Technical Report IDE0746, June 2007

Implementation of Protocol Stacks

PhD Research First Year Report

Yan Wang

Centre for Research on Embedded Systems

School of Information Science, Computer and Electrical Engineering

Halmstad University

Box 823, S-30118 Halmstad, Sweden

Contents

1 Introduction **3**
- 1.1 Introduction to Network Protocol Software . . . 3
- 1.2 Overview of the IPS Project . . . 4
- 1.3 Organization . . . 4

2 The IPS Project **4**
- 2.1 Motivation . . . 4
- 2.2 Requirements and Challenges . . . 5
- 2.3 Phases . . . 6

3 Methodology **7**
- 3.1 What is a DSL? . . . 7
- 3.2 What is a DSEL? . . . 8
- 3.3 When to develop a DSL? . . . 9
- 3.4 How to design a DSL? . . . 9
- 3.5 How to implement a DSL? . . . 11
- 3.6 How to evaluate a DSL? . . . 11

4 Related Work **12**
- 4.1 System Support Approach . . . 13
  - 4.1.1 Ensemble . . . 13
  - 4.1.2 X-kernel . . . 15
  - 4.1.3 The Linux TCP/IP Implementation for Embedded Systems 16
  - 4.1.4 UIP . . . 17
- 4.2 Language Based Approach . . . 18
  - 4.2.1 FoxNet . . . 19
  - 4.2.2 TAP . . . 20
  - 4.2.3 Promela++ . . . 21
  - 4.2.4 Prolac . . . 21

5 A Framework for Protocol Stacks **22**
- 5.1 Background . . . 23
- 5.2 Morpheus . . . 23
- 5.3 Our Approach . . . 26
  - 5.3.1 Modular Development . . . 26
  - 5.3.2 Built-in Mechanisms . . . 28
- 5.4 Examples . . . 29
  - 5.4.1 A Simple Virtual Protocol . . . 29
  - 5.4.2 A Transport Protocol . . . 29
  - 5.4.3 A Protocol Stack . . . 30

6 Conclusion and Future Work **30**

Abstract

Implementation of network protocol stacks is time-consuming and error prone due to its complex and performance-critical nature. The project *Implementation of Protocol Stacks* (IPS), addresses this problem by developing a high-level statically typed language to facilitate the efficient implementation of network protocol stacks. This report presents the accomplishments for the first year of PhD student Yan Wang working in the IPS project. The research started with two studies, conducted to gain a clear understanding on the methodology of developing a domain specific language and approaches of implementing a protocol stack efficiently. The first attempt in practice is a framework for the protocol stack implementation written in Haskell, whose preliminary result shows a working framework for composing protocol stacks with modularity, generality, and flexibility.

1 Introduction

1.1 Introduction to Network Protocol Software

A network protocol [PD03] is a specification of packet formats and rules of operation for governing interactions between communicating peers, e.g., data representation, authentication, error detection, congestion control. The implementation of a network protocol is how the software performs the functionalities of the protocol in a given platform. In programming terms, packet formats are implemented as data structures with a specific machine-independent encoding, and protocol rules of operations are typically expressed in terms of a set of states and actions.

To make design and evaluation easier, an individual protocol often has one single purpose. Protocols with various tasks can work together by layering them into a protocol stack [PD03], which is a particular software implementation of a protocol suite. In practical implementations, protocol stacks are often divided into three sections for media, transport, and applications, as well as two interfaces: media-to-transport interface and application-to-transport interface.

Network protocol stack implementation is time-consuming and error prone due to its complex and performance-critical nature. It should hide the details of the underlying hardware, recover from transmission failures, ensure that messages are delivered to the application processes in the appropriate order, and manage the encoding and decoding of data. Because of complicated functionality in a distributed concurrent program, correctness is hard to achieve, and this situation is exacerbated by the additional requirement of high performance. Therefore, it is worth researching on how to make network protocol implementations both easier and more likely to be correct without compromising high performance.

1.2 Overview of the IPS Project

The IPS project, is devoted to protocol stack implementation aiming at developing a high-level statically typed language addressing protocol stacks with correctness, efficiency and maintainability in focus. Its research goals are defining high-level constructs and notations, organizing program design, facilitating program writing and reading, hiding the intricacies of low level details, as well as enabling performance and robustness, which demand a special combination in the domain knowledge and the language development expertise.

Our method is to provide a programming language with specialized constructs for protocol stack implementations. In this way we hope that we can offer a means to organize protocol stacks by composing layers, as their specifications or reference implementations are understood. We will apply regular compiler optimizations and protocol-oriented compiler optimizations, as well as program transformations techniques to achieve efficient implementations. The envisioned language will also provide constructions for typical services, like multiplexing and timers, and will then be used as a kernel. Finally, it will be improved by developing more tools for analysis, simulation, testing, profiling and verification.

1.3 Organization

The rest of this report is structured as follows. Chapter 2 briefly describes the IPS project, and Chapter 3 explores the methodology that will be used in this project. Chapter 4 investigates the state-of-the-art in protocol stack implementation domain. Chapter 5 shows our first attempt in practice, which is a framework for the implementation of protocol stack written in Haskell. Finally, Chapter 6 concludes and provides some directions for future work.

2 The IPS Project

2.1 Motivation

This project is motivated by the fact that many companies are re-implementing well-known infrastructure protocols as well as implementing new application protocols due to the fact that network software is changing in response to new network hardware, new application requirements, the integration of previously disjoint communication systems, and the changing scale of networks. Thus, developing and modifying protocols is an ongoing process.

However, implementing network protocols is error-prone and time-consuming due to its complex and performancecritical nature. The specification of most modern protocols is enormous and mapping the specification into code is not straight-forward that makes correctness is hard to be achieved. Many mature techniques enable optimizations to make protocol code more efficient but also tend to make it more complicated as well as introduce more error sources.

Network protocols are typically programmed in traditional languages like C. However, these languages are not well suited for the purpose of the network protocol implementation. First, language abstractions are not provided for manipulating specific network protocol behaviors. Some key elements, e.g., buffers, timers, threads, and messages, are coded explicitly by functions in libraries. The alternative is dedicated constructs that are more straight-forward and allow consistency properties and type-correctness to be checked. Second, the optimizations at compile-time for a traditional languages offer limited opportunities related to protocol stacks. The compiler is only amenable to conventional optimizations, and protocol-specific optimizations in general can only be achieved manually. Third, the flexibility to control memory consumption and hardware leads to more opportunities for mistakes. Furthermore, the few constrains imposed on the programmer together with the involved low level programming leads to fairly unreadable code.

All these factors make network protocol implementations in conventional languages to be hard to read, hard to write, and therefore hard to maintain and debug. Domain specific languages (DSL) [Hud97] can offer a solution to all these problems. Such a language usually supports an appropriate but restricted suite of notations and abstractions explicitly while hiding common program patterns implicitly. With domain specific knowledge in mind, DSLs can support domain-oriented compiler optimizations, constrain enforcement, transparent multiprocessing, language-level debugging as well as enforce systematic reuse in a graceful way. Since DSLs capture the semantics more precisely, they enable more properties for program verification. Moreover, the offered natural vocabulary makes the code more readable, writable, and therefore maintainable. DSLs are being successfully used in a variety of areas including financial products [Lex07], communication services[CHR+03], hardware device design[BCSS98], cryptographic algorithms[Cry07] and network protocols[Mcg04, BMvE98, CRL96].

For these reasons, there is a potentially large payoff for investing in a special purpose language to provide powerful program development support for protocol stack implementation by either making bottlenecks together with their solutions more available or realizing infrastructure automatically. Optimally, this language should be robust, expressive, and easily usable, so that it can be integrated into the protocol development and standardization processes.

2.2 Requirements and Challenges

The objective of our project is to develop a high-level statically typed language tailored to the implementation of protocol stacks. It is a challenging task, requiring the special combination of research skills in software engineering, programming languages, language technology, systems and networks.

