Signatures for a Network Protocol Stack: A Systems Application of Standard ML*

Edoardo Biagioni Robert Harper Peter Lee Brian G. Milnes

Abstract

Advanced programming languages such as Standard ML have rarely been used for systems programming tasks such as operating systems and network communications. In order to understand more fully the requirements of systems programming, we have implemented a suite of industry-standard network communication protocols in a completely type-safe extension of Standard ML. While the implementation has only recently become operational, we already observe acceptable communications throughput. We make careful use of the Standard ML modules system, with the core component of the implementation being a signature which is generic to all communications protocols. This generic protocol is then specialized for specific protocols, and these are implemented by functors parameterized by generic protocols. This leads naturally to a layered system structure and also provides an important and useful "mix-and-match" capability in composing protocols into complex networking systems.

We have found the advanced features of Standard ML, in particular the modules system, static typing, and higher-order functions, to be extremely useful in building complex communications systems. The type compatibility of the various components of a system is guaranteed by the compiler. Furthermore, we find it significant that most of the information needed to understand the structure and interactions in our code can be obtained from a study of the signatures alone. Perhaps most important is that we have been able to use the expressive power of Standard ML modules to give concrete expression to previously *ad hoc* system-structuring concepts developed by other researchers in the field of network communications. For language designers and implementors, our experience has also pointed out specific areas for further work that may lead to advanced languages that are useful for systems programming.

*This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F196228-91-C-0168.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

1 Introduction

The implementation of network communication protocols is a complex task requiring careful design to achieve high performance and reliability. An efficient implementation must exhibit both high throughput (transmission capacity per unit time) and low latency (time per transmission). It must achieve these goals while ensuring that data transmission functions reliably over unreliable media. In addition to the intrinsic complexity of the task, the problem of implementing a network protocol is compounded by the need to interface with hardware device drivers and to offer a standard interface to higher-level clients. These demands stress conventional software engineering methodology to the point that network protocols are often regarded as prime examples of the need to violate principles of structure and safety. To achieve efficiency and express low-level operations, many implementations use programming languages such as C or C++ that support low-level operations and let the programmer violate abstraction boundaries. *Ad hoc* techniques are used to compensate for the lack of linguistic support for program structuring, multi-threading, and storage management. The resulting programs are generally difficult to modify and maintain.

The purpose of the Fox Project is to investigate whether this unfortunate state of affairs is essential. In particular the project is investigating the suitability of modern programming languages based on rigorous semantic foundations for systems programming applications such as the implementation of network protocols. The overall goals of the project are to advance the design of programming languages by using them to solve systems programming problems and to advance the art of systems programming through the use of programming languages that support modularity, type checking, and higher-order functions.

This paper describes an implementation of the standard TCP/IP protocol stack in an extension of the Standard ML (SML) language. In structuring our implementation, we have made careful use of the Standard ML modules system. We define a generic signature for all protocols and specialize this for specific protocols. Functors that implement various protocols may then be parameterized by generic protocols, thereby providing a "mix-and-match" capability in composing protocols into complex networking systems. The type compatibility of the various components of a network system is checked by the compiler. We find it significant that most of the information needed to understand the structure and interactions in our code can be obtained from a study of the signatures alone.

The system we have built uses the SML module language to express design abstractions. These abstractions have been described informally by other researchers in the networking field but have never been concretely expressed in implementations due to the lack of appropriate support in the programming languages used.

In addition to modules, SML has safety guarantees about type and storage use that assist in the development and maintenance processes by eliminating type errors and storage use errors.

We have extended the SML language to make it easier to express low-level operations and to make it easier for the compiler to produce efficient code for these operations. The extensions we use include first-class continuations, byte arrays, 8-bit, 16-bit, and 32-bit values and operations, and functions to access the system and the hardware.

Section 2 summarizes the terminology and concepts used to describe the design and implementation of network communications software. Section 3 recasts these concepts in terms of ML language structures and presents our signatures and the high-level design choices that they reflect. Section 4 reports on the experience we have had with our initial system, and Section 5 concludes by summarizing our results and looking ahead.

2 Network Communications Systems

The terminology used for network communications systems often depends on the specific type of system being discussed, and is not always used consistently. To fix terminology we define the terms *layer*, *protocol*, *protocol stack*, *instance*, and *peer*.

A network communications system can be viewed as an implementation of an abstract machine. Application programs (*clients*) communicate through an interface represented by this abstract machine. A client may use this interface as a building block in building further abstractions. The term *layer* refers to a network abstraction.

In keeping with conventional usage, we use *protocol* to refer to an implementation of a layer, that is, an algorithm that realizes a network abstraction. A *protocol stack* is an ordered collection of protocols layered on top of each other, each protocol implementing its abstraction in terms of the abstraction upon which it is layered.

An *instance* of a protocol is the execution of a protocol on a particular system. There could potentially be several instances of the same protocol within the same system at the same time. In practice the state of an instance can be used to represent the instance, so that code can be shared among instances. The *peer* of a given instance `A` is the instance `B` with which `A` is communicating. Whenever `A` communicates with itself, `A` is its own peer. Typically, peers run on separate machines (*hosts*) connected on a network.

