M. Harvan and J. Schönwälder

TinyOS Motes on the Internet: IPv6 over 802.15.4 (6lowpan)

Matúš Harvan is a doctoral student in the Information Security group at the Swiss Federal Institute of Technology (ETH) Zurich. He received his BSc degree in 2005 and MSc degree in 2007 from the Jacobs University Bremen. His research interests are network security and usage control.

Dr. *Jürgen Schönwälder* is Associate Professor of Computer Science at Jacobs University Bremen, Germany. He received his doctoral degree in 1996 from the Technical University Braunschweig, Germany. His research interests are distributed systems, network management, wireless sensor networks, and network security.

ABSTRACT

Wireless sensor networks have so far used custom, light-weight network protocols. Given the common presence of 802.15.4 radio interfaces, it becomes feasible to connect motes directly to the global Internet using the 6lowpan adaptation layer. By natively supporting IPv6, motes become first-class Internet citizens capable of communication with any other IPv6-enabled host and benefit from the standardized and already established technology. To this end, a 6lowpan/IPv6 stack has been implemented for TinyOS 2.0. The paper gives an overview of this implementation, describes the motivations behind design decisions, provides an evaluation of the implementation, and briefly compares it to other implementations.

I INTRODUCTION

Wireless sensor networks consist of numerous tiny nodes equipped with various sensors and a radio interface for communication. Among the applications are environment monitoring such as forest fire detection and water or air quality monitoring, wildlife monitoring, smart spaces, medical systems and robotic exploration. Due to the nature of the application, access to the motes may not be feasible after initial deployment. Hence, the devices have to run for extended periods of time on battery power, resulting in low-power, energy-saving designs.

Traditionally, wireless sensor networks have used custom, light-weight network protocols such as Active Messages [1]. However, given the common presence of an 802.15.4 radio interface [2] on the motes and the 6lowpan adaptation layer [3], [4] allowing the exchange of IPv6 packets over 802.15.4 links, enabling IPv6 connectivity in wireless sensor networks and connecting them to the global Internet becomes feasible. By natively supporting the IPv6 protocol, these devices become first-class Internet citizens capable of communication with any other IPv6-enabled host and benefit from the standardized and already established technology and a plethora of readily available applications. To this end a 6lowpan/IPv6 stack [5] has been implemented for TinyOS 2.0 [6], an embedded operating system commonly used in wireless sensor networks.

The implementation includes the 6lowpan adaptation layer with fragmentation and fragment reassembly and compression of the IPv6 and UDP headers. While the full ICMPv6 protocol is not supported, the ICMP echo mechanism and support for the UDP protocol have been implemented. The implementation has been tested on the MicaZ and TelosB motes from CrossBow Technologies.

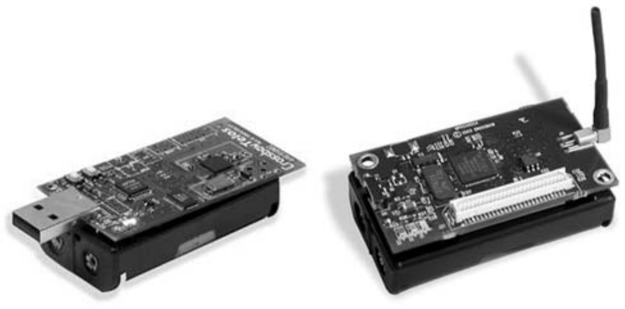

Fig. 1 TelosB and MicaZ motes (see www.xbow.com)

The rest of this document is structured as follows. The hardware platforms and the TinyOS operating system are briefly introduced in Section II, the 802.15.4 standard and the 6lowpan adaptation layer in Section III. The implementation is described in Section IV and evaluated in Section V. Related work is described in Section VI and the document concludes in Section VII.

II HARDWARE PLATFORMS

The hardware platforms used, the TelosB and MicaZ motes, are shown in Fig. 1. Both are tiny, low-power motes with restricted resources and equipped with a Chipcon CC2420 chip providing 802.15.4 connectivity. The TelosB motes feature a Texas Instruments MSP430 MCU. It is a 16-bit RISC MCU clocked at 8 MHz

10.1515/piko.2008.0042

and has 16 registers. The platform offers 10 kB of RAM, 48 kB of flash memory and 16 kB of EEPROM. Requiring at least 1.8 V, it draws 1.8 mA in the active mode and 5.1 μA in the sleep mode. The MicaZ motes feature an Atmel AVR Atmega128L MCU. It is an 8-bit RISC MCU with 32 registers. The platform offers 4 kB of RAM, 128 kB of flash memory and 4 kB of EEPROM. Requiring at least 2.5 V, it draws 8 mA in the active mode and 15 μA in the sleep mode. The mote is similar to the Mica2 mote, but has a different radio interface.

Clearly, these motes are suitable for low data rate applications requiring only minimum data processing. Spending most of their time in the sleep mode, the motes can run for several years on 2 AAA batteries. Target costs of less than 10 cents per mote would enable networks with potentially thousands of devices.