The first challenge is that protocol stacks are difficult to construct due to their complex and performance-critical nature, e.g., handling complicated control flow, ensuring the reliability, availability and security. Some principles such as information hiding and encapsulation should be conformed in each layer of the protocol stack, and in the meanwhile the per-layer cost should be minimized

as far as possible. Moreover as low-level programming is involved, they must interface with bare hardware in network devices, and at the same time implement complicated real-time algorithms.

Another challenge is that it is difficult to generalized truly efficient constructs and abstractions to accommodate a wide range of protocols. Since protocols have very different characteristics that vary so greatly in purpose and sophistication: from connection-oriented to message-oriented, from synchronous to asynchronous and from reliable to unreliable. In the meantime, it is also not easy to arrange the low-level and complex protocol formats without losing domain-specific information.

Finally, developing a new programming language is a complicated process, since the complex language processors must be involved in. Such as in terms of implementing a compiler, semantic analysis, code generation, code optimization, type checking and error messages are challenge tasks.

In a word, IPS project demands a good language designer as well as a good language implementer together with the specific knowledge of network software in mind.

2.3 Phases

Briefly speaking, there are three phases involved in this project. Initially, we concentrate on the study of the philosophy and the understanding of the application domain. It includes exploring the possible approaches to be used, investigating related work, analyzing typical protocol stack implementations, identifying the abstractions used and characterizing their semantics. The outcome of this phase will be the language abstractions that can automatically supply the predictable code and data structures appropriate for a given protocol together with the design disciplines in programming.

The next phase is the implementation of a strong type programming language. The outcome of this phase will be a prototype of the language together with its compiler which has capability to captures those protocol abstractions and disciplines from the first phase, and generate intermediate code in a conventional programming language. Moreover, a series of experiments should be done.

At a further phase, the research will focus on protocol-oriented compiler optimizations to achieve high performance by investigating the specific common behaviors of protocols. The output of this phase will be a stand alone programming language which generates efficient code and some other incorporated tools, such as debugger and profiler. Furthermore, this language should be more portable to different kinds of computer architectures.

3 Methodology

3.1 What is a DSL?

A *Domain Specific Language* (DSL)[Hud97, Dsl07] is a programming language dedicated to a particular application domain or problem. It has natural vocabulary for concepts in domain expertise and is more declarative than imperative. It usually only offers an appropriate but restricted suite of notations and abstractions, and has ability to develop software in the domain quickly and effectively. Since the offered guidelines and built-in functionalities enforce the re-use as well as optimizations by hiding common program patterns implicitly, a DSL provides a concise way to write better programs with less code. The domain engineer is the person supposed to use a DSL rather than the skilled programmer, who only needs to concentrate on what to compute but not how to compute.

The use of DSLs is not new. Classical and well-known examples include SQL for database queries, VHDL for hardware design and BNF for syntax specification. The Cryptol language[Cry07] is a relative new DSL. As a profitable commercial product, it enables constructions of cryptographic software with ease, reliability, and high assurance. Another example is LexiFi[Lex07], which provides a complete solution for designing, pricing, analyzing, and processing complex financial products. It works well for a wide customer community: structurers, salespeople, quantitative analysts, risk managers and middle office professionals as well as in various fields: investment banks, asset managers and private banks.

A DSL can offer the domain-specificity in better ways [MHS03] than a *General Purpose Language* (GPL) with an application library. Firstly, the domain specific notations of DSLs are usually beyond the limited user-definable notations offered by GPLs. Secondly, DSLs offer appropriate constructs and abstractions with natural vocabulary in problem domain, which can not always be mapped straightforwardly to functions or objects in GPLs. Finally, since DSLs capture the semantics of the application domain more precisely, i.e. no more and no less, they enable more properties for programs to be verified.

Figure 1 [Hud97] roughly shows the difference between DSLs and GPLs in terms of the payoff of software development. The point is that the initial cost of a DSL's design and development is high, although it will gain in productivity and reduce maintenance costs later. Firstly, designing a DSL can be fairly difficult. It is more likely to be designed from scratch, which might lead to incorrect design especially at the first time. And this kind of startup would cost more than once during the life-cycle of software system in the case of the design is totally wrong. Secondly, developing a DSL is costly, since the complex language processor must be implemented. Such as, code-generation, optimization, type-checking and error messages are the challenge tasks in the implementing the compiler for a DSL. Finally, following the evolution of a DSL is not easy. A DSL will keep growing in the procedure of development, since new modules and data structures would be added. As a result, all of the difficulties associated with those evolutions will be invoked, e.g., redesign grammar, rewrite parser

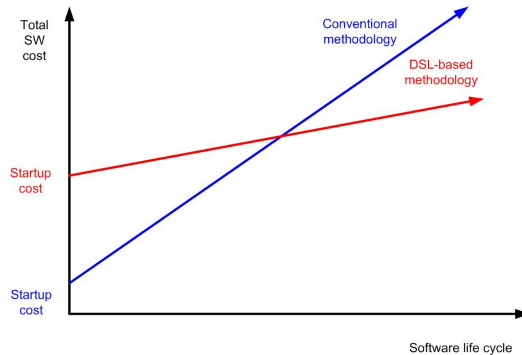

Figure 1: The Payoff for Software Development

and compiler.

3.2 What is a DSEL?

One of the techniques which can both lead to the effective use the methodology of DSL and avoid the great pains in startup stage is *Domain Specific Embedded Language* (DSEL). As [Kam98] argued,

> "Embedding is the process of implementing a language by defining functions in an existing "host" language; the host language with these added functions is the new language, so that the new language has all the power of the host language."

For example, in the case of Java, a DESL can be built by using Java classes and methods.

Instead of bothering with non-domain-specific aspects, a DSEL borrows most of the design decisions from the host language, it can reuse syntax and semantics and can inherit compiler and tools. With such infrastructure, a DSEL would only concentrate on semantic issues. Therefore it is relatively easy to do, and as powerful as a DSL in traditional way.

Another advantage of DSEL is that it has all the power of the host language. In other words, we can gain access to the characteristics of the host language which normally is a fully fledged programming language. For example, Lava[BCSS98] gets to use higher-order-functions from its host language, Haskell, to describe circuits containing look-up-tables easily, in contrast with VHDL descriptions of such circuits tend to be long and hard to read. This advantage also overcomes one of the significant drawbacks of DSLs. Since a DSL only focuses on domain-specific features with the assumption that other features are not needed, its design tends to be weak and poor. In the case of this DSL achieves widespread use, it would be too late to know some new programming features are necessary.

3.3 When to develop a DSL?

To aid the general DSL development, a systematic survey[MHS03] identifies a common situation suitable for designing a new DSL from the technical point of view. A number of decision patterns are defined as follows from the experience of successful developed DSLs.

Notation If the required domain specific notations can not be offered by GPLs with an existing library, it is possible to offer such notations explicitly by a DSL.

Task automation If programming tasks are tedious with the same patterns, it is possible to generate the repeat code automatically by an application generator of an appropriate DSL.

Data structure representation and traverse If the complicated data structures are hardly written and maintained, it is possible to express them easily and reliably as well as their traverses by a DSL.

AVOT If domain specific analysis, verification, optimization, and transformation of application programs written in a GPL are usually not feasible, it is possible to make the source code pattern well-defined and less complexity by a DSL.

Interaction If the specification of input is complicated and repetitive, it is possible to supply the text or menu based interaction by a DSL.

Moreover, the GUI construction is usually handled by DSL as well as the configuration and adaptation of system.

From the business intent point of view, when to develop a DSL also depend on the size of the user community, and the potential economical benefit it can bring.

