simply holds the packets source and destination address as well as an hops-left-counter that is decreased by every forwarding node[15]. With this it is possible to route packets over the link layer transparently to the upper layers. In this case the network topology looks like all nodes are in a single broadcast domain to the network layer. Hence, only link-local addresses are needed for communication within the network. This header adds another 5 bytes of overhead.

2.4 Compressed Header Sizes

6loWPAN compresses the IPv6 header to only 2 bytes. On the other side it adds only an 1-2 byte dispatch header. This means an IPv6 packet using UDP as a transport protocol, which would add 44 bytes overhead to the raw payload, if transmitted uncompressed, is compressed by 6loWPAN to only 7 bytes(1 dispatch + 2 compressed IPv6 + 4 UDP)[12]. Even if the other headers for fragmentation and mesh addressing are added, the overhead would not exceed 17 bytes, which means still half of the overhead size can be saved. Figure 1 shows a frame with mesh routing, fragmentation and header compression.

3. UIP

uIP was developed by Adam Dunkels[7] along with the Contiki operating system for tiny sensor networks[8] in 2003. It was one of the first fully working IP stacks for resource constraint systems. In the first version uIP already supported all mandatory IPv4 features as requested by [17], and omits only some of the lesser used features like IP options, thus being fully compatible to any other IP-Stack like the BSD stack implementation[7]. In 2008 uIP has been extended to support IPv6 as well[10]. Up to now uIP is the smallest implementation of a complete IP stack.

3.1 Attributes

In order to cut down memory consumption to an absolute minimum, the uIP implementation uses a shared buffer for incoming and outgoing packets. This buffer is only big enough to hold one packet of maximum size. uIP holds one packet in the buffer at a time and overwrites the buffer every time a new packet arrives, but not before the application has processed the data. As most radio or ethernet

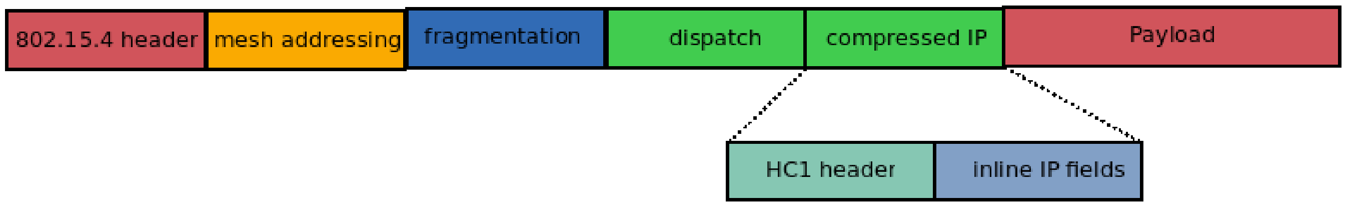

Figure 1: 802.15.4 frame with all 6loWPAN headers references [12]

controllers have internal memory to buffer about 5 packets, this behavior will not lead to packet loss, if more than one packet arrives, before the application is able to process the data[7]. Another consequence is that outgoing packets may be overwritten right after being sent. Furthermore, uIP does absolutely no dynamic memory allocation.
uIP consists of an implementation of IP as a network layer and UDP or TCP as a transport layer. The link layer is not specified and can be chosen independently. It is possible to use 6loWPAN as well as any other implementation.
uIP is fully compatible with other TCP/IP implementations since it supports all features that are needed for host-to-host communication as required by RFC 1122[3]. Additionally, uIP can reassemble only one fragmented packet at a time, for the reason that reassembling has to be done in a separate buffer. On the other hand, not all features for interfacing between the application and the network stack are implemented. As a result of this uIP does not support soft error reporting or dynamically configurable type-of-service bits[7]. For routing uIP uses the RPL protocol explained in the next section.
For ICMP just the minimal subset of operations, that are mandatory for interoperability, is supported. Meaning that uIP is solely capable of sending echo reply messages, by swapping the source and destination address[7] of incoming echo request messages.
As a transport layer uIP supports both TCP and UDP. Like any other part of the uIP stack the TCP implementation is done to ensure full interoperability, while it omits all features, which are not mandatory, to save memory and quicken the network operations. Listening and connecting ports are supported as well as sending data works normally. But there is no support for a sliding window mechanism, because as mentioned above the packet buffer can only hold a single packet at a time. Additionally, as no sent packets are buffered and any incoming packet would overwrite the packet buffer, retransmission has to be done manually. This means that TCP reports to the application, if a packet was not acknowledged correctly and the application then has to ensure the packet is retransmitted. Congestion Control is elided as it is not needed, too[7].