A well-known example is TCP/IP, which consists of three layers. The IP layer supports unreliable transmission of data using hardware-independent addressing. The TCP layer supports reliable delivery of correct[1] data. TCP is built on top of IP. IP itself is layered on top of hardware-specific protocols such as Ethernet or ATM. Data handed by TCP to IP may or may not be delivered to the intended destination, may be corrupted, or may be delivered multiple times. TCP attaches a unique identifier and check bits to each message sent, and exchanges messages with its peers to retransmit any messages that have been lost or corrupted.

We say that an implementation of a network communication system is *well-structured* if:

- it has a distinct protocol for each layer
- all the peers of each instance of a protocol are other instances of the same protocol,
- the layered structure is realized using modules and explicit interfaces.

It is a simple observation that many implementations of network communication systems are not well-structured according to our definition. One reason for this situation is that network software is usually implemented in C, which fails to support modules with clean interfaces.

Another reason for the poor structure of many implementations is the lack of modularity of the TCP and IP protocols. Specifically, the TCP protocol expects to make use of data structures internal to IP. This leads to an instance of the TCP protocol receiving information from the peer of the IP protocol, which violates the principles of good structure. Protocols defined within the ISO reference model [5] are generally more modular.

Another explanation for the lack of structure within implementations is that some optimizations are only possible if layer boundaries are violated. For example, a clean, but inefficient, implementation of a layered protocol would copy the data and perform a context switch every time the data had to cross the boundary between two layers. A more optimized implementation would only copy the data when absolutely necessary, and avoid context switches between layers by using the same thread of control (where possible) to execute the code for all the layers. Taking this to an extreme, all the boundaries between layers are removed. This merging of layers is similar to the loop fusion done by Fortran compilers and to the deforestation techniques used by functional language compilers. It reduces intermediate storage and the operations required to store and retrieve the data, and therefore saves time and space. Unlike these compilation techniques, the layers of communication software are typically merged by hand and optimized at the source code level, in a technique called *Integrated Layer Processing* or ILP[2]. The disadvantage of using ILP at the source code level is that it completely destroys the modular structure of the protocol implementation.[2]

One recent and notable exception to these poorly structured network implementations is the software produced by the x-kernel project [9], which has developed implementations of protocol stacks that are well-structured and highly modular. In the x-kernel, all layers declare an interface containing the same set of procedures each with the same set of arguments. This structure of procedures and its calling convention is called the *meta-protocol*. As a result, a protocol need not know which other layer it is layered on, and protocols can be layered almost arbitrarily to build custom protocol stacks that achieve the different performance tradeoffs required by different applications.

The x-kernel does not use ILP. As pointed out by Clark and others [4], many of the costs of traditional protocol implementations are not due to layering, and can instead be

[1] The correctness is probabilistic, and incorrect data can, with very low probability, be delivered by a correct implementation of TCP

[2] Abbott and Peterson [1] have addressed this by developing a specialized language for protocol data processing

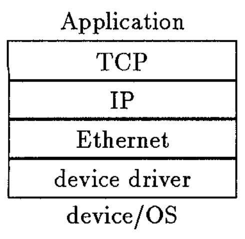

Figure 1: A Standard Protocol Stacks.

traced to avoidable inefficiencies in the layered implementations. By concentrating on avoiding unnecessary context switches and data copying, the x-kernel achieves efficient data transmission and reception while retaining a clean design whose structure is the same as that of the protocol stack [9]. This shows that while integrated layer processing may improve performance, a well-structured design can achieve good performance even without ILP.

Our work has been inspired by and has taken many ideas from the design of the x-kernel. Unlike the x-kernel, we have developed formal interfaces for our protocols and given them concrete expression in SML. In fact we believe that the x-kernel ideas can only be given full expression in a language like SML. The next section explains the general principles we followed in developing the formal interfaces and the details of the interfaces, and explain how the interfaces fit in our implementation of the TCP/IP protocol suite.

3 Structuring Communication Systems in Standard ML

We have developed a well-structured implementation of a simplified version of the TCP/IP protocol suite[3] in a language based on Standard ML. We refer to our implementation as the *Fox Net*. This section gives an overview of the Fox Net, omitting details in the interest of describing the overall structure and how it is realized in SML.

In our overview we present SML ***signatures***, ***structures***, and ***functors***. An SML signature specifies the interface of a module. A module may implement multiple, different signatures. A signature may be thought of as a collection of conditions a candidate implementation of the interface must satisfy. An implementation of an interface specified by a signature is an SML structure that must have a concrete type for every type specified by the signature and a value for every value specified by the signature. A functor is essentially a parameterized structure whose parameters are structures. A functor, when applied to actual arguments, computes a new structure. We refer to this as the ***instantiation*** of a functor. A functor may be instantiated multiple times, with the same or different parameters, each time producing a new structure. Functor instantiation can involve execution of arbitrary code to build the resulting structure. This may be exploited in a number of ways, including code selection based on instantiation-time flags [6] and optimizations based on pre-computing those values that are computable at instantiation time.