3.4 How to design a DSL?

As [Con04] argued that the design of a DSL is a very subjective process but has a lot of similarities with a craft. To spread the DSL approach more widely, they outline the following systematic steps in designing a DSL from a notion of a program family, which can be used to extract common program patterns for syntactic constructs design and exhibit the implementation expertise for domain-specific optimizations. After identifying a program family, developers can naturally exploit a library and suggest domain specific operations, values and states for defining a domain abstract machine. Then the main ingredients formed by the abstract machine and the common program patterns lead to the design of a DSL.

- Identify a program family
  It is an intuitive process to identify a program family, which is a set of related programs with enough similar characteristics. A study on the commonalities shared in this set and a study of the scope of computations are involved in. Normally, there is a large set of programs devoted to a particular domain. If the scope of computations is too wide, it would lead to a general purpose language. To delimit the scope of computations, two strategies are suggested. It could be either a specific class in horizon or in vertical, e.g., only address protocols in transport layer or only focus on the TCP/IP suit in protocol implementation domain. Moreover the specific analysis of a program family can lead to identify repetitive program patterns, which can be abstracted into syntactic constructs.

- From a program family to a library
  A program family naturally leads to the development of a library. It includes the identification of domain specific operations, which are corresponding to the re-usable software components abstracted over commonalities. It also includes versions of operations, variations of implementations, and parameterizations, which are translated from variabilities. And the objects manipulated by the library functions contribute to identify domain specific objects.

- From a library to an abstract machine
  The abstract machine defines the run-time model for a DSL, which is a combination of a domain specific set of instructions and states. Library functions explicitly or implicitly pass arguments to represent some notion of states, which not only models the run-time environment, but also ensures the safety of some instructions. The common domain specific operations in the library contribute to identify instructions as well as states. Domain specific instructions are similar to library operations in terms of dynamic semantics, but they are generated by a compiler or invoked by an interpreter, rather than directly used by a programmer.

- From an abstract machine to a DSL
  After define the abstract machine, actions are staged into two categories: one is the compiler actions which correspond to the static semantics of a language, the other is the run-time actions which correspond to the dynamic semantics. The designer can handle these two kinds of actions respectively, e.g., check the accessibility of register statically in the case of device driver example, as well as encapsulate the buffer overflow run-time check into each function systematically by the compiler. Finally, domain specific optimizations are decided by exhibiting the implementation expertise according to the analysis result of the program family.

3.5 How to implement a DSL?

[MHS03] also explores the following possible approaches for implementing a DSL.

- Interpreter
  DSL constructs are recognized and interpreted using a standard fetch-decode-execute cycle. It is appropriate for languages having a dynamic character or if the execution speed is not an issue. The DSL implemented by this approach can be easily extended and have great control over the execution environment.

- Compiler
  DSL constructs are translated into constructs in base language constructs and library calls. Normally high-level programming languages come with a compiler. The DSL implemented by this approach can be statically analyzed.

- Preprocessor
  DSL constructs are translated into constructs in the base language. Macros and subroutines are typical language extension mechanisms used for DSL implementation. The DSL implemented by this approach can be limited by the processor of the base language.

- Embedding
  DSL constructs are embedded in existing GPL by defining new abstract data types and operators. The DSL implemented by this approach is able to inherit most of the design decisions from host language, but can also be limited by the syntax of its host.

- Extensible compiler or interpreter
  GPL compiler or interpreter is extended with domain specific optimization rules or code generations. Interpreter is easier to be extended than the compiler.

- Hybrid
  A DSL is implemented by combining some of the above approaches.

3.6 How to evaluate a DSL?

Furthermore, [Con04] suggests the following three criteria to evaluate a DSL:

- Performance
  As a critical measure, performance assesses the value of a DSL. A DSL should demonstrate that it enables optimizations that are beyond the reach of compiler for GPLs, and it does not entail any performance penalty.

- Robustness
  It is significant to demonstrate that a DSL is able to detect bugs as early

as possible during the development process. Mutation analysis is one of the approaches to assess robustness. It measures how many errors would be detected statically or dynamically by introduces mutation errors with correct syntax.

- Conciseness
  Since the syntax of a DSL may differ from that of a GPL, the comparison between DSL programs and the equivalent programs written in a GPL is not easy. Some comparison can distort measurements, such as counting the number of lines or characters. From the experience of previous works, the number of words is a meaningful measurement.

Of course, we can also treat a DSL just as a programming language and evaluate it in a regular way, e.g., ease of programming, abstraction level, readability, maintainability, and usability.

4 Related Work

As we mentioned, in the initial phase of the project, we concentrate on understanding the application domain for exploring the possible approaches to be used in our project, which should lead to efficient implementation. This chapter describes our survey on several existing systems for implementation of protocol stacks.

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 the following two approaches: the system based approach and the language based approach.

The system based approach provides systematic software architectures for constructing protocol stacks. [HP91, Hay98] aim at better expressing the modular structures and protocol layer composition to achieve concise and efficient implementations. Although they work well in the intended environments, their applicability to embedded systems is restricted by their complexity in terms of the large code size and runtime footprint. uIP [Dun07] offers a non-layered implementation of the TCP/IP protocol stack intended for memory-constrained embedded systems. It provides the necessary protocols for Internet communication, 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 almost all these cases, the optimization has to be applied manually, which is a challenge task even for the skilled protocol implementers. Moreover, they only provide runtime support and have been coded in a traditional language, which suffers from a lack of protocol-specific knowledge.

The language based approach has been studied extensively during the past decade. It intends to 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 such design philosophy

using language constructs for either ease of programming [KKM99] or reusability and optimizations [AP93]. Others [Mcg04, CRL96, BMvE98] have focused on both verification and code generation, which highlight the power of a domain specific language linked with formal methods. Their compilers can generate models for the model checker as well as executable code. For example, TAP [Mcg04] is effective to describe asynchronous message-passing network protocols. Teapot [CRL96] has been designed for writing cache coherence protocols. Both of them heavily specialize for one particular protocol category but ignore the protocol construction handling. Promela++ [BMvE98] 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.

4.1 System Support Approach

Many operating systems support abstractions for encapsulating protocols. For example, to accommodate differences in different protocol layers, Berkeley Unix [Ste93] defines three different interfaces: driver to protocol, protocol to socket, and socket to application. Such abstractions are useful because they provide a common interface to collection of dissimilar protocols, thereby simplifying the task of composing protocols.

But not all operating systems provide explicit support for implementing network protocols. Systems like uIP[Dun07], move responsibility for implementing protocols out the kernel, they view each protocol as an application that is implemented on top of the kernel, and provide no protocol-specific infrastructure.

There are other approaches try to better express the modular structure and composition of protocol layers. Some of them very broadly follow the standard network layers, e.g., x-kernel[HP91]. The others deconstruct traditional protocols into building blocks by themselves, e.g., Horus[RBM96] and Ensemble[Hay98]. Those efforts demonstrate that an implementation methodology is more suited to the task at hand can help build cleaner and more robust implementations and can do so without exacting performance penalties.

4.1.1 Ensemble

Ensemble[Hay98] is a reliable group communication toolkit developed at Cornell University and Hebrew University of Jerusalem. It is implemented in the Objective Calm programming language, which is a dialect of ML[MTH90].

It provides a very flexible framework that relies heavily on the layering model: high-level protocols are implemented by composing a stack of tiny simple protocol layers. Each Ensemble layer has its own local state, the header it places on messages, and the handler for communicating with adjacent layers. Layers interact with the environment only through the event communication which guarantees that the behavior of a stack of layers is completely described through the event communications and the updates of the local state in individual layers.

Ensemble extracts a small number of common sequences of operations that occur in protocol stacks into event traces as the basic unit of optimization. For each event trace, a condition is identified which must hold for enabling the trace, and the event handler executes all the operations in this trace, e.g., sending and receiving messages, transferring protocol states, and managing reconfigurations. Target on event traces, Ensemble employs several optimization strategies that greatly reduce the overheads introduced by layering:

- Improve the speed of the computation
  Ensemble removes the explicit use of events in protocol layers. Since event traces encompass the entire life of events, the contents of events can be kept in local variables, which are often placed in registers by compiler. Furthermore, all functions in the trace handler are completely inline. It results in large basic blocks which can be optimized in a traditionally way.