802.15.4 header	802.15.4 header
[Mesh Addressing Header]	*[Mesh Addressing Header]*
[Broadcast Header]	*[Broadcast Header]*
[Fragmentation Header]	*[Fragmentation Header]*
IPv6 dispatch (0x41)	HC1 dispatch (0x42)
IPv6 header (uncompressed)	HC1 Header
UDP header (uncompressed)	HC_UDP Header
UDP payload	HC1 (IPv6) in-line carried fields
	HC_UDP (UDP) in-line carried fields
	UDP payload

Fig. 2 802.15.4 frames with 6lowpan payload. The 6lowpan optional headers are shown in italics and square brackets. On the left is a 6lowpan packet carrying uncompressed IPv6 and UDP headers. The difference to a non-6lowpan IPv6 packet is the dispatch field before the actual IPv6 header. The packet on the right carries compressed IPv6 (HC1) and UDP (HC_UDP) headers. The dispatch field is followed by the HC1 and HC_UDP compression headers and the in-line carried non-compressed fields.

A commonly used operating system for motes is TinyOS [6]. TinyOS is an event-driven embedded operating system designed for extremely restricted devices such as the wireless sensor network motes. It has a very small footprint, with the core OS requiring only 400 bytes of code and data memory. The system provides a set of reusable components which can be combined together to produce an application. An important property of TinyOS is the absence of dynamic memory allocation – all memory is allocated statically.

Currently there are two versions of TinyOS, 1.1 and 2.0. The newer 2.0 version is not backwards compatible with the 1.1 version. In this project, only the 2.0 version has been used.

III IPV6 OVER 802.15.4 (6LOWPAN)

The 6lowpan specifications developed by the IETF specify how IPv6 packets are transmitted over 802.15.4 links.

The 802.15.4 standard [2], [7] specifies the physical (PHY) and medium access control (MAC) layers for low data rate wireless personal area (up to 10 meters) networks. The standard provides for low complexity, low power consumption, low data rate wireless connectivity among a wide range of inexpensive devices. Operating in the unlicensed frequency bands of 2 : 4 GHz, 900 MHz and 868 MHz, it provides bandwidths of 250, 40, and 20 kbps, respectively. The standard provides for two types of addresses, globally unique 64-bit extended IEEE addresses and 16-bit short addresses unique only within a personal area network. In the following, we assume 16-bit short addresses.

Depending on the mode in which 802.15.4 is used, a single frame can at best accommodate 102 bytes of payload since the 802.15.4 frame is limited to 127 bytes. The 802.15.4 specification distinguish between Full Function Devices (FFDs, usually mains powered) and Reduced Function Devices (RFDs, usually battery powered). While 802.15.4 networks can have a rather complex structure, such as a superframe structure with a PAN coordinator or a mesh topology, the IPv6 network layer treats the 802.15.4 as a simple link. Hence, mesh routing functionality must be provided below the IPv6 layer and above the 802.15.4 MAC layer.

The 6lowpan adaptation layer allows to transport IPv6 [8] packets over 802.15.4 [2] links. To meet the IPv6-required MTU of at least 1280 bytes with the 802.15.4 layer offering at most 102 bytes of payload per frame, a fragmentation mechanism below the IP layer is specified using an optional Fragmentation Header before the actual IPv6 header. Support for mesh networking is provided by the optional Mesh Addressing and Broadcast Headers. The former carries the Originator and Final Destination link-layer addresses as the 802.15.4 header contains the intermediate source and next-hop addresses in a mesh networking scenario. The latter contains a sequence number used to detect duplicated frames. Both headers are carried at the beginning of the 802.15.4 payload.

Furthermore, mechanisms for compressing the IPv6 header, in the ideal case, from 40 bytes down to 2 bytes and the UDP header from 8 bytes down to 4 bytes are specified. To distinguish between a compressed and uncompressed header, a 1-byte dispatch value is prepended before the IPv6 header. The optional 6lowpan headers mentioned earlier also start with a dispatch value allowing the recipient to determine what types of headers are present.

The IPv6 compressions scheme is called *HC1*. If used, the IPv6 header is replaced by the 1-byte long HC1 header followed by the in-line carried non-compressed fields. Various bits of the HC1 header specify which fields of the IPv6 header are compressed and which are carried in-line. After the in-line carried fields follows the next header such as a UDP or ICMP header.

The UDP compression scheme is called *HC_UDP*. It can only be employed in conjunction with the HC1 compression scheme. Then the HC1 header is followed by the 1-byte HC_UDP header. Various bits of the HC_UDP header specify which fields of the UDP header are compressed and which are carried in-line. The HC_UDP header is then followed by the in-line carried HC1 fields and the non-compressed inline carried UDP header fields. Afterwards follows the UDP payload.

Examples of 6lowpan packets with and without compression are shown in Fig. 2. The format of the 6lowpan adaptation layer and details of the compression schemes are specified in [4].

IV IMPLEMENTATION

The initial goal for the implementation was to support replying to an ICMP echo request message (ping) and exchanging of UDP datagrams. Only the bare minimum necessary for meeting that goal was implemented. Additional and possibly not needed features would mean not only more time for development, but a more severe consequence in an embedded scenario would be the increased code size and memory requirements.