3.2 Routing

uIP uses the Routing Protocol for Low Power and Lossy Networks(RPL) [1]. This protocol is especially designed for large networks of resource constraint nodes managed by a few central *supernodes* and optimized for multipoint-to-point traffic[4]. RPL is a distance vector protocol that uses a destination orientated directed acyclic graph(DODAG) for routing. One of the *supernodes* serves as the root of the DODAG and all other nodes build up multipoint-to-point routes to this root. Every node in the DODAG has a rank, denoting its distance from the root node, and a set of parent nodes, which are closer to the root. One of this parents serves as the preferred parent.
To create the DODAG each node that is part of the DODAG sends DODAG information objects(DIO) via link-local multicast containing its own rank. A new node, which has received some of this messages, can determine its own rank, which has to be greater than its parents rank, and starts itself sending DIOs. To avoid count-to-infinity problems a node that is part of the DODAG can only lower its rank if it receives a DIO from a parent with a lower rank.
After the DODAG has been formed, messages can be sent to the root by forwarding to the nodes preferred parent. To send messages from the root to nodes in the networks DODAG advertisement objects(DAO) are sent towards the root, while every node that forwards the object adds its address. With this the root node has a source route to the node, that issued the DAO, once it arrives at the root. For node to node communication messages are sent first to the root from where the messages are sent towards the destination nodes via the source routes.

3.3 Usage

Since uIP is designed for the use together with Contiki as an operating system, there are two ways of programming applications that use uIP for communication.
The first way to program with uIP is the event driven interface, which was initially the only way to program with uIP. Here the application is invoked every time an event on the IP stack occurs, like incoming data or a new connection request[7]. After processing the incoming data the program has to hand control back to the uIP stack, so that outgoing packets can be sent and new arriving packets can be processed. This means that the application has to be built like a state-machine, testing on each incoming event what caused the event to be posted. For the purpose of testing which event has occurred the API defines functions such as uip_newdata(), uip_acked() or uip_connected(). On every new event each of this functions has to be called to determine which is the appropriate action to be taken. While this is perfectly sufficient for small programs, it would cause too much overhead in more complex applications. These would be easier to develop and more efficient with an API that allows sequential code. Listing 1 shows a short example code using the event driven API.

```
void example() {
  example_state state;
  if(uip_connected()) {
    uip_send("Welcome\n",9);
  }
  if (uip_acked()) {
    if (state == WELCOME_SENT) {
      state = WELCOME_ACKED;
    }
  }
  if (uip_newdata()) {
    uip_send("ok",2);
  }
}
```

Listing 1: example program with event driven API references[1]

For this purpose protosockets, which can be thought of as lightweight versions of the BSD sockets, were provided[1]. With this API the networking code can be written sequentially, similar to networking applications on a PC. Like with normal sockets every connection is associated with an protosocket and the protosocket has to be handed over to the underlying operating system every time an action like sending or receiving data has to be carried out. In opposite to the event driven API, where control must be handed back to the stack after processing a packet, protosockets also support blocking calls for receiving data, which is shown in listing 2.

```
PT_THREAD example() {
  PSOCK_BEGIN(s);
  PSOCK_READTO(s, '\n');
  if (strcmp(inputbuffer,"HI") != 0) {
    PSOCK_CLOSE(s);
    PSOCK_EXIT(s);
  }
  PSOCK_SEND(s, "ok", 2);
  PSOCK_CLOSE(s);
  PSOCK_EXIT(s);

}
```

Listing 2: example program with protosocket API references[1]

3.4 Memory Footprint

uIP is highly scalable at compile time. The programmer can for example decide on the number of maximum simultaneously open connections as well as the amount of RAM reserved for the packet buffer. Furthermore, support for TCP, UDP and IPv6 can be deactivated, if not needed, to save further memory.
In a minimal configuration it is possible to run uIP with only 200 bytes of RAM but with this configuration throughput is very low and it is usable only in simple environments that send only very small packets. A configuration that is usable in a more generic scenario would consume about 2 kbytes of RAM. The stack implementation needs about 5.1 kbytes of ROM[7].