- Compress the size of message headers
  In layering model, pushing and popping header cost significantly. Ensemble improves bandwidth efficiency and performance by reducing the sized of message headers. All headers are divided into three classes: addressing headers, constant headers and non-constant headers, and two of which are compressed. Since addressing headers for one connection do not change very often, connection identifiers are used to compress many addressing headers and constant headers into 4 bytes value by using MD5 hashing. An additional field namely "multiplexing index" of connection identifier is used to multiplex several virtual channels, which is generated for the trace handler.

- Reorder operations
  Ensemble borrows the approach [vR96] to reduce the latency of the trace handlers without decreasing the amount of computation. It divides the send and delivery processing into two phases: pre-processing phase and post-processing phase. The pre-processing may be done before the post-processing on each layer. And the post-processing would be delayed until the next sending or delivering, which makes a message can be passed to adjacent layers without changing the state of current protocol layer immediately.

One of the significant contributions of Ensemble is to determine the impact of using advanced functional programming language such as ML in the implementation of communication systems. Since Ensemble is the variant of Horus and both of them have been developed by the same research group but coded in ML and C respectively, their similarities make the comparison possible and efficient. After variety comparisons, the conclusion shows that ML allowed high level of abstraction in the system, and significantly reduced the size of protocol layers as well as it supports enough management for low-level details. From the experimental result, Ensemble achieved better performance than Horus.

4.1.2 X-kernel

[HP91] presents the x-kernel, which provides an explicit architecture for modular protocol construction and composition aiming to facilitate the implementation of efficient communication protocols. It started as an operating system kernel, and then evolved to a subsystem which can be installed in other operating systems. The major contribution of this work is that it demonstrates that x-kernel is both general and efficient enough for implementing complex network protocols by composing a collection of micro-protocols hierarchically without compromising performance.

There are three primitive communication objects supported by x-kernel: *protocols*, *sessions*, and *messages*. Protocol objects serve two functions: one creates session objects, and another demultiplexes the receiving messages from session objects. A session objects is an instance of a protocol created at runtime, which is the local representation of the network connection. Message objects visit a series of sessions via *push* or *pop* operation when they flow up or downward. A composition of protocol and session objects forms a path through the kernel that message follow. In a simple way, x-kernel views a network protocol as a collection of *protocols* and *sessions* which exchange a set of *messages*.

Studies [Cla82, Cla85] show that one of the primary factors leads to inefficient protocol implementation is the hierarchical structure, since the large overhead would be involved in communication and synchronization between layers. X-kernel addresses this problem by integrating the following several basic low level techniques, which are optimized for network protocol processing.

- Process management
  Process per message is the key aspect of x-kernel design. Processes are associated with messages rather than protocols where processes are allowed to change from the user mode to the kernel mode by a system call or from the kernel mode to the user mode by an *upcall*[Cla85]. *Upcall* allows a lower layer to "ask advice" to an upper layer in order to obtain information about the type of service that it is providing, so the lower layer can optimize its performance according to the type of service. Process per message management makes it possible to shepherd messages through the protocol stack without context switches and it permits more parallelism and supports synchronous message transmitting.

- Buffer management
  X-kernel supports a buffer management scheme that allows protocol implementation to avoid unnecessary data copying. Buffer manager routine allocates two successive and individual buffers as an entire buffer space, which allows multiple references without incurring any data copying.

- Event management
  Event manager routines provide an alarm clock facility. For example, a timer event can be registered as a procedure with the event manager, and protocols can be timeout if the message has not been acknowledged in time.

It also provides several features on high level for achieving high performance.

- Define explicit structure for protocols
  Each protocol in x-kernel is partitioned into two disjoint components for making code easier to write and understand. The protocol object is responsible for the protocol-wide issues, which switches messages to the right session. And another component is session object, which is responsible for connection-dependent issues corresponding to the real work of protocol.

- Compose protocols dynamically on a per message basis
  At run time, processing message will invoke the open operation which determines then next lower level protocol it should open. Such as, an application protocol opens a TCP session, sequentially TCP make decision to open an IP session. By this way, only the needed protocols can be configured, and there are not statically bound between protocols.

- Make the interface to protocols uniform
  It provides uniform interfaces to protocol objects aimed at improving the performance of most common patterns of interaction, since the significant cost of changing abstractions would be avoided. Moreover, it makes performance predictable when the new protocol is designed. Because the cost of individual micro-protocol is already known, the cost of composing those protocols will be known also.

- Use control operations to gain the same advantage available to ad hoc implementations
  X-kernel is limited to give the ability to one protocol to access information from the others due to the hierarchical structure. For example, TCP needs to know the MTU from the underlying network protocol to make decision on packet fragmentation. Instead of looking for this kind of information from the global data of other protocols, x-kernel implementation learns it by invoking a control operation, which is able to access all the information protocols need.

4.1.3 The Linux TCP/IP Implementation for Embedded Systems

The Linux operating system is one of the most popular operating systems for embedded systems as well as one of the most successful platforms for modern networking. As the popularity of Linux increases, the combination of the Linux OS with a stable TCP/IP stack[Ste93] has become the primary choice for many embedded systems used in many diverse applications.

There are several special requirements[Her04] for TCP/IP stack implementation of a generic embedded system, e.g., timer facility, concurrency and multitasking, buffer management, link layer facility, low latency, minimal data copying. Linux is able to provide a robust, mature and stable TCP/IP stack implementation to support all these networking needs. For example, fixed-length buffer systems are commonly used in embedded systems. Normally the buffer

pool was pre-allocated at boot-up time, which avoids heap fragmentation problem but limited in the fixed maximum number of buffers. In Linux, the networking buffers, namely socket buffers, are allocated from a slab cache, which both solves the fragmentation problem and has scalability without preconfigured upper bound. Another example is the amount of copying necessary to move a packet from the application down through the stack to the transmission media is especially performance sensitive in embedded systems. Linux supports scatter-gather DMA to allow the direct transmission of lists of TCP segments. And when data is transferred through a socket, copying can be also avoided by mapping data directly from the kernel space into the user space.

Generally speaking, Linux processes a packet through TCP/IP stack as follows[CP02]. A packet is constructed by the application, and is send through the socket API from top to down on one side of networks. This packet is passed through to the transport protocol in the correct family, e.g., UDP and TCP. The kernel triggers the appropriate state machine for the transport protocol and then passes the packet to the corresponding IP protocol, which does route work as well as other concrete operations, e.g., adding the header and queuing the data for driver output. The device is woken and transmits the packet out onto the transmission medium. Then the packet traverses a link to a router, where it would be scheduled in packet queuing code, and be forwarded over an Ethernet. Finally the driver at the far side will wake up caused by the coming of this packet. The interrupt service routine pulls it off a device and puts it in memory, and invokes an interrupt to show that the packet is ready. IP protocol checks the packet and carries out necessary reassembly work. If the packet is for local delivery, it will be demultiplexed to the appropriate transport handler receive function. Transport protocol then does the further packet header checking, updates the state machinery, and then demultiplexs it further to the correct socket. Once the packet is put into the appropriate socket receive queue, it calls wakeup on any process waiting for data.

Two guiding principles allow protocol stacks to be implemented as shown in the OSI model under the Linux system: *information hiding* and *encapsulation*[CP02]. Each layer hides its implementation details from the adjacent layers above and below. In addition, as two machines communicate, each layer in the transmitting machine has a peer-to-peer relationship with the same layer in the receiving machine. Encapsulation is the primary mechanism that allows the layers to hide information from the layers above and below. As the data being readied for transmission travels down the stack, each layer puts its header in front of the data it receives from the layer above. In the receiving machine, each layer strips its header off after receiving the data from the layer below.

