INTEGRATING A DATA DESCRIPTION LANGUAGE WITH PROTOCOL STACK DEVELOPMENT

Yan Wang and Verónica Gaspes
Halmstad University, Sweden
yan.wang@hh.se, veronica.gaspes@hh.se

ABSTRACT

Communication software, most notoriously protocol stacks, are an area of growing interest. Many companies implement new or revised protocols for new application requirements, and reimplement well-known infrastructure protocol stacks to accommodate to new hardware and software platforms. However, due to the complexity and performance-critical nature of communication software, implementing protocol stacks remains a time-consuming and error-prone task with considerable impact on time to market, scalability and maintainance. The work at hand investigates how to provide program development support for protocol stack implementation to make it easier and more likely to be correct while respecting non-functional constraints. We present a language-based approach for the implementation of protocol stacks. We define a domain-specific embedded language, IPS, for declaratively describing overlaid protocol stacks. In IPS a high-level packet specification is described using a data description language which is compiled into a.) an internal data representation, and b.) packet processing functions in C. Both are then integrated into the dataflow framework of a protocol overlay specification. IPS generates highly portable C code for various architectures from this source. We present the compilation framework for generating packet processing and protocol logic code, and a preliminary evaluation.

KEY WORDS
Network programming, Communication software, Domain-specific languages.

1 Introduction

In both research and business enterprises that deal with networked systems, the implementation of protocol stacks is a central issue. Newly designed or revised protocols are implemented for new application requirements; well-know infrastructure protocols are reimplemented to adapt to new emerging hardware and software architectures. However, implementing protocol stacks is tedious, error-prone and time-consuming due to the complex and performance-critical nature of communication software. The specifications of most modern protocols are quite large. And specifications are often not mapped into code in a straightforward manner, which makes it difficult to both achieve and check correctness. In addition, a range of mature optimization techniques designed to make protocol code more efficient tend to make implementations more complicated and are new sources of errors. In order to improve on time to market, scalability, maintainability and product evolution, even programming efficiency is relevant.

In this work, we take a language-based approach in the form of a Domain-Specific Language (DSL) [1] to address these issues and provide development support for protocol stack implementation. A DSL is a programming language that closely models a particular application domain or problem of knowledge or expertise, where the concepts are tied to the constructs of the language. Since the program written in a DSL is at the level of abstraction of the application domain, it is expressive, concise and self-explanatory. It allows domain-oriented compiler optimizations, constraint enforcement, and language-level debugging. We present a Domain-Specific Embedded Language (DSEL), Implementation of Protocol Stacks (IPS), which combines a data description language for specifying protocol packets with a framework for constructing overlaid protocol stacks. The technique of embedded languages [2] facilitates its development by using a host language platform. By compiling IPS code into C in a later step, IPS code exposes good performance, as well as low energy consumption and memory usage.

IPS is partially based on the Data Description Calculus (DDC) [3] which uses types from dependent type theory to describe various forms of ad-hoc data. Types in DDC are interpreted as parsing functions that produce both internal representations of the external data and parse descriptors pinpointing errors in the original source. The high-level IPS specification is compiled into two parts: a.) an internal representation, e.g., a C data type, and b.) packet processing functions, i.e. marshaling and parsing. The internal representation can be integrated into the implementation of protocol logic, while the packet processing functions will be combined into the dataflow framework generated from the IPS protocol overlay specifications. With our language, protocol stack programmers can declaratively specify and construct a complex protocol stack using simple and explicit high-level abstractions, and automatically generate highly portable C code for various architectures from this source. This is clearly an innovation, it facilitates specification, encourages modularity and reuse of code, and enforces correctness by sound methodology.
The remainder of this paper is structured as follows. Section 2 introduces the domain-specific embedded language IPS which we have developed. We describe a protocol stack specification in IPS, notations and meaning of the language constructs for specifying protocols and their relationship in a layered hierarchy, and summarize the main design decisions of our approach. Section 3 includes the preliminary evaluation results, Section 4 discusses related work, and Section 5 concludes and points out future work.

