TinySPOTComm: Facilitating Communication over IEEE 802.15.4 between Sun SPOTs and TinyOS-Based Motes

Daniel van den Akker[1], Kurt Smolderen[1,2], Peter De Cleyn[1], Bart Braem[1], and Chris Blondia[1]

[1] PATS, Dept. of Mathematics and Computer Science,
University of Antwerp – IBBT,
Middelheimlaan 1, B-2020, Antwerp, Belgium
{Daniel.VanDenAkker,Kurt.Smolderen,Peter.DeCleyn,
Bart.Braem,Chris.Blondia}@ua.ac.be
http://www.pats.ua.ac.be/
[2] Dept. of Industrial Sciences and Technology,
Karel de Grote-Hogeschool,
Salesianenlaan 30, B-2660, Hoboken, Belgium

Abstract. The increasing popularity of sensor network has spawned a wide range of platforms and frameworks for sensor network development. While in theory nodes based on different frameworks should provide radio stack compatibility, in practice this is rarely the case. We explore this problem by providing a case study and introduce TinySPOTComm, a customized radio stack for the Sun SPOT platform which allows for radio communication between IEEE 802.15.4 based TinyOS motes and Sun SPOTs. The TinySPOTComm radio stack remains fully compatible with the Sun SPOT radio stack and its network performance is only marginally affected in comparison to the default Sun SPOT radio stack. Performance tests have shown good results when communicating between TinyOS motes and Sun SPOTs. The round trip time, when measured between a Sun SPOT and a TinyOS mote, is affected by no more than 15%, in comparison to the RTT between two TinyOS motes. In the same scenario an increase in throughput of more than 50% has been measured.

Keywords: Sun SPOT, TinyOS, 802.15.4, compatibility, sensor network.

1 Introduction

Over the years sensor networks have become increasingly popular, not only as a research topic, but also for real-life applications. Because of this, many frameworks for sensor network development now exist, each with its own programming language preference and supported hardware platforms. Unfortunately, communication between nodes using different sensor network frameworks is, despite compatibility at the hardware level, generally not possible.

N. Komninos (Ed.): SENSAPPEAL 2009, LNICST 29, pp. 177–194, 2010.

This paper attempts to provide further insight into this problem by means of a case study. More specifically, we focus on the radio stack compatibility between two distinct types of sensor nodes: TinyOS-motes [1] and Sun SPOTs [2].

These two platforms are, amongst others, further discussed in section 2. An introduction to the IEEE 802.15.4 [12] and LowPAN [10] protocols, which are both commonly used in sensor networks, is provided in section 3. The radio stacks of these two platforms are further discussed in section 4. In section 5 we discuss the effect of the changes proposed in section 4 on the wireless networking performance, and we conclude our paper in section 6.

2 Current Frameworks for Sensor Network Development

In this section several sensor network development frameworks are discussed.

2.1 TinyOS

TinyOS has been widely accepted as a well supported, highly usable and efficient framework for sensor network development. It is, as the name suggests, a 'Tiny' Operating System designed for energy-constrained devices and for sensor nodes in particular. Sensor nodes running TinyOS are usually referred to as TinyOS-motes. TinyOS is written in nesC, a component-based version of the C programming language. A nesC application consists of multiple components. Each component is expected to both provide and use interfaces. The provided interfaces allow other components to interact with the component, while used interfaces define the functionality required by it. Unlike interfaces in other languages, nesC-interfaces are bidirectional since they specify both commands and events. Commands are functions implemented by the component providing the interface and may be invoked by other components requiring the interface. Events work in the opposite direction. They are implemented by requiring components and may be invoked from the component providing the interface. Commands may therefore be regarded as the equivalent of methods while events resemble call-back functions.

Based on this principle, TinyOS uses an event-driven asynchronous approach to concurrency. Unlike regular programming-languages, most operations are executed asynchronously. When performing operations, such as sending packets, a command is issued to initiate the operation. This command immediately returns and does not block until the operation is complete. Instead an event is used to signal completion. Furthermore, TinyOS does not support multithreading, since it would require a separate stack for each thread and introduce a significant thread-swapping overhead. Instead, so called Tasks may be submitted which are then sequentially executed by the only available thread. As a result, TinyOS applications have a very small memory-footprint and require very little computational power.

Although TinyOS allows for the development of efficient, modular applications and is supported on numerous hardware platforms, it has a few major drawbacks. One of the largest issues is that due to the structure of nesC and the used concurrency model, writing TinyOS applications is a complex task. Furthermore, debugging these applications is equally challenging. It either requires applications to be simulated, which prevents low-level device interactions to be tested, or requires specialized tools and hardware, such as a JTAG controller, to directly interface with the mote's microcontroller. Moreover, it is only possible to debug the C code generated by the nesC compiler. As a result of these restrictions, TinyOS may only reach a specific target audience. A second problem of TinyOS is that, since nesC compiles both application-specific and general-purpose TinyOS-originated code into a single binary, it is not possible to update the application code without replacing the entire binary. As a result it is generally not possible to reprogram TinyOS-motes once they have been deployed.

Levis et al. [3] addresses this problem by providing Maté, a byte code interpreter implemented on top of TinyOS. Maté is based on the observation that many sensor network applications rely on a common set of components and services. Maté therefore specifies a specialized instruction set containing not only regular instructions such as arithmetic operations but also more elaborate instructions such as the 'send' instruction which is used to transmit data. Furthermore the Maté-instruction set provides eight 'user' instructions which allow Maté to be extended with domain-specific functionality. Application code is broken into capsules, each containing up to 24 instructions, which are then transmitted to the individual motes and scheduled for execution. As a result, sensor network applications may be altered in an efficient way, even after the sensor network has been deployed.