4.1.4 UIP

UIP[Dun07] is a user level TCP/IP protocol stack implementation for memory-constrained embedded systems, such as sensor nodes in wireless sensor networks. It is open source written in C with a very small code size as well as a very small memory usage. The main contribution of uIP is that it demonstrated the

TCP/IP protocol stack implementation can be light-weight for tiny devices.

For reducing the code size and the memory requirement, uIP only implements all RFC requirements that affect host-to-host communication, and removes certain mechanisms between the application and the stack, such as reporting TCP error condition to the application. It argues that this removing will not compromise the generality, since few applications use those features. And from the source code of view, the implementation of uIP is not in a layered fashion; instead it uses tightly coupled way to save code space.

A single global buffer is used in uIP to hold packets and a fixed table is used to hold connection state. The incoming packet is placed in this global buffer and it has to be processed immediately, otherwise the next incoming packet will be dropped in order to avoid overwriting. If packets arrive at the time of the application is processing the data, they must be queued in network device. Normally the on-chip buffers can contain at least 4 maximum sized Ethernet frames. And if the buffer is full, this packet is dropped. The outgoing packet uses the same global packet buffer to store TCP/IP header as well as the data temporary. The application has to reproduce the outgoing data for retransmission if necessary, since the outgoing data is not queued in the buffer.

In order to handle concurrency in a small amount of memory, uIP uses an event-driven model rather than the most common multithread model. In uIP, the application is invoked in response to certain events, such as data arriving on a connection request, rather than the stop-and-wait semantics of BSD socket API. Since event-driven model uses only one stack, it avoids the overhead of task management, context switching and allocation of stack space for the tasks in multi-thread model. As a result, the memory requirements are reduced.

4.2 Language Based Approach

Another approach to implement network protocol stacks is the language based approach, which either modifies an existing programming language by adding extra abstractions or creates a new language to explicit support protocol development. Both of them provide the type support to the domain abstractions as well as conventional ones.

Functional languages are not often associated with the development of network protocol stacks, mainly due to the lower performance and lack of support for system programming than more conventional languages such as C language. However, the high level of discourse and concise syntax provided by functional languages makes it easier to develop network protocols than conventional approaches based on an imperative language. FoxNet[Bia01] is such a well-known previous work related to the development of TCP/IP stacks in functional languages. It is a TCP/IP stack together with a web server both developed in Standard ML, which demonstrates that functional languages are suited for system development.

The new programming languages in special domains normally have natural vocabulary as well as syntax and semantics that are easy to be understand and used. Some of them are designed primarily to make implementation easy. Such

as, Prolac[KKM99] is intended to make protocol implementation readable to human beings. Others are aimed primarily at making correctness verification easy as well as implementation. Such as, TAP[Mcg04] and Promela++[BMvE98] have two execution models: one for model checking, one for generating execution code.

4.2.1 FoxNet

FoxNet[Bia01] is the communication part of Fox Project that aims at writing system level code in high-level programming language. FoxNet has been used to build a safely composable TCP/IP protocol stack to provide the network service for a standard web server. It is able to provide powerful high level constructs to write efficient modular code as well as support low level constructs by extending the Standard ML language. For example, it includes 32-bits integer as a new type in SML, which is useful in the implementation of network software.

The overall design of FoxNet resembles the x-kernel with one primary difference that is the FoxNet is implemented in SML. There are three key features provided by SML, which are very useful in the implementation of the FoxNet.

Firstly, SML is a rich module language. It integrates signature to define the interface of modules, and the compiler checks the signatures conformance of implementations automatically. Structure, signature and functor are three SML constructs used in FoxNet. Each protocol in FoxNet must conform to a specified Protocol signature, which is ensured by signature inheritance and type sharing.

Structure encapsulates SML values, such as integers, list and functions. A structure satisfies a signature if it contains all of the set of type and value bindings within the signature, and it also can contain functions which are written as functors.

Signature specifies a set of type and value bindings. It is a specific view of the structure, but only allowing access from code outside of the structure to those bindings that appear in the structure.

Functor is used to compose structures together, which take structures, types and values as arguments and produced another structure. In FoxNet, one module responds to one functor.

Secondly, SML is a type-safe polymorphic language with type inference, automatic storage management and garbage collection, whose type system is almost enough to capture the compilation errors, except for the nested polymorphism handling. On the other hand, data copy can be caused by strict type system if type cast is necessary, such as marshaling and unmarshaling headers.

Finally, high-order functions can be both constructed and returned as values at run-time in SML, which are used to define richer and more appropriate

interfaces for modules than x-kernel. For example, the meta-protocol is expressed explicitly as PROTOCOL signature in FoxNet, which has a *send* and a *receive* operation. The implementation of *send* and *receive* must be a high-order function, which includes executable codes as well as values.

4.2.2 TAP

The Timed Abstract Protocol (TAP)[Mcg04] notation is a DSL for describing asynchronous message-passing network protocols precisely and verifiablely. Coupled with Austin Protocol Compiler (APC), protocol descriptions in TAP can be transformed into executable C code.

In TAP, a network protocol consists of two or more processes, which communicate by sending messages across channels. The highlight of TAP is that it provides a direct model of time, which makes temporal behavior easier to express. From an abstract point of view, the Tap notation can be described as followed.

- *Message* is a sequence of fields, where each field is either an integral value or a uninterpreted byte array.
- *Channel* is a queue of messages that is identified by the abstract address of the sending and receiving processes.
- *Processes* is a combination of a local state and a set of actions describing the behavior of the process.
- *Action* is a combination of the circumstances and the statements describing when and what should be performed by a process.
- *Statement* includes *send*, *:=*, *skip*, *if*, *do*, *act-in* and *function call*.

The TAP has two equivalent execution models. The abstract execution model supports protocol design, comprehension and verification. The concrete execution model supports simple and efficient protocol implementation. In high-level abstract model, global atomicity and immediate message propagation limit the number of states and transitions between states. The message faults abstracts the general fault of a protocol and the timeout behavior abstracts the passage of real time without reference to a clock. Furthermore, the global fairness ensures progress in the protocol. In low-level concrete model, local atomicity models the execution of a group of processes and the channels between processes can take an arbitrary time to deliver a message. Local fairness allows a computation to delay an action by executing other actions and messages and timeouts are acted on fairly.

The APC translates the specification into an executable system based on the concrete execution model. Since TAP does not contain many features to a general-purpose programming language, APC provides for protocols to call arbitrary C functions. It allows file input/output, cryptography, database and

buffer management. A protocol implementation generated by the APC is invoked from an external program and in turn invokes other external functions. The interfaces among them are described by C functions and datea structures. The APC runtime library is built on top of a base network protocol, which is limited to UDP in the current version.

4.2.3 Promela++

Promela++[BMvE98] has been designed to facilitate the development of modular and efficent protocol code whose source code can be converted to Promela for verification, as well as transformed into efficient C code for execution. It reduces the complexity of developing new protocols and highlights the power of a DSL linked with formal methods.

As a high-level language, Promela++ provides a rich set of language constructs to make specification easier for a wide variety of protocols. It encapsulates protocol layering and composition of such layers by providing explicit language mechanisms. In more detail, it provides:

- The layered specification of protocols.
- The composition of layers into protocol implementations using an event-based mechanism.
- The identification of fast paths though programmer annotation.
- The encapsulation of protocol state and message headers.

Promela++ is closed to C/C++ programming model, which is easy and attractive to C programmers. In a Promela++ program, only the control sturcture of a protocol layer is specified in Promela++. The low level network access and data-manipulation routines are blocks of embedded C code, which is therefore not checked by the verifier.

Promela++ is limited by the lack of explicit memory allocation primitives which makes message allocation operations difficult. It does not support timers which makes the implementation of a number of widely-used protocols such as TCP becomes quite cumbersone. It provides explicit language mechanisms to encapsulate and composite protocol layers where the adjacent layers communicate neatly using FIFO message queues. However, this scheme has to afford a time-consuming context switch between the communicating processes that could overburden the runtime memory.

