An Adaptive Communication Architecture for Wireless Sensor Networks

Adam Dunkels
adam@sics.se

Fredrik Österlind
fros@sics.se

Zhitao He
zhitao@sics.se

Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden

Abstract

As sensor networks move towards increasing heterogeneity, the number of link layers, MAC protocols, and underlying transportation mechanisms increases. System developers must adapt their applications and systems to accommodate a wide range of underlying protocols and mechanisms. However, existing communication architectures for sensor networks are not designed for this heterogeneity and therefore the system developer must redevelop their systems for each underlying communication protocol or mechanism. To remedy this situation, we present a communication architecture that adapts to a wide range of underlying communication mechanisms, from the MAC layer to the transport layer, without requiring any changes to applications or protocols. We show that the architecture is expressive enough to accommodate typical sensor network protocols. Measurements show that the increase in execution time over a non-adaptive architecture is small.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Network Operating Systems*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Wireless sensor networks, Protocol stack

1 Introduction

As sensor networks move towards increasing heterogeneity [19], assumptions on how sensor network protocols operate are challenged. For instance, if an application is to run over multiple link layer technologies, applications and protocols cannot rely on the existence of specific link layer mechanisms such as link layer retransmissions. This prompts us to rethink existing work on communication architectures for sensor networks.

Having a common sensor network architecture has many benefits in terms of interoperability and code reuse. The sensor network community has recently investigated several different sensor network architectures, such as the SP architecture by Polastre et al. [30] and the modular network architecture by Cheng et al. [16]. However, neither SP nor the architecture by Cheng et al. solves the problem of how to adapt the protocols running on top of the architectures so that they can communicate with the lower layer protocols.

SP does not specify any protocol headers and can therefore be adapted to many underlying protocols. By not specifying protocol headers, SP leaves the problem of adapting to the network protocols to the application programmer. The modular network layer by Cheng et al. partitions network layer functionality into a set of abstract modules. Each module defines its own protocol headers. To add additional protocols, the application programmer must define clear-cut module boundaries and specify packet sub-headers.

In this paper, we present Chameleon, a communication architecture for sensor networks. The Chameleon architecture consists of two parts: the Rime communication stack and a set of packet transformation modules. The Chameleon architecture is designed to be able to adapt to a variety of different underlying protocols and mechanisms while being expressive enough to accommodate typical sensor network protocols.

One of the main problems of specifying an interoperable communication architecture is finding a universal header format [16]. Such a header format must be both expressive enough to encompass all communication patterns supported by the architecture and flexible enough to allow for future expansion of the architecture. For a sensor network architecture, the problem is even more challenging, as the header format must be small enough to be efficient over low-power radio links with small maximum packet sizes.

Chameleon takes a drastically different approach to the problem of finding a common packet header format: Chameleon does not define any packet headers at all. Rather, Chameleon uses *packet attributes*, an abstract representation of the information usually found in packet headers.

Packet headers are produced by separate header transformation modules that transform application data and packet attributes into packets with headers and payload. By using

SenSys'07, November 6–9, 2007, Sydney, Australia.

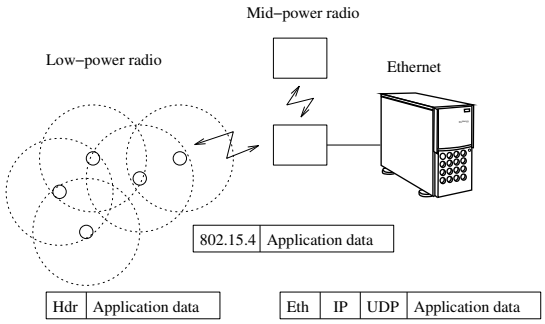

Figure 1. A tiered architecture running multiple link layer protocols; a low-power radio (e.g. CC1100), a mid-power radio (e.g. 802.15.4), and Ethernet.

different Chameleon modules, it is possible to create packets that conform to any given packet header specification. For example, Figure 1 shows a network with three different underlying network protocols: a low-power radio such as the CC1100, a mid-power 802.15.4 radio, and TCP/IP over Ethernet. This network uses three Chameleon modules: one that produces bit-packed compressed link-layer headers for the low-power radio, one that produces 802.15.4 MAC-layer packets, and one that produces UDP/IP transport-layer packets. Unlike other communication architectures, the application running on top of Chameleon does not need to be changed to make it run over those different underlying communication mechanisms. Packet header transformations are, however, not enough to mimic the operations of such protocols. In some cases, a Chameleon header transformation module must implement parts of the protocol logic of the protocol it mimics. For example, the Chameleon header transformation module that produces UDP/IP packets must implement the ARP protocol to resolve IP addresses to Ethernet MAC addresses.

The second part of the Chameleon architecture is the Rime communication stack. Rime provides a set of basic communication primitives ranging from best-effort single-hop broadcast and best-effort single-hop unicast, to best-effort network flooding and hop-by-hop reliable multi-hop unicast. We have selected the Rime primitives based on analysis of the communication requirements of typical sensor network protocols.

We have implemented the Chameleon architecture under the Contiki operating system [13] and evaluate it on Tmote Sky motes [31].

We make three main contributions in this paper. First, we present a solution to the cross-layer information-sharing problem of a layered communication stack by separating the protocol logic from the details of the packet headers. We show that the use of packet attributes instead of packet headers allows applications to access low-level information without violating the layering principle, while execution-time performance is on par with a traditional packet header-based implementation. Second, we present the Rime protocol stack, a lightweight layered communication architecture for sensor networks that reduces the complexity of implementations of network protocols. We show that the communication primitives in the stack map onto typical sensor network protocols: data dissemination, data collection, and mesh routing protocols. Third, we show that the use of packet attributes makes it possible to adapt the output from the protocol stack to other communication protocols such as link and MAC layer protocols and TCP/IP.

The rest of this paper is structured as follows. We present the background to our communication architecture in Section 2. In Section 3 we present the high-level design of the Chameleon architecture. Section 4 presents the Rime protocol stack and Section 5 the Chameleon packet transformation module. The implementation of the architecture is discussed in Section 6. We implement a set of sensor network protocols in Section 7 and Chameleon modules in Section 8. We evaluate the architecture in Section 9. We review related work in Section 10 and conclude the paper in Section 11.

2 Background

An adaptive communication architecture for sensor networks must be able to support both typical sensor network protocols running on top of the communication architecture, and MAC and link layer protocols on which the architecture runs. In this section, we review the high-level architectural issues with communication stacks and the requirements from sensor network protocols as well as underlying MAC and link layer protocols and standards.

2.1 The Narrow Waist

One of the primary design challenges of a network architecture is where to place the narrow waist of the architecture; the fixed point around which the rest of the network architecture grows. The narrow waist allows for different protocols running above the waist and different technologies running below it.

Previous work in communication architectures for sensor networks [10, 16, 30] have placed the narrow waist of the sensor network protocol stack below the network layer and above the link layer. With this placement of the narrow waist, the primary communication primitive is best-effort single-hop broadcast. This placement of the narrow waist also allows for other mechanisms, such as congestion control, to be efficiently implemented with the architecture [6].

Our architecture corroborates the view that the narrow waist of a sensor network architecture should be single-hop best-effort broadcast. Additionally, we show that richer communication primitives, such as multi-hop communication, can be naturally incorporated into such a sensor network communication architecture, while keeping the narrow waist located below the network layer. Also, our architecture shows this placement of the narrow waist does not restrict the use of complex underlying protocols, nor of cross-layer information sharing.

2.2 Address-free Protocols

One important class of sensor network protocols is the address free protocols. Address-free protocols are protocols that do not explicitly use node addresses. Perhaps the most commonly used example of address-free protocols are data dissemination protocols [21, 25]. In a data dissemination protocol, data is sent from a source node to all other nodes in the network. Neither the source, nor any other node in the

network, need to know the address of the receiving nodes. Instead, the nodes can use broadcast to send data to all their single-hop neighbors, which in turn can rebroadcast the data to all their single-hop neighbors. The nodes must ensure, however, that the network is not overloaded and therefore must engage in some form of congestion control scheme.

2.3 Name-based Protocols

While address-free protocols play an important role in sensor networks, name-based protocols are also frequently used. In name-based protocols, nodes are explicitly named, usually with their node identification address. The perhaps most prominent example of a name-based protocol is unicast multi-hop routing, where data packets are sent from one specific node to another specific node in the network. The sender knows the address of the receiving node, and intermediate nodes on the route between sender and receiver know the addresses of the receiver.

Data collection protocols are a hybrid form of address-free and name-based protocols. In a data collection protocol, the participating nodes send data to one or more sink nodes in the network. The data packets are forwarded in a multi-hop fashion towards any of the sink nodes. The participating nodes do not need to know the address of the sink node that will eventually receive the data. However, the nodes typically must know the address of their single-hop neighbors. To send data towards a sink node, a node sends its data to the single-hop neighbor that is, in some sense, closest to a sink. Thus the protocol is address-free in the multi-hop sense, but not in the single-hop sense.