While both Maté and TinyOS allow the development of efficient sensor network applications, they both require a profound knowledge about low-level (nes)C or even assembler programming. As a result both exhibit a very steep learning curve and therefore only appeal to a specific audience.

2.2 Java Based Frameworks

To enable the use of sensor networks by a broader community, various Java virtual machines for sensor nodes have been proposed. NanoVM [4], for instance, is an open-source Java VM designed to run on the AVR ATmega8 CPU. While it can run using only 8KB of program flash and 1KB of RAM, it only supports a very limited subset of the Java language specification and the JDK. AmbiCompVM [5] an extension to NanoVM does support most of the Java specification. In order to limit the memory footprint and CPU usage, compiled class-files are transcoded for the target platform and then statically linked into a single binary. VM* [6] takes this principle even further, and synthesizes a specialized VM for each application. Since these platforms are not open-source, their benefit for researching sensor networks is limited.

2.3 Squawk and Sun SPOTs

In April 2007, Sun introduced their own platform for sensor network development. Specialized sensor nodes, called Sun SPOTs [2] (Small Programmable Object Technology), are equipped with the Squawk VM, a Java virtual machine that was originally developed for the next generation smart cards [7] and has been refined for sensor network usage.

While Squawk shares many characteristics with other embedded device JVMs, such as a specialized compressed byte code format, the open source Sun SPOT platform provides many features that greatly facilitate the development of Sun SPOT applications. Not only is the Squawk VM fully compliant with Java ME [2]. It also provides an extensive debugging framework and allows applications to be migrated between devices. Furthermore, the Sun SPOT platform allows for over-the-air deployment of Sun SPOT applications. Despite these features, Squawk is still required to run on sensor nodes, which usually have a limited memory capacity. Squawk therefore runs directly above the hardware and does not require an operating system.

However, unlike most other VMs Squawk is almost entirely implemented in Java itself [2]. This is a result of the observation that the Java language is well suited to express most VM functionality. Therefore only the actual byte code interpreter was written in C. All other features such as the thread scheduler and the garbage collector have been written in Java. Consequently all device drivers have also been written in Java and may easily be modified for domain specific purposes. Like with other VMs the standard Java byte codes are translated into a more compact byte code format. Furthermore Squawk uses a "Split VM" architecture. Instead of directly performing the loading process off the Sun SPOT device, class files are first loaded on the host where the application is deployed from. The internal object memory representation of these classes is serialized into suite files which are then deserialized on the Sun SPOT, where they are placed in predetermined memory areas. As a result both the memory footprint and the time required to launch an application are reduced. Despite these optimizations, the Squawk VM requires hardware that is more powerful than what is usually found in TinyOS based sensor networks.

A Sun SPOT consists of an ARM920T processor, running at 180 MHz, 512KB RAM- and 4MB flash memory [2], while the Crossbow TelosB mote running TinyOS, for instance, uses a TI MSP430 micro controller with only 10KB RAM running at 4MHz [9]. As a result, the average battery lifetime of a Sun SPOT is a few orders of magnitudes less than is actually required for a sensor network. Sun SPOTs are therefore most suited for rapid prototyping of sensor network applications.

3 IEEE 802.15.4 and LowPAN

Sun SPOTs and many types of TinyOS-programmable sensor nodes [8] are equipped with an IEEE 802.15.4 compliant radio to perform wireless communication. Furthermore, the network layer of the Sun SPOT stack is heavily based on

the IETF's LowPAN [10] specification. Therefore, in the following, we introduce these protocols.

3.1 IEEE 802.15.4

As Akyildiz et. al [11] point out, the use of power-limited devices in wireless sensor networks requires power-optimized radio protocols. Since existing protocols, such as IEEE 802.11, consider power-consumption as a secondary concern, the IEEE 802.15.4 standard was introduced. This standard provides both PHY and MAC layer protocols for low-power low-bandwidth wireless networks and is used for instance in the proprietary ZigBee radio stack. At the physical layer, IEEE 802.15.4 can operate on a variety of frequency bands, including the 2.4GHz ISM-band in which 802.15.4 defines 16 separate channels. DSSS and OQPSK are used to send symbols, each containing 4 bits of data, at a rate of up to 62.5 ksymbols or 250kbits per second.

Above the physical layer, the MAC layer provides two modes of operation: beacon-enabled mode and non-beacon-enabled mode. The beacon-enabled mode provides many interesting features such as contention-free channel access and polling-based frame reception. In non-beacon-enabled mode, all frames are sent using the contention based CSMA-CA algorithm. Since neither TinyOS nor the Sun SPOT library supports beacon-enabled mode, this paper focuses on the non-beacon-enabled mode. For simplicity's sake, in what follows we refer to the non-beacon-enabled mode of the IEEE 802.15.4 MAC layer protocol as 'the MAC layer'.

The IEEE 802.15.4 standard specifies four different MAC frame-types: beacon, management, data and acknowledgement frames. For this discussion only data and acknowledgement frames are relevant, since management frames are used in neither TinyOS nor the Sun SPOT radio stack and beacon frames are only relevant when using beacon-enabled mode. Data frames are used to transfer higher-level data between nodes. The sender of a data frame may request that frame to be acknowledged. The receiver is then required to send an acknowledgement frame exactly 12 symbol periods (192 μs) after receiving the data frame. This interval is the maximum time allowed for changing the radio between RX- and TX-modes.

The MAC Layer divides all nodes operating on the same channel into multiple Personal Area Networks (PANs). Each of these PANs is identified by a unique identifier. Although this separation does not prevent communication between nodes of different PANs, this mechanism does allow multiple networks to operate independently on the same channel. More importantly, the communication between nodes of the same PAN may be optimized if a so called PAN-Coordinator is present. In that case a node may request its coordinator for a temporary, PAN-specific, 16-bit short address. When granted, this address is then used for PAN-local communication instead of the node's unique 64-bit extended address. By using this mechanism, the overhead on local communication can thus be largely reduced.