4.2.4 Prolac

Prolac[KKM99] is an advanced modular object-oriented language designed for creating efficient and readable protocol implementations, such as TCP. Its compiler has been implemented in C++ programming language.

Prolac is designed as an expression language as well as Lisp, SML or Haskell but with fewer complicated control constructors. For example, Prolac has no

looping construct, which can be expressed by recursion. It has light-weight syntax, since the substance of the code is the only thing on display, which makes small methods more readable. All of C's operators plus a few additions are usable in the expressions of Prolac.

Compared with C function bodies, the method bodies of Prolac tend to be very short. Most of the methods have only 5 lines or less code. Large methods are encouraged to be broken up into small sensible-meaning pieces. By this way, it would be easy and efficient to give names for those computation parts which make protocol specification easier to understand.

The basic unit of program in Prolac is module, which is associated with data and method as the class in many object-oriented languages. And each piece of code in Prolac belongs to some module. Module operators are Prolac's simple and powerful mechanism for manipulating modules, which provides a very clean way to give information to the compiler. So far, Prolac support 6 module operators: *Hide*, *Show*, *Rename*, *Using*, *Notusing* and *Inline*.

Prolac works as follows. First of all, a Prolac specification file is stored in a single file. And then the compiler of Prolac processes this file, two output files written by C programming language are produced. One of the output files is a header file containing C structure definitions which responds to Prolac structures, and the other one is C source file containing definitions for any exported Prolac methods. These two output file can be used directly as networking protocol implementation in Linux kernel 2.2.

There are three stages involved in the compiler processing. The first stage includes parsing and translating the source text to parse trees without looking up any identities. In the second stage, the compiler resolves and reports semantic errors, looks up identities, and translates the parse tree to an intermediate representation. Finally, optimization and compilation are accomplished, and the input resource is translated to the output code in C programming language.

The primary limitation of Prolac is its compiler relies on C language. If some features of a protocol are not suitable to be implemented in Prolac, it will quote C code directly. Such as there are 1300 lines of Prolac code and 700 lines of C code in its TCP implementation.

5 A Framework for Protocol Stacks

In this chapter we present a framework for developing protocol stacks in Haskell. The framework contributes a variety of primitive building blocks corresponding to parts of protocols as well as combinators for putting blocks together to form protocols and stacks. To develop a protocol stack the programmer has just to choose what blocks to use, redefine some of their functionality and put them together in the desired fashion. The abstractions and structuring principles are borrowed from [AP93, Abb93].

5.1 Background

We take the DSEL approach as a light-weight way to construct our languages for the implementation of protocol stacks. [Kam98] argued that embedding works particularly well when the host language is a functional language, e.g., Haskell and ML, which is relatively easy and produces good results. [Hud97] also described a couple of other benefits provided by Haskell, which make it a suitable host language for DSELs. For instance, it is lazy evaluation, it allows recursive definitions polymorphism. Under their suggestions, we choose Haskell as our host language.

We borrow the abstractions and structuring principles defined in Morpheus [AP93, Abb93]. Since one of the commonly used techniques to make developing protocol correctly and efficiently is building modular protocols, e.g., Horus and Ensemble. A protocol is divided into distinct reusable building blocks that can be developed individually and recomposed in different ways. Morpheus is such a language designed for enabling flexible development through the modular architecture, which allows gluing the set of selected micro-protocols together into a protocol to meet the desired properties. Each micro-protocol conforms to one of *Shapes* in Morpheus, which enables the protocol-oriented compiler optimizations. This proposal is started by Morpheus, but left uncompleted as well as some of other good ideas.

We think that it would be a good starting point to reference an existing proposal to gain insights in the abstractions we need and the intricacies to be solved. Optimally, we can identify the domain problems and gather domain knowledge from the experience of this practical work.

5.2 Morpheus

Morpheus [AP93, Abb93] is designed as a special purpose programming language to facilitate the implementation of communication protocols. Its focus is protocol abstractions and protocol-oriented compiler optimizations which are based on the experience gained from implementing the x-kernel[HP91].

Morpheus introduces the concept of a *Shape* as the building blocks for protocol implementations, where each *Shape* corresponds to one of three categories related to protocol functionality. This division aims at providing more opportunities for code and data structure reuse. Additionally, developing, verifying, implementing, and maintaining a collection of simple protocols is much easier than for an equivalent complex protocol. Arbitrary protocol functionality can be implemented by composing one or more protocols with three different *Shapes* as shown in Figure 2. These *Shapes* are:

- *Multiplexor* is responsible for any multiplexing and demultiplexing.
- *Worker* is responsible for "real" protocol work.
- *Router* is responsible for routing based on host addressing.

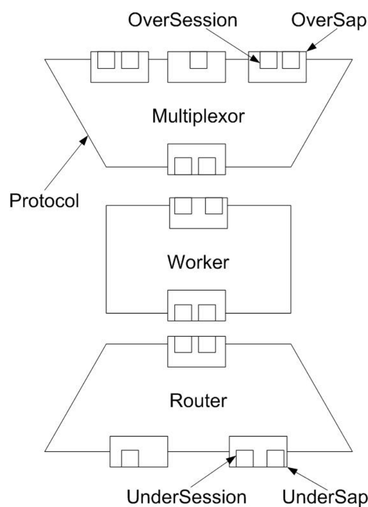

Figure 2: Shapes and Base Classes

Morpheus is an object-based design. It predefines a collection of base classes corresponding to the fundamental elements of Morpheus' model of protocols. The programmer can implement a specific protocol by deriving and modifying these base classes to produce subclasses. Inheritance from the base class is used to supply predefined code and data structures. These five base classes: *Protocol*, *OverSap*, *UnderSap*, *OverSession*, and *UnderSession*, are also depicted in Figure 2, and some of them are nested in others.

- *Protocol*
  Protocol represents an active instance of a particular implementation of a protocol specification, and includes *OverSaps* and *UnderSaps* as components.

- *OverSap* and *UnderSap*
  A matched pair of *OverSap* and *UnderSap* represents a *Service Access Point* (SAP), which is a communication service interface between one higher-level particular protocol entity and another protocol entity with an address which can be uniquely identified. The communication service user side of a SAP is represented by an UnderSap, while the service provider side is represented by an OverSap. An *OverSap* includes *OverSessions* as components, as well as an *UnderSap* includes *UnderSessions* as components.

- *OverSession* and *UnderSession*
  A matched pair of *OverSession* and *UnderSession* represents a session, which is an end point of a data stream or abstract communication channel between two specific SAPs. An *UnderSession* represents a one-way

interface from the protocol entity that uses the channel to the protocol entity that provides the channel; an *OverSession* represents that interface in a reverse way.

Besides abstractions, Morpheus also concentrates on reducing per-layer overhead by using protocol-oriented compiler optimizations. There are five optimizations employed by Morpheus for latency.

- Dedicated Message Registers
  Morpheus designates two aside registers, one for the passing message and the other is for the header pointer, which will not be released until this message passes through the entire protocol stack or a second message must be passed. Since Morpheus generates one thread for each new message, there would be several threads working on passing messages in parallel. And how to restore and reallocate those two registers would be a problem.

- Short-circuit return
  Under the thread per message context, the last action of every send or deliver functions is calling the corresponding send or deliver function from adjacent protocol. In Morpheus, before jumping to the lower function, the current function restores all registers, and then gives the start point of the current function's caller as the return address. This optimization saves one assemble instruction per layer. Furthermore, in the case of no more send or deliver function in the current procedure, four assemble instructions are saved, which are saving and restoring the return address register and updating and restoring stack pointer.