2.4 Neighborhood Abstractions

Many sensor network algorithms operate on collections of nodes that are physically or logically close to each other. A number of programming abstractions for developing such algorithms have been constructed [18, 26].

While the neighborhood abstractions typically hide the communication behind a layer of abstraction, the nodes that participate in a neighborhood abstraction typically interchange messages with each other. Some messages are sent directly to a specified neighbor, whereas other messages are broadcast to all neighbors. Others need scoped flooding to reach all n-hop neighbors [18]. Similarly, some messages are of higher importance than others and may therefore need to be sent using a reliable communication channel, while others can be sent using best-effort messages. A communication architecture for sensor networks must handle such communication patterns.

2.5 MAC Protocols

One of the primary purposes of sensor network MAC protocols is to reduce the energy consumption of the sensor nodes. Radio communication is typically one of the most energy consuming activities [15] and the reception energy is often as high as the transmission energy. The MAC protocol must therefore turn the radio off as often as possible, while being awake long enough to allow for communication with other sensor nodes.

Many MAC protocols for sensor networks exist, both time-slotted TDMA protocols [32] and contention-based CSMA protocols [1]. In TDMA-based protocols, each node is given a time slot in a local time schedule. A node can send only during its own time slot. CSMA-based protocols use channel sampling to ensure that only one sender is active at any given time. Power-optimization techniques such as low-power listening [29] and strobed preamble [4] is used to keep the energy consumption down while remaining responsive. With such mechanisms, packets typically cannot be instantly transmitted but must be queued before the actual transmission occur.

Sensor network MAC protocols often treat unicast and broadcast traffic differently. Unicast traffic only needs to reach the receiver, and therefore all other nodes can switch their radio off during the packet transmission. Broadcast traffic, however, reaches all local neighbors and all nodes must therefore be awake during the transmission.

Many MAC and link layer protocols support automatic packet acknowledgements for unicast traffic [1]. Packets that should be automatically acknowledged are tagged with a special bit to indicate that they are to be reliably transmitted. If the sender does not receive an acknowledgement, the packet is retransmitted.

Different MAC and link layer protocols use different addressing modes, and may even support multiple addressing modes. One example of such a link layer is 802.15.4, where packets can have either 64-bit addresses or 16-bit addresses. There is also an option to completely turn off addresses in packet headers.

An adaptive sensor network communication architecture must be able to support all of the above mechanisms: packet queuing, different handling of broadcast and unicast packets, automatic packet acknowledgement and retransmission, and different addressing modes.

2.6 Related Standards

The ZigBee Specification is an industry standard for a range of short-range, low data rate control and sensing applications. The ZigBee stack employs a layered architecture that provides end-to-end reliable data transfer on top of IEEE 802.15.4 via its network layer and application support sublayer. ZigBee further specifies how applications can be constructed by requiring a pre-defined application profile along with associated commands, called clusters, for all nodes in the network. ZigBee supports mesh routing protocol based on AODV [28].

The Internet protocol family, TCP/IP, has traditionally been viewed as unsuitable for sensor networks. To make TCP/IP viable for wireless sensor networks, we have previously suggested the use of simplified implementations of the TCP/IP protocol stack [12] and optimizations such as header compression [14]. Recently, these ideas have been picked up both in standardization efforts such as the 6lowpan IETF working group [27] and by industry [8]. For example, to meet the strict energy constraints of sensor networks, 6lowpan nodes do not fully comply with the IPv6 standard and the IPv6 headers are compressed before transmission.

Ideally, an adaptive sensor network architecture should allow for protocols running on top of the architecture to be compatible with such standard protocols.

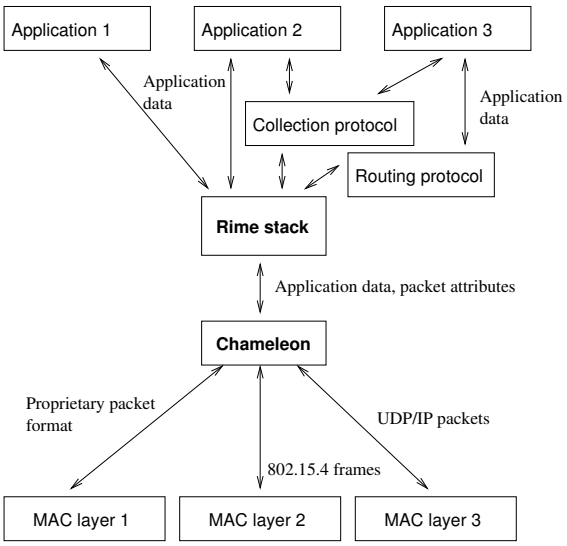

Figure 2. The Chameleon architecture. Applications and network protocols run on top of the Rime stack. The output from Rime is transformed into different underlying protocols by header transformation modules.

3 The Chameleon Architecture

The Chameleon architecture is an adaptive communication architecture for sensor networks. The purpose of the architecture is threefold. First, the architecture is designed to simplify the implementation of sensor network communication protocols. This is done through the use of the Rime protocol stack. Second, the architecture allows for sensor network protocols that are implemented on top of the architecture to take advantage of the features of underlying MAC and link layer protocols. This is done by using packet attributes instead of packet headers. Third, the architecture allows for the packet headers of outgoing packets to be formed independently of the protocols or applications running within the architecture. Separate packet transformation modules handle packet header construction.

The Chameleon architecture draws from previous work on sensor network architecture [10, 16, 30] and is inspired by work in the area of distributed programming [20] and general-purpose network architecture [3, 9].

Figure 2 shows the Chameleon architecture. The architecture contains three parts: the Rime stack, which provides a set of communication primitives to applications running on top of the stack; a set of network protocols running on top of the Rime stack; and the Chameleon header transformation modules, which create packets and packet headers from the output of the Rime stack. Applications run either directly on top of the Rime stack, or on top of communication protocols that run on top of Rime.

The Chameleon header transformation modules can produce either tightly bit-packed packet headers or headers that conform either to specific MAC or link layer protocols, or to other communication protocols. Some header transformation modules also implement parts of the protocol logic of the protocols they mimic.

Applications and protocols pass application data down to the Rime stack. The Rime stack adds packet attributes to the application data before it passes the application data and packet attributes to the underlying Chameleon header transformation module. The header transformation module constructs packet headers from the packet attributes and sends the final packets to the link-level device driver or the MAC layer. The MAC layer can inspect the packet attributes to decide how the packet should be transmitted. For example, broadcast packets may be sent differently from unicast packets, and packets that need single-hop reliability can be sent with link-layer acknowledgements turned on.

3.1 Separation of Protocol Logic and Protocol Headers

The protocol logic in the Rime stack does not deal with low-level details of packet headers such as the placement, structure, and alignment of header fields. Rather, all management of such low-level details is contained in the header transformation modules.

Instead of using packet headers, the Chameleon architecture uses packet attributes. Packet attributes contain the same information that normally is found in packet headers. The packet attribute information is a more abstract representation of the packet header information. Table 3.1 lists the pre-defined packet attributes in the Chameleon architecture. Both applications and lower layer protocols may define additional packet attributes.

The pre-defined packet attributes include the sender and receiver addresses, packet IDs, packet types, the number of times that a packet has been forwarded, as well as feedback information from the lower layers, such as the estimated link quality, and information about radio congestion.

Each packet attribute has a scope. The scope of a packet attribute specifies how far the attribute will follow the packet. Attributes with scope 0 will only follow the packet within the node, attributes with scope 1 will be transmitted in packet headers but will not be forwarded across more than one node, and attributes with scope 2 will follow the packet to the final recipient in case of a multi-hop packet.

3.1.1 Header Field Alignment

The headers in general purpose communication protocols, such as the protocols in the TCP/IP stack, are typically defined so that all header fields are aligned on even byte-boundaries. The reason for this is that many microprocessors cannot access quantities that are not properly aligned.

Protocol designers must ensure that all header fields are properly aligned, and must therefore sometimes insert padding bytes into the packet headers [24]. Low-power radio protocols, however, must reduce their header size to a minimum and therefore in many cases cannot afford to align all header fields.

With Chameleon's packet attributes approach, the protocol implementations do not have to deal with low-level alignment of header fields. Rather, all low-level header alignment details are contained in the header transformation modules.

3.1.2 Byte Ordering

Protocols headers are typically designed to allow for hosts with different byte order to communicate with each other.

Table 1. Pre-defined Chameleon packet attributes. The scope specifies if the attribute terminates at the multi-hop receiver (2), the single-hop receiver (1), in the local node (0).