The main restriction of the implementation was the amount of RAM available on the target platform, i.e., 4 KB on the MicaZ. Although aiming for an embedded implementation, easily readable and maintainable code was preferred over optimizing to squeeze into the least possible amount of memory at the cost of hard to understand programming constructs, hacks or munging of code into a few large functions for saving space on the stack. Furthermore, the design should allow to easily add more functionality in the future.

```
interface UDPClient {
   command error_t listen(uint16_t port);
   command error_t connect(const ip6_addr_t *addr,
                           const uint16_t port);
   command error_t sendTo(const ip6_addr_t *addr,
                          uint16_t port,
                          const uint8_t *buf,
                          uint16_t len);
   command error_t send(const uint8_t *buf,
                        uint16_t len);
   event void sendDone(error_t result, void* buf);
   event void receive(const ip6_addr_t *addr,
                      uint16_t port,
                      uint8_t *buf, uint16_t len);
}
```

Fig. 3 UDPClient interface

A Application interface

An application can interact with the 6lowpan/IPv6 stack via the UDPCLIENT interface, shown in Fig. 3. The interface represents a single UDP endpoint and allows to send and receive UDP datagrams. An application can register for listening on a UDP port with the LISTEN() command or specify the remote communication endpoint using the CONNECT() command. The SEND() and SENDTO() commands allow to send datagrams. The application provides a buffer with the datagram payload, which will be used by the IP stack until the SENDDONE event signals that the datagram has been sent. Receipt of a datagram is signaled with the RECEIVE event.

The IPv6 stack supports one link-local and one global IPv6 address. Currently, the application cannot change the IPv6 addresses. The prefix of the global address is hardcoded. The interface identifier for both addresses is calculated from the device's link-layer address as specified in the 6lowpan specification [4].

B Functional overview

As the implementation was designed to be easily extensible, each network layer and protocol is handled by a separate function rather than combining the handling of multiple protocols for the sake of efficiency.

The receipt of an 802.15.4 frame is signaled by TinyOS' ACTIVEMESSAGEC component with the RECEIVE event of the RECEIVE interface. The payload is then passed to the LOWPAN_INPUT() function, which processes the optional Mesh Addressing, Broadcast and Fragmentation Headers, if present. After a packet has been successfully reassembled from fragments or an unfragmented packet was received, the remaining payload starting with the 6lowpan-encapsulated IPv6 header is passed to the LAYER3_INPUT() function. From the dispatch value preceding the IPv6 header it learns whether the IPv6 header is *HC1*-compressed or not. The remaining payload starting with the IPv6 header is then passed to IPV6_INPUT_COMPRESSED() or IPV6_INPUT_UNCOMPRESSED(). Based on the IPv6 NEXT HEADER field, the corresponding function for processing the next layer is called and the remaining payload starting at the respective header is passed. This is ICMP_INPUT() for the ICMP protocol. The UDP header is handled by UDP_INPUT_UNCOMPRESSED() or UDP_INPUT_COMPRESSED() if it is HC_UDPCOMPRESSED. The ICMP_INPUT() function inspects the ICMP header, verifies the checksum and possibly initiates a reply to an ICMP echo request. The UDP input processing functions also verify the checksum and possibly notify the application via the RECEIVE event of the UDPCLIENT interface, including a pointer to the UDP payload and information about the sender. A sequence diagram showing how a UDP datagram is received can be found in Fig. 4.

Functions for sending a packet are also separate for each protocol and headers for the corresponding layers are prepended to the buffer. The sending of a packet is initiated by the application or by the ICMP_INPUT() function when replying to an ICMP echo request. The application uses the SEND() or SENDTO() command of the UDPCLIENT interface, resulting in a call to one of the UDP output functions, UDP_OUTPUT_UNCOMPRESSED() or UDP_OUTPUT_COMPRESSED(), depending on whether the UDP header should be compressed or not. ICMP_INPUT() would invoke ICMP_OUTPUT() in order to send a reply to an ICMP echo request. The UDP or ICMP output functions would determine