[3]Our simplified implementation would be a full TCP/IP if it implemented IP *options*. These are part of the standard, but are not required for normal operation, and we do not support them yet

3.1 Protocol Building Blocks

The individual protocols in a protocol stack are implemented by a collection of functors that define a layer in terms of the layer below it. Here, we start with a simplified example in which each functor has only one parameter. This code builds a stack containing TCP, IP, Ethernet, and a device interface.

```
functor Tcp(structure Lower:PROTOCOL):
            TCP_PROTOCOL =...
functor Ip (structure Lower:PROTOCOL):
            IP_PROTOCOL =...
functor Eth (structure Lower:PROTOCOL):
            ETH_PROTOCOL =...
functor Eth_Device ():DEVICE_PROTOCOL=...

structure Device_Instance = Eth_Device ()
structure Eth_Instance =
    Eth (structure Lower = Device_Instance)
structure Ip_Instance =
    Ip (structure Lower = Eth_Instance)
structure Tcp_Instance =
    Tcp (structure Lower = Ip_Instance)

val connection=Tcp_Instance.active_open(...)
```

This code produces the standard protocol stack shown in Figure 1.

Each of the protocol functors is parameterized by a structure which must satisfy the PROTOCOL signature. This parameter represents the lower layer protocol. At the same time, each individual functor has an interface defined by a different signature, such as IP_PROTOCOL or TCP_PROTOCOL. These *specific* signatures define the interface for one particular protocol and are defined as enrichments of the PROTOCOL signature. The PROTOCOL signature is *generic* and defines the minimal set of features that every protocol should satisfy, and is therefore analogous to the meta-protocol of the x-kernel. Since PROTOCOL is used to constrain the parameter, only the types and values specified in PROTOCOL are available to each functor from the actual protocol that is supplied as a parameter.

The signatures of the individuals protocols in a stack are derived from the generic signature using a form of specification inheritance expressed by the SML include construct:

```
signature PROTOCOL = sig ... end
signature TCP_PROTOCOL = sig
  ...
  include PROTOCOL
    sharing type ...
  ...
end
```

By including the generic signature in their definition, each of the specific signatures inherits all the types and values declared in the PROTOCOL signature. Sharing constraints bind the types declared in PROTOCOL to types that are declared in the specialized signature. Because the specialized signatures inherit all the types and values of PROTOCOL, they are enriched descendants of the same signature. Any functor that satisfies a specialized signature also satisfies the PROTOCOL signature, and any instantiation of such a functor can be used as a parameter to other protocol functors.

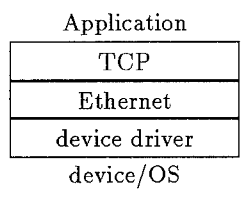

Figure 2: A Non-Standard Protocol Stacks.

To illustrate the modularity afforded by the use of functors to define protocols, we can build a non-standard protocol stack with TCP running directly over Ethernet.

```
    structure Tcp_Over_Ethernet =
        Tcp (structure Lower = Eth_Instance)
```

This is a special-purpose protocol that reliably sends packets of any length, but only between hosts attached to the same Ethernet. The protocol stack is shown in Figure 2. With such a protocol, it might be desirable to turn off TCP checksums, which are expensive to compute and not as strong as the Ethernet Cyclic Redundancy Check.[4] By adding to the TCP functor a new parameter that specifies whether checksums should be computed, we can specify different behavior for different instances of TCP.

```
    functor Tcp (structure Lower: PROTOCOL
                 val compute_checksums: bool):
                   TCP_PROTOCOL = ...
    structure Tcp_Instance =
        Tcp (structure Lower = Ip_Instance
             val compute_checksums = true)
    structure Tcp_Over_Ethernet =
        Tcp (structure Lower = Ethernet_Instance
             val compute_checksums = false)
```

In practice, each functor has several such parameters.

Through the use of functors we have constructed truly modular building blocks that may be "glued" together by instantiation of these functors. Since the interface to the lower-level protocol is constrained to the generic protocol signature, we now have the components of a modular implementation of the TCP/IP network protocol. These components can be used as building blocks to build other interesting protocols. The example of a non-checksumming TCP over Ethernet describes a protocol stack that is potentially much faster than the regular TCP/IP and therefore attractive for applications that only need to communicate across a single Ethernet. This protocol can be built by re-using the TCP/IP code without changing either the TCP or the Ethernet code. With these software building blocks we thus achieve a high degree of code re-use, an important software engineering goal, without sacrificing modularity or type safety.

We can use functors to build what the x-kernel project calls *micro-protocols*, small protocols which can be added to a stack to achieve particular goals. For example, we might have a micro-protocol whose function is to checksum outgoing data and discard incoming segments with incorrect checksum. Another micro-protocol might sequence data by attaching a sequence number to each outgoing segment and delivering incoming segments to the higher layer in the order specified by the sequence number. Compositions of such micro-protocols could be used to build protocols with the functionality of monolithic protocols such as TCP or IP.

[4] That is, the probability of random bit errors going undetected is higher when using only the TCP checksum than when using only the Ethernet CRC.

3.2 The PROTOCOL Signature in Detail

The TCP/IP protocol stack implementation sketched in the previous section uses the PROTOCOL signature to define the generic interface between layers: the PROTOCOL signature is thus the key to freely composable protocols. Since the compiler enforces strict adherence to the requirements expressed in a signature, all protocols must match the PROTOCOL signature. Furthermore, the specific protocol signatures are all derived from the generic signature using a form of specification inheritance. The design of the PROTOCOL signature therefore plays a central role in the overall architecture of the implementation.