Attribute	Type	Scope
End-to-end sender	Address	2
End-to-end receiver	Address	2
End-to-end packet type	Integer	2
End-to-end packet ID	Integer	2
Hops	Integer	2
Time to live	Integer	2
Single-hop sender	Address	1
Single-hop receiver	Address	1
Single-hop packet type	Integer	1
Single-hop packet ID	Integer	1
Retransmissions	Integer	1
End-to-end reliable	Integer	0
Single-hop reliable	Integer	0
Maximum retransmissions	Integer	0
Link quality estimate	Integer	0

The most common byte order in communication protocols is the so-called network byte order, which is different from the byte order used in most microprocessors. Thus protocol implementations must explicitly convert all multi-byte header field quantities into network byte order before they are written to a packet header, and conversely convert incoming header fields to host byte order.

By using packet attributes instead of packet headers, the protocol implementation does not need to know what the byte order of the transmitted packets is, but can access the packet attributes in host byte order. The Chameleon header transformation modules convert all multi-byte packet attributes into header fields with network byte order.

3.1.3 Cross-layer Bit-packed Header Fields

Many protocol header fields require only a few bits of information. Typical examples of such fields are flag fields or fields that specify the type of a packet; if the packet is a data packet or an acknowledgment packet. While it is possible to hold single-bit fields in separate byte fields, to reduce the total size of the header many single-bit fields are typically packed into a single byte header field. Reducing the size of the header is particularly important in sensor networks, where packet sizes are small. For example, the radio chip on the Tmote Sky board restricts the size of radio packets to 128 bytes.

Manual bit packing of single-bit header fields have several disadvantages. First, the protocol implementation must be aware that certain fields are single-bit fields and that those header fields must be accessed using bit shifting and Boolean logic expressions. Second, the protocol implementation must be aware that the memory location of a single-bit field may be shared with other single-bit fields. The protocol implementation must therefore make sure that the other single-bit fields are not overwritten when writing to a single-bit field. Third, protocols at different layers cannot pack their single-bit fields into one, but must use different bytes for their single-bit fields even if the total sum of bits in the single-bit fields is smaller than the size of a byte.

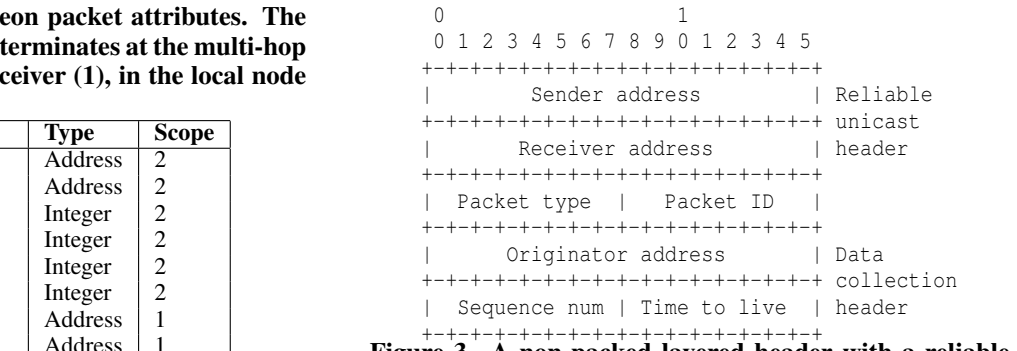

Figure 3. A non-packed layered header with a reliable hop-by-hop header and a data collection header. Due to alignment, the collection tree header must be located at an even location.

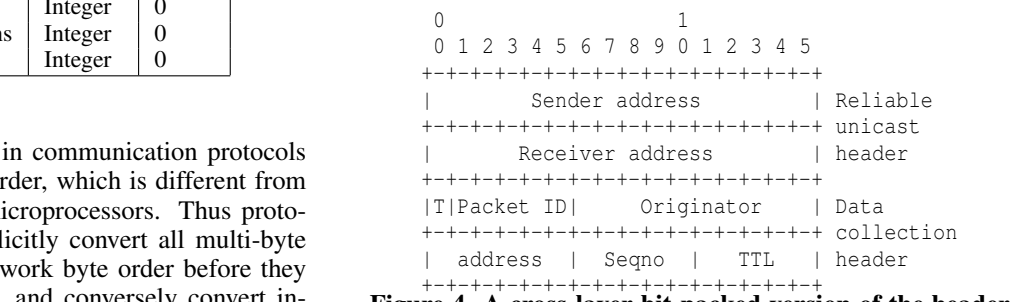

Figure 4. A cross-layer bit-packed version of the header in Figure 3 with 1-bit packet type, 5-bit packet ID and sequence numbers, and a 5-bit time to live field. The packed header is two bytes shorter.

In conventional layered communication architectures, it is in general not possible to pack bit-sized header fields from different layers into a single byte field. Instead, bit-sized header fields must be contained in separate byte-sized fields. An example of such a header is shown in Figure 3.

The packet attributes in the Chameleon architecture both simplify the implementation of bit-packed header fields, and make it possible to bit-pack header fields both within a layer and between layers. Protocol implementations do not have to be aware that certain packet attributes are single-bit values. The implementation can access the attribute like any other attribute. Furthermore, the memory location of a packet attribute is not shared with other attributes. The implementations therefore do not need to worry about overwriting other attributes when writing to a single-bit attribute. Moreover, since the header transformation modules have access to all packet attributes of each packet, it can efficiently pack all single-bit attributes into single-byte packet headers. This is shown in Figure 4, which shows a cross-layer bit-packed version of the header in Figure 3.

3.1.4 Header Compression

Header compression is a mechanism that reduces the size of transmitted headers by sending only header information that is strictly needed. Header compression can be used both to increase throughput over slow links [23] and to reduce the energy cost and packet loss rates in wireless networks [11]. In wireless sensor networks, header compression

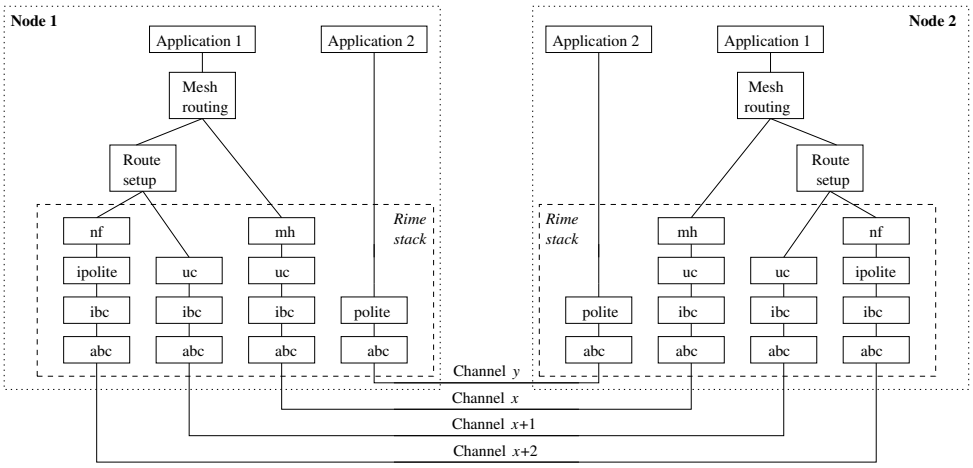

Figure 5. Two applications communicating using Rime. One uses a mesh routing protocol running on top of the Rime stack and one uses the Rime stack directly. Each communication path uses its own logical channel. See Section 4.1 for explanations of the names in the protocol stack.

has recently been used to reduce the size of IPv6 headers as part of the 6lowpan standardization effort [27].

Traditional header compression requires the header compression module to both parse the original header format, and to bit-pack the optimized header format. With packet attributes, header compression is naturally included in the architecture. Header parsing is performed on incoming packets, and the production of bit-packed headers is, as discussed above, already a part of the architecture.

3.2 Logical Channels

Communication in the Chameleon architecture uses different logical channels. Each channel has its own set of protocols and packet attributes. The communicating parties must agree beforehand on the particular set of protocols to be used for a particular channel.

Figure 5 illustrates the concept of logical channels in the Chameleon architecture (see Section 4.1 for explanations of the names in the protocol stack). Two applications, Application 1 and Application 2, run on two different nodes and communicate with each other using four logical channels, y, x, $x+1$, and $x+2$. Application 1 uses a mesh routing protocol, which in turn uses a route discovery protocol, and the best-effort multi-hop unicast Rime primitive, mh. Both nodes know that the mh primitive uses logical channel x, that the route discovery protocol uses channels $x+1$ and $x+2$, and that channel y is used by Application 2. Both nodes have agreed on this channel configuration before the communication is set up. The situation is similar to how all Internet hosts agree on that TCP port 80 is used for HTTP communication and that TCP port 25 is used for SMTP.

The logical channels are opened at run-time. When an application opens a logical channel for a stack of Rime primitives, the primitives register the packet attributes they use with Chameleon. Chameleon uses this information both when constructing outgoing headers and when parsing incoming headers.