2 Implementation of Protocol Stacks

A domain-specific framework for describing protocols protocol stacks should provide a concise and structured development process, while keeping the language small and focused on the specific task. IPS describes protocols as separate modules and combines them into protocol stacks (graphs), capturing the core features of protocol stacks as we find them in their specifications: a) packet format descriptions including physical layout, dependency of fields and semantic constraints, b) cleanly separated descriptions of different protocols and their implementation details, c) separation of packet processing from protocol logic, d) explicit declaration of assumptions made about related protocols e) a concise way to overlay individual protocols into protocol stacks (and graphs).

2.1 Protocols in IPS

We illustrate our language IPS following the specification of Rime [4], a lightweight layered communication stack for sensor networks sketched in Figure 1.

![Figure 1. The communication primitives in the Rime stack](image1)

IPS supports protocol development by providing a high level of abstraction. The implementation of protocols are concise in the sense that protocols are expressed with few statements and declarations, reducing verbosity by hiding implementation details and making programs easier to read, understand and debug.

Figure 2 shows the implementation in IPS of the protocol ibc (identified best-effort single-hop broadcast), whose task is to add the sender address to outgoing packets in the Rime stack [4].

As we see, only few crucial properties have to be specified for a protocol definition in IPS. The definition of ibc is introduced by the keyword protocol followed by comma-separated component definitions enclosed by curly brackets. Name is the identity of a protocol used to be recognized by others, "ibc" in our example. Packets are one of the fundamental protocol abstractions and thus an integral part of the language (specified by packet). A packet is a sequence of header fields (here for ibc, fields header0 and header1) followed by a payload. Both syntactic and semantic properties are specified here: The first header field header0, of type int id 0 and size 2 (bits), is used to route incoming packets to the upper encapsulated protocols. The 2 bits allow for 4 possible protocols to be located on top of ibc, but is constrained to values 0 or 1 in the specification. The incoming and outgoing C packet buffer is specified by bufferin and bufferout respectively. For Rime, a single buffer rimebuf is used for both incoming and outgoing packets to reduce memory footprint. Furthermore, a protocol has to specify how to transmit its packets via its lower-layer protocols (send function), and how to pass its receiving packets to the upper-layer protocols (receive function). IPS offers a seamless model for thinking about packets: packets can be specified by packet formats and the fields of a packet can be referred in the operations directly.

**Figure 2. Ibc protocol implementation in IPS**

```plaintext
ibc :: Protocol
ibc = protocol{
    name = "ibc",
    packet = header0:header1:payload,
    bufferin = "rimebuf",
    bufferout = "rimebuf",
    send = ibcsend,
    receive = end
}

where
    header0 = int 0 2 |* constraint
        constraint x = (x==0)||((x==1)
    header1 = int 1 16
    localAddr = cterm "rimeaddr_node_addr.u16[0]"
    ibcsend = (ifE (upperProtocolIs "uc")
        {-then-} (header0 = 1)
        {-else-} (header0 = 0))
    ; header1 = localAddr
    ; gotoSap 0
    ; end
```

The task of ibc is to add the sender node's local address localAddr as a header field. It is obtained from a C array rimeaddr_node_addr.u16 (which is indicated by the prefix cterm). Furthermore, field header0 indicates the upper protocol in a send operation, 1 indicating protocol "uc", otherwise 0. After adding header field values in header1, the packet is passed to the next lower layer through gotoSap 0 (Sap, i.e. service access point, being the interface between two adjacent protocols). When a packet is received by the ibc, it immediately passes the packet to the upper layer. In other words, the receiving function receive does nothing and consists only of the mandatory end keyword.