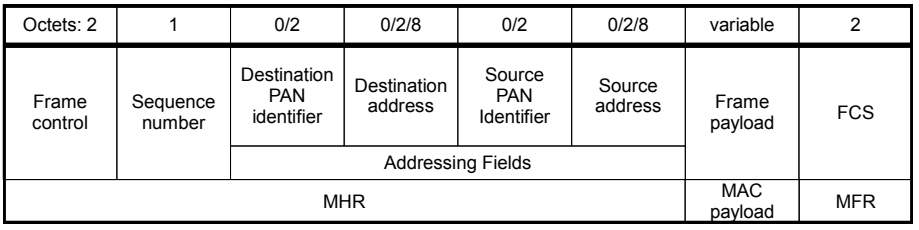

Fig. 1. The IEEE 802.15.4 MAC Frame format. Reproduced from [13].

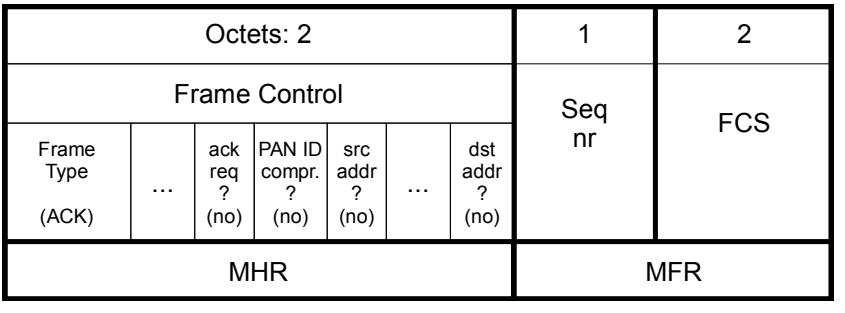

Fig. 2. The IEEE 802.15.4 Ack-frame format

We now explain the IEEE 802.15.4 frame format shown in figure 1. This frame format is used by both TinyOS and the Sun SPOT library. Each MAC frame can contain up to 127 bytes and consists of a MAC Header (MHR), the MAC Payload and a MAC Footer (MFR) containing a 16-bit frame check sequence. The MHR contains a 16-bit frame control field, followed by a 1-byte sequence number and the addressing fields. The frame control field contains the following information relevant to this paper:

- **Frame type**: the type of the frame being sent.
- **Ack. Request**: used to signal to the receiver that the frame should be acknowledged.
- **PAN ID Compression**: if set, the packet is being sent between nodes with the same PAN id and the source PAN id is not present.
- **Dest. addressing mode**: specifies whether the destination address is 16 bit long, 64 bit long, or not present in the frame.
- **Src. addressing mode**: specifies whether the source address is 16 bit long, 64 bit long, or not present in the frame.

The address fields of the MHR consist of the PAN-id and address of the source and destination node. Each of these fields may, depending on the frame type, be absent from the MHR. Data frames being sent between nodes of the same PAN-id, for instance, do not specify the destination PAN-id field. Likewise,

acknowledgement frames, as illustrated by figure 2 contain no addressing information at all. They only contain the sequence number of the data frame being acknowledged. As a result the length of the MHR is dependant on which fields are present and the length of the source and destination address.

For more information about the IEEE 802.15.4 protocol, we refer to [12].

3.2 LowPAN

In order to integrate sensor networks with other networking technologies, a general network layer such as IPv6 is required. Unfortunately this protocol cannot be used directly on top of IEEE 802.15.4 since, amongst others, IPv6 packets are too large to fit in a single IEEE 802.15.4 MAC frame (at most 114 bytes). To resolve this problem, the IETF specifies an intermediate layer that provides the needed services to support IPv6 on IEEE 802.15.4-based sensor networks [10]. This layer is commonly referred to as the LowPAN layer (or 6LowPAN when talking about IPv6).

The first service provided by this LowPAN layer is fragmentation. As IPv6 packets may contain up to 1280 bytes and IEEE 802.15.4 frames only contain 127 bytes, fragmentation and reassembly is required to transfer IPv6 packets between sensor nodes. Furthermore, the LowPAN specification also requires that all nodes in a single PAN should be seen by IPv6 as being on the same network-link. As a result the LowPAN layer also provides meshing and multihop broadcasting in order to manage the routing of packets between nodes of the same PAN.

Furthermore, the LowPAN layer can also perform IPv6 header compression. Due to its limited relevance to this paper, we refer to [10] for more information about IPv6 header compression.

In order to ensure extensibility, the LowPAN specification does not define a single 'LowPAN-header' but instead defines a separate header for each provided service. In order to distinguish between headers of different services, a 1-byte dispatch- or type-value is defined for each header. When a LowPAN frame is sent, the relevant headers are created, prepended with their dispatch-byte and then stored in a fixed order at the start of the payload of an IEEE 802.15.4 MAC frame.

An example of such a frame is provided in figure 3. Not all headers need to be present. If, for instance, an IPv6 packet is small enough to fit into a single LowPAN frame no fragmentation header is used.