The process of constructing and parsing headers is deterministic and reversible. When a packet is sent on a channel, Chameleon uses the attribute specification to construct the packet header. Similarly, when a packet arrives on the channel, Chameleon parses the header using the same attribute specification.

Chameleon is free to treat different logical channels differently in terms of how headers are constructed and what physical device will be used to send the packet. The mapping of channels to output devices is done either at compile time, at system boot-up, or at run-time. Chameleon can multiplex multiple logical channels over a single physical link by, e.g., explicitly transmitting the channel number in a packet header, or it can use different physical radio channels for different logical channels.

3.3 Buffer Management

The buffer management in the Chameleon architecture and the Rime stack is simple. All packets, both outgoing and incoming, are stored in a single buffer, called the Rime buffer. The Rime buffer contains both the application data and the packet attributes. All access to the Rime buffer is done at a single priority level so no locking mechanisms need to be used. Device drivers do not write to the buffer from their interrupt handlers, but must write to the buffer at system priority.

Protocols that need to queue packets, such as MAC protocols that wait for the radio medium to be free, can allocate so-called queue buffers to hold the queued packet. Queue buffers are dynamically allocated from a pool of queue buffers. The contents of the Rime buffer, including the packet attributes, are copied into the queue buffer when it is allocated.

Rime does not specify how the queue buffers are managed after they are allocated. If a protocol queues more than one queue buffer, it is up to the protocol to define how the queue is handled. For example, a MAC protocol may decide to send high priority packets before low priority packets once the radio medium is free.

3.4 Lightweight Layering

The Rime stack is built around a lightweight layering principle. The communication primitives are designed in a layered fashion, where more complex communication primitives build on simpler ones. This is inspired by work in the area of distributed programming [20], where many simple layers are used to implement complex mechanisms such as network consensus. The design with many simple layers allows for provable properties of composition of layers; we leave to future work to investigate if provable properties are possible in the Rime stack.

For sensor networks, the lightweight layering principle has several benefits. First, as the communication primitives are simple, they are easy to implement and test. Second, the memory footprint of the implementations of the primitives is small, which is important for memory-constrained sensor nodes. Third, as applications may attach to any layer of the stack, the applications can express precisely how much of the communication features that they need. In more heavyweight-layered stacks, such as the TCP/IP protocol stack, it generally is not possible to express such fine-grained feature requirements. For example, a TCP/IP application that needs congestion control but not guaranteed delivery cannot express this within the TCP/IP protocol architecture.

3.5 Header Transformations

The header transformation modules in Chameleon produce headers from the packet attributes supplied by the Rime stack. Chameleon can transform the packet attributes into an arbitrary packet header format. By transforming the packet attributes into a standard packet format, the Chameleon architecture can become compatible with another node that implements the standard. However, header transformations alone are not enough to mimic another communication protocol.

3.6 Header Transformations Are Not Enough

The header transformation mechanism is able to construct headers that are compatible with any communication protocol. However, a communication protocol is not defined by its protocol headers, but also by its protocol logic. In many cases, the Rime protocols already implement the protocol logic required to fulfill the impersonated protocol. In those cases, the Chameleon header transformation module only needs to create headers that match the impersonated protocol.

In case the protocol to be impersonated contains protocol logic not implemented by the Rime protocol, the Chameleon module must itself implement the missing parts of the protocol logic of the protocols that it impersonates. For example, a UDP/IP header transformation module must implement the ARP protocol if it is running over Ethernet, and a header transformation module that translates a reliable bulk-transfer Rime protocol into a TCP stream must implement the SYN-ACK exchange before data transmission can start.

3.7 Feedback from Lower Layers

The protocols implemented in a header transformation module may need to send feedback up to the application running on top of Rime. Examples of this include both congestion notification and estimates of the radio link quality.

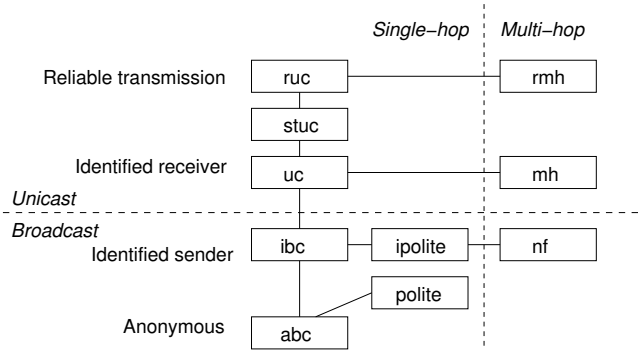

Figure 6. The communication primitives in the Rime stack and how they are layered.

Chameleon uses packet attributes to provide feedback from the header transformation modules to the Rime stack. When an event occurs that needs be forwarded to the application, Chameleon associates the event with the channel on which the event occurred. The next time the channel is active and a packet is sent towards the local application, Chameleon sets the appropriate packet attribute for the packet that is sent up through Rime. The feedback information may also be piggybacked on acknowledgement packets that Chameleon produces for the benefit of the application.

4 The Rime Protocol Stack

The Rime protocol stack provides a set of communication primitives, ranging from best-effort local neighbor broadcast and reliable local neighbor unicast, to best-effort network flooding and hop-by-hop reliable multi-hop unicast. Applications or protocols running on top of the Rime stack may use one or more of the communication primitives provided by the Rime stack.

4.1 Rime Communication Primitives

The protocols in the Rime stack are arranged in a layered fashion, where the more complex protocols are implemented using the less complex protocols. The communication primitives in the Rime stack and how they are arranged is shown in Figure 6.

We have chosen the communication primitives in the Rime stack based on what typical sensor network protocols use. Applications or protocols running on top of the Rime stack attach at any layer of the stack and use any of the communication primitives.

The Rime stack supports both single-hop and multi-hop communication primitives. The multi-hop primitives do not specify how packets are routed through the network. Instead, as the packet is sent across the network, the application or upper layer protocol is invoked at every node to choose the next-hop neighbor. This makes it possible to implement arbitrary routing protocols on top of the multi-hop primitives.

4.1.1 Anonymous Best-effort Single-hop Broadcast

The anonymous best-effort single-hop broadcast primitive (abc) is the most basic communication primitive in Rime. The abc primitive provides a way for upper layers to send a data packet to all local neighbors that listen to the

channel on which the packet is sent. No information about who sent the packet is included in the transmission.

All other Rime primitives are based on the abc primitive. Normally, however, the abc primitive is not used directly by applications or protocols that run on top of the Rime stack. When a packet is received by the abc module, the module immediately passes the packet to the upper layer.

4.1.2 Identified Best-effort Single-hop Broadcast

The identified best-effort single-shop broadcast primitive (ibc) sends a packet to all local neighbors. The ibc primitive adds the single-hop sender address as a packet attribute to outgoing packets. All Rime primitives that need the identity of the sender in the outgoing packets use the ibc primitive, either directly or indirectly through any of the other communication primitives that are based on the ibc primitive.

4.1.3 Best-effort Single-hop Unicast

The best-effort single-hop unicast primitive (uc) sends a packet to an identified single-hop neighbor. The uc primitive uses the ibc primitive and adds the single-hop receiver address attribute to the outgoing packets. For incoming packets, the uc module inspects the single-hop receiver address attribute and discards the packet if the address does not match the address of the node.

4.1.4 Stubborn Single-hop Unicast

The stubborn single-hop unicast primitive (suc) repeatedly sends a packet to a single-hop neighbor using the uc primitive. The stuc primitive sends and resends the packet until an upper layer primitive or protocol cancels the transmission. While it is possible for applications and protocols that use Rime to use the stubborn single-hop unicast primitive directly, the stuc primitive is primarily used by the reliable single-hop unicast (ruc) primitive.

Before the stuc primitive sends a packet, it allocates a queue buffer, to which the application data and packet attributes is copied, and sets a timer. When the timer expires, the stuc primitive copies the queue buffer to the Rime buffer and sends the packet using the uc primitive. The stuc primitive sets the number of retransmissions for a packet as a packet attribute on outgoing packets.

4.1.5 Reliable Single-hop Unicast

The reliable single-hop unicast primitive (ruc) reliably sends a packet to a single-hop neighbor. The ruc primitive uses acknowledgements and retransmissions to ensure that the neighbor successfully receives the packet. When the receiver has acknowledged the packet, the ruc module notifies the sending application via a callback. The ruc primitive uses the stubborn single-hop unicast primitive to do retransmissions. Thus, the ruc primitive does not have to manage the details of setting up timers and doing retransmissions, but can concentrate on dealing with acknowledgements.

The ruc primitive adds two packet attributes: the single-hop packet type and the single-hop packet ID. The ruc primitive uses the packet ID attribute as a sequence number for matching acknowledgement packets to the corresponding data packets.