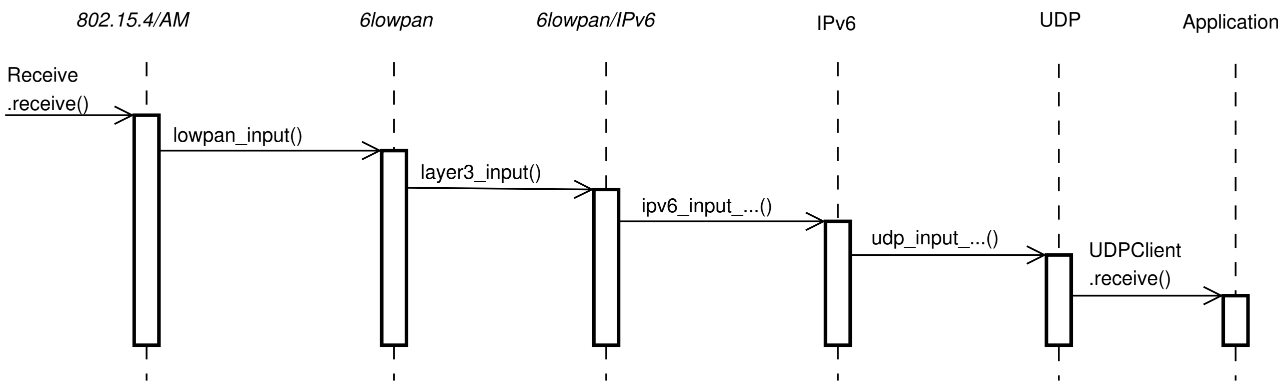

Fig. 4 Receiving a UDP packet

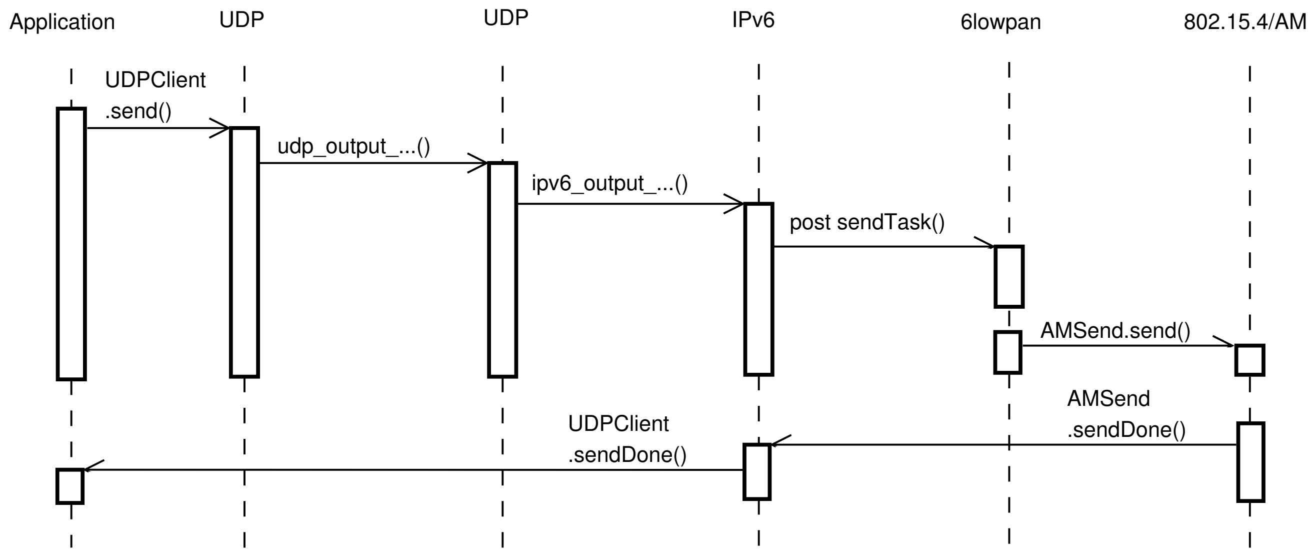

Fig. 5 Sending a UDP packet

the correct IPv6 source address for the packet unless specified by the calling function, calculate the checksum, prepend the header for the respective protocol to the buffer and call the IPv6 output function. This can be IPV6_OUTPUT_COMPRESSED() or IPV6_OUTPUT_UNCOMPRESSED(), depending on whether the IPv6 header should be compressed. In contrast to the receiving case, the choice of calling the compressed or uncompressed version is done at compile time. After filling in the IPv6 header, the SENDTASK task is scheduled for sending the packet. It uses the SEND() command from the AMSEND interface provided by TinyOS' ACTIVEMESSAGEC component to send the 802.15.4 frame. If the packet does not fit into one frame, the correct fragmentation header is added by SENDTASK and after sending a frame SENDTASK is scheduled again for sending subsequent fragments. An example showing the sending of a UDP datagram is shown in the sequence diagram in Fig. 5.

The reasons for scheduling the SENDTASK task rather than sending the packet directly are:

– The destination's link-layer address has to determined before the packet is sent, e.g. by neighbor discovery. This requires sending a neighbor solicitation and waiting for a neighbor advertisement to be received. The sending of a packet and waiting for a reply could not be done without exiting the IPv6 output function and returning control to TinyOS. The link-layer address is also used by the HC1 compression of the IPv6 header.
  Note that the requirement for initiating the neighbor discovery process can be mitigated by assuming that the node never initiates a connection. If a packet is always sent as a response to a received packet, then the IPv6 address to link-layer address mapping can be learned from the received packet. Hence the node would not have to initiate a neighbor discovery on its own. Another possibility would be to use a fixed neighbor table mapping or to use the link-layer broadcast address.
– The packet may require 6lowpan fragmentation. This requires sending of more than one frame. However, after requesting the sending of the first fragment, the function would have to return control to TinyOS. Only after the SENDDONE event of the AMSEND interface is signaled can the sending of the next frame be requested.
– While the fragments of a packet are being sent, another packet may be received by the radio. By splitting the sending of different fragments into separate executions of the SENDTASK task, it is possible to receive a packet even before all frames of an outgoing packet have been sent. If there is enough memory and buffer space, the received packet could be processed rather than discarded. For example, it may be possible to reply to a neighbor solicitation, which does not require fragmentation, while sending a large UDP datagram requiring fragmentation. Note that the TelosB platform offers more than twice as much memory as the MicaZ platform and the queuing design allows to trade more memory for being able to respond to such neighbor solicitation as described in the above example.

C Buffers

A data structure for representing packets, LOWPAN_PKT_T, was designed to efficiently accommodate both a short unfragmented packet and a long packet after fragment reassembly (see Fig. 6). Furthermore, an application sending a UDP datagram provides a buffer with the UDP payload. This buffer is reused rather than copied into another buffer within the IPv6 stack and headers for the various layers are prepended.

```
typedef struct _lowpan_pkt_t {
    uint8_t *app_data;              /* app data buffer */
    uint16_t app_data_len;
    uint8_t *app_data_begin;
    uint8_t app_data_dealloc;

    uint8_t header[LINK_DATA_MTU]; /* header */
    uint16_t header_len;
    uint8_t *header_begin;

    uint16_t dgram_tag;             /* fragmentation */
    uint16_t dgram_size;
    uint8_t dgram_offset;

    ip6_addr_t ip_src_addr;         /* IPv6 addresses */
    ip6_addr_t ip_dst_addr;

    hw_addr_t hw_src_addr;          /* 15.4 addresses */
    hw_addr_t hw_dst_addr;

    uint8_t notify_num;              /* UDPClient handle */
    struct _lowpan_pkt_t *next;
} lowpan_pkt_t;
```