The definition of the PROTOCOL signature is given in Figure 3. This signature is closed, meaning that it does not refer to any external types or modules other than those in the language definition. In particular the types are unconstrained in the PROTOCOL signature, allowing a given protocol implementation to choose their definitions as appropriate for that protocol. The signature of a specific protocol, such as IP_PROTOCOL, will usually specify or constrain these types, for example by defining a specific notion of address suitable for IP implementations. In addition to declaring a number of unspecified types, the PROTOCOL signature defines the types of the arguments and results of the operations common to all protocols so that other protocols may use them without relying on any specific implementation. This is crucial to achieving a modular protocol stack, and fundamental to structured system design.

Research in implementations of networking protocols has produced useful principles to guide the design of good implementations. One of these is that copying the data unnecessarily can be one of the most time-consuming operations operations, and therefore avoiding unnecessary copying is one of the most important achievements of a good design. Another principle is that the control flow of a program should match the data flow, so for example the receive operation, which delivers data from a lower layer to a higher layer, should be implemented as a so-called *upcall* from the lower to the higher layer [3].

With these remarks in mind, we now turn to a more detailed description of the PROTOCOL signature.

The type address is used to identify a peer of the protocol (or application) layered above the protocol represented by this signature. An address_pattern is a specification of a set of possible peers from which a connection request will be accepted. A connection is a handle for sending data and closing or aborting a connection.

The type of incoming messages is the same across all Fox Net protocols, and bound to the abstract data type Receive_Packet.T. This abstract data type provides modular and efficient access to the data and to the message headers and trailers while avoiding unnecessary copying. The same is true for Send_Packet.T, to which the type of outgoing messages is bound.

```
signature PROTOCOL = sig
  eqtype address
  eqtype address_pattern
  eqtype connection
  type incoming_message
  type outgoing_message

  val initialize: unit -> int
  val finalize: unit -> int

  val active_open: address * (connection -> incoming_message -> unit) -> connection
  val passive_open: address_pattern * (connection -> incoming_message -> unit)
                -> (connection * address)
  val close: connection -> unit
  val abort: connection -> unit

  val send: connection -> outgoing_message -> unit

  type control
  type info
  val control: control -> unit
  val query: unit -> info

  exception Initialization_Failed of string
  exception Protocol_Not_Initialized of string
  exception Invalid_Connection of connection * address option * string
  exception Bad_Address of address * string
  exception Open_Failed of address * string
  exception Packet_Size of int
end (* sig *)
```

Figure 3: Signature PROTOCOL

A protocol may need to acquire system resources before it can operate. The lowest layer of the Fox Net, for example, must acquire resources from the operating system[5] so that it can communicate with the device driver.[6] These resources must be explicitly released once the application no longer needs them, because the resources include hardware devices or operating system resources that will not be released by the SML garbage collector. The `finalize` call releases all resources held by the protocol and the corresponding `initialize` call allocates the resources. Both functions may be called multiple times, but resources are only allocated on the first `initialize` and released on the matching `finalize`. The count of unmatched initialize calls is returned by both functions.

A connection can be opened either by `active_open` or `passive_open`. Both of these functions take as one of their arguments a packet handler function that will be called when a packet is received. The `connection` value returned by either of the open calls can be used to `send` any number of packets, and finally to `close` or `abort` the `connection`. The difference between `close` and `abort` is that the latter will terminate even if the peer is no longer reachable.

A protocol always has a `send` function, but needs no `receive`: as data to be sent is given to the protocol when the higher layer protocol calls `send`, so data to be received is given to the protocol when the lower layer protocol calls the handler that was given as a parameter to one of the open functions. This handler call is an upcall [3] since the lower layer protocol calls a function from a higher layer. Using upcalls for receive helps achieve high performance by allowing the transfer of packets from the bottom to the top of the stack with no context switches. We note that higher-order functions are a clean and easy way of implementing upcalls.

The `control` function implements operations specific to each protocol, and query delivers protocol-dependent information. For example, IP must be configured to use a specific address as a *default gateway*, whereas TCP needs to be told when a packet has been consumed and more packets can be accepted from the network. Likewise, each protocol has information it can return, information that may be different from that of every other protocol. By binding the generic `control` and `info` types to appropriate specific types in the specific signatures, we specify a different set of operations for each protocol; for example the IP `control` type allows the specification of a default gateway address. If a protocol-specific operation or protocol-specific information is needed by a higher-level protocol, as is the case for the elements of the IP header that must be included in the TCP checksum, we can pass an encapsulation of that function as one of the parameters to the higher-level protocol functor; in the case of TCP we pass a function which computes the checksum of the required elements of the IP header. When

[5] Our implementation is built on top of the Mach 3.0 operating system [10]

[6] At the moment, the lowest level of our system communicates with the device driver which resides in the Mach micro-kernel. Our system resides entirely in user space.

```
signature IP_PROTOCOL = sig
  datatype ip_address =
           Address of {ip: ubyte4, proto: ubyte1}

  datatype ip_address_pattern =
           Pattern of {protocol: ubyte1, source_ip: ubyte4 option}

  datatype ip_control =
           Set_Default_Gateway of {gateway: ubyte4}
         | Set_Specific_Gateway of {destination: ubyte4, gateway: ubyte4}
         | Set_Interface_Address of string * ubyte4
         | Disable_Interface of string

  datatype ip_info =
           Info of {max_packet_size: ip_address -> int,
                    interfaces: (string * ubyte4 option),
                    local_address: ubyte4 -> (string * ubyte4),
                    packets_sent: int,
                    packets_received: int,
                    packets_discarded: int}

  include PROTOCOL
   sharing type address = ip_address
       and type address_pattern = ip_address_pattern
       and type incoming_message = Receive_Packet.T
       and type outgoing_message = Send_Packet.T
       and type control = ip_control
       and type info = ip_info
end (* sig *)
```

Figure 4: Signature for the IP Protocol Layer

building custom protocols, only the encapsulation of the lower-level protocol function as parameters to the higher-level functor needs to be re-implemented.