The application or protocol that uses the ruc primitive can specify the maximum number of transmissions that the ruc module should attempt before the packet times out. If a packet times out, the application or protocol that sent the packet is notified with a callback.

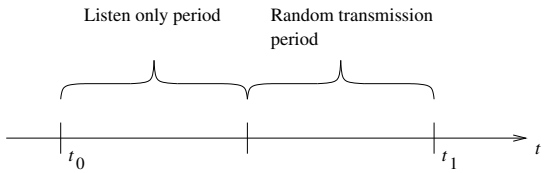

Figure 7. Timeline of the algorithm used by the polite broadcast primitive.

4.1.6 Polite Single-hop Broadcast

The polite single-hop broadcast primitive (polite) is a generalization of the polite gossip algorithm from Trickle [25]. The polite gossip algorithm is designed to reduce the total amount of packet transmissions by not repeating a message that other nodes have already sent. The purpose of the polite broadcast primitive is to avoid that multiple copies of a specific set of packet attributes is sent on a specified logical channel in the local neighborhood during a time interval.

The polite broadcast primitive is useful for implementing broadcast protocols that use, e.g., negative acknowledgements. If many nodes need to send the negative acknowledgement to a sender, it is enough if only a single message is delivered to the sender.

The upper layer protocol or application that uses the polite broadcast primitive provides an interval time, and message along with a list of packet attributes for which multiple copies should be avoided. The polite broadcast primitive stores the outgoing message in a queue buffer, stores the list of packet attributes, and sets up a timer. The timer is set to a random time during the second half of the interval time, as shown in Figure 7.

During the first half of the time interval, the sender listens for other transmissions. If it hears a packet that matches the attributes provided by the upper layer protocol or application, the sender drops the packet. The send timer has been set to a random time some time during the second half of the interval. When the timer fires, and the sender has not yet heard a transmission of the same packet attributes, the sender broadcasts its packet to all its neighbors.

The polite broadcast module does not add any packet attributes to outgoing packets apart from those added by the upper layer.

4.1.7 Identified Polite Single-hop Broadcast

Identified polite single-hop broadcasts (ipolite) works in the same way as the polite primitive but adds the identity of the sender as a packet attribute through the use of the ibc layer.

4.1.8 Best-effort Multi-hop Unicast

The best-effort multi-hop unicast primitive (mh) sends a packet to an identified node in the network by using multi-hop forwarding at each node in the network. The application or protocol that uses the mh primitive supplies a routing function for selecting the next-hop neighbor. If the mh primitive is requested to send a packet for which no suitable next hop neighbor is found, the caller is immediately notified of this and may choose to initiate a route discovery process.

```
nf_send(hops)
  packetattr_set(TTL, hops)
  packetattr_set(E2E_SENDER, our_address)
  packetattr_set(E2E_PACKETID, seqno)
  last_sender ← our_address
  last_id ← seqno
  ipolite_send({E2E_SENDER, E2E_PACKETID},
               default_interval)
  seqno ← seqno + 1

received_ipolite_packet
  if(packetattr_get(E2E_SENDER) ≠ last_sender &
     packetattr_get(E2E_PACKETID) ≠ last_id)
    last_sender ← packetattr_get(E2E_SENDER)
    last_id ← packetattr_get(E2E_PACKETID)
    if(call_upper_layer() ≠ 0)
      ttl ← packetattr_get(TTL) - 1
      if(ttl > 0)
        packetattr_set(TTL, ttl)
        ipolite_send({E2E_SENDER, E2E_PACKETID},
                     default_interval)
```

Figure 8. Implementation of the best-effort network flooding primitive, nf, in pseudocode.

Routing schemes that are not based on traditional routing tables, such as attribute-based routing [22] or opportunistic routing [2] are implemented using the routing function supplied by the upper layer application or protocol. When a next-hop neighbor has been found, the mh primitive uses the best-effort unicast primitive to send packets to it.

4.1.9 Hop-by-hop Reliable Multi-hop Unicast

The hop-by-hop reliable multi-hop unicast primitive (rmh) is similar to the best-effort multi-hop unicast primitive except that it uses the reliable single-hop primitive for the communication between two single-hop neighbors.

4.1.10 Best-effort Network Flooding

The best-effort network flooding primitive (nf) sends a single packet to all nodes in the network. Like the Trickle protocol [25], the nf primitive uses polite broadcasts at every hop to reduce the number of redundant transmissions. Unlike Trickle, however, the nf primitive does not perform retransmissions of flooded packets and packets are not tagged with version numbers. Instead, the nf primitive sets the end-to-end sender and end-to-end packet ID attributes on the packets it sends. A forwarding node saves the end-to-end sender and packet ID of the last packet it forwards and does not forward a packet if it has the same end-to-end sender and packet ID as the last packet. This reduces the risk of routing loops, but does not eliminate them entirely as the nf primitive saves the attributes of the latest packet seen only. Therefore, the nf primitive also uses the time to live attribute, which is decreased by one before forwarding a packet. If the time to live reaches zero, the primitive does not forward the packet.

Figure 8 shows an implementation of the nf primitive in pseudo-code.

4.2 Attribute Specification

Each Rime primitive keeps a list of the packet attributes that the primitive uses. The list also contains the number of bits that the primitive needs for each attribute. For instance, a type attribute may need to use a single bit only, whereas a time to live field may need five bits. Table 2 shows the attribute specification for the reliable multi-hop (rmh) primitive. Since the rmh primitive uses the ruc primitive, the attribute specification of rmh includes the attribute specification of ruc and the primitives on which ruc builds.

Table 2. Packet attributes used by the rmh primitive.

Primitive	Attribute	Bits	Scope
ibc	Single-hop sender	Addr len	1
uc	Single-hop receiver	Addr len	1
stuc	Retransmissions	4	1
ruc	Single-hop reliable	1	0
ruc	Single-hop packet type	1	1
ruc	Single-hop packet ID	2	1
ruc	Max. rexmit	4	0
rmh	End-to-end sender	Addr len	2
rmh	End-to-end receiver	Addr len	2
rmh	Time to live	5	2

Chameleon uses the packet attribute specification both when constructing packet headers and when parsing incoming headers. When an application opens a logical channel, Chameleon associates the attribute specification of all communication primitives used on the channel with the channel.

4.3 Cross-layer Information Sharing

The communication primitives in the Rime stack use packet attributes to pass information between layers. Once a packet attribute has been set, it is not removed as the stack processes the packet. This is different from how packet headers are processed in most layered stacks. Traditionally, the packet headers that belong to a particular layer is removed from the packet after the header has been processed.

5 Header Transformations

The Chameleon header transformation modules produce the headers for outgoing packets before passing the packets to the MAC or link layer. If Chameleon is not able to immediately send the packet, e.g. because an underlying MAC layer has turned the radio device off, Chameleon queues the outgoing packet for later processing.

Chameleon provides a default header packing mechanism that is used for logical channels where no other Chameleon module has been setup. The header packing mechanism packs the packet attributes of outgoing packets into a packet header based on the packet attribute specification used on the channel. The header packing is deterministic; the packing mechanism always produces the same packet header from the same packet attributes.

Incoming packets on a channel for which no other Chameleon module has been setup are handled by the default header unpacking mechanism. The default header unpacking mechanism does the reverse of the default header packing mechanism: it parses incoming bit-packed headers and turns them into packet attributes before sending the incoming data and packet attributes to the Rime stack. The header unpacking mechanism uses the packet attribute specification for the logical channel that received the packet.

The attribute specification for the receiving channel must match the attribute specification on the sending channel.

```
struct ibc {
  struct abc abc;
  const struct ibc_callbacks *callbacks;
};

struct ipolite {
  struct ibc ibc;
  const struct ipolite_callbacks *callbacks;
  struct ctimer timer;
  struct queuebuf *packet;
};
```

Figure 9. The implementation of the data structures of the layered ipolite and ibc primitives. The data structure includes the data structure of the lower layer primitive.

This may seem like an unusually strict requirement, but it is not any different from the situation in traditional networking; all Internet hosts must know that TCP port 80 is designated for HTTP communication.

The single-hop reliable communication primitives in the Rime stack benefit from automatic packet acknowledgement if the underlying link layer provides such mechanisms. Chameleon supports this by forging acknowledgement messages that are sent up to Rime when a link layer packet has been automatically acknowledged. Similarly, Chameleon intercepts acknowledgement messages sent from Rime and drops them; if the data packet arrived in the first place, the sender has already received a forged acknowledgement at the remote node.

6 Implementation

We have implemented the Chameleon architecture—the Rime stack and header transformation modules—in the C programming language using the Contiki operating system [13]. The implementation does, however, not use any Contiki-specific functionality and should therefore be portable to a wide range of embedded operating systems, including TinyOS.