4. BLIP

BLIP stands for Berkley low power IP and is a IPv6 stack developed for TinyOS by David Culler[11]. It was first developed under the name b6loWPAN and later renamed to BLIP, because BLIP was at this time already more than just an implementation of 6loWPAN but instead a fully working IP stack. As a part of TinyOS it was written in nesC, an enhancement to the language C especially written for TinyOS. There are implementations of BLIP for micaZ, TelosB and iMotes[16].

4.1 Attributes

Unlike uIP BLIP implements not only network and transport layer, but it also uses b6loWPAN as an adaptation layer between link and network layer. Therefore, BLIP must always use IPv6 for communication. BLIP also supports mesh under routing, explained in section 2.3, meaning that routing is done transparently by the adaptation layer[18].
The network layer uses a routing protocol called HYDRO, which is explained in the following section. Additionally, the network layer is able to do neighbor discovery, using ICMPv6, which is fully supported, as opposed to uIP, which only supports echo messages. Also BLIP can configure link-local addresses and global addresses, either via stateless autoconfiguration or via DHCPv6, if a router is reachable[18][11]. The network layer is additionally responsible for retransmitting packets that get lost over a single hop. This enables the stack to reroute the packet if the network topology changed during transmission.
BLIP defines basic Quality of Service(QoS) classes. The most important are to indicate high priority data and contrary to denote latency-tolerant packets, which can be buffered and sent in large bulks of packets for energy efficiency[11]. As a transport protocol it currently supports UDP, but there is already a prototype for TCP[16].

4.2 Routing

BLIP uses the HYDRO routing protocol[6]. It was designed to combine the support for many-to-one traffic, which is needed for data collection, and one-to-one traffic, needed to propagate commands in the network. For the HYDRO protocol there are two kinds of communication points. The sensor nodes, which acquire data and send them to a remote server or communicate in the network, and at least one border router, which connects the network to the outside world. The border router has to store an overview over the complete network and should install routes onto the nodes. This network overview is created from topology reports it gets from the nodes. To hold traffic for creating and maintaining the network overview as little as possible, the topology reports are sent piggybacked with the sensor data. Such a topology report holds information about the node's neighbors and the estimated transfer costs to this neighbor. However, the construction of this global network topology is complicated, due to the fact that the sensor nodes have not enough memory to hold a complete list of their neighbors. This means that the created network topology does not represent the complete network with all its links, but only a subset, which is nevertheless sufficient for routing.
The border router stores this subset of the complete network topology in the so called link state database. If there is more than one border router, they have to be connected in order to share their particular link state database. From this database a best effort route between every two nodes in the network can be derived. This routes are installed in the corresponding node to improve point-to-point communication.
The nodes on the other hand are to constraint in terms of memory to hold a complete routing table. Instead they use

a distributed DAG(directed acyclic graph) to provide them with a reliable route to the next border router. In this distributed DAG every node holds the address of a node nearer to the border router. In practice more than one route to the border router is stored to improve reliability[6]. The nodes also maintain flow tables, which hold the installed routes from the border router.
The nodes can forward packets in 3 steps:

1. If the packet contains a valid source route it is used
2. If there is an entry in the flow table for the desired destination, the previously installed route is used
3. If neither a source route is provided nor a matching entry in the flow table exists, the packet is redirected to the border router, from where it is sent to the correct node