IPS allows code reuse by overwriting part of the spec-
ification. For example, in the Rime stack, the protocol ipolite (identified polite single-hop broadcasts) works in the same way as the protocol polite but located on the top of ibc to identify the sender. It can be realized by reusing the existing polite implementation and only overwriting the protocol name as shown in Figure 3.

```
import Polite(polite)
ipolite :: Protocol
ipolite = polite{
    name = "ipolite"
}
```

Figure 3. Ipolite protocol implementation in IPS

The basic protocol type Protocol can be extended to new protocol types; new state information is added by additional declarations, and protocol behaviors are extended by additional procedure code. For example, protocol stuc (stubborn unicast) in the Rime stack provides a reliable transmission by repeatedly sending a packet to a single-hop neighbor. As Figure 4 shows, the IPS specification for stuc uses a different type RetransmissionProtocol. This type requires to specify a timer retransmissiontimer with an id 0 and an expire interval 200(ms), as well as a retransmission function retransmission with parameters parameters and function body functionbody.

```
stuc :: RetransmissionProtocol
stuc = retransmissionprotocol{
    name = "stuc",
    bufferin = "rimebuf",
    bufferout = "rimebuf",
    packet = header0:payload,
    send = stucsend,
    receive = stucreceive
    retransmissiontimer = stuctimer
    retransmission = function0
}
```

Figure 4. Stuc protocol implementation in IPS

2.2 Protocol stacks

In IPS, any protocol type is derived from Protocol, e.g., RetransmissionProtocol. Therefore, all the protocol abstractions have a uniform protocol interface and can be composed as building-blocks. IPS provides a basic combinator <|> to build protocol stacks in a strictly linear way. Introduced by the keyword stack, a number of protocols are enumerated to form a top-down stack. For example, we can overlay the separately defined protocols abc, ibc and ipolite in a protocol stack, as shown in Fig 5.

```
stack1 = stack
( protocol ipolite [ ] -1)
( protocol ibc [(header ibc 0)\* (\x->x==*0)] 0)
( protocol abc [(header abc 0)\* (\x->x==*0)] 0)
```

Figure 5. A protocol stack implementation in IPS

```
stack2 = stack
( protocol polite [ ] -1)
( protocol ibc [(header ibc 0)\* (\x->x==*0)] 0)
( protocol abc [(header abc 0)\* (\x->x==*0)] 0)
```

Figure 6. A protocol graph implementation in IPS

Care must be taken that all protocols in one protocol stack(graph) use the same protocol buffers for raw data, i.e., bufferin and bufferout. This is ensured by the combi-

nators for stack construction and stack merging. Whenever two protocols with non-matching buffer names are used,
the user receives an informative error message instead of generated code.

2.4 Design Decisions

Domain-specific embedded language: It is fairly difficult and time-consuming to design and implement a DSL from scratch, including grammar, parser, code generator or interpreter etc. Much of the startup cost goes to non-domain specific parts, e.g., variables and arithmetic types, which can be avoided by using an embedded approach as a shortcut [1]. A Domain-Specific Embedded Language (DSEL) embeds domain-specific concepts and features into an existing host programming language and uses its implementation and tools, thereby accelerating implementation. Reusing syntax and semantics of the host language for non-domain specific aspects, specific data types and operators are defined as a library. While providing a look-and-feel of special syntax similar to a real DSL, DSELS are much easier to maintain. When new functions and procedures are added to the DSEL, they will be seamlessly integrated into the former parts without any changes to the existing tool set.

Using Haskell as a host language: We have chosen the functional language Haskell as our host language for embedding IPS because of a number of advantages it has over other languages. Its first and major advantage is that it supports higher-order functions. Embedding a DSEL in Haskell can be realised as a higher-order algebraic structure, a first-class value that has the appearance of special syntax. Other advantages of Haskell are a relatively lightweight syntax (a property which is inherited from the host language), and the static type system which provides an automatic compile-time check of specific usage constraints for the DSEL. Haskell has previously been used to embed DSELS for different domains: Lava [5] for hardware design, LexiFi [6] for evaluating financial contracts, Cryptol [7] for code encryption and decryption, and so on.