All of the functions are designed to return to their caller in the course of normal operations. When an unusual event is detected, an exception is raised. The PROTOCOL signature defines six generic exceptions that may be raised by any protocol implementation. Protocol implementations only raise exceptions from this group.

3.3 A Specific Protocol Signature

As mentioned earlier, the signatures of specific protocols are derived from the generic protocol signature by constraining the implementations of the types in the PROTOCOL signature using sharing specifications, and by specifying additional operations (if any) specific to that protocol. As an example, consider the signature of the IP protocol, given in Figure 4. This signature binds the type address to the IP address type, correspondingly for address_pattern, binds control and info to corresponding types meaningful for IP, and binds incoming_message and outgoing_message to the types Receive_Packet.T and Send_Packet.T that were described in the previous section.

The IP address and address pattern types are declared as record-valued *datatypes*: the datatype declaration makes the types unique for each structure matching this signature, and the records make the structured values self-documenting. We use record-valued datatypes extensively in our signatures.

The IP protocol definition [8] specifies a 4-byte *IP number* which identifies a host, and a 1-byte field which identifies the particular layer above IP to which packets will be given. Our IP address type specifies both values, since this combination uniquely identifies the peer of the protocol that uses this instance of IP to send data.

The IP address pattern always has a field to identify the layer above IP, and optionally specifies the remote host by its IP number. This flexibility is used to wait passively for packets for a specific protocol, with the packets being either from a specific remote host or from any host. Once a packet matching the pattern is received, the passive open completes and the packet is delivered to the specified handler.

The control type offers four control operations. Two allow the enabling and disabling of a specific interface. We require the local address for the interface to be specified when enabling an interface. The other two control operations allow the specification of a gateway (also known as *router*) for IP packets that must be forwarded to other networks. The gateway can be set either for all packets that cannot otherwise be routed, or specifically for a particular destination address. This control operation can be used when a corresponding *ICMP redirect* packet[7] is received from a gateway.

The info type returns a variety of information used both by the layers above IP and for monitoring the system. In the first category are the maximum packet size and the local address, which are used by TCP and UDP (via functor pa-

[7]ICMP is the Internet Control Message Protocol, which lets IP systems communicate to each other information about the network.

rameters; Ethernet has corresponding functions, which can be used for Tcp_Over_Ethernet). The other values returned are useful for monitoring the system. Some of these values are functions since in general it would be either too expensive or outright impossible for every call to query to compute all the values that may be of interest. Instead we return a function: this delays evaluation, thereby allowing the caller to ask for exactly the required data at the time it is needed.

4 Evaluation

We have completed an initial implementation of the TCP/IP protocol stack in ML. This includes implementations of the TCP, UDP, IP, ARP, and Ethernet protocols, and of an Ethernet device interface. The device interface uses the mach_msg call in the Mach 3.0 micro-kernel to send and receive packets. The implementation is coded in a type-safe extension of Standard ML with functions to access mach_msg and with efficient support for manipulation of 32-bit data.

We have tested and run the entire protocol stack, and it successfully communicates both with other instances of itself and with the standard implementations available under Mach and SunOS.

In this section we compare the Fox Net implementation of the TCP/IP protocol stack to that of the x-kernel version 3.2, released in February 1993.

	Layer					
	Device	Eth	IP	UDP	TCP	Total
Signatures	1	1	6	1	8	17
Functors	3	2	8	2	9	24
Total	4	3	14	3	17	41

Table 1: Number of Modules in each Layer

Figure 1 shows the number of signatures and functors defined in our protocol implementation. This gives an idea of the complexity of the implementation and the extent to which we have exploited the modules system. The number of structures created depends on the exact protocol stack that is being built.

The complexity of the implementation can also be seen by comparing the number of files in the Fox Net and x-kernel protocol implementations, shown in Table 2. In this table, the numbers for IP also include the ARP protocol, and for the x-kernel also the vchan protocol. The Fox Net protocol implementations altogether have about 7,000 non-comment, non-blank lines of code whereas the x-kernel protocol implementations have about 8,400, again showing approximate equality in the complexity of the two systems.

	Device	Eth	IP	UDP	TCP	Total
Fox Net	3	3	14	3	17	40
x-kernel	2	2	20	4	21	49

Table 2: Number of Files in each Layer

The figures in Table 3 give the size of .o files in the x-kernel, and the count of bytes of executable code and static data for the Fox Net. It should be noted that the Fox Net sizes include a significant amount of debugging code.