4.3 Usage

The BLIP API is closely related to programming with sockets on UNIX systems. The structs containing the address are defined like in Linux:

```
struct in6_addr
  {
    union
      {
      uint8_t   u6_addr8[16];
      uint16_t  u6_addr16[8];
      uint32_t  u6_addr32[4];
      } in6_u;
#define s6_addr       in6_u.u6_addr8
#define s6_addr16     in6_u.u6_addr16
#define s6_addr32     in6_u.u6_addr32
  };

struct sockaddr_in6 {
  uint16_t sin6_port;
  struct in6_addr sin6_addr;
};
```

Listing 3: structs for IPv6 addresses and port

For programming with UDP BLIP provides the three functions. error_t bind(uint16_t port) is for binding a socket to a specific port and the functions error_t send_to() and error_t recv_from() serve for sending and receiving data.
The TCP implementation is still experimental and cannot accept more than one connection. The API is similar to BSD sockets, with functions to actively connect to an port and passively listening for incoming connection requests.

4.4 Memory Footprint

Together with the HYDRO routing protocol BLIP needs about 2.5 kilobytes of RAM and 9.4 kilobytes of ROM[6][11]. Especially in code size BLIP is significantly bigger than uIP. This is because BLIP also contains the underlying link layer, whereas uIP only implements the network and transport layer. Furthermore, BLIP fully supports ICMPv6 and DHCPv6. Table 1 briefly compares the features supported by uIP and BLIP and table 2 shows the memory usage of both stacks.

	uIP	BLIP
OS	Contiki	TinyOS
IP	IPv4 and IPv6	IPv6 only
TCP	YES	Prototype
UDP	YES	YES
ICMP	echo only	YES
Mesh Under	NO	YES
Route over	NO	YES

Table 1: Comparison between the main features of uIP and BLIP references [18]

	uIP	BLIP
RAM	200 byte - 2 kbyte	1 kbyte
ROM	5.1 kbyte	9.4 kbyte

Table 2: Memory footprint of uIP and BLIP

5. EVALUATION

This section will show the usage of IP in two simple network environments, to examine its impact on throughput and the effectively available bandwidth. For the evaluation we assume typical IEEE 802.15.4 standard parameters, namely a bandwidth of 250 kbit/s and all frames should be of maximum frame length, hence 127 byte. In the following sections plain IPv6 without any compression is compared against compression according to the 6loWPAN standard. UDP is used as transport protocol.
The following formula is used to compute the maximal achievable data rates: $(1 - \frac{headersize}{127byte}) \cdot 250\frac{kbit}{s}$. The term $1 - \frac{headersize}{127byte}$ is the percentage of the frame that can be used for the payload. Multiplied with the available maximum data rate this gives us the data rate for payload transfers.

5.1 Single Hop

The first network layout simulates a link-local transmission from node 1 to node 2. Figure 2 depicts this scenario. Plain IPv6 together with UDP headers add 48 bytes of overhead to every transferred frame. This equals to 37.7 % of the whole frame. Thus the total available data rate is reduced by 34.6 % leaving a total of 155 kbit/s for payload transfer. Opposed

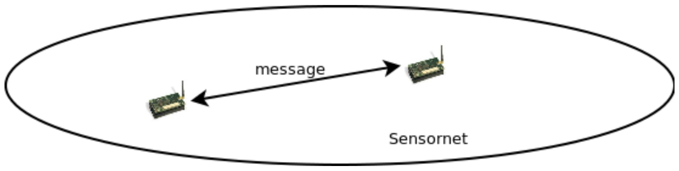

Figure 2: First setup with just two nodes

to this transmission with 6loWPAN adds only 7 bytes of overhead as discussed in section 2.4. This is only 5.5 % of the total frame length, meaning that a data rate of 236 kbit/s is still available for payload transmission. Considering a packet larger than 127 bytes, the packet must be fragmented, which adds the 5 bytes fragmentation header to each frame. With this there are 226 kbit/s left for payload transmission.
These values show that using 6loWPAN pays off even in a single hop transfer, since uncompressed IPv6 consumes nearly one third of the available frame length.