Fig. 6 The lowpan_pkt_t structure

Important factors in the design have been the maximum size of a 6lowpan payload after fragment reassembly and the size of an 802.15.4 payload. The former is 1280 bytes while the latter is at most 102 bytes. By adding up the sizes of the various supported headers (6lowpan, IPv6, UDP and ICMP), it turns out that any combination of the headers fits into the 102 bytes of an 802.15.4 payload. Therefore, LOWPAN_PKT_T contains two types of buffers. One is a statically allocated buffer of size 102 bytes, the HEADER. The other one is a pointer to the application-provided data buffer, the APP_DATA, and is not allocated within structure. This allows to change the owner of that buffer from one packet to another without having to copy the contents of the buffer.

When sending a UDP packet, the application-supplied buffer is used as APP_DATA and the headers are prepended to the header buffer. As already discussed, all headers certainly fit into this buffer. When replying to an ICMP echo request, the APP_DATA buffer is reused for sending the same ICMP payload in the reply.

If an unfragmented packet is received, it fits into the header buffer. A fragmented packet would after reassembly be stored in an APP_DATA buffer provided by the fragment reassembly code.

D Fragment reassembly

In order to reassemble a 6lowpan-fragmented packet, a buffer of size up to 1280 bytes is needed. Although the fragment header contains the size of the whole packet, the absence of dynamic memory allocation in TinyOS does not easily allow to allocate a buffer of just the right size. Hence, a fragment reassembly buffer of 1280 bytes was chosen to accommodate for the largest possible case. The buffers are managed by a pool component with compile time configurable size, allowing to change the number of fragmented packets than can be reassembled concurrently.

The 6lowpan specification [4] requires different behavior for different types of overlapping fragments. If a fragment is received that overlaps another fragment and differs from the overlapped fragment in either size or offset then the already accumulated fragments shall be discarded. However, if the fragment were just a duplicate of another already received fragment, the already accumulated fragments do not need to be discarded. To be able to determine whether a fragment is just a duplicate or really a different overlapping fragment, the information about the size and offset of each already received fragment has to be stored until the complete datagram is reassembled.

In order to satisfy the above mentioned requirement, a linked list is used. Elements of this list contain information about the offset and length of the received fragments belonging to a given datagram. Note that the information about received fragments could also be stored in an array instead of a linked list. This would save the memory needed for the linked list pointers, but would not allow moving items between lists for different datagrams being concurrently reassembled. By default, there are fifteen times as many list elements available as the number of buffers available for fragment reassembly. The motivation is that by using the full size of the 802.15.4 frames, a 1280 bytes long payload would be fragmented into 15 fragments. Storing the size and offset of a fragment requires 2 bytes. Assuming 15 fragments per packet, this requires 30 bytes per packet.

The alternative to the linked list would be a bitmap. Each bit in the bitmap would then represent the information whether the data at the corresponding offset has been received. With offsets of fragments being always multiples of 8 bytes and the total packet length of 1280 bytes, a bitmap of 20 bytes would suffice. Although the bitmap suffices for keeping track of which parts of the payload have been received, it does not allow to determine whether a received overlapping fragment is only a duplicate or really a differing overlap.

While the linked list approach may allow to detect a broken fragment sender, it requires more memory and additional complexity in the implementation. Furthermore, an incorrectly fragmented datagram should be detected anyway, as ICMP, TCP and UDP all use a checksum with IPv6. Hence, relaxing the requirement in the format draft would seem useful from the implementation point of view. A comment on this issue and a suggestion to change the draft has been sent to the 6lowpan mailing list.

E Link-layer

A significant drawback of the implementation and a limitation to interoperability is that TinyOS 2.0 does not contain a proper 802.15.4 stack. Jan Flora from the University of Copenhagen has implemented an 802.15.4 stack for TinyOS 1.1, available in the TinyOS 1.1 distribution under CONTRIB/DIKU. However, the porting to TinyOS 2.0 has staggered on a problem with timers not providing the accuracy required by the 802.15.4 specification.

Instead of a proper 802.15.4 stack, TinyOS networking is based on Active Messages [1]. These are sent in 802.15.4 data frames, but the required 802.15.4 functionality is not implemented. The Active Message header used is identical to the 802.15.4 header except for an additional field AM TYPE. As this is in the payload part of the frame, it would not interfere with the correct functioning of 802.15.4. However, setting this field to a different value as part of the payload is not easily supported by TinyOS. Changing this would require modifying parts the TinyOS code for Active Message processing and for using the CC2420 radio chipset to not send the AM TYPE field.

As the goal of the described work was a 6lowpan implementation rather than an 802.15.4 implementation, the choices were to tunnel the 6lowpan payload as Active Messages payload, i.e., have the AM TYPE field at the beginning of the 802.15.4 payload or to modify TinyOS to get rid of that field. However, such modifications would be rather unlikely to get accepted back into TinyOS. Furthermore, they would still not guarantee interoperability with other 802.15.4 devices and 6lowpan implementations as the remaining functionality of 802.15.4 would still be missing. Hence, the easier approach was taken and the 6lowpan payload is simply tunneled as Active Message payload and prefixed with the AM TYPE field in the 802.15.4 payload. An interoperability frame format has been defined in [9] to allow coexistence of Active Messages and 6lowpan packets by using 6lowpan dispatch codes. As the document promises an implementation to appear soon in the CVS version of TinyOS, it will likely be possible to use the whole 802.15.4 payload for 6lowpan in the near future.