Modular architecture similar to specification: Protocol specifications are usually written in a modular fashion, specifying a whole stack of layered protocols. This reduces complexity and makes reuse and configuration possible. It should be straightforward for a protocol stack implementation to contain code in the same fashion as the specification, and would render code easy to read, understand and trace back for debugging and maintaining. IPS enforces a strict layered and fully modular structure. The basic building-blocks are protocols implemented as separate modules, which are then overlaid to produce a whole protocol stack. However, modularity is one of the chief villains in attempting to obtain good performance due to the large overhead involved in interfacing between modules. Thus, we generate lower-level monolithic code from modules instead of running IPS directly.

Using C as a target language: In order to achieve good performance and hardware support, IPS programs are compiled into highly portable non-architecture specific C-implementations, which will be compiled further into machine code for different platforms in a later step. C is the target language of choice, because it is established for system programming and traditionally related to operating systems development. A standards-compliant C program can be compiled for a very wide variety of computer platforms and operating systems with little or no change to its source code.

Automatic packet processing code from packet descriptions: Packets are semistructured data which can be specified formally by using dependent types, e.g., the physical organization, dependencies among field contents and constraints over the values. In general, packet formats and specifications are independent of any given machine’s architecture. We take an approach similar to data description languages, which automatically generate data processing code from data format descriptions. In our case, all packet processing is contained in the automatically generated library from packet descriptions. It enables to build up strong intuitions from high-level perspective which can be straightforwardly mapped from packet figure and explanation in protocol specification. Since some standard tasks common to all protocol implementations are fully automated, our approach liberates protocol stack implementation from low-level data manipulation related to the wire format of packet, and thus substantially reduces the complexity of such implementations.

Separate packet processing and protocol logic: A major task of protocols is to process packets. However, packet processing functions are actually independent pieces of code without much interaction with the rest of the protocol implementation. Packet processing can be separated from protocol logic[4, 8] where all management of packets is dealt with in one place. Realizing this principle avoids the low-level details of packet headers to spread over all the source code, and mimics the description style of a protocol specification, where packet formats are specified by tables and protocol logic is specified in running text.

Global code generation for the protocol stack: While keeping source specifications modular, the central protocol logic implementation in IPS uses a global code generation. The separation of concerns solves one of the drawbacks exposed by modular layered protocol implementations: cross-layer information-sharing. The central protocol logic implementation has access to information from packet headers at all levels of the protocol stack. Packet processing code remains modular, but makes its information available to other layers. Furthermore, it enables bit-sized header field packing between layers. This is especially important when protocols are lightweight and only have a few header fields with a small number of flags and type fields.
3 Evaluation

We wish to demonstrate that IPS is able to significantly reduce the implementation complexity of protocol stacks, while showing acceptable resource requirements and performance, when compared to the hand-crafted code written by specialists in protocol stack implementation. To keep the example manageable, we only show the single-hop part of the Rime stack.

3.1 Lines of code

The primary metric of a domain specific language like IPS is ease of programming. It measures how well a DSL fits the problem domain from the point of view of supporting software development. In the case of protocol stack implementation, a DSL should offer meaningful and intuitive abstractions to support a straightforward mapping between protocol specification and implementation. This is a qualitative metric and is therefore hard to quantify, but our examples in Section 2 give an idea of the correspondence between the protocol stack specification and its implementation. A DSL should also relieve the programmer from expressing low-level details and hide them behind the high-level abstractions provided. This property shows in the lines of code for the implementations. To substantiate that IPS reduces the complexity of protocol stack implementation, we compare the Rime implementation written in IPS with the corresponding original implementation written in C, using the lines of code used to implement the protocol stacks as an indicator of the implementation complexity. Of course, this measure is affected by aspects not inherent to the programming language, but rather due to programming style and conciseness. Results have to be interpreted with care and minor differences would not indicate differences in the complexity which we intend to measure here. However, our results are clear enough to show a general trend and obvious differences.