5.2 Multi Hop

In this scenario we consider 4 nodes in a string connection, as shown in figure 3, and we assume a message is sent from node 1 to node 4 over two hops(nodes 2 and 3). In this setup the node's bandwidth is not only consumed by them-selves but also by the next node, which blocks the communication medium while it transmits the message to the next hop. Furthermore, interference plays a role as two nodes may not be close enough to directly communicate with each other, but one node sending may still cause interference on the other node. This leads to another limitation of the effectively usable bandwidth.

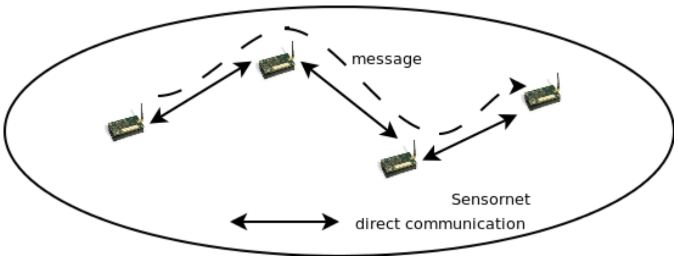

Figure 3: Message being sent over 3 Hops

Normal Interference Radius

Here we assume that transmission and interference radius are the same and a node only blocks the communication channel for the nodes it can directly communicate with. In this scenario the bandwidth for the initial sending node is halved, because for every packet, the node sends, the next hop also has to transmit the packet. Therefore, the node can effectively use only half of its original bandwidth, hence 125 kbit/s.
At the nodes that serve as hops, but not as communication endpoints, each packet consumes three times the bandwidth that one transmission of the packet would cost. This is because at first the packet must be transferred from the previous hop to the node. After that the node has to forward the packet and at last the next hop is also retransmitting the packet, blocking the communication channel for another length of the packet.
Using IPv6 and UDP without any compression would consume 37.7 % of the total frame length. Leaving a data rate of only 78 kbit/s for the payload. With 6loWPAN 118 kbit/s remain. A fragmented packet consumes another 5 bytes for overhead as well as a packet that is routed on the link layer. In this case the data rate effectively available for payload transmission is 113 kbit/s. If both cases, fragmentation and mesh under routing, apply the needed headers consume 13.3 % if the frame length, leaving a data rate of 108 kbit/s.

Double Interference Radius

In a more realistic scenario interference would not only occur between nodes, which could directly communicate with each other, but have a far wider influential radius. For this example setup we consider the sphere of interference to be twice as big as the radius in which actual communication is possible. As shown in figure 4 this would mean that node 1 would not only block the communication channel for node 2 but also for node 3, despite there is no direct communication possible between node 1 and 3. In this scenario the initially

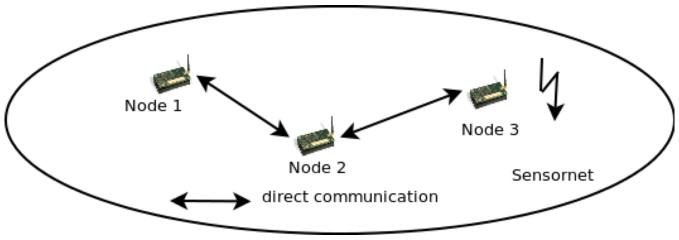

Figure 4: Example for doubled interference radius

sending node has only one third of its original bandwidth for transmission available, because the communication channel is not only blocked from the next node sending, but also from the node hereafter. This means the node has only 83 kbit/s of effective usable bandwidth.
Here plain IPv6 would leave only 51 kbit/s for payload transmission. 6loWPAN on the other hand leaves 78 kbit/s for not fragmented packets, 75kbit/s for fragmented packets and 71 kbit/s for fragmented packets that are routed mesh under.
The different data rates achievable with and without 6loWPAN are summarized in table 3.