F 6lowpan for Linux

Most PCs today do not have an 802.15.4 interface and common operating systems such as Linux or the BSDs do not include a 6lowpan implementation. To allow for exchanging packets between the motes and a Linux PC, a translating daemon has been developed to use a mote as an 802.15.4 interface. The scenario is shown in Fig. 7.

The base station mote runs the TinyOS sample application *BaseStationCC2420* forwarding traffic between the 802.15.4 and the USB interface of a mote. This mote is connected to the PC via the USB interface.

The translating daemon on the PC is a C program exchanging packets between the USB interface and a tun interface. The latter is a virtual network interface allowing a user space process to read and write packets to it. The daemon decapsulates the 6lowpan-encapsulated IPv6 packets received from the mote and 6lowpan-encapsulates the plain IPv6 packets received on the tun interface. This allows to use standard IPv6 applications on Linux for communication with the motes without modifying the Linux kernel. Furthermore, by enabling IPv6 forwarding on the PC, the motes can be connected to the global Internet.

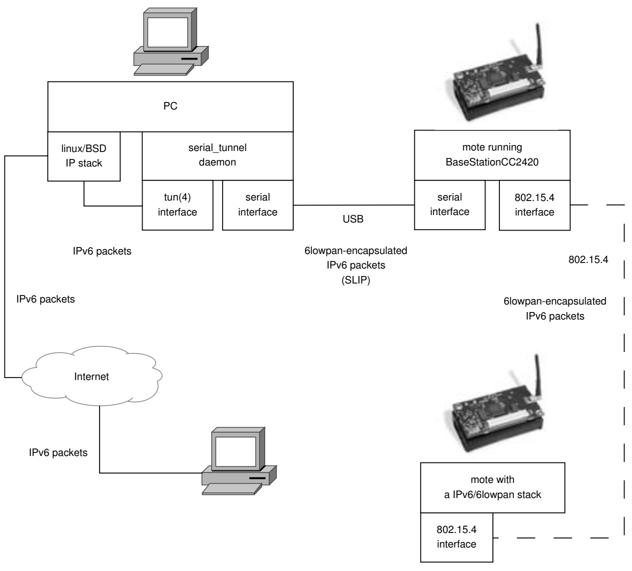

Fig. 7 The motes and the Internet

Note that the usage of the base station mote and the translating daemon is not an architectural or design limitation. It is just an additional bonus allowing to use a mote as an 802.15.4 interface for a PC without an 802.15.4 interface.

V EVALUATION

The implementation was tested using a scenario as shown in Fig. 7. A TelosB mote with the BaseStationCC2420 application was used as an 802.15.4 interface for a Linux PC running the translating daemon. Two other TelosB motes and a MicaZ mote were flashed with the 6lowpan implementation. The motes were successfully replying to ICMP echo requests initiated from the PC as well as exchanging UDP datagrams. Both short unfragmented packets as well as large, fragmented packets of sizes up to 1280 bytes were successfully exchanged. Measured round-trip times for ICMP echo packets of various sizes are plotted in Fig. 8. The scenario for the measurements was as shown in Fig. 7 using a TelosB mote as the base station. The ICMP echo packets were exchanged between the PC and another TelosB mote. Steps in the plot indicate a change in the number of fragments needed to transport a certain packet size.

In order to test the UDP protocol an application with a minimal command line interface over UDP was developed. It offers commands to toggle the leds, activate the speaker on the sensorboard of a MicaZ mote and request light and temperature sensor readings. For testing, responses of various sizes can be requested.

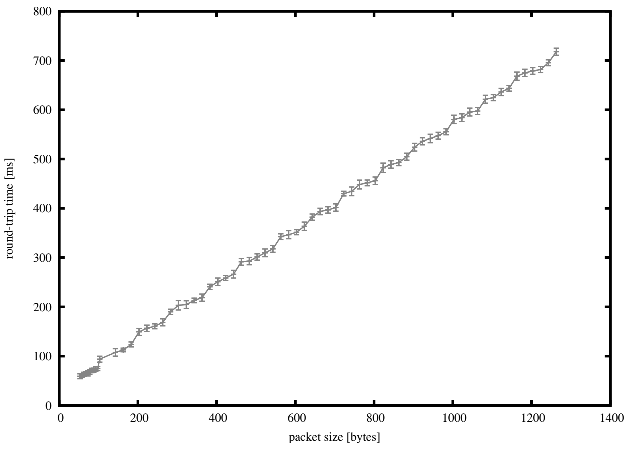

Fig. 8 Ping round-trip times. The packet size is the complete layer3 header and payload length, excluding the 6lowpan optional headers such as the Fragmentation Header. Round-trip times shown are averages over 50 pings for packet size up to 103 bytes and 20 pings for larger packets. Error bars show one standard deviation. Steps in the plot indicate a change in the number of fragments.

Fragment reassembly on the mote was found to work sporadically as not all fragments leaving the PC seemed to have arrived at the mote, but adding a USLEEP(10000) command to sleep 10ms before sending subsequent fragments in the daemon on the PC has solved the problem with losing fragments. The likely cause of the dropped packets was sending too many packets in a very short time to the base station mote.