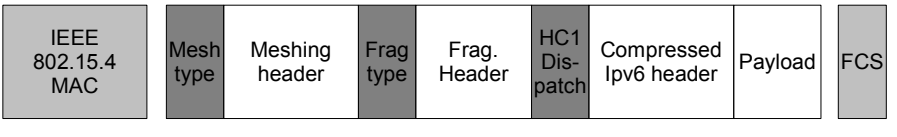

Fig. 3. Example LowPAN frame

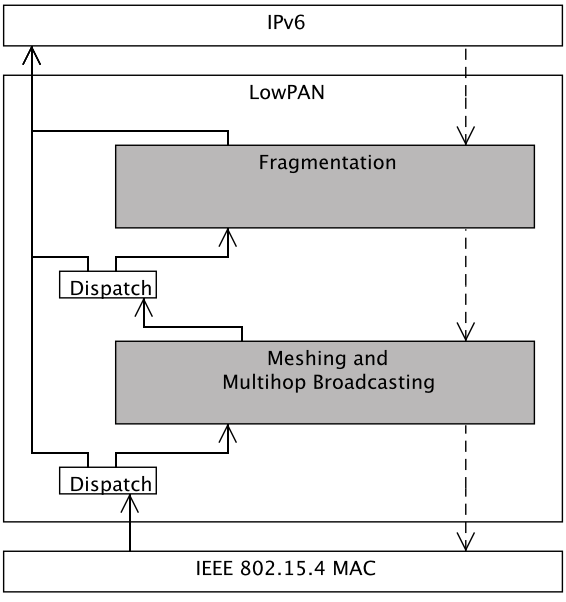

Fig. 4. The LowPAN layer architecture

Upon packet reception, the LowPAN headers are processed by the appropriate services in the same order they were stored. Since each service may decide that no further headers are to be processed, these services may be regarded as separate 'sublayers' inside the LowPAN layer.

Figure 4 shows the resulting architecture and the flow of incoming and outgoing packets. While separate headers exist for the meshing and for the multihop broadcasting service, the broadcasting header may only appear if a meshing header is present. We therefore regard these services as being a single layer.

As explained above, each service is assigned a separate dispatch value. While some of these values are already in use to support the current LowPAN services, many values remain unused to allow future extensions to the LowPAN layer. In order to allow non LowPAN-enabled nodes to coexist with LowPAN implementing nodes, a range of dispatch-values has been reserved. These special 'Not a LowPAN' or NALP dispatch-values indicate that the frame should be discarded by the LowPAN layer upon packet reception.

4 Radio Stack Compatibility

We now discuss the radio stacks of both the TinyOS and the Sun SPOT platform. These platforms both based their radio stacks on the IEEE 802.15.4 standard and the LowPAN specification. Unfortunately both do not provide fully compliant implementations. Consequently, their radio stacks are not compatible with each other. The resulting issues manifest both at the MAC and the LowPAN layer.

While these issues could be resolved by providing compliant implementations of the IEEE 802.15.4 MAC protocol and the LowPAN specification for both platforms, this solution would require the radio stack of both the Sun SPOT and the TinyOS platform to be altered. Sun SPOTs are however best suited for rapid prototyping of sensor network applications. The main application of this paper therefore lies in integrating SunSPOTs with existing TinyOS-based networks. Altering the TinyOS radio stack would thus require the sensor network to be completely redeployed. A more viable solution is therefore to modify the Sun SPOT radio stack in order to provide compatibility with the TinyOS radio stack.

We introduce the TinySPOTComm project. This project consists of a number of modifications to the Sun SPOT stack, that allow for radio communication with an unaltered TinyOS stack. Furthermore the TinySPOTComm stack remains fully compatible with the default Sun SPOT stack. The changes made by the TinySPOTComm project, as discussed below, are based on version 4.0 (blue) of the Sun SPOT library and TinyOS version 2.1.0. Since hardware-compatibility between the two investigated platforms is one of the key assumptions in our research, the TelosB TinyOS-mote was used as it is based on the same RF-chip as the Sun SPOT platform, namely the TI CC2420 RF Transceiver.

4.1 MAC Layer

As mentioned above, the radio stacks of Sun SPOTs and TinyOS-motes are both based on the IEEE 802.15.4 specification for wireless sensor networks. Unfortunately, this standard is generally not fully implemented. This is also the case for TinyOS and the Sun SPOT library. Both provide only partial implementations, which are not compatible with each other. The following issues were identified:

16- vs. 64-bit addressing. As mentioned in section 3, the IEEE 802.15.4 standard provides two different addressing modes. The Sun SPOT radio stack uses the 64-bit extended addresses, while in TinyOS only 16-bit addresses are used, without the required PAN Coordinator. Since no 64-bit to 16-bit translation is being performed, this issue prevents communication between Sun SPOTs and TinyOS-motes. This issue may not be resolved by the use of a PAN Coordinator since TinyOS is to remain unaltered and lacks the functionality required to communicate with one. Consequently the most viable solution is to build support for 16-bit addressing into the Sun SPOT stack. In order to maintain compatibility with unmodified Sun SPOTs, the required changes have to be as least intrusive as possible. The usage of short addresses is therefore as much as possible hidden from both the MAC layer and the rest of the radio stack. For this purpose a two-way conversion between short and extended addresses is used. Since the Sun SPOT radio stack implementation makes use of a separate 'RadioPacket'-class to perform all packet-related operations, this conversion has been inserted into this class. As a result the rest of the Sun SPOT stack is largely unaware of the existence of 16-bit addressed hosts. By default, the conversion is implemented by assuming a unique 48 bit prefix is shared between the extended address of

all Sun SPOTs. In reality each extended address is comprised of a 32 bit vendor specific prefix followed by a 32 bit device identifier. As all Sun SPOTs are manufactured by Sun, the 32 bit vendor specific prefix is shared between all Sun SPOT devices. The device identifier is uniquely coupled to each individual Sun SPOT. As a result, this assumption fails to hold when the extended addresses of two Sun SPOTs differ within the 16 most significant bits of the device identifier. In order to circumvent this issue, the TinySPOTComm stack allows the address conversion to be redefined by extending the 'IEEEAddressHash' class.