Table 1. Lines of code for Rime implementations

<table>
<thead>
<tr>
<th>Protocol(s)</th>
<th>C</th>
<th>IPS</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>65</td>
<td>16</td>
<td>0.25</td>
</tr>
<tr>
<td>ibc</td>
<td>88</td>
<td>18</td>
<td>0.20</td>
</tr>
<tr>
<td>uc</td>
<td>91</td>
<td>22</td>
<td>0.24</td>
</tr>
<tr>
<td>stuc+ruc</td>
<td>97 + 72</td>
<td>39</td>
<td>0.23</td>
</tr>
<tr>
<td>polite</td>
<td>89</td>
<td>20</td>
<td>0.22</td>
</tr>
<tr>
<td>ipolite</td>
<td>89</td>
<td>3</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 1 lists the lines of code in the Rime implementation in C and in IPS. For Rime implemented in C, we count statements as lines of code, thus excluding comments and header files. Accordingly, the numbers for the implementation in IPS count IPS statements, excluding comments and module specification statements. We see that the numbers for IPS are considerably smaller than the ones for Rime’s original implementation, generally less than 25% of the lines of C code. In addition, we see that the code for ipolite is extremely short, because ipolite does not add substantially new functionality. In fact, ipolite is identical to the previously-defined protocol polite, only situated in other place in the protocol graph. While the C implementation duplicates a considerable amount of code, IPS can reuse the previous code and only overwrite the protocol name as shown previously.

3.2 Code footprint

Aside from source code size, we measure the code footprint of generated code, which is an important property for resource-constrained systems. Table 2 lists the code memory footprint in bytes of the Rime implementations in hand-crafted C code and IPS, both compiled for the COOJA simulator [9].

Table 2. Static memory footprint of Rime implementations

<table>
<thead>
<tr>
<th>Protocol(s)</th>
<th>C</th>
<th>IPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>4408</td>
<td>6852</td>
</tr>
<tr>
<td>ibc</td>
<td>5192</td>
<td>7408</td>
</tr>
<tr>
<td>uc</td>
<td>5492</td>
<td>7148</td>
</tr>
<tr>
<td>stuc+ruc</td>
<td>9160 + 8380</td>
<td>19322</td>
</tr>
<tr>
<td>polite</td>
<td>7312</td>
<td>8896</td>
</tr>
<tr>
<td>ipolite</td>
<td>7928</td>
<td>8896</td>
</tr>
</tbody>
</table>

The code footprint of IPS-generated code is bigger than the one from the hand-crafted implementation. The latter uses packet attributes instead of packet headers, and a general transformation module to transform packet attributes into packets with headers and vice-versa. Thus, packet processing code is separated from the proper Rime stack implementation. In our implementation, the packet processing C code resulting from our code generation is integrated with the protocol logic implementation, which increases the code footprint for each protocol. The memory footprint at runtime will also depend on the size of dynamically allocated buffers, which is determined by the programmer’s implementation decisions, and therefore not measured here.

