

IPS: Implementation of Protocol Stacks for Embedded Systems

Yan Wang
Halmstad University, Sweden
yan.wang@ide.hh.se

ABSTRACT

In both research and business enterprises dealing with sensor networks, the implementation of communication protocol stacks is a central issue. It has an impact on time-to-market, scalability, maintainability and on the possibility of making fair comparisons. Network software, in particular for small embedded systems with strict non-functional requirements, is complex, error-prone and with many possible bottlenecks. Our work addresses these issues using a language based approach. It aims at a programming language supporting abstractions suitable for the implementation of protocol stacks. Language technology in the form of runtime system, type system and compiler transformations can then be used to generate efficient implementations.

1. INTRODUCTION

In both research and business enterprises dealing with sensor networks, the implementation of protocol stacks is a central issue. Implementing protocol stacks is tedious, error-prone and time-consuming. It is even more so when targeting small embedded systems, e.g., wireless sensor network nodes, which usually have additional, non-functional constraints. Thus, implementations have to minimize energy consumption, memory usage as well as other computation resources. In order to improve on time-to-market, scalability, maintainability and product evolution, even programming efficiency is relevant.

To make design and evaluation easier, protocols are normally engineered using layers as modules. But modularity is one of the chief villains in attempting to obtain good performance due to the large overhead involved in interfacing between modules. Therefore, there is an enormous potential payoff for automatizing performance optimizations when implementing protocol stacks. This opens opportunities for a language based approach in the form of a domain specific language (DSL) [9].

DSLs offer appropriate constructs and abstractions with natural vocabulary in the problem domain leading to a good fit between the language's features and the domain engineer's knowledge and experience. Also, domain-specific optimizations can be automatized: a compiler that translates constructs that have a very specific meaning is in a better position to do more specific optimizations. Moreover, DSLs enforce a good implementation discipline by restricting the design space. This can also be used by the compiler for optimizations that tackle the entire program — in the case of protocol stacks this could be cross-layer optimizations. Last but not least, DSLs enable more properties of programs to

be verifiable, since DSLs capture the semantics of the application domain more precisely, i.e., no more and no less.

1.1 Thesis Statement

We propose to design *Implementation of Protocol Stacks* (IPS), a DSL to facilitate protocol stack implementations targeting resource constrained embedded systems. IPS code will be compiled into C code for real-world use and have correctness, efficiency and maintainability in focus. We have decided to delimit the scope of the computations by addressing asynchronous message-passing protocols on single processor distributed real-time embedded systems.

IPS will address the protocol stack as a whole to enable cross-layer compiler optimizations and address the resource usage by means of a suitable runtime system. There are four criteria for measuring the success of IPS.

Performance: Generate C code which do not have worse measures than the hand-crafted code in terms of the expenditure of computer resources, e.g., energy and memory.

Robustness: Hide the intricacies of low-level details by using high-level strongly typed disciplines that allow consistency properties and type-correctness to be checked.

Ease-of-programing: Offer a rich set of meaningful and intuitional data and control constructs that suffice to specify a wide variety of protocol stacks – to support straightforward mapping from protocol specifications.

Usability: Impose constraints on the structure and interface of protocol stacks to enforce good implementation disciplines based on accumulated experience.

1.2 Expected Contribution

IPS is expected to have three main contributions. (a) IPS allows the programmers to specify network protocol stacks in a high-level language while retaining the efficiency of compiled code in terms of energy consumption and memory usage. (b) IPS provides a unified framework as well as an appropriate but restricted suite of notations and abstractions supported by its runtime system, allowing programmers to concentrate on the functionalities of protocols rather than details of arranging code. (c) IPS enables flexible composition of modules in different levels, making it possible to construct a new protocol stack by reusable modules with writing few new codes.

2. RELATED WORK

There has been a steady stream of research over the years addressing the problems of protocol implementation. Previous work can be said to take one of two approaches.

System based approaches provide systematic software architectures for constructing protocol stacks. [10, 8, 6] aim at better expressing the modular structures and protocol layer composition to achieve efficient and concise implementations. But their applicability to sensor networks is restricted by their complexity in terms of large memory usage. uIP [5] offers a non-layered TCP/IP stack implementation with a very small code size and RAM requirements. However, its good performance mainly depends on only supporting the bare necessary functionalities which is not general enough. In all these cases, the optimizations have to be applied manually, which is a challenge task even for the skilled protocol implementers. On the contrary, domain-specific optimizations in IPS is supposed to be automatized.

Language based approaches have been studied extensively during the past decade. They deal with the limitations of the system based approach by defining new notations to exploit protocol-specific behaviors. Some DSLs decompose complex protocols into modules and enforce a design philosophy using language constructs for either ease of programming [11] or reusability and optimizations [1]. Others [13, 4, 3] have focused on both verification and code generation — compilers can generate models for model checker as well as executable code. It highlights the power of a DSL linked with formal methods. IPS differs from the above approaches, as it focuses on cross-layer optimizations with energy and memory efficiency in mind, which makes it more suitable for embedded systems.