The communication state for the Rime primitives are implemented as C structures as shown in Figure 9. A program that communicates using a Rime communication primitive statically allocates memory for the C structure corresponding to the communication primitive. Layering is implemented as nested structures. Each layer adds its own state variables in the structure. Additionally, each layer adds a pointer to a set of callback functions.

Packet attributes are stored in a fixed size array with one entry for each packet attribute. The default configuration has 15 attributes.

7 Network Protocols

To explore the expressiveness of the Rime communication primitives, we implement four types of typical sensor network protocols on top of Rime; one data collection protocol similar to MintRoute [33] and CTP [17], two data dissemination protocols based on Trickle [25] and Deluge [21], and one mesh routing protocol based on AODV [28].

7.1 Data Collection

The data collection is an address-free protocol that sends messages towards a sink node somewhere in the network. The protocol is address-free in the sense that the originating nodes do not send their messages to a specific addressed node. Instead, the nodes send their messages towards the nearest sink in the network.

The protocol does two things. It first builds a tree that originates at the sink node. The nodes build the tree by sending periodic announcements containing the number of hops away from the sink. After having built the tree, the nodes start sending messages towards the root of the tree. The protocol sends the messages using hop-by-hop reliable unicast.

To implement the data collection protocol with Rime, we use two Rime primitives: the reliable multi-hop module primitive for sending messages towards the sink, and the identified polite broadcast primitive for setting up the collection tree. The implementation uses two Rime channels: one reliable hop-by-hop (ruc) channel for sending data packets and one identified polite broadcast channel for sending neighbor request and solicitation messages.

7.2 Data Dissemination

Data dissemination is an important mechanism in sensor networks. Data dissemination is used to distribute commands or new code to nodes in the network [25]. We have implemented two data dissemination protocols with Rime: a single-packet dissemination protocol based on Trickle [25] and a multi-packet dissemination protocol based on Deluge [21].

The single-packet dissemination protocol runs on top of the network flooding (nf) primitive, which in turn uses the identified polite broadcast primitive. Packets to be disseminated are sent with a specific sequence number that is picked by the original sender. If a node receives a packet with a sequence number that is higher than its own sequence number, it resends the packet using the nf primitive. The resulting protocol behaves as the original Trickle protocol; only the implementation of the protocol is different.

The multi-packet dissemination protocol uses the single-packet dissemination protocol to send the data. Negative acknowledgements and repair packets are sent with identified polite broadcasts. The protocol thus uses three logical channels: two for the single-packet dissemination protocol and one for the identified polite broadcasts.

7.3 Mesh Routing

The mesh routing protocol is based on AODV [28]. Each node keeps a list of destination node addresses together with the address of the next-hop node. To setup a routing path through the network, the protocol floods the network with route request packets. When a node receives a route request packet, it sets up a backward path to the sender of the route request, and rebroadcasts the route request towards other neighbors. If a node receives a route request for its own address, the node sends a route reply packet back to the sender of the route request. Since nodes along the path towards the sender of the route request have a route back to the sender, the route reply can be sent along this route. The route reply is sent using multi-hop unicast. Nodes that forward the

route reply set up a route towards the receiver of the original route request packet. Data packets are sent using multi-hop unicast.

The Rime implementation of the mesh routing protocol uses the network flooding primitive (nf) to send route requests to the entire network and the best-effort multi-hop forwarding primitive (mh) to send multi-hop unicasts.

8 Chameleon Modules

To explore how well the Chameleon architecture is able to adapt to underlying protocols and mechanisms, we have implemented Chameleon modules that mimic the 802.15.4 header format, the X-MAC MAC protocol [4], and TCP/IP. The 802.15.4 Chameleon module does not implement the full 802.15.4 protocol, but simply constructs 802.15.4-compatible frames based on information in the packet attributes coming from Rime. The X-MAC module implements the X-MAC protocol, which treats broadcast and unicast packets differently. The TCP/IP Chameleon module is based on uIP [12] and constructs UDP packets and TCP segments from the packet attributes; reliable unicast packets are sent over TCP connections, and best-effort and broadcast packets are sent over UDP.

9 Evaluation

The primary metric of a communication architecture is how well its communication abstractions fit the problem domain. This is a qualitative metric and is therefore not possible to quantify. The secondary metric is the overhead of an implementation of the architecture. For sensor networks, we measure the overhead in memory footprint, energy consumption, and run-time performance. For sensor network programs, memory footprint typically is determined at compile time. For dynamically allocatable buffers, the memory footprint is compile-time configurable and it is interesting to measure how well a typical configuration works. Energy consumption is typically determined by the time the radio device can be asleep. This is, however, dependent on the MAC protocol and not on the communication architecture. Nevertheless, the size of packet headers matters for the energy consumption as larger packet headers require more transmission and reception time, which amounts to a higher energy consumption. The run-time performance is number of microprocessor cycles required to execute the implementation of the architecture.

We evaluate the qualitative aspect of how well the communication primitives in the Rime stack map onto sensor network protocols. This is done in Section 7 by implementing sensor network protocols with Rime. Similarly, we evaluate the flexibility of Chameleon by implementing a set of underlying protocols with Chameleon. We do this in Section 8.

In this section, we quantitatively evaluate the complexity of the protocol implementations, the memory footprint of our implementation of the Rime stack, and the run-time overhead of our implementation of the architecture. Our results show that the complexity of the protocol implementations in Rime is lower than similar implementations for previous communication architectures for sensor networks, that the memory footprint is small, and that the execution-time overhead of the implementation is small.

Table 3. Comparison of implementation complexity of the data collection and data dissemination protocols.

Module	Statements (SP)	Statements (Rime)
Data dissemination	109	28
Data collection	138	72

9.1 Complexity of Protocol Implementations

One of the design goals of the Rime stack is to reduce the implementation complexity of communication protocols for sensor networks. To evaluate if Rime reduces the implementation complexity of sensor network protocols, we compare the Rime implementation of the data collection, data dissemination, and mesh routing protocols with the corresponding implementations of similar protocols implemented within the SP architecture [30][1].

We use the number of program language statements used to implement the protocol as an approximation of the implementation complexity. While this approximation is affected by many other factors other than the implementation complexity, such as programming language terseness and programming style, it allows us to see general trends. If one protocol implementation consists of radically more program language statements than another does, this may hint towards the first being more complex than the other is.

Table 3 lists the number of program language statements in the implementations of the Trickle data dissemination protocol and the data collection protocol in SP and in Rime. The numbers for Rime are C statements, excluding comments and header files, and the numbers for SP are nesC statements, excluding comments and module specification statements. We see that the number of program language statements in the Rime implementations is much smaller than for SP. We did not have a mesh routing implementation in SP to compare with, but the TinyAODV code in TinyOS 1.x[2] consists of 496 statements, while the mesh routing and route discovery code in Rime consists of only 105 statements.

9.2 Memory Footprint

Due to the limited memory size of sensor network nodes, the memory footprint of the Chameleon architecture is an important measure of it feasibility and usefulness on memory-constrained sensor network nodes.

The total memory footprint is composed of two parts: the memory footprint of the compiled code, which always is present in on-chip ROM, and the memory footprint of the data memory required to run the code. The size of the data memory includes the buffers, both the Rime buffer and all queue buffers, and all protocol state.

Table 4 lists the memory footprint of the Rime modules, compiled for the MSP430 microcontroller. The code was compiled with a Rime buffer size of 128 bytes and 4 queue buffers. If all modules are to be included in the final system, the total memory footprint of the compiled code is less

[1]The SP implementations were provided by Joe Polastre.

[2]The TinyAODV code is from tinyos-contrib-1.1.0/contrib/hsn/tos/lib/AODV_Core.nc and tinyos-contrib-1.1.0/contrib/hsn/tos/lib/AODV_PacketForwarder.nc.

Table 4. Static memory footprint of Rime compiled for the Tmote Sky/MSP430. The RAM for the communication primitives is the per-channel RAM footprint.

Module	ROM (bytes)	RAM (bytes)
rimebuf	416	168
queuebuf	320	730
abc	134	2
ibc	114	2
uc	124	2
stuc	272	16
ruc	254	4
polite	390	15
ipolite	406	15
mh	246	2
rmh	266	3
nf	356	7

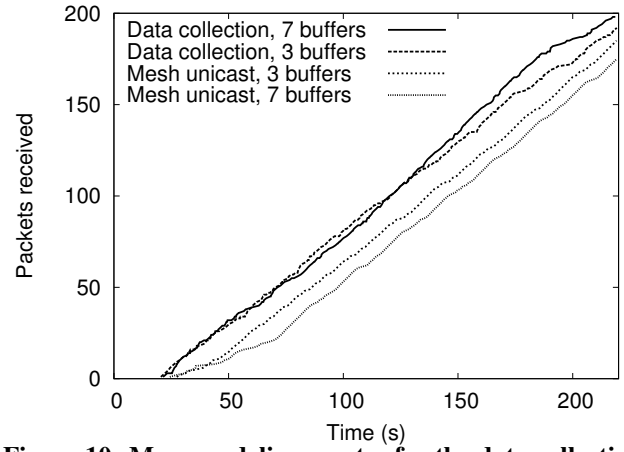

Figure 10. Message delivery rates for the data collection and mesh routing protocols with three and with seven queue buffers.

than 10% of the available ROM and RAM on the Tmote Sky platform.