Secondly, the configuration of the address-recognition had to be altered. In the unmodified Sun SPOT stack, the CC2420 chip is configured to only accept broadcast and matching 64-bit addressed unicast frames. Since TinyOS only uses 16-bit addresses, the radio was configured to also accept frames addressed to the 16-bit representation of the Sun SPOTs extended address.

Software versus Hardware Acknowledgements. According to the IEEE 802.15.4 standard, frames with the 'ACK' bit set, should be acknowledged after exactly 12 symbol periods (192 μs)

The Sun SPOT stack implements this behavior by using the automatic acknowledgement feature provided by the radio chip. Unlike the Sun SPOT library, TinyOS handles ACKs, by default, in software rather than in hardware. This is due to the fact that in TinyOS a received packet is not guaranteed to be transferred from the CC2420 chip to the micro controller [15]. Consequently, the use of hardware-ACKs may result in false acknowledgements. The TinyOS developers therefore chose to handle ACKs in software. Because of this, ACKs sent from a TinyOS-powered mote, are sent with a delay that is too large to be accepted by Sun SPOTs. By increasing the Sun SPOTs ACK timeout from 864μs to 992μs, TinyOS-originated ACKs are accepted by Sun SPOTs. It should be noted that increasing the ACK timeout is not entirely without risk. Since in the IEEE 802.15.4 standard, acknowledgements do not contain the address of the host which sent the original packet, a packet and its acknowledgement are only related by their respective sequence number. Consequently, a false acknowledgement will occur if an unrelated packet with the same sequence number is acknowledged while the sender of the original packet waits for an acknowledgement. The chance of a false acknowledgement is proportional to the ACK timeout, which should therefore be kept as small as possible. In a TinyOS network however, the proposed increase should not pose a problem as TinyOS requires an ACK timeout of 8000μs. This relatively large timeout value is the result of restrictions on the hardware level. On certain hardware platforms, such as the Crossbow TelosB mote, the radio chip shares its bus to the microcontroller with other peripherals. By increasing the ACK timeout value, it is no longer necessary for the radio chip to keep the bus occupied while waiting for an acknowledgement. As a result the other peripherals on the bus (persistent storage in the case of the TelosB mote) may be accessed by the microcontroller while the radio is busy.

4.2 Network Layer

As mentioned above, the Sun SPOT stack heavily relies on the LowPAN specification to provide routing, meshing and fragmentation. Although most LowPAN functionality is implemented, the Sun SPOT library does not support IPv6 and instead uses the IEEE 802.15.4 extended addresses to identify nodes in the network. Based on the 'multiple header' principle of the LowPAN specification, the Sun SPOT library provides an extensible LowPAN implementation. It allows so called 'ProtocolHandlers' to be coupled to specific dispatch-bytes. Upon packet reception, a packet is relayed to the ProtocolHandler associated with the dispatch-byte of the packet. This mechanism is, for instance, used by the radiogram (the equivalent of UDP) and radiostream (the equivalent of TCP) protocols to allow Sun SPOT applications to access the network. These ProtocolHandlers are only used for protocols which are not defined in the LowPAN specification. If meshing, multihop broadcasting or fragmentation headers are present, these are directly handled by the LowPAN layer itself.

In contrast, TinyOS does not seem to provide any network layer functionality. Instead the payload of each TinyOS-originated data packet commences, by default, with a byte containing '0x3f'. LowPAN implementing nodes interpret this byte as a 'NALP' value and consequently discard TinyOS-originated packets. The second byte is used for multiplexing, as to allow multiple data-flows. Since

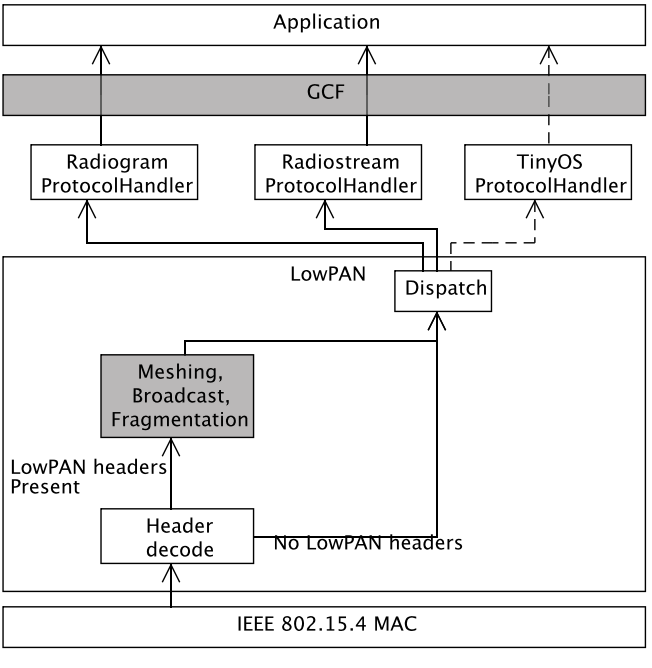

Fig. 5. The LowPAN Layer in the Sun SPOT architecture

```
try {
  DatagramConnection conn =
    (DatagramConnection) Connector.open(
      "radiogram://0014.4F01.0000.116B:65");
 Datagram dg = conn.newDatagram(22);
  dg.writeChars("Hello World");
  conn.send(dg);
} catch (IOException e) {
...
}
```

Fig. 6. This code sends the string "Hello World" to SunSPOT '0014.4F01.0000.116B' on port 65, using radiograms

```
try {
  DatagramConnection conn =
    (DatagramConnection) Connector.open(
      "tinyos://0014.4F01.0000.0001:65");
  Datagram dg = conn.newDatagram(22);
  dg.writeChars("Hello World");
  conn.send(dg);
} catch (IOException e) {
...
}
```

Fig. 7. This code sends the string "Hello World" to the TinyOS-mote with short address '0001', using 65 as multiplexing value

no alterations are to be made to the TinyOS radio stack, the best solution to this problem would be to bypass the LowPAN layer for all TinyOS-originated packets. Fortunately, the Sun SPOT LowPAN implementation allows for this behavior to be implemented with only minor changes.