3.3 Size of packet headers

Both memory consumption at runtime and packet processing time (as well as energy consumption) are affected by the packet size. As the amount of data depends on the particular application, we completely left it out from our measurement, but we have explained that IPS, by design, uses bit-oriented headers. In IPS-generated code, fields from different layers are presented as field handles specifying physical locations. The generated packet processing library is bit-oriented rather than byte-oriented, i.e., parsing
and marshaling functions can extract and construct packet fields crossing byte boundaries. Small header fields will be packed into the same byte, even when they belong to different protocols. This bit-oriented implementation saves considerable memory by efficiently using all available bits in a transmitted byte, and considerably reduces the size of the transmitted header, as we will exemplify by the protocols abc, ibc, uc and ruc.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>2-bit Packet type flag</td>
</tr>
<tr>
<td>ibc</td>
<td>2-bit Packet type flag</td>
</tr>
<tr>
<td>uc</td>
<td>2-bit Packet type flag</td>
</tr>
<tr>
<td>ruc</td>
<td>2-bit Packet type flag</td>
</tr>
<tr>
<td></td>
<td>16-bit Sender address</td>
</tr>
<tr>
<td></td>
<td>16-bit Receiver address</td>
</tr>
<tr>
<td></td>
<td>5-bit Packet ID</td>
</tr>
<tr>
<td></td>
<td>5-bit Retransmission counter</td>
</tr>
</tbody>
</table>

Figure 7. Header fields of selected Rime protocols

Lowest protocol in the stack (and thus at the beginning) is abc, which encodes its packet type (the routing to the upper layers) in one header fields requiring 2 bits. The ibc above equally requires one 2-bit field for the packet type, and adds an address field, the length of which we specified as 16 bits. Ibc's layered protocol uc again has two header fields which require 2 and 16 bits individually. More header fields follow in the upper layers of ruc/stuc: another packet type flag (2 bits), a packet ID (5 bits), a retransmission counter (5 bits), and more fields which we do not discuss further here.

Figure 8. Non-packed layered header of the Rime stack (fields byte-aligned).

With byte alignment for each protocol stack layer, these headers would have to be located as shown in Figure 8. Even more empty "padding" space is required when the target hardware uses word-alignment and a word size of 16, 32 or 64 bits. Figure 9 shows how IPS can align the header fields for different protocols without any padding requirements. The header fields we have shown for an abc/ibc/uc/ruc packet take 50 bits and are efficiently compressed into 7 bytes, whereas the byte-aligned version requires 9 bytes.

3.4 Performance

Another important metric is the performance of the generated code from source code of a DSL. Typically, latency and throughputs are the runtime performance indicators for a protocol implementation. At the protocol level, the time to transfer a complete packet involves the time taken for processing a packet, e.g., how long does an outgoing packet take for the first bit to send (latency), plus how long does it take for the remaining bits to send (the length of the packet divided by the throughput). As the throughput mainly depends on the hardware, we only measure the latency.

We measure and compare the runtime performance of the Rime implementation generated by IPS and the original Rime implementation by experiments. The application scenario is a sensor network with 25 sensor nodes in a square lattice for measuring temperature. One node acts as a sink node which collects all temperature information (4 bytes of application data) measured in the network. Each node runs a Rime protocol stack, exchanging packets with its single-hop neighbor by using the single-hop unicast protocol stack of Rime. Our test scenario is simulated by the Contiki simulator COOJA [9]. To make sure all the experiments work on the same input, we use 10000 already-collected temperature values. We measure the amount of time that is needed to process a packet for sending/receiving, by taking timestamps before calling and after returning from the Rime stack. We carried out all experiments on the same computer. Each experiment was repeated 3 times and the lowest value was taken.

Figure 10 and Figure 11 report the estimated average time for sending/receiving a packet, and how they develop over time in the test setup for both the IPS and hand-crafted implementation. We see that the execution time resulting from the generated and the hand-crafted code are approximately the same. This is not surprising, since the C code resulting from our code generation uses techniques similar to the hand-crafted implementation, i.e., bit-oriented packing to reduce the size of header. While our measurements are slightly slower, the reduction is not significant.

Figure 9. Bit-oriented Rime packet header generated by IPS (no alignment required).
4 Related Work

We are not the first ones to be interested in using high level formal descriptions to generate programs that deal with tedious tasks. In this section we reflect on how our work relates to, and in some cases builds on, earlier attempts.