6. CONCLUSION

Since the usage of IP in WSNs and their integration into the Internet brings many advantages, a lot of effort has been put to overcome the challenges that using IP in WSNs poses. The 6loWPAN standard made it feasible to use IPv6 in low throughput environments by compressing the relatively large overhead of IPv6 to a few bytes.
Another issue was that most IP stacks were too big to be used on resource constraint nodes with less than 100 kilobytes of RAM. Here uIP and BLIP have shown that it is possible to implement a complete IP stack, which uses only a few kilobytes of RAM and ROM, while being fully interoperable with other IP stacks.
Since traffic in WSNs happens to be mostly multipoint-to-point traffic, in opposite to classic networks, where most traffic is point-to-point, there is still a need for specialized routing protocols. Recently there has been some work in this area, especially by the IETF which has brought up RPL to create a standard routing protocol for WSNs.

7. REFERENCES

[1] http://www.sics.se/contiki/.
[2] J. A. H. R. Adam Dunkels, Thiemo Voigt and J. Schiller. Connecting wireless sensornets with tcp/ip networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, 2004.
[3] R. T. Braden. Rfc 1122: Requirements for internet hosts — communication layers, Oct. 1989. Status: STANDARD.

	Single Hop	Multiple Hops	Multiple Hops Double Interference Radius
IPv6	155 kbit/s	78 kbit/s	51 kbit/s
6loWPAN	236 kbit/s	118 kbit/s	78 kbit/s
6loWPAN & Fragmentation	226 kbit/s	113 kbit/s	75 kbit/s
6loWPAN & Mesh Under	/	113 kbit/s	75 kbit/s
6loWPAN & Mesh Under & Fragmentation	/	108 kbit/s	71 kbit/s

Table 3: Comparison of data rates in different environments

[4] T. Clausen and U. Herberg. Multipoint-to-point and broadcast in rpl. In *Proc. 13th Int Network-Based Information Systems (NBiS) Conf*, pages 493–498, 2010.

[5] P. A. C. da Silva Neves and J. J. P. C. Rodrigues. Internet protocol over wireless sensor networks, from myth to reality. *Journal of Communications*, 5(3), March 2010.

[6] S. Dawson-Haggerty, A. Tavakoli, and D. Culler. Hydro: A hybrid routing protocol for low-power and lossy networks. In *Proc. First IEEE Int Smart Grid Communications (SmartGridComm) Conf*, pages 268–273, 2010.

[7] A. Dunkels. Full tcp/ip for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services*, May 2003.

[8] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *First IEEE Workshop on Embedded Networked Sensors*, November 2004.

[9] A. Dunkels, T. Voigt, and J. Alonso. Making tcp/ip viable for wireless sensor networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN2004)*, January 2004.

[10] M. Durvy, J. Abeillé, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels. Making sensor networks ipv6 ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session*, Raleigh, North Carolina, USA, Nov. 2008. Best poster award.

[11] J. W. Hui, A. R. Corporation, and D. E. Culler. Ip is dead, long live ip for wireless sensor networks. In *The 6th International Conference on Embedded Networked Sensor Systems (SENSYS'08)*, pages 15–28. ACM, 2008.

[12] J. W. Hui and D. E. Culler. Extending ip to low-power, wireless personal area networks. *Internet Computing, IEEE*, 12(4):37–45, July-Aug. 2008.

[13] V. Jacobson. Rfc 1144: Compressing tcp/ip headers for low-speed serial links, Feb. 1990. Status: PROPOSED STANDARD.

[14] W. F. Karl Mayer. Ip-enabled wireless sensor networks and their integration into the internet. In May, editor, *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, 2006.

[15] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of ipv6 packets over ieee 802.15.4 networks. RFC 4944 (Proposed Standard), Sept. 2007.

[16] K. Nithin. Blip: An implementation of 6lowpan in tinyos, November 2010.

[17] J. Postel. Rfc 791: Internet protocol, Sept. 1981. Status: STANDARD.

[18] F. B. Ricardo Silva, Jorge S Silva. Evaluating 6lowpan implementations in wsns. In *CRC '09: Proceedings of the 9Âł ConferÃłncia sobre Redes de Computadores*, 2009.

[19] C. Westphal and R. Koodli. Stateless ip header compression. In *Proc. IEEE Int. Conf. Communications ICC 2005*, volume 5, pages 3236–3241, 2005.