Figure 5 shows the resulting architecture. Since packets starting with a 'NALP' byte do not contain any meshing or fragmentation headers, the LowPAN implementation attempts to dispatch the packet to the corresponding ProtocolHandler. Since, by default, no ProtocolHandlers are registered to handle NALP values, the packet is discarded (as required by [10]). TinyOS originated packets may therefore be intercepted by registering a specialized 'TinyOSProtocolHandler' to the dispatch value 0x3f. This ProtocolHandler is also responsible for sending packets to TinyOS nodes. Since the LowPAN layer normally uses 64-bit addressing, this ProtocolHandler is responsible for creating RadioPackets using 16-bit addressing. The created packet is then passed directly to the MAC Layer.

By extending this TinyOSProtocolHandler, new protocols being developed on top of the TinyOS-stack may be ported to the Sun SPOT platform. The current implementation of the TinyOSProtocolHandler is used to provide application

level compatibility with the default TinyOS stack. This is done by translating TinyOS' multiplexing byte into a port number and by adding a new 'tinyos://' handler to the Sun SPOTs Generic Connection Framework (GCF) [14]. Using this GCF handler, communicating with TinyOS-motes is done similar to regular communication between Sun SPOTs. An example of this is provided in figures 6 and 7.

5 Performance

5.1 Setup

In order to establish the performance of the TinySPOTComm stack (the Sun SPOT stack patched with the changes proposed in the previous section), the single-hop delay and throughput between Sun SPOTs and TinyOS motes were measured. The delay was measured by performing a ping-test from a client to a server node and recording the round-trip time. The average round trip time was then calculated over 200 test runs. To measure the throughput, unicast packets with the ACK-request flag set, were continuously sent from the client node to the server node. After the test-run had completed, throughput was derived from the number of packets received by the server. An average value was calculated over seven test-runs of 60 seconds. These round trip time and throughput tests were run using only TinyOS motes, only Sun SPOTs or between a Sun SPOT and a TinyOS mote, with the TinyOS mote acting as server and the Sun SPOT acting as client and vice versa.

Furthermore, the network performance of the TinySPOTComm stack was compared to that of the regular Sun SPOT stack. For this purpose the delay and throughput tests were also performed using the TinyOS-incompatible radiogram protocol provided by the Sun SPOT library. These tests were then run between two Sun SPOTs equipped with the default Sun SPOT radio stack and between two Sun SPOTs using TinySPOTComm stack. Unfortunately, the use of radiograms limits the number of bytes that may be sent in an IEEE 802.15.4 frame to 123 instead of 127 bytes. This is due to the fact that the Sun SPOT library is overly cautious when calculating the number of available payload bytes. Firstly the Sun SPOT library reserves two bytes in the MAC header to store the source PAN id, regardless of whether that field is present or not. Secondly, two extra bytes are reserved by the LowPAN implementation to allow for the use of extended dispatch-fields in a LowPAN packet. Given the limited number of ProtocolHandlers registered with the LowPAN layer, these large dispatch-fields remain currently unused. In order to compensate for this inequality, all throughput tests were performed using 123-byte frames.

Due to it's limited relevance to this paper, the energy consumption of the Sun SPOT and TinyOS radio stacks was not measured. The TinySPOTComm project does not alter the TinyOS radio stack and therefore does not affect it's energy consumption. Furthermore, Sun SPOTs only have a battery lifetime of a few hours to a few days at most and are best suited for rapid prototyping. As a result 'prototyped' sensor network applications usually only need to run

for a few hours or are deployed on a Sun SPOT connected to an external power source.

All measurements were obtained using Crossbow TelosB motes equipped with TinyOS 2.1.0 and Sun SPOTs running either version 4.0 (blue) of the Sun SPOT library or our custom TinySPOTComm stack.

5.2 Round Trip Time

Figure 8 displays the average and the standard deviation of the round trip times measured between TinyOS motes and Sun SPOTs. The smallest average is measured when two TinyOS motes are used. When one TinyOS mote is replaced with a Sun SPOT, the round trip time increases, and it is the largest when only Sun SPOTs are used. This increase in round trip time is to be expected since the Sun SPOT library provides a more advanced network layer than TinyOS. Furthermore, different threading models are used by TinyOS and the Sun SPOT JVM. As a result, in TinyOS a received packet is handed almost directly to the application while the Sun SPOT radio stack requires a received packet to be handled by several different threads before it is delivered to the application. This may also account for the increase in delay. The tests show however that the round trip time is increased by no more than 10% to 15%. It is therefore unlikely to cause any major issues.