- Procedure Cloning
  The send or deliver function for each session is generated at runtime by filling up a template with arguments which is generated by Morpheus for each protocol at compiler time. The purpose of this cloning is that the constants are hardwired into the code, which reduce the number of executed instructions and eliminate the memory access.

- Language constructs for frequent tasks
  Frequent protocols tasks are abstracted as constructs in Morpheus. It is more efficient than using utility routines from the library, since the linkage code of procedure is eliminated and compiler has more information for optimization of the code.

- Eliminating header bounds checking
  Since Morpheus can predict the largest message header for each protocol, it always allocates sufficient header space to each message. As the result, the boundary of message would not be checked anymore.

Unfortunately, the design and implementation of Morpheus has been left uncomplete. There is no compiler, no formal semantics and no grammar for this language. All the concepts have been tested only by hand coding. To

force clean protocol designs and enable domain specific optimizations, it puts many constraints on the programmer. And only the asynchronous and unicast protocols fit into its framework. As a result, existing protocol specifications may not be implemented in Morpheus.

5.3 Our Approach

5.3.1 Modular Development

Modular development as one of the techniques commonly used in the software engineering community has been successfully applied in many domains. This technique is also used in Morpheus as we mentioned in 5.2, which is able to decompose a complex protocol into several simple micro-protocols, and each of them has to be constrained to one of three *Shapes*: *Multiplexor*, *Router*, or *Worker*. With distinguishing *Shapes*, the micro-protocols with the same *Shape* would be similar enough that maximizes the code and data structures reuse. We keep and extend this concept by employ a hierarchical modular model, where each module can be independently developed and customized. Therefore, the programmers are able to reuse, configure and assemble modules into a new protocol stack with writing few new codes.

A framework with three levels is identified, where each level works at various granularities as follows.

- Micro-protocol level
  Our framework supports micro-protocols as the lowest level modules which conform one of the *Shapes*, and nests *Saps* and *Sessions* as entities as Morpheus does. Because the behaviors of micro-protocols can be partly predicted according to their *Shapes*, an explicit template is provided to generate micro-protocol modules by our framework. This template hides the predicted implementation details inside the module, and takes higher-order user-declared functions as arguments for special behaviors of variant micro-protocols. For example, sending and delivering message through *Sessions* specify the operations of the *Session* within a micro-protocol which can be also considered as nested entities. In this way, programmers can concentrate on the semantic of the micro-protocol rather than bother with its structure and relationships with adjacent micro-protocols.

- Protocol level
  The protocol level module is supposed to match the specification of the practical protocol. It is actually a micro-protocol graph, which is composed by a set of micro-protocol modules with different *Shapes*. The required properties of the protocol specification are distributed over the micro-protocol graph. For some particular protocol states, which handle the entire protocol globally, we put them into the top or bottom micro-protocol of the graph. In the case of connection-oriented protocol, the

nested *Session* is established and closed dynamically according to the control messages; in the case of connectionless protocol, the default *Session* is always standby.

- Protocol stack level
  From the architecture point of view, our framework clearly represents the protocol stack structure, indicating how protocols are combined together, as well as the nesting relationship between modules. A protocol stack is configured by a set of protocols according to the relation table where all relationships between protocols for this stack are specified.

Module interfaces are well-explicated for both the service that the module provides and the service it requires from the rest of the system, which allows for nearly arbitrary composition of modules.

- Interface for micro-protocol modules
  Each micro-protocol module has to specify its interfaces to the above and below micro-protocol modules at source code level, which includes the names of the micro-protocol modules and the corresponding SAPs. The structure of micro-protocol graph is built statically at initialization time according to the interfaces they declared. The interactions of micro-protocol modules are call-oriented at runtime. It means after the current micro-protocol module performs its own operations, it will sequentially invoke the corresponding operations of the adjacent micro-protocol module by function calls, which includes connection establishing and closing as well as messages sending and receiving.

- Interface for protocol modules
  The protocol module works in the similar way as the micro-protocol module does, whose interface includes names of the declared above and below protocol modules. If there are more than one top micro-protocol in the micro-protocol graph, one micro-protocol shaped *Router* is enforced, which takes all top micro-protocols in former graph as the lower layer modules. In the case of multiple bottom micro-protocols, this routine works similarly: a micro-protocol shaped *Multiplexor* is imposed on the bottom of the graph. In this way, the protocol module has the uniform interface that only has one *OverSap* as well as one *UnderSap*. The protocol modules communicate with each other by function calls as the micro-protocol modules does. For example in the case of sending messages, after the current protocol module complete its own operations, its bottom micro-protocol will sequentially invoke the corresponding function of the top micro-protocol of the below protocol modules. Since the structure of the protocol stack has been already built, i.e. the direction of data flow has been decided, the interaction between adjacent protocol modules usually only involves messages as arguments.

5.3.2 Built-in Mechanisms

This section presents three mechanisms which are built in our framework to enforce constrains on software architecture.

- Thread per message

  Our framework uses thread per message model for passing message up and down, in which threads are associated with messages rather than protocol layers. The main thread of the system listens for incoming or outgoing messages, and forks a new thread for each message. The thread shepherds a message from a user process to a network device or in the reverse way without context switches, which make up the latency caused by hierarchical structured architecture. It also permits more parallelism and supports synchronous message transmitting. The thread terminates after this message passes through the protocol stack.

  The downside of thread per message is that it would be too many threads running parallel in the case of high throughput. As the protocol stack implementation is intended for embedded devices in our case, which usually do not produce lots of data, the drawback of thread per message would not become a serious problem.

- Blocking operations prevention

  Locks are used for accessing shared data, such as the state information of micro-protocol. As we mentioned above, thread per message strategy treats each protocol as a more or less static piece of software. In our case, all the entities in micro-protocol module are static, except the *Session* needs to be maintained dynamically. Potentially blocking operations include the opening and closing *Session* as well as sending and delivering message through *Sessions*.

  To avoid blocking, each *Session* runs independently in our framework, so it will not cause blocking by transmitting messages through different *Sessions*. Furthermore, before invoke the next operation from adjacent protocol, the resource of current protocol used by a *Session* will be released. As a result, no blocking will be caused by the same *Session* which runs at different protocol layers. Finally, since each micro-protocol only performs a simple functionality, messages will pass it through very soon, which reduces the holding time in micro-protocols for each thread.

- Flow control mechanism

  Flow control plays a significant role in controlling congestion, which is able to offer low-loss and low-delay real-time services without the need for complicated admission control, bandwidth reservation or packet scheduling mechanisms. Our framework supports flow control at the granularity of micro-protocol as Morpheus does. But instead interposing a dedicated reusable micro-protocol that enforces flow control policy, our framework provides explicit flow control templates at two endpoint micro-protocols

of a protocol graph in a protocol module. In detail, the bottom micro-protocol informs the far side peer the number of messages can be delivered by using the *grantDelivers* operation and only delivers messages with credits. Similarly, the top micro-protocol responds for the peer's delivery credits and uses the *grantSends* operation to handle how to send messages.

As a result, it is unnecessary to propagate flow control information between the adjacent micro-protocols up and down within a protocol graph, which means the internal flow control policy is "infinite credits". The default policy for protocols is also "infinite credits", unless programmers specify their own ways to stop messages without credits in different cases by filling up the *grantDelivers* and *grantSends* templates.

5.4 Examples

This section presents a couple of examples of protocols and an example of protocol stack in our framework.

5.4.1 A Simple Virtual Protocol

To illustrate how our framework works, we start from showing a simple virtual protocol in Figure 3. The functionalities of this protocol are fragmenting and reassembling packets, filtering out any duplicate or out-of-order incoming packets, as well as encapsulating and unpacking, which are implemented respectively by three micro-protocol modules. Packets to be send move from up to down, and received packets move from down to up. For simplicity, this protocol does not enforce any flow control policy. As the shown pseudo code for each module, programmers only need to fulfill two functions: *Send* and *Deliver*. And there is almost no code or data structure for non-specific work has to be implemented by programmers, e.g., assemble the modules, connect this protocol to the adjacent layers, which are provided by the framework implicitly.