3. A DSL FOR IMPLEMENTATION OF PROTOCOL STACKS

3.1 Problem Domain and Proposed Solution

A Modular Architecture: Complex systems like network protocols are normally implemented in modular fashion to reduce complexity, make reuse and configuration possible. Inspired by Morpheus[1], we propose to offer fine-grained modules at three granularity levels to avoid the surplus functionalities from coarse-grained modules. Similarly to Morpheus, IPS will support some desirable features. For example, each module in the lowest level – *micro-protocol* level has to be constrained to one of three *Shapes*, i.e., *multiplexor*, *router* and *worker* corresponding to the type of fine tasks. This constraint contributes to more predictable and reusable code for the compiler, since *Shape* distinguishing maximizes the commonality of different *micro-protocols*.

IPS differs from Morpheus in significant ways. Firstly and most remarkably, IPS will employ compiler techniques to integrate all modules belonging to one protocol stack into a tightly coupled compiled-code to eliminate the overhead across modular boundaries and achieve small code footprint. IPS will also pay more attention to energy consumption and memory usage by incorporating a suitable runtime system. IPS will be more expressive than Morpheus. For example, UDP can be implemented in IPS by composing several *micro-protocols*, whereas this can not be done in Morpheus. IPS focuses on protocol stacks, where Morpheus is geared for protocols. Finally, the implementation of Morpheus has been left incomplete, but IPS will be a complete DSL with its type system, compiler and runtime system.

A Type System for Packets: Types have remarkable highlights which can be used for dealing with complex data

like network packet. First, type-directed programs retain the detail domain specific information and deeper semantic properties, e.g., the physical layout of packets and user constraints. Second, types describe both the syntax and semantic properties of data formats which can be dually interpreted into canonical in-memory representations and parse descriptors. Third, types enable to build up strong intuitions from high-level perspective that avoid low level tedious code.

Similarly to PacketTypes [12], we propose to compile type descriptions of packet into parsers which can marshal packets into C data structures and generate detected error information automatically. Additionally, we plan to enable static analysis, e.g., predicting the stack and heap requirements of packet processing. We also plan to enable partial evaluation to reduce the runtime packet processing. e.g., split a packet related source code into static and dynamic parts to make it specialize better.

A Concurrency Model: By introducing a language we can choose a suitable computation and concurrency model. For example, we might want to have threads and blocking primitives [5] or reactive objects and events [14]. we will base our choice on experiments with both and the choice will have an impact on the runtime system of our language.

Protothreads [5] are a lightweight stackless type of threads which allow concurrent program to be written in a sequential fashion. State machines can be expressed implicitly by using protothread constructs, e.g., `Pt_begin`, `Pt_wait` and `Pt_end`, which can be compiled into C functions as [5] defined.

Reactive objects [14] provide a consistent model of event-driven concurrency. It can relieve objects from transparent blocking and protect state consistency automatically, since every object is an autonomous unit of execution and its executable code is defined in terms of non-blocking methods. A protocol can be represented by a reactive object consisting of its local state and a set of methods describing its behavior.

Region-based Memory Management: Explicit dynamic memory management interface is flexible and efficient but unsafe. In contrast, automatic dynamic memory management (garbage collection) can improve code quality, but it often involves runtime overhead in terms of processor overhead and additional memory requirement.

Region-based memory management [15] provides a solution to this dilemma. It is able to get the safety without paying the runtime cost of garbage collection by freeing memory when dynamic context exits. In the case of protocol stack implementation, the *region* can be session-related packet pool: every memory request for packet can be allocated from the corresponding packet pools based on sessions, and a packet pool will be automatically released when its associated session terminates. This discipline not only avoids fragmenting memory into a patchwork of holes of different sizes, but also guarantees the safety of memory since memory deallocation is not allowed to be expressed explicitly. Moreover, it makes consistency properties checking possible, e.g., packet *p* can not exist after session *s* terminates.

Soft Timers: Timers are needed by protocol implementations that allow actions to be triggered after a given time interval has elapsed. However, conventional timers invoke designated event handlers periodically in the context of a hardware interrupt, which cause high CPU overhead due to saving and restoring state as well as cache and *Translation Lookaside Buffer* (TLB) pollution. Therefore, they are not optimal for the resource constrained embedded system in

which energy efficiency is one of major concerns.

We propose to employ soft timer [2] to reduce the hardware interrupt penalty. It is an operating system facility that checks pending timer expiration across a large amount of useful states where a system very frequently reaches, e.g., at the end of executing a hardware device interrupt handler. We plan to implement soft timer in the runtime system for IPS. It should only be invoked by a IPS construct, which can be expressed as an event handler function coupled with a time interval.

3.2 Methodological Approach

The methodological approach will be centered around the following main streams: (a) Analyze and experiment with well-known implementations targeting resource constrained embedded systems. (b) Extract common program patterns including operations, data structures and states for syntactic constructs design. (c) Capture the static and dynamic semantics for defining an abstract machine for protocol stack implementations. (d) Exhibit the implementation expertise for protocol-specific optimizations by taking advantage of well-known implementation techniques that address standard bottlenecks [16].