6lowpan header compression for the IPv6 and UDP headers, *HC1* and *HC_UDP*, respectively, are supported as well as uncompressed headers. Non-zero flow labels and traffic classes and compressed UDP port numbers length is not byte-aligned. Hence, inline-carried fields after these ones would also not be byte-aligned. It was suggested on the 6lowpan mailing list to change the format in a way minimizing the number of fields not starting on a byte-boundary and hence requiring less complexity in the implementation. In particular, non-zero flow label and traffic classes are very uncommon. Hence, they could be padded with additional four bits when carried inline to end on a byte boundary. The order of the HC_UDP in-line carried fields could be changed so that the UDP port numbers would be carried as the last rather than the first ones. This would minimize the number of fields not aligned on a byte-boundary.

The main limitation to interoperability with other 6lowpan implementations is the absence of a proper 802.15.4 stack in TinyOS 2.0 and the presence of the 1-byte AM TYPE field in the beginning of the 802.15.4 payload before the 6lowpan payload.

Although the implementation supports the ICMP echo mechanism and the UDP protocol, many features required for IPv6 implementations are missing. Among others, features of the Neighbor Discovery have not been implemented, IPv6 extensions headers are not processed, IPv6 fragmentation is not supported and ICMP error messages are not generated. While many of these could be added, it is unclear whether they make sense in an embedded system. For example, is one willing to trade decreased battery life for regular neighbor advertisements or neighbor unreachability detection? Or if an error is encountered while processing a received packet, should a 1280 bytes long ICMP error message be sent back? Should one be sent back at all? Furthermore, many of these issues, such as Neighbor Discovery optimizations, and which features are required, e.g., IPSec support, are still under discussion in the working group.

The design goal of the implementation was to write easily extensible code rather than optimize for smallest possible code size and memory usage. The implemented 6lowpan/IPv6 stack compiled for the TelosB platform with TinyOS 2.0 and the testing application requires 13448 bytes of ROM and 2196 bytes of RAM. By disabling fragment reassembly, the RAM requirement can be decreased to 794 bytes.

VI RELATED WORK

Our work uses the 802.15.4 radio interface as specified in [2]. Some properties of this radio interface and its implications for carrying IPv6 traffic over 802.15.4 are described in [10]. An overview of the recent evolution of the 802.15.4 standards can be found in [11].

Several 6lowpan implementations for wireless sensor networks have been announced while this project was in progress. The company *Arch Rock* has announced a commercial 6lowpan implementation *Primer Pack/IP* in March 2007. As this is a commercial implementation, technical information is scarce. The company *Sensinode* has released a GPL-licensed 6lowpan implementation called *NanoStack v0.9.4* in April 2007. It is claimed to be up to date with version 12 of the 6lowpan format draft and to include IEEE 802.15.4 Beacon-mode support. The source code, however, does not seem to include 6lowpan fragmentation support and UDP checksumming.

uIP[12] is a TCP/IPv4 stack implementation developed by Adam Dunkels. It runs on 8-bit micro controllers with a few hundred bytes of RAM. It has been ported to TinyOS 1.1 by Andrew Christian from the Hewlett-Packard Company. The port is available in the TINYOS-1.X/CONTRIB/HANDHELDS/TOS/LIB/UIP/ directory of the TinyOS 1.1 distribution but is not easy to port to TinyOS 2.0. The uIP code is also included in the Contiki [13] embedded operating system. The system requires at least 3876 bytes of ROM and 230 bytes of RAM on the MSP430 platform such as the TelosB motes. For interfacing with a PC without an 802.15.4 interface, Contiki allows to use a mote as an 802.15.4 interface and the tunslip daemon. A comparison of ping round-trip times between the 6lowpan implementation for TinyOS and the uIP IPv4 implementation for Contiki is shown in Fig. 9. The measurement scenario was as shown in Fig. 7 using one TelosB mote as the base station. The ICMP echo packets were exchanged between the PC and another TelosB mote. While Contiki features lower round-trip times, they are still in the same order as the TinyOS times. Contiki is by default not doing IP fragment reassembly or header compression, resulting in simpler processing than required for 6lowpan. A further cause for the differences might be the different 802.15.4 implementations. Note that with the Chipcon CC2420 chip, a significant portion of the MAC layer functionality, such as the back-off and random wait schemes, are implemented in software.

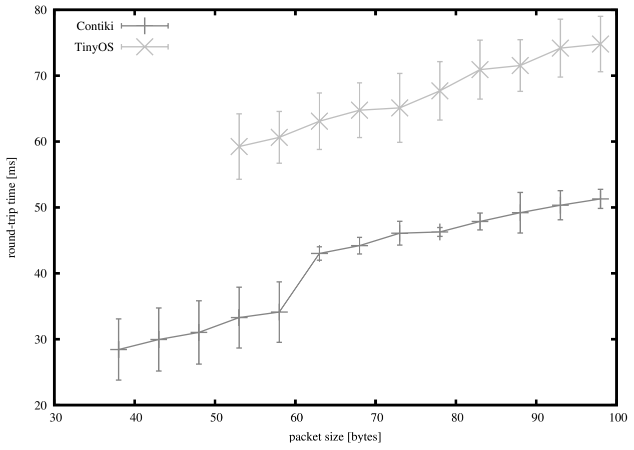

Fig. 9 Ping round-trip times for Contiki and the 6lowpan implementation for TinyOS. The packet size is the complete layer3 header and payload length. Round-trip times shown are averages over 50 pings.