5.4.2 A Transport Protocol

Despite decomposing a protocol into a set of reusable building blocks is conceptual elegance, protocol implementation based on this approach is hardly to be caught on, particularly at transport level. [CHM+05] demonstrated that it is possible to build transport layer protocols by this approach by providing several examples in a diverse range, which are able to cover the needs of most point-to-point applications well.

To check the practicability of our framework, we choose one of the transport protocols introduced in [CHM+05], which allows an application to retry transmission of a message to a potentially different destination. It is decomposed into a number of building blocks as illustrated in Figure 4. And we easily port it to our framework with several simple changes as Figure 5 shows. The red arrows represent the sending data flow, and the black arrows represent the receiving data flow. All building blocks with multiple above and below interfaces are

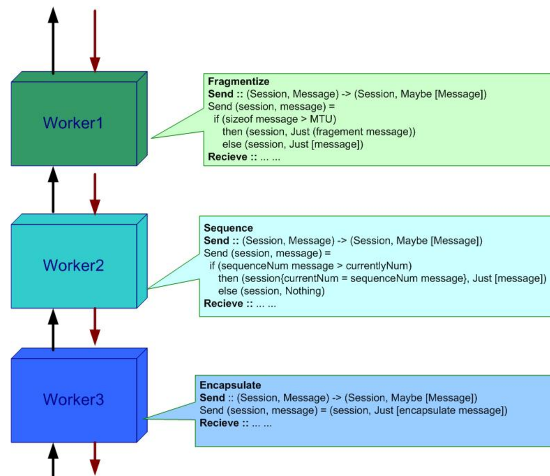

Figure 3: A Simple Virtual Protocol

translated into a combination of two micro-protocols shaped *Multiplexor* and *Router* respectively. For example, the congestion control building block(CC Tx) of Figure 4 is split into two micro-protocols in Figure 5, since it interfaces more than one above micro-protocols as well as below building blocks. To uniform the interface with the adjacent protocols, a micro-protocol shaped *Router* is put on the top of the protocol graph, as well as a micro-protocol shaped *Multiplexor* at the bottom. With this example, we show that our framework is enough to describe a wide range of protocols.

5.4.3 A Protocol Stack

On the base of previous two examples, we can build the whole protocol stack by binding those two protocols together as Figure 6 shows. The blue dashed box is corresponding to the simple protocol introduced in 5.4.1, and the green dashed box is corresponding to the transport layer protocol introduced in 5.4.2. Since the interface of each protocol graph is uniform, it makes the combination easily with few new code.

6 Conclusion and Future Work

This report gives an introduction of the IPS project and presents the insights gained from related work. It also describes a framework for implementation of protocol stacks in Haskell, which is the primary accomplishment for the first year in the IPS project, whose preliminary result suggests that it is a working framework for composing protocol stacks with modularity, generality, and flexibility.

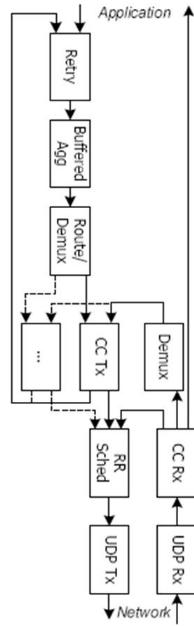

Figure 4: Bulding Blocks for a Transport Protocol

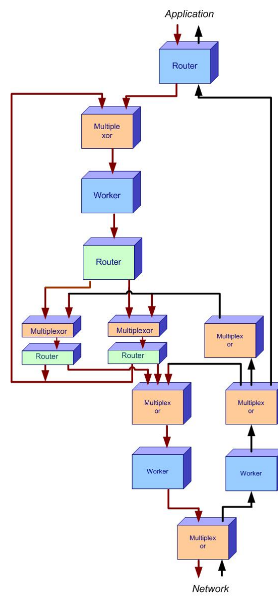

Figure 5: A Transport Protocol

Figure 6: A Protocol Stack

In the short term, we will concentrate on testing this framework by implementing more practical protocols as well as integrating it into a real operating system, e.g., House, which is an OS written in Haskell. And we will write a parser to interpret a raw buffer into the packet according to its format.

In the long term, we will study on how to incorporate solutions to common bottlenecks in automatic way, and how the language can look like. The output of the study would lead to design a prototype of the language together with a series of experiments to explore its possibility.

References

[Abb93] Mark Bert Abbott. *A Language-based Approach to Protocol Implementation*. PhD thesis, University of Arizona, 1993.

[AP93] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM transactions on networking*, 1(1), 1993.

[BCSS98] Per Bjesses, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *ACM International Conference on Functional Programming*, 1998.

[Bia01] Edoardo Biagioni. A network protocol stack in Standard ML. Technical report, University of Hawai'i, 2001.

[BMvE98] Anindya Basu, J. Gregory Morrisett, and Thorsten von Eicken. Promela++: A language for constructing correct and efficient protocols. In *INFOCOM*, pages 455–462, 1998.

[CHM+05] Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Sean Rhea, and Timothy Roscoe. Finally, a use for componentized transport protocols. In *HotNets IV*, 2005.

[CHR+03] C. Consel, H. Hamdi, L. Reveillere, L. Singaravelu, H. Yu, and C.Pu. Spidle: A DSL approach to specifying streaming applications. In *Second International Conference on Generative Programming and Component Engineering*, 2003.

[Cla82] D.D. Clark. Modularity and efficiency in protocol implementation. Technical report, MIT Lab. Comput. Sci., 1982.

[Cla85] D.D. Clark. The structuring of systems using upcalls. In *ACM Symposium on Operating Systems Principles*, 1985.

[Con04] C. Consel. From a program family to a domain-specific language. *Domain-Specific Program Generation*, 2:11, 2004.

[CP02] Jon Crowcroft and Iain Phillips. *TCP/IP and Linux Protocol Implementation: Systems Code for the Linux Internet.* Robert Ipsen, 2002.

[CRL96] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *PLDI '96*, pages 237–248, New York, NY, USA, 1996. ACM Press.

[Cry07] http://www.cryptol.net, April 2007.

[Dsl07] http://www.ece.ubc.ca/ elec571f/lectures.html, February 2007.

[Dun07] Adam Dunkels. *A Language-based Approach to Protocol Programming Memory-Constrained Networked Embedded Systems.* PhD thesis, Swedish Institute of Computer Science, 2007.

[Hay98] Mark Garland Hayden. *The Ensemble System.* PhD thesis, Cornell University, 1998.

[Her04] Thomas F. Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems.* Charles River Media,, 2004.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), 1991.

[Hud97] Paul Hudak. Modular domain specific languages and tools. Technical report, Department of Computer Science, Yale University, 1997.

[Kam98] Samuel N. Kamin. Research on domain-specific embedded languages and program generators. *Theoretical Computer Science*, 14(1), 1998.

[KKM99] E. Kohler, M.F. Kaashoek, and D.R. Montgomery. A readable tcp in the Prolac protocol language. In *ACM SIGCOMM '99 Conference*, 1999.

[Lex07] http://www.lexifi.com, April 2007.

[Mcg04] Tommy Marcus Mcguire. *Correct Implementation of Network Protocols*. PhD thesis, The University of Texas at Austin, 2004. Supervisor-Mohamed G. Gouda.

[MHS03] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. Technical report, Stichting Centrum voor Wiskunde en Informatica, 2003.

[MTH90] R. Milner, M. Tofte, and R. Harper. The definition of standard ml. The MIT Press, 1990.

[PD03] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A systems approach*. Morgan Kaufmann Pub, 2003.

[RBM96] Robbert Van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communications system. Technical report, Department of Computer Science, Cornel University, 1996.

[Ste93] W. Richard Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley Professional, 1993.