Data description languages [8, 10, 11, 12] are modern approaches for describing data formats using dependent types. PacketTypes [8] is a data description language that uses types for packet descriptions. It introduces packet types to provide a programmatic description and automated generation of parsing functions and other tools for packet processing that can be rapidly adapted to implement packet filters. It provides only one basic type, i.e., bit, and type constructors for repetition and sequencing. In order to cope with data dependency, fields are allowed to have attributes that can be referred to in restriction clauses. It supports overlaying a packet specification in another specification's field — typically in the payload field of overlaid protocols. In our work we replace PacketTypes' ad-hoc notion of attributes with a richer dependent type system based on the Data Description Calculus (DDC), which is a formal system using dependent types to capture the principles of data description languages. We also allow for a richer set of basic types instead of just bits. In this way we can deal with more semantic constraints and consistency conditions that are beyond the scope of PacketType.

There has been a steady stream of research over the years on automating the generation of protocol implementations from specifications. Some of the early work decompose complex protocols into modules and enforce such design philosophy using language constructs for either ease of programming [13] or reusability and optimizations [14]. Others have focused on formal protocol specification for both verification and code generation. They primarily aim at making correctness verification as easy as implementation. The typical approach to this, taken by TAP [15], Teapot [16] and Promela++ [17], is to have two execution models: one for model checking, one for generating executable code. TAP [15] is effective to describe asynchronous message-passing network protocols. Teapot [16] has been designed for writing cache coherence protocols. Both of them heavily specialize for one particular protocol category and ignore the protocol construction handling. Promela++ [17] is a more closely related work, which provides explicit language mechanisms to encapsulate and compose protocol layers where the adjacent layers communicate neatly using FIFO message queues. However, the downside of this scheme is a time-consuming context switch between the communicating processes which could overburden the runtime memory. Promela++ does not support some necessary primitives, e.g., timers and memory allocation, thus its source code has to include blocks of C code when needed. To the best of our knowledge, no existing language covers both automatic packet processing and protocol overlaying so far.

5 Conclusions and Future Work

We present a language-based approach to network programming, a domain specific language to facilitate protocol stack implementation. Our prototype language IPS captures the core features for the implementation of protocol stacks. It provides high-level notations and abstractions and hides the intricacies of low-level details, e.g. packets, buffers, timers and so on. IPS thereby provides a unified framework for specifying protocols, composing protocols to protocol stacks and graphs. The high-level specification allows programmers to concentrate on the functionality of protocols rather than on the details of how to structure and optimize code for packet processing and protocol logic. It enables flexible composition of modules, making it possible to construct a new protocol stack by reusing existing modules instead of rewriting code. The internal compilation of IPS into C code allows the programmers to specify protocol stacks in a high-level language while ensuring good performance and high portability. As the preliminary experimental evaluation shows, IPS is able to achieve the high efficiency of compiled code in terms of energy consumption and memory usage.

An essential addition to the language IPS is an appropriate runtime system, which we plan to develop in the future. So far, some of the IPS features rely on particular runtime primitives (e.g., protothreads and timers) of Contiki [18], our primary testing platform. Decoupling IPS from Contiki can be achieved by giving different options at compile time to employ different runtime primitives. If
the stack implementation is part of a full operating system (OS), existing OS primitives can be used. For cases without a (suitable) OS on the target platform, our own specialized runtime system will be used as a loadable kernel module, providing a suitable concurrency model, efficient memory management, and operating system facilities needed by protocol implementation, like timers.

Furthermore, we would like to investigate cross-layer compilation techniques, optimizations addressing the protocol stack as a whole, in order to reduce the performance penalty for layering, and to produce more reliable and efficient code. As we have embedded our language in Haskell, rewriting rules with domain specific knowledge in the Glasgow Haskell Compiler (GHC) [19] to optimize the program would be another direction that we are going to study.

We intend to implement a series of practical protocols in the very near future, to explore the expressiveness of IPS, to guide further improvements and extensions towards a more full-fledged language, and then formally fix syntax and semantics of essential IPS abstractions.

References