The largest module in terms of RAM footprint in Table 4 is the queuebuf module. The queuebuf module contains statically allocated memory for all queue buffers in the stack and the maxiumum number of queue buffers therefore determines the RAM footprint of the module. How many queue buffers that are needed by Rime depend on the applications running on top of Rime and on the environment in which they run.

To determine a reasonable configuration of available queue buffers, we simulate a network of 25 nodes in a square lattice. Each node run a data collection application and an application that uses the mesh routing protocols. We measure the amount of queue buffers that are needed. With an interval of 10 seconds, each node sends a data collection message towards the sink and a mesh routing message. The sink is located in the top right corner, and the mesh routing message is sent to the node in bottom right corner. With this configuration, the maximum amount of queue buffers used in the network is seven. The number of received packets is shown in Figure 10. After 220 seconds, the number of delivered data collection packets is slightly higher with an unlimited amount of queue buffers. The number of delivered mesh routing messages is, however, higher with the limited amount of queue buffers. The reason for this is that the mesh routing protocol, which does not allocate queue buffers, does not have to compete with the data collection traffic that cannot allocate enough queue buffers.

Table 5. Static memory footprint of five header transformation modules compiled for the Tmote Sky/MSP430.

Module	ROM (bytes)	RAM (bytes)
Bit-pack	1090	0
Non-opt	472	0
802.15.4	642	0
UDP/IP	1658	104
TCP/IP	6042	162

The memory requirements for five header transformation modules are given in Table 5. The UDP/IP module includes an implementation of the ARP protocol and the TCP/IP module contains the entire uIP TCP/IP stack [12]. The ROM requirement of uIP and the ARP module is 5312 bytes.

9.3 Execution Time Overhead

The purpose of the Chameleon architecture is not to optimize the execution time but to provide an adaptive communication architecture for sensor network applications and protocols. Nevertheless, it is interesting to study the execution time characteristics of the architecture as it provides insights into the behavior of a packet attribute-based architecture such as Chameleon. We find that the execution time overhead of the packet attribute-based Chameleon architecture is small compared to a packet header-based architecture.

We first measure the Chameleon architecture as a black box. To measure the effect of the execution time overhead of Chameleon, we implement two versions of the Chameleon architecture: one using packet attributes, and one using traditional packet headers. We run both implementations on Tmote Sky nodes [31] using the X-MAC protocol [4] with a 9% duty cycle. Both implementations construct 802.15.4-compatible MAC-level headers. We run two experiments. First, we let two Tmote Sky nodes ping-pong packets between each other using the best-effort unicast primitive. Second, the two nodes ping-pong packets using the best-effort local area broadcast primitive. During both experiments we measure the round-trip time of the packets. The results of the measurements are show in Figure 11. The round-trip time, which is completely dominated by the time spent waiting for the other node to wake up, is not noticeably different between the two implementations.

To study the performance effect of packet attributes, we measure and compare the run time of the Rime stack implemented with packet attributes and with packet headers. We run the Rime data collection protocol on nine Tmote Sky motes. We run the MSP430 microcontroller at a speed of 2.45 MHz and measure the CPU cycles with a hardware timer running at the CPU speed. We measure the time from the application's invocation of the entry function into the

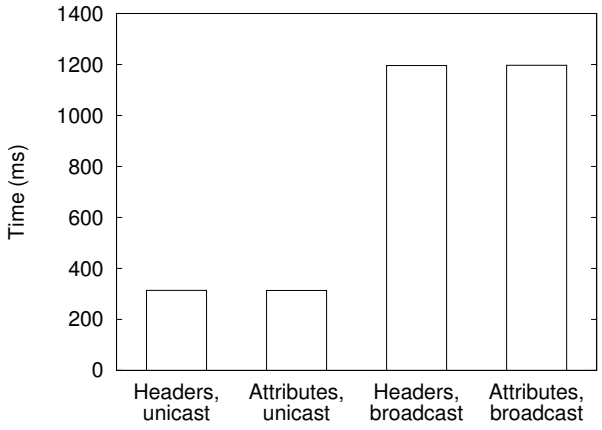

Figure 11. The round-trip time for ping-ponging a packet between two Tmote Sky motes using the X-MAC protocol. The overhead for packet attributes is negligible.

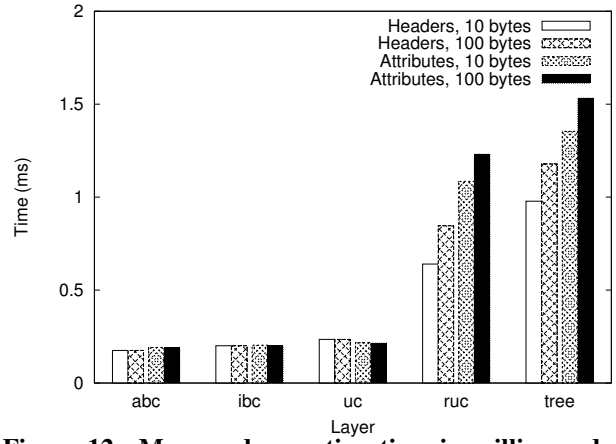

Figure 12. Measured execution time in milliseconds at different layers, with 10 and 100 byte packets, with and without packet attributes.

Rime stack, to the packet is transmitted. Packets are formed as 802.15.4 frames. The packet attribute-based implementation uses the 802.15.4 Chameleon module, whereas the packet header-based implementation uses a tailored MAC driver that is able to produce 802.15.4-compatible frames. We measure the execution time at different points in the stack: at the data collection layer (tree), at the reliable unicast layer (ruc), at the best-effort unicast layer (uc), at the identified broadcast layer (ibc), and at the anonymous broadcast layer (abc). The measured execution time does not include the overhead of the device driver for the radio chip, which typically is on the order of a few milliseconds on the Tmote Sky. To study the impact of packet size, we run two experiments, one with 10-byte packets and one with 100-byte packets.

The results of the execution time measurements are shown in Figure 12. We see that for the lowest three layers in the stack, the execution time for the packet attribute-based implementation is similar to or lower than the execution time for the packet header-based implementation. For the reliable unicast and the data collection layers, however, the execution time is higher for the packet attribute-based implementation. The reason for this is that the reliable unicast module copies the outgoing packet to a queue buffer. In the packet attribute-based, the prototype implementation copies a list of all possible packet attributes to the queue buffer. In the packet header-based implementation, the queue buffer operation only copies the packet header, which typically is smaller than the list of available packet attributes. This is, however, an artifact of our current prototype implementation and it is most likely possible to optimize this operation.

To quantify the run-time overhead of different header transformation modules, we measure the processor cycles spent in each transformation module. We perform two experiments, one with the single-hop unicast module (uc) and one with the data collection protocol, which uses both best-effort broadcast messages and reliable unicast messages. We take timestamps before calling the header transformation module and after returning from the call.

Table 6. Measured execution time in cycles of different header transformation modules. 1000 cycles is approximately 0.4 ms.

Module	Data collection	Unicast
Non-packed, input	991	738
Non-packed, output	584	458
Bit-packed, input	1160	739
Bit-packed, output	932	457
802.15.4, input	1424	503
802.15.4, output	956	348
UDP/IP, input	2103	904
UDP/IP, output	1735	512

Table 6 shows the measured CPU cycles for the different header transformation modules. The overhead is larger in the data collection case because the data collection protocol has more headers than the unicast protocol. The 802.15.4 header transformation module is the fastest in the unicast case because it is optimized for this case; it checks if the packet attributes match the 802.15.4 header and if so quickly produces a matching header. If more packet attributes are present, extra headers are put after the standard 802.15.4 header, which increases the execution time. UDP/IP calculates the IP checksum, which increases its execution time.

9.4 Energy Trade-offs in Bit-packed Headers

The cross-layer bit-packed header transformation module in Chameleon can reduce the size of transmitted packet headers, which leaves more room for data in each packet, and reduces the risk of bit errors in the header [11]. In this context, it is interesting to study if the header packing also reduces the energy consumption. Since energy is spent on both constructing and transmitting headers, the reduction in transmission energy must be more than the increase in processing energy for constructing the bit-packed headers. To evaluate the energy consumption, we use the energy estimation mechanism in Contiki [15]. The mechanism computes an energy estimate from pre-measured current component draw and real-time measurements of radio transmission and listening time, and CPU processing and sleeping time.