	Device	Eth	IP	UDP	TCP	Total
Fox Net	82	16	200	46	282	626
x-kernel	11	10	66	14	73	174

Table 3: KBytes of Object Code

4.1 Performance

The performance of a protocol implementation is generally measured by both *latency* and *throughput*. Latency is the time between sending a packet and its reception by the peer, and is measured in seconds. Throughput is the amount of data that can be sent to a peer in a given time, and is measured in bits per second.

In our experiments, the performance is measured on an isolated 10Mb/s ethernet network between pairs of identical 64MB DECstation 125s (with 25MHz MIPS/R3000 CPU's) running Mach 3.0 version MK 83+NETFIX. For these preliminary numbers (the protocols are still under development) the machines were running in multi-user mode and ran the Andrew File System, but were otherwise unloaded.

Measuring latency directly would require either instantaneous communication between the peers or synchronized clocks. Instead of measuring the latency directly, we measure round-trip time by having a protocol instance send a small packet to its peer which responds as quickly as possible by sending a return packet; the time between sending the packet and receiving the response is the round-trip time, which is twice the latency. The round-trip time for each protocol in the Fox Net and the x-kernel is shown in Table 4.

	Round Trip (ms)		
	IP	UDP	TCP
Fox Net	19.6	22.5	54.8
x-kernel	3.5	3.9	4.9

Table 4: Measured Round-Trip Time of Protocol Stacks.

Directly measuring throughput of an unreliable protocol such as IP or UDP is only feasible if the receiver is at least as fast as the sender. We have chosen instead to measure throughput using a simple flow control strategy in which confirmation messages are transmitted by the receiver back to the sender to signal that all transmitted data has been received and more data can be sent.

Throughput in general is known to be affected by the size p of packets sent as well as by whether the packets are *fragmented* to fit large segments in smaller hardware packets; with our flow control scheme throughput is also affected by n, the number of bytes before a confirmation message is sent. The TCP protocol has its own flow control and segmentation mechanisms, and its performance is less dependent on n or p than on the size w of the TCP *window* parameter, which plays a role similar to n.

In the x-kernel, fragmenting a large packet is faster than sending smaller unfragmented packets, whereas in the current implementation of the Fox Net the opposite is true.

Because of these factors there can be no single number representing the throughput of a protocol, and no implementation is likely to be "faster" than any other implementation for all values of n, p, and w. We have chosen to report the performance of our system with $n = 24,576$, since this is the size of the network data buffers reserved by the version of the Mach operating system that we use, and allows us to operate reliably even if the receiver is slow. We have chosen to present performance both with large fragmented packets, $p = 24,576$, and with the largest packets that can be sent without fragmenting, using $p = 1472$ for UDP and $p = 1480$ for IP. For TCP we have arbitrarily chosen a window size $w = 4,096$, which is the window size used by default on many systems in common use.

All our throughput tests measure the real time needed to send approximately 2 million bytes of data. The results are shown in Table 5.

	Speed (Mb/s)				
	IP	IP-frag	UDP	UDP-frag	TCP
Fox Net	2.3	1.3	2.1	1.4	0.3
x-kernel	2.2	4.4	2.0	4.3	2.4

Table 5: Measured Throughput of Protocol Stacks.

Table 4 shows the latency of the Fox Net to be between six and ten times greater than for the x-kernel. Table 5 shows that the throughput of the Fox Net is equal to the x-kernel for unfragmented IP and UDP, but up to eight times less for TCP. Since the latency is measured with small packets and the throughput with large packets, this suggests that our per-packet costs are relatively higher than those of the x-kernel, whereas our per-byte costs, which usually dominate when sending large packets, are comparable to those of the x-kernel except when fragmenting data.

The throughput figures tell us the x-kernel is faster when fragmenting than when sending packets without fragmentation; since it takes more work to fragment and reassemble a packet than not to, it is fair to assume that the x-kernel fragmentation code is especially optimized, which is not true for the Fox Net. Optimizing the Fox Net code to the same degree as the x-kernel might yield comparable throughput.

The throughput and latency figures both show the Fox Net implementation of TCP to be substantially slower than the x-kernel implementation. The latter is derived from the BSD implementation available in most versions of Unix, and is a highly-tuned implementation of a complex protocol; it is therefore not surprising that it is substantially faster than our implementation, which was entirely written by one of the authors in less than a year.

Because there are ample "opportunities for optimization"[8] still ahead of us, we regard these performance figures as preliminary. It is however satisfactory to note that the throughput of our IP and UDP protocol implementations is already in the same order of magnitude as that of production implementations that are in common use.

4.2 Programming Language

The features of SML that helped us in our work include the module system, static type checking and type inference, higher-order functions, and automatic memory management. Perhaps because we are building systems which are designed to be implemented using monomorphic languages, one feature we have not used extensively is polymorphism.

Modules are crucial for structuring the system, and functors in particular permit parameterization of system components, which greatly improves code re-use. Specifically, we have shown in this paper how signatures and functors may be used to express directly the x-kernel notions of meta-protocol and micro-protocol.

Higher-order functions are also used heavily. For example, in a protocol, send is a higher-order function that yields a procedure of type `outgoing_message -> unit` when given a `connection` as an argument. The use of a functional result allows us to arrange for send to compute a specialized message-sending procedure that takes advantage of specific characteristics of the connection. Thus, that part of the cost of sending messages that is specific to the connection may be amortized over multiple message-sends. Another example of the use of higher-order functions is seen in `active_open`, which takes a functional argument representing the standard notion of an upcall. Lower-level protocols can use the upcall function to handle incoming messages, thereby eliminating many of the costs of data copying and context switching that might otherwise occur as messages proceed from layer to layer.