3.3 Current Status

So far, we have built a modular framework for protocol stacks. A complex protocol stack can be constructed by reusable protocol modules, which can be composed by micro-protocol modules. Modules communicate with each other by function calls where messages are usually involved as arguments. In this framework, thread-per-message strategy is enforced. Preliminary experimental results suggest that this framework is both expressive and precise enough to describe a wide range of protocol stacks.

We implemented a parser generator for packet types according to the semantics of *Data Description Calculus* (DDC) [7], which is served as the semantic foundation for data description languages, e.g., Packet Types [12]. This parser generator can interpret packet format descriptions in DDC to parsing functions which produce both C representations of packets and parse descriptors pinpointing errors of the original raw buffer data. The integration of packet parsing functions with protocol stack implementations is in progress.

To illustrate the preliminary design of IPS, we show some examples. We start with two primitive constructs, i.e., $\langle | \rangle$ and $\langle \rightarrow \rangle$, which are used to combine protocols into layer and combine layers into stack respectively. For example:

```
Stack TcpIp = Tcp<|>Udp<|>Icmp<→>Ipv4<|>Ipv6
```

IPS typechecks operators for each composition to ensure that the entire composition is type safe. Protocols in IPS is composed by fields and micro-protocols. For example:

```
Protocol Ipv4 = {
  State      Ipv4State;
  PD         Ipv4Packet;
  Initialize Ipv4Ini(...);
  Send       Ipv4Send(...);
  Receive    Ipv4Recieve(...);}

```

Field `Ipv4State` typed `State` and `Ipv4Packet` typed `PD`, which hold the information of Ipv4 states and the description of Ipv4 packet type. The following three items are micro-protocols. Since they have identifiable interfaces based on their types, they can be developed individually and integrated easily at compile time. The detail of Ipv4 packet type description is shown below. Besides the regular structure, the necessary constraints appear in a `where` clause, which

identify an Ipv4 packet. It will be compiled into parsing functions automatically that can be used in `Ipv4Recieve` as a default packet parser. Again, all the constructions and operations will be type checked at compile time.

```
Ipv4Packet := {byte4 version;
               byte4 ihl;
               .....
               bytestring options;
               bytestring payload;}
where {version.value = 0x04;
      options.numbytes = ihl.value * 4 - 20;}

```

4. REFERENCES

- [1] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, 1993.
- [2] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, 2000.
- [3] A. Basu, J. G. Morrisett, and T. von Eicken. Promela++: A language for constructing correct and efficient protocols. In *INFOCOM*, pages 455–462, 1998.
- [4] S. Chandra, B. Richards, and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *PLDI '96*, pages 237–248, New York, NY, USA, 1996. ACM Press.
- [5] A. Dunkels. *A Language-based Approach to Protocol Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, Swedish Institute of Computer Science, 2007.
- [6] D. Engler, D. Wallach, and M. Kaashoek. Design and implementation of a modular, flexible, and fast system for dynamic protocol composition. Technical Memorandum TM- 552, MIT, May 1996.
- [7] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *POPL '06*, pages 2–15, New York, NY, USA, 2006. ACM Press.
- [8] M. G. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.
- [9] P. Hudak. Modular domain specific languages and tools. In *The Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
- [10] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 1991.
- [11] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the prolog protocol language. In *SIGCOMM*, pages 3–13, 1999.
- [12] P. J. McCann and S. Chandra. Packet types: Abstract specification of network protocol messages. *SIGCOMM Comput. Commun. Rev.*, 30(4):321–333, 2000.
- [13] T. M. Mcguire. *Correct Implementation of Network Protocols*. PhD thesis, The University of Texas at Austin, 2004. Supervisor-Mohamed G. Gouda.
- [14] J. Nordlander, M. Jones, M. Carlsson, D. Kieburtz, and A. Black. Reactive objects. In *ISORC '02*, 2002.
- [15] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.
- [16] G. Varghese. *Network algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Pub, 2004.

Biographical Sketch

Yan Wang
Center for Research in Embedded Systems
Halmstad University

Box 823, SE-301 18
Halmstad, Sweden
`yan.wang@ide.hh.se`

Expected Date of Dissertation Submission: 2011

Advisors and Affiliations:

- Ph.D. Verónica Gaspes
Associate Professor in Computer Science
Center for Research in Embedded Systems
Halmstad University
`veronica.gaspes@ide.hh.se`
- Ph.D. Dimiter Driankov
Professor in Computer Engineering
AASS Research Center
Örebro University
`dimiter.driankov@tech.oru.se`
- Ph.D. Thorsteinn Rögnvaldsson
Professor in Computer Science
Intelligent Systems Lab
Halmstad University
and
Professor in Learning Systems
AASS Research Center
Örebro University
`thorsteinn.rognvaldsson@ide.hh.se`