Table 7. The estimated power consumption in mW of the bit-packed header transformation module and the non-packed header transformation module. The total power is rounded to two significant digits.

Module	CPU power	Transmit power	Total power
Non-packed	0.0401	0.237	0.28
Bit-packed	0.0401	0.237	0.28

We let the data collection application send data once every two seconds on nine Tmote Sky boards. We compare the estimated energy consumption of the bit-packed header with that of a non-packed header. In both experiments, we transmit 16 bytes of application data. By bit-packing the data collection protocol headers, the data collection header size is reduced with 33% from twelve bytes (Figure 3) to eight bytes (Figure 4). While this does increase the available space for data in each packet, and reduces the risk of bit errors in the header, it does not seem to significantly reduce the energy consumption. Table 7 reports the estimated energy consumption for the data collection application for both the bit-packed headers and the non-packed headers. All reported data are averaged over six runs of 200 seconds each. While our measurements show slightly lower energy consumption with a bit-packed header, the reduction is not significant.

10 Related Work

Culler et al. [10] highlight the need for defining a sensor network architecture and discuss where to place the narrow waist of the architecture. The authors conclude that, in order to allow for both address-free and name-based protocols, the narrow waist should be placed between the network layer and the link layer. Polastre et al. [30] implement and evaluate SP, a sensor network architecture based on the proposal by Culler et al. The SP architecture provides an abstraction of the link layer and mechanisms for managing next-hop neighbors and their sleep cycle schedules. The SP architecture has inspired the design of the Chameleon architecture and parts of the SP architecture, such as the placement of the narrow waist, can be found in the Chameleon architecture. However, Rime provides a wide range of communication primitives for simplifying protocol implementations that SP does not. Furthermore, the header optimization and transformation mechanisms in Rime do not have any counterpart in SP.

Cheng et al. [16] propose a modular network layer that is intended to run on top of SP. The modular network layer provides multi-hop routing functionality that SP does not provide. The network layer provides a software architecture that is designed to simplify the implementation of routing protocols. Rime is different in that Rime itself provides a rich set of communication primitives instead of relying on the protocol or application developer to implement such primitives. The modular network layer by Cheng et al. is layered and the protocol developer must specifically define packet headers. The architecture suffers from the same problems as other layered stacks in that cross-layer information sharing is difficult and in that the architecture does not easily allow bit-packed headers and header optimizations.

The challenges of constructing a communication architecture for sensor networks is very different from the challenges in constructing a general-purpose communication architecture for the Internet [7]. Nevertheless, our work is inspired by the ideas behind the Plutarch architecture by Crowcroft et al. [9] and the role-based architecture by Braden et al. [3]. The Plutarch architecture does not specify the particular protocols to be used within the architecture, which is similar to how the Chameleon architecture does not specify the particular message formats used within the architecture.

The role-based architecture by Braden et al. shares many similarities with the packet attribute and header transformation mechanisms in the Chameleon architecture, such as the use of packet attributes instead of packet headers. However, there are a number of significant differences. First, the role-based architecture is designed to be fully modular, with the possibility to freely compose roles. In contrast, we deliberately designed the Chameleon architecture and the Rime stack to be strictly layered in order to reduce the complexity of the architecture, which is important for resource constrained sensor nodes. Second, unlike the Chameleon architecture, the role-based architecture does not specify any communication protocols or mechanisms, but is only a framework in which protocols can be implemented. The Chameleon architecture includes a set of communication primitives that are designed for the specific application domain of sensor networks. Third, since the problem scope of a general-purpose Internet-scale communication architecture is much larger than that of a sensor network communication architecture, the role-based architecture includes several mechanisms for controlling the access to packet attributes. In this sense, the Chameleon architecture is significantly less complex.

Chang and Gay [5] present network types, an extension to the nesC language that solves parts of same problem that the header transformation modules in Chameleon do: protocol implementations do not need to implement header field alignment and header field byte ordering. Unlike Chameleon, however, network types do not address the cross-layer information sharing problem, but still require protocol headers to be strictly separated.

A number of programming primitives for sensor networks have been proposed and investigated [18, 26]. Rime is different from this body of work in that the communication primitives provided by Rime are more low-level. The Rime communication primitives can be used to implement higher level programming primitives such as logical neighborhoods [26] or generic role assignment [18].

11 Conclusions

We present a novel communication architecture for wireless sensor networks that provides a set of communication primitives that map well onto the communication primitives used by typical sensor network protocols. The architecture does not use packet headers but instead uses packet attributes, an abstract representation of the information commonly found in packet headers. Packet attributes make it possible for the architecture to adapt to a variety of underlying protocols and mechanisms such as MAC and link layer

protocols. The overhead in terms of memory footprint and execution time is low. Thus an adaptive communication architecture such as Chameleon can be efficiently used for wireless sensor networks.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Governmental Agency for Innovation Systems. We are grateful to Joe Polastre for access to his SP code. Many thanks to Gertjan Halkes and John Heidemann for interesting discussions that helped improve this work, and to our paper shepherd Chenyang Lu for reading and suggesting improvements to the paper.

12 References

[1] IEEE standard 802.15.4. IEEE Computer Society, October 2003.

[2] S. Biswas and R. Morris. Opportunistic routing in multi-hop wireless networks. In *Proceedings of the ACM SIGCOMM '05 Conference*, Philadelphia, Pennsylvania, August 2005.

[3] R. Braden, T. Faber, and M. Handley. From protocol stack to protocol heap: role-based architecture. *SIGCOMM Comput. Commun. Rev.*, 33(1):17–22, 2003. ISSN: 0146-4833

[4] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320, Boulder, Colorado, USA, 2006. ISBN: 1-59593-343-3

[5] K. K. Chang and D. Gay. Language support for interoperable messaging in sensor networks. In *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 1–9, Dallas, Texas, 2005. ISBN: 1-59593-207-0

[6] J. I. Choi, J. W. Lee, M. Wachs, and P. Levis. Opening the sensornet black box. In *Proceedings of the International Workshop on Wireless Sensornet Architecture (WWSNA)*, Massachusetts, USA, April 2007.

[7] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: defining tomorrow's internet. *IEEE/ACM Trans. Netw.*, 13(3):462–475, 2005. ISSN: 1063-6692

[8] Arch Rock Corporation. A sensor network architecture for the ip enterprise. In *Proceedings of the 6th international conference on Information processing in sensor networks, demo session*, Cambridge, Massachusetts, USA, 2007.

[9] J. Crowcroft, S. Hand, R. Mortier, T. Roscoe, and A. Warfield. Plutarch: an argument for network pluralism. In *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, Karlsruhe, Germany, 2003.

[10] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, and J. Zhao. Towards a Sensor Network Architecture: Lowering the Waistline. In *Proceedings of HotOS 2005*, 2005.

[11] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. *ACM/Baltzer Journal on Wireless Networks*, 3(5), 1997.

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

[14] A. Dunkels, T. Voigt, and J. Alonso. Making TCP/IP Viable for Wireless Sensor Networks. In *First European Workshop on Wireless Sensor Networks (EWSN 2004)*, Berlin, Germany, January 2004.

[15] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He. Software-based online energy estimation for sensor nodes. In *Proceedings of the Fourth IEEE Workshop on Embedded Networked Sensors (Emnets IV)*, Cork, Ireland, June 2007.

[16] Cheng T. E., R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A modular network layer for sensornets. In *Proceedings of OSDI 2006*, Seattle, Washington, USA, November 2006.

[17] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis. TEP 123: Collection Tree Protocol. Technical report. http://www.tinyos.net/tinyos-2.x/doc/

[18] C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 230–242, San Diego, California, USA, 2005.

[19] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006.

[20] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[21] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. SenSys'04*, Baltimore, Maryland, USA, November 2004.

[22] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[23] V. Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, Internet Engineering Task Force, February 1990.

[24] S. J. Leffler and M. J. Karels. Trailer encapsulations. RFC 893, Internet Engineering Task Force, 1984.

[25] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of NSDI'04*, March 2004.

[26] L. Mottola and G. Picco. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks (InterSense '06)*, page 8, Nice, France, May 2006.

[27] G. Mulligan, N. Kushalnagar, and G. Montenegro. IPv6 over IEEE 802.15.4 BOF (6lowplan). Web page. Visited 2005-02-21. http://www.ietf.org/ietf/04nov/6lowpan.txt

[28] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc on-demand distance vector (aodv) routing. RFC 3561, Internet Engineering Task Force, 2003.

[29] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, Baltimore, MD, USA, 2004. ISBN: 1-58113-879-2

[30] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys*, 2005.

[31] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.

[32] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 171–180. ACM Press, 2003. ISBN: 1-58113-707-9

[33] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, Los Angeles, California, USA, 2003. ISBN: 1-58113-707-9