Automatic memory management has helped us avoid many painful bugs and achieve high performance. However, extended pauses adversely affect the performance of a protocol which is expected to react to an incoming message within milliseconds. Furthermore, we observe that such pauses can cause packets in an unreliable protocol to be dropped. Thus, we plan to replace the current stop-and-copy garbage collector with an incremental garbage collector that will cause fewer extended pauses.

The Fox Net does make use of polymorphism, for example in the signature and implementation of utility modules which are used at different types. However, neither the `PROTOCOL` signature nor the specific signatures have any polymorphic types. It is possible that as we continue to study the problem we will isolate more patterns of control that are common in networking code, and that we will abstract these patterns using polymorphism.

Our FoxML extension to Standard ML has some new types and operations on those types. The new types are byte arrays, 8-bit, 16-bit, and 32-bit unsigned integers, and continuations. Signatures for these types are in Appendix A.

Byte arrays are used to store data in memory contiguously. This allows simple control over the layout of data in memory, to the extent that data can be encoded as bytes, and helps achieve seamless communication with parts of the system that are not written in SML, specifically the operating system and the hardware devices. Both of these expect data to be presented with a specific layout in memory, and byte arrays help us store the data in the proper format. It should be noted that byte arrays only allow for the simplest non-nested layouts, and we have in fact needed to use one unsafe operation to implement the nested data structure required by Mach. We are actively working to address this problem, and in the future we expect to be able to safely control memory layout for data structures.

Unsigned integers of different sizes are useful both to implement specific operations and to store fields of specific size. An example of the former is the TCP sequence number field, the computation of which requires 32-bit modulo

[8] To borrow a phrase from a well-known systems researcher.

arithmetic. An example of the latter is the IP protocol field, which can take values between 0 and 255 and always has one byte reserved for it in data structures. We are able to store these unsigned integers into byte arrays and retrieve them from byte arrays. To achieve good performance we have restricted the byte array indexing operations to work only with indices that are multiples of the size of the value. In general, we have been able to confirm the experience of others that careful control of data representations and layout is crucial in realistic systems programming.

Continuations are created by `callcc` and invoked by `throw` [7]. We have used continuations to write, entirely in FoxML, a coroutine package that implements all the multithreading required by our current implementation. Likewise, we have been able to implement a simple timer package using only FoxML. Since the TCP protocol is designed to be implemented using a number of timers and threads, being able to implement the coroutine and timer packages in FoxML has given us the benefit of custom threads and timer packages without having to debug a threads package written in an unsafe language.

Though our software is by no means bug-free, there are classes of bugs that are common in large software systems that we have not seen, most significantly undetected memory usage errors. We do not suffer from null pointer dereferencing, incorrect allocation, incorrect de-allocation, or from undetected access outside of an array. It is well known that these are some of the most common and insidious of errors. In fact we have been able to find multiple such bugs in the SML runtime system and in the Mach 3.0 micro-kernel, both written in C. We attribute this not to exceptionally careful design on our part, but to the emphasis on type safety in SML.

With three people working full-time on the software, we have observed that in most cases when an interface changes, its signature will change and the compiler will report any module that depends on the interface and has not been updated. It is not an unusual experience for us to have multiple people significantly change a number of modules, test them individually, fix any incompatibilities unearthed by the compiler, compile them, and have everything run flawlessly the first time. Such behavior is not normally observed with programs written in other languages.

5 Concluding Remarks

We have designed and built a modular implementation of a standard and widely used network communications protocol. The division of the system into independent modules with a well-defined interface between protocols ensures that protocols can be composed into a variety of protocol stacks without sacrificing type safety. The design hinges on a generic protocol signature that contains exactly those types and values that we expect every protocol implementation to support. This signature is enriched and specialized to produce the specific signatures for the individual protocols. Protocol implementations that satisfy a specific signature also satisfy the generic signature. Our protocol implementations are parameterized to run on top of any protocol that satisfies the generic signature, and can therefore be easily composed to form different special-purpose protocols.

Our design makes heavy use of higher-order functions and the modules system of SML. In fact, we have found that many of the standard techniques for structuring operating systems and network communication systems can be expressed directly and elegantly in SML. Thus, in contrast to C, we have been able to explain system-structuring techniques in concrete terms (that is, expressing them as SML code), as opposed to relying on the abstract and usually *ad hoc* descriptions typical of the systems-programming literature.

In the near future, we plan to improve the performance of our system through profiling, identifying bottlenecks, optimizing our code, and improving the compiler. We also plan to continue improving the language to allow clean, efficient, and type-safe expression of low-level operations. We plan to add new protocols so we can experiment with more combinations. Eventually, we would like to extend our work to other areas of systems programming.

Specific features that we would like to see in the language are type abbreviations in signatures, which would let us specify types in signatures without using datatypes, and safe ways to build data structures with control over the layout as well as the ability to specify that certain memory structures may not be relocated by the garbage collector.