5.3 Throughput

The average and standard deviation of the throughputs that are reached when sending data between TinyOS motes and Sun SPOTs is displayed in figure 9. In

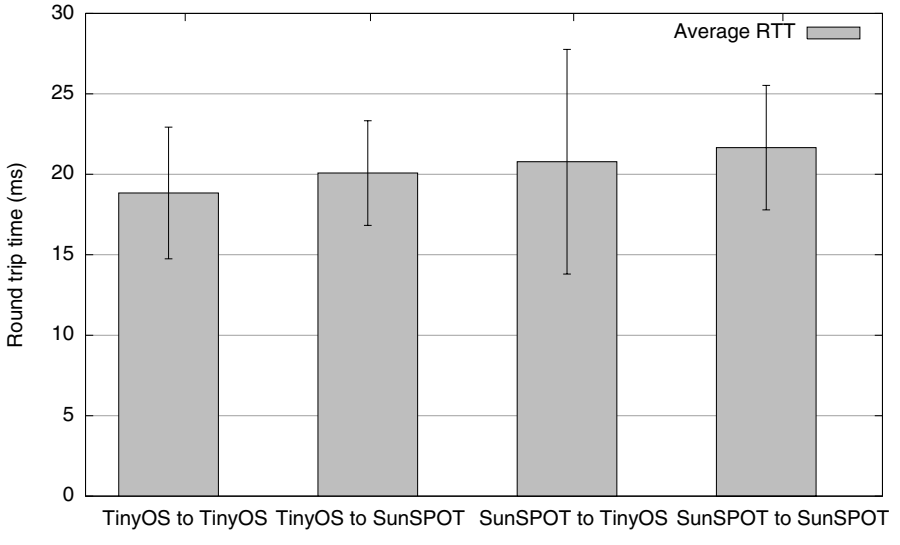

Fig. 8. Average Round Trip Times

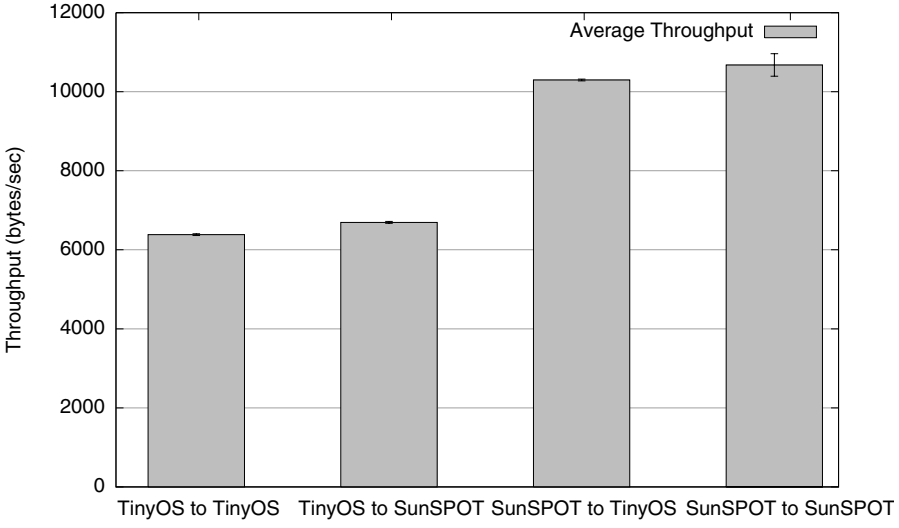

Fig. 9. Average Throughputs

contrast with the round trip time, the throughput is maximal when measured between two Sun SPOTs and it is minimal when only TinyOS motes are used. This is probably due to the fact that the used TelosB motes have significantly lower hardware specifications than Sun SPOTs. Interestingly, this bottle-neck is most prominent when the TelosB mote is used as a client node. When a Sun SPOT is used to unicast packets to a TelosB mote, the throughput is only slightly smaller than if two Sun SPOTs are used. Secondly, a sudden increase in the standard deviation can be observed when only Sun SPOTs are used. This is caused by the garbage collector of the Squawk VM interrupting the server node test application during the performance test. This phenomenon is not visible in the other test setups, since it only affects the Sun SPOT server node and only when the throughput is large enough. When the garbage collector is explicitly invoked at the end of each test-run, the standard deviation is equally large as in the other test setups, while the average throughput remains unaltered. Do note that the throughputs displayed in figure 9 do not approach the theoretically possible maximum throughput of the IEEE 802.15.4 standard (250kbit/s). This is due to the fact that power consumption is more important than throughput in the application domain targeted by the IEEE 802.15.4 standard and the Sun SPOT and TinyOS platforms.

5.4 Impact on Sun SPOT Performance

Figures 10 and 11 show the results of the radiogram based delay and throughput tests. From these results, it is clear that the network performance of the