While an IP stack can be implemented on the motes, it is also possible to use a proxy-based scheme. In this case a special proxy server is employed as a gateway separating the sensor network and the IP network. This allows to freely choose the communication protocol used within the sensor network. Although limited to IPv4, the Sensor Internet Protocol (SIP) [14] is an example of such a proxy scheme.

VII CONCLUSION

A 6lowpan/IPv6 stack has been implemented for the TinyOS 2.0 operating system and was tested on the TelosB and MicaZ hardware platforms. Using the translating daemon on the PC and a mote as the base station, it is possible to exchange IPv6 packets between the motes and a PC. In case IP forwarding is set up on the PC and a properly assigned and routable global IPv6 prefix is used, the motes can be connected to the global Internet.

It should be stressed that the translating daemon and the base station mote are not an architectural or design limitation. It is only an additional bonus allowing a PC without an 802.15.4 interface and using an operating system without a 6lowpan implementation to participate in 802.15.4 6lowpan networks by using the mote as an 802.15.4 interface.

More information about the implementation can be found in [5]. The source code is available from TinyOS CVS [15]. The main contributions of the described work are:

– A freely available 6lowpan implementation has been released under a BSD-style licence. Note that the other free implementation, from Sensinode, is GPL-licenced, which may be a problem for commercialization.

– The 6lowpan implementation has been integrated into the TinyOS operating system so that future releases of the system would natively support 6lowpan.
– Practical experience from implementing the 6lowpan specification [4] and noticed shortcomings of the specifications have been provided back to the IETF working group. The issues are mainly related to the fragment reassembly and byte-alignment of the in-line carried fields when headers are compressed.

As future work, the issue with the AM TYPE field will be resolved. It would also be useful to develop a native Linux/BSD kernel driver for 802.15.4 interfaces attached directly via 802.15.4 interface cards so that using the base station mote would become obsolete and normal PC boards can be used more easily as full function devices.

Two mesh routing algorithms, namely LOAD and DYMOlow have been prototyped in TinyOS 2.0 [16], [17] and the Mesh Addressing and Broadcast Headers could be used with these mesh routing algorithms for mesh networking below the IPv6 layer. Finally, existing Internet application protocols such as SNMP can be implemented in TinyOS on top of the 6lowpan stack for collecting sensor values.

REFERENCES

[1] J. Hill; P. Bounadonna; and D. Culler: Active Message Communication for Tiny Network Sensors. http://webs.cs.berkeley.edu/tos/papers/ammote.pdf.

[2] IEEE: IEEE Std. 802.15.4-2003, October 2003.

[3] N. Kushalnagar; G. Montenegro; and C. Schumacher: IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919, Intel Corp, Microsoft Corporation, Danfoss A/S, August 2007.

[4] G. Montenegro; N. Kushalnagar; J. Hui; and D. Culler: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, Microsoft Corporation, Intel Corp, Arch Rock Corp, September 2007.

[5] Matúš Harvan: Connecting Wireless Sensor Networks to the Internet – a 6lowpan Implementation for TinyOS 2.0. Master's thesis, School of Engineering and Science, Jacobs University Bremen, May 2007.

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

[7] E. Callaway; P. Gorday; L. Hester; J.A. Gutierrez; M. Naeve; B. Heile; and V. Bahl: Home Networking with IEEE 802.15.4: A Developing Standard for Low-Rate Wireless Personal Area Networks. IEEE Communications Magazine, 40 (8): 70-77, August 2002.

[8] S. Deering and R. Hinden: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, December 1998.

[9] Jonathan Hui; Philip Levis; and David Moss: TEP 125: TinyOS 802.15.4 Frames. http://www.tinyos.net/tinyos-2.x/doc/txt/tep125.txt.

[10] K. Srinivasan; P. Dutta; A. Tavakoli; and P. Levis: Some Implications of Low Power Wireless to IP Networking. In: Fifth Workshop on Hot Topics in Networking (HotNets-V). ACM, November 2006.

[11] L.D. Nardis and M.-G. Di Benedetto: Overview of the IEEE 802.15.4/4a standards for low data rate Wireless Personal Data Networks. In: Proc. of the 4th IEEE Workshop on Positioning, Navigation and Communication 2007 (WPNC '07), Hannover, March 2007. IEEE.

[12] A. Dunkels: Full TCP/IP for 8-bit architectures. In: Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03), May 2003.

[13] Adam Dunkels; Björn Grönvall; and Thiemo Voigt: Contiki – a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the First IEEE Workshop on Embedded Networked Sensors (IEEE Emnets 2004), 2004.

[14] X. Luo; K. Zheng; Y. Pan; and Z. Wu: A TCP/IP implementation for wireless sensor networks. In: IEEE International Conference on Systems, Man, and Cybernetics, 2004.

[15] Matúš Harvan: 6lowpan implementaion source code (TinyOS CVS). http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x/tos/lib/net/6lowpan/.

[16] V. Iliev: Mesh Routing for Low-Power Mobile Ad-Hoc Wireless Sensor Networks using LOAD. BSc Thesis, May 2006.

[17] I. Zarow: Mesh Routing for Low-Power Mobile Ad-Hoc Wireless Sensor Networks using DYMO-low. BSc Thesis, May 2006.