Our work has also helped us identify desirable improvements to our current language implementation, which is derived from the SML/NJ compiler. For example, the current implementation is unable to generate invariants for loops and use them to reduce the amount of work to be done inside the loop. This could be used for example to eliminate bounds check inside loops which access arrays, e.g. copy loops. Another interesting challenge is to allow some separate compilation, but have the final optimization of the resulting code occur at link time. Such optimizations would let the compiler specialize polymorphic functions on the types of values that will actually be supplied to them, do cross-module register allocation, and so on. Finally, it is possible to imagine that further delaying code generation until runtime might also prove beneficial. For example, when our `send` routine returns a function that is specialized for the connection, it is possible that a run-time code generator might be invoked to generate specialized and highly optimized code for this function.

An area of considerable interest within the network community is Integrated Layer Processing (ILP). A program that has been restructured for ILP will combine all its manipulation on incoming or outgoing data into a single integrated loop, so a single pass is made over the data. This has significant performance benefits, but as traditionally implemented results in a breakdown of modularity, since the code for all layers of the protocol stack must be present in the single loop over the data. From the point of view of compilation, ILP is reminiscent of deforestation; we suspect that a compiler performing suitable optimizations, perhaps involving link-time or run-time code generation, might achieve much the same performance as ILP while preserving source code modularity.

Our TCP/IP implementation is a first step towards establishing the feasibility and usefulness of advanced programming languages for developing real-world systems, particularly network communications systems. While the performance of our implementation in SML does not yet match that of the most highly tuned C implementation, we are encouraged to have already achieved reasonable throughput, with as yet little tuning or performance analysis.

Besides performance, software quality considerations are also important. Here, we can already claim to have implemented a system with important reliability benefits. In particular, the language guarantees that many common errors will not happen: our system will not "dump core". We are keeping records of our software development process, and plan to present more details in a future paper.

6 Acknowledgements

We would like to thank Ken Cline, Elmootazbellah Elnozahy, Nick Haines, Greg Morrisett, and Eliot Moss for their assistance and contributions to the Fox Net project. Brian Bershad, Alessandro Forin, David Golub, Chris Maeda, and Mary Thompson have made available to us valuable information about Mach, and Larry Peterson, Hilarie Orman, and Ed Menze have helped us with the x-kernel and given us much encouragement. Matthias Felleisen and anonymous reviewers have read early drafts of the paper and contributed to its quality.

References

[1] Mark B. Abbott and Larry L. Peterson. Automated integration of communication protocol layers. Technical Report TR 92-24, Department of Computer Science, University of Arizona, December 1992.

[2] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, September 1990.

[3] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th Symposium on Operating Systems Principles*, Orcas Island, Washignton, December 1985. ACM.

[4] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications*, 27(6), June 1989.

[5] J. D. Day and H. Zimmerman. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[6] Nicholas Haines, Edoardo Biagioni, Robert Harper, and Brian G. Milnes. Note on conditional compilation in Standard ML. Technical Report CMU-CS-93-172, School of Computer Science, Carnegie Mellon University, June 1993.

[7] Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *The Journal of Functional Programming*, 1994. to appear.

[8] USC Information Sciences Institute. Internet protocol. RFC 791, September 1981.

[9] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.

[10] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi. Mach: A foundation for open systems (operating systems). In *Workstation Operating Systems: Proceedings of the Second Workshop on Workstation Operating Systems*, September 1989.

A Signatures of SML Extensions

```
type 'a cont
val callcc: ('1a cont -> '1a) -> '1a
val throw: 'a cont -> 'a -> 'b

signature BYTEARRAY = sig
  eqtype bytearray
  exception Size
  exception Subscript
  exception Range

  val array: int * int -> bytearray
  val sub: bytearray * int -> int
  val update: bytearray * int * int -> unit
  val length: bytearray -> int

  val extract: bytearray * int * int -> string
  val fold: (int * 'a -> 'a)
        -> bytearray -> 'a -> 'a
  val revfold: (int * 'a -> 'a)
           -> bytearray -> 'a -> 'a
  val app: (int -> 'a) -> bytearray -> unit
  val revapp: (int -> 'a) -> bytearray -> unit
end

(* UBYTES matches Byte1, Byte2, Byte4 *)
signature UBYTES = sig
  eqtype ubytes
  type bytearray

  val + : ubytes * ubytes -> ubytes
  val - : ubytes * ubytes -> ubytes
  val * : ubytes * ubytes -> ubytes
  val div: ubytes * ubytes -> ubytes
  val mod: ubytes * ubytes -> ubytes
  val min: ubytes * ubytes -> ubytes
  val max: ubytes * ubytes -> ubytes

  val > : ubytes * ubytes -> bool
  val >= : ubytes * ubytes -> bool
  val < : ubytes * ubytes -> bool
  val <= : ubytes * ubytes -> bool

  val print: ubytes -> unit
  val makestring: ubytes -> string

  val << : ubytes * int -> ubytes
  val >> : ubytes * int -> ubytes
  val !! : ubytes -> ubytes
  val xor: ubytes * ubytes -> ubytes
  val || : ubytes * ubytes -> ubytes
  val && : ubytes * ubytes -> ubytes

  val to_int: ubytes -> int
  val from_int: int -> ubytes
  val update: bytearray * int * ubytes -> unit
  val sub: bytearray * int -> ubytes
end
```