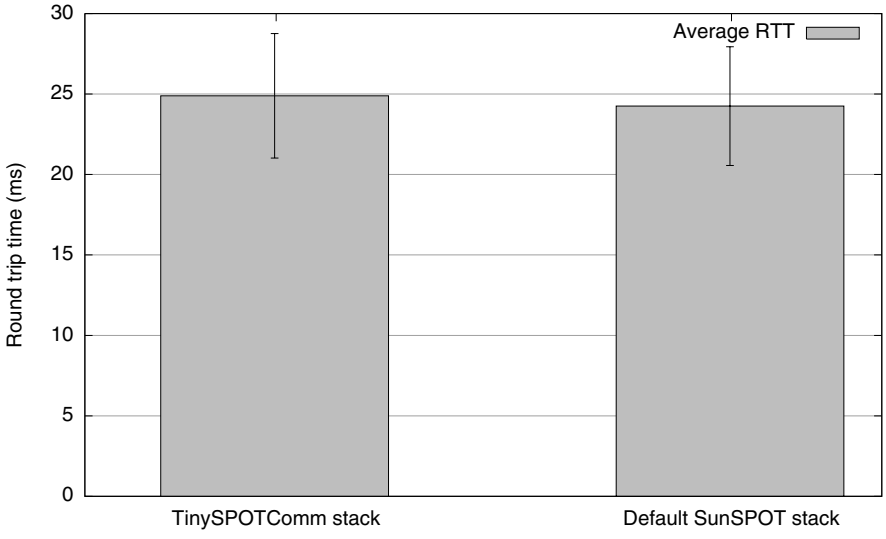

Fig. 10. Average Round Trip Times when using radiograms

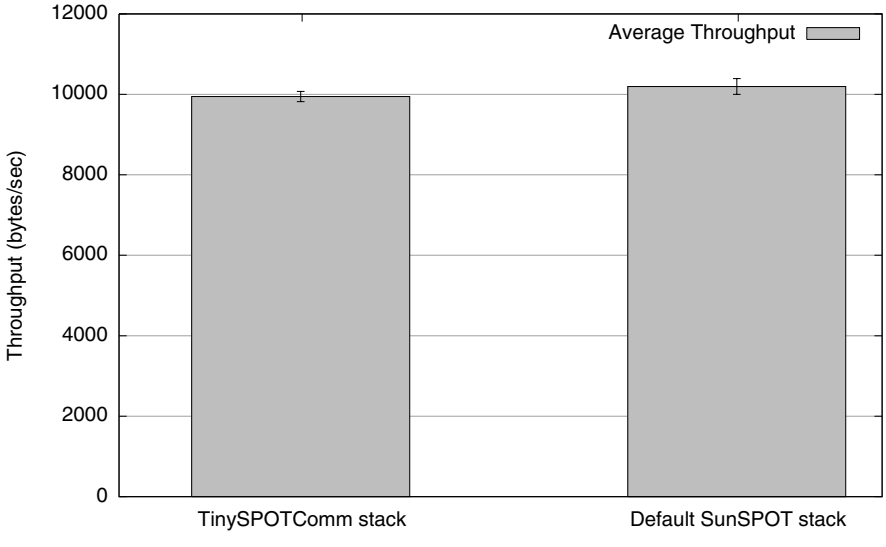

Fig. 11. Average throughput when using radiograms

TinySPOTComm stack is only marginally smaller than that of the default Sun SPOT stack. The use of the TinySPOTComm stack only increased the round trip time by 1.03%. The original Sun SPOT stack only achieves a throughput that is 1.02% larger than the TinySPOTComm stack. As with the Sun SPOT to

Sun SPOT throughput test in section 5.3, the relatively large standard deviation of the measured throughputs is caused by the garbage collector.

6 Conclusion and Future Work

Wireless sensor networks are becoming increasingly popular. Despite hardware compatibility, sensor nodes which are programmed using different frameworks are often incapable to communicate with each other. We have investigated this problem by focussing on the radio stacks of the Sun SPOT and TinyOS platform. The TinySPOTComm project introduced in this paper, provides a set of modifications to the Sun SPOT stack, that allow for communication with TinyOS-motes. The TinySPOTComm stack remains fully compatible with the default Sun SPOT radio stack and we have shown that its network performance is only marginally smaller than that of the default radio stack. While the TinySPOTComm stack does allow for communication between Sun SPOTs and TinyOS motes, the use of this modified radio stack would become unnecessary if both the Sun SPOT and the TinyOS radio stack were to be made fully IEEE 802.15.4-compliant. The Sun SPOT platform already provides a respectable but not yet fully compliant implementation and a working group [16] has been founded to provide an IEEE 802.15.4 compliant radio stack for TinyOS.

The TinySPOTComm stack may be improved by providing compatibility with blip [17], a LowPAN implementation for the TinyOS platform. Unfortunately, it is impossible to extend the functionality of the TinyOSProtocolHandler in order to gain this compatibility as the blip stack both lacks both the 'NALP' byte and the multiplexing byte used by the standard TinyOS stack. A possible solution would be to inject IPv6 functionality into the existing LowPAN implementation. The TinySPOTComm stack may be further improved by analyzing and reducing the effect on the round trip time of using multiple threads to handle incoming packets.

References

1. The TinyOS community website, `http://www.tinyos.net`
2. Simon, D., Fuentes, C., Cleal, D., Daniels, J., White, D.: Java(TM) on the bare metal of wireless sensor devices: the squawk Java virtual machine. In: Proceedings of the 2nd international conference on Virtual execution environments, pp. 78–88 (2006)
3. Levis, P., Culler, D.: Maté: a tiny virtual machine for sensor networks. ACM SIGOPS Operating Systems Review (2002)
4. Harbaum, T.: The NanoVM - Java for the AVR (2005), `http://www.harbaum.org/till/nanovm/`
5. Saballus, B., Eickhold, J., Fuhrmann, T.: Towards a Distributed Java VM in Sensor Networks using Scalable Source Routing, `http://i30www.ira.uka.de/research/documents/p2p/2007/saballus07distributed.pdf`

6. Koshy, J., Pandey, R.: VMSTAR: synthesizing scalable runtime environments for sensor networks. In: SenSys 2005: Proceedings of the 3rd international conference on Embedded networked sensor systems, pp. 243–254 (2005)
7. Shaylor, N., Simon, D.N., Bush, W.R.: A java virtual machine architecture for very small devices. SIGPLAN Not. 38(7), 34–41 (2003)
8. Beudel, J.: Metrics for Sensor Network Platforms. In: Proceedings of REALWSN 2006 (2006)
9. Datasheet for the Crossbow TelosB mote, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/TelosB_Datasheet.pdf
10. Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks (2007)
11. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A Survey On Sensor Networks. Communications Magazine, 102–114 (August 2002)
12. IEEE Computer Society: IEEE Std 802.15.4-2006 (September 2006)
13. IEEE Computer Society: IEEE Std 802.15.4-2003 (October 2003)
14. Enrique Ortiz, C.: The Generic Connection Framework (2003), http://developers.sun.com/mobility/midp/articles/genericframework/
15. Moss, D., Hui, J., Levis, P., Choi Il, J.: CC2420 Radio Stack (2007), http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep126.html
16. The TinyOS IEEE 802.15.4 Working Group, http://www.tinyos.net/scoop/special/working_group_tinyos_154
17. The Berkeley IP Information project, http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip