A Structured TCP in Standard ML*

Edoardo Biagioni

Abstract

This paper describes the design of an implementation of the Transmission Control Protocol using an extension of the Standard ML (`SML`) language. `SML` supports higher-order functions, modularity, and type-safe module composition. We find that by using `SML` we can achieve good structure and good performance simultaneously. Good structure includes a modular decomposition of the protocol stack and of the TCP implementation, a control structure that imposes a total ordering on all events and processes them synchronously, and a test structure that allows component testing to catch problems before system integration. Strategies that help achieve good performance include using fast algorithms, using language constructs that make it easy to stage function evaluation, and language implementation features such as compacting garbage collection.

1 Introduction

The design of programming languages has made great strides in the past twenty years. However most programs, and especially networking and systems programs, are still written in languages that already existed twenty years ago or in languages that carry all the baggage of twenty-year-old languages and have but few of the features of advanced programming languages. It is hard to find significant systems projects implemented in advanced languages.

These statements raise the parallel questions of whether it is possible to improve the practice of systems programming by using advanced languages, and whether advanced languages really are well-suited to systems programming. The Fox Project is addressing both questions by producing implementations of network code in the Standard ML (`SML`) [12, 11] programming language.

*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 number F19628-C-0168.

To appear in SIGComm, August 1994, London, England.

It is common practice today to write systems and networking software in either `C` or `C++`. The advantages of these languages include the speed of the compiled programs, the ability to express low-level operations, and the compatibility with existing systems. These advantages are considered more important than the drawbacks of these languages, including lack of modularity, lack of typing, lack of automatic storage management for dynamic data, and in general lack of safety: type safety, which prevents type and argument mismatches, storage safety, which prevents memory misuse, and module safety, which prevents violations of data abstraction boundaries. Advanced languages such as `SML` address these problems, but have traditionally been considered too inefficient, and insufficiently expressive of low-level operations, to be useful for systems programming.

The Fox Project has used an extension of `SML` to develop an implementation of the TCP/IP protocol stack. Our goals have been to explore the advantages (and drawbacks) of developing networking software in advanced languages and to study the features that advanced languages need to allow the expression of operations common in systems software. We have chosen to implement the TCP/IP protocol suite not because it is especially simple or well-structured, since it is neither, but because it is relatively well-understood and frequently used and implemented, and can therefore be taken to represent much of the common practice of systems programming.

In this paper, we concentrate on the design of our implementation of the TCP protocol and on its realization in `SML`. In particular, we describe the implementation's module structure, which reflects the structure of the specification, the control structure, which is synchronous except for the order in which asynchronous events are queued, and the test structure, which allows detailed and accurate component testing.

We claim that the combination of structured design and advanced languages can lead to implementations of systems code that are more robust than currently available implementations, and that the the software engineering cost of building and maintaining these implementations may be substantially less than for current implementations. We also believe that structured implementations in advanced languages can be made to be as fast as comparable implementations in other languages, though the maturity of our current implementation is as yet insufficient to demonstrate this.

Section 2 briefly describes some the features of `SML` that we refer to in describing our implementation; section 3 describes the general structure of our protocol stack; and section 4 describes in detail the structured aspects of our implementation of TCP. Section 5 reports on and discusses the performance of this implementation, Section 6 describes related work, and section 7 summarizes our results.

2 Programming Language

The Standard ML programming language is a typed polymorphic language with type inference, automatic storage management and garbage collection, higher-order functions, and a well-developed module system [12]. We treat each of these in turn.

In a typed language, only operations defined on particular types can be applied to values of these types. For example, in SML it is impossible to dereference an integer or add two pointers. SML is typed, but also polymorphic, so SML does allow functions that can be applied to values of different types; the restriction is that the function implementation must be such that no knowledge is required about the type. For example, the `length` function returns the length of its argument, whether it be a list of reals, integers, or a list of elements of any other type. SML also has type inference, which means the programmer almost never needs to explicitly specify the type of an expression; the compiler automatically infers the most general possible type for each expression, and only rejects the program if no legal assignment of types to expressions is possible.

Typing helps prevent type mismatches such as passing an Ethernet address to a function from the IP layer.

Storage management in SML is automatic, as it is for C and C++ stack variables but not for C or C++ heap storage. Because of typing, there is no way for a program to get at the bit representation of a pointer. SML's garbage collector can take advantage of this to move data and compact the heap, allowing the use of fast garbage collection algorithms [1]. With a compacted heap, heap allocation can be fast (as fast as stack allocation or more so), and automatic storage management can therefore be much faster than the heap memory management of C or C++ [21].

Automatic storage management is useful when memory requirements vary with time. For example IP fragment reassembly may on occasion need buffers for reassembling a large number of packets simultaneously, but normally won't. In such cases, manual storage management is often slow, may be hard to use, and always poses the danger of storage leaks and memory misuse in case of programming errors. In contrast, automatic storage management can be faster, is simple, makes it harder to inadvertently program storage leaks and, in combination with a typing system, completely eliminates memory misuse.

There is legitimate concern over the pauses that are often associated with garbage collection. With a generational copying garbage collector such as provided by the SML/NJ compiler, however, most pauses are short and generally interfere with the operation of a protocol no more than the pauses imposed by the multitasking operating system. We return to this point in Section 5.

Higher-order functions are functions that are available as values at run-time, and functions that may constructed and returned by other functions as their result. In languages with higher-order functions, functions are constructed at run-time by associating predefined code with an environment constructed at run-time; the association of a function and an environment is called a *closure*. Higher-order functions can be used to stage a computation. For example, a function can take one argument, perform part of the computation, and return a new function that will complete the computation. In the closure of the returned function are the partial results from the earlier part of computation. If the returned function is used multiple times, the cost of the first part of the computation is amortized over the uses of the returned function. In languages that provide higher-order functions, the partial results need not be available to any but the function that will use them, thus preventing inadvertent misuse.

Our project uses higher-order functions to stage the send computation, for example. In each layer of our protocol stack the call to send is staged so the early stage accepts a connection and figures out which lower-layer send function to call, and the later stage can directly use this send function. Other optimizations are also possible.

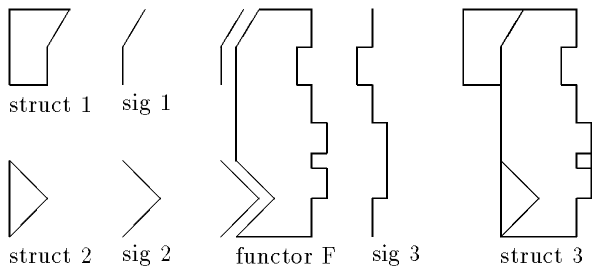

Figure 1: Geometric Analog of the SML module System.

Modules constructs in SML include *signatures*, *functors*, and *structures*. Signatures are interface descriptors; structures are modules with types and values (values include functions); and functors are modules that are parametrized by types, values, and structures, and yield new structures each time they are *instantiated* with their parameters. Figure 1 shows how structures, functors, and signatures fit together.

A structure *satisfies* a signature if it contains all of the values, types, and sub-structures specified in the signature. A structure may satisfy multiple signatures, and conversely there may be multiple structures that satisfy the same signature.

Functor instantiation is used to build new structures, and has the same purpose as the link phase of conventional compilers. In our project, as discussed below, we use functor instantiation to build protocol stacks using only the SML language, rather than a special linker/loader.

Our project uses an extended SML which has first-class continuations, 1-byte, 2-byte, and 4-byte unsigned integer types, and in-lined byte arrays. Further extensions are under consideration.

Continuations are control constructs used to capture the

```
signature PROTOCOL = sig
  eqtype address
  ...
  val active_open: address
        * (connection -> incoming_message -> unit)
        -> connection
  val send: connection -> outgoing_message -> unit
  ...
end
signature TCP_PROTOCOL = sig
  eqtype lower_layer_address (* not specified here *)
  datatype tcp_address =
        Address of {remote_peer: lower_layer_address,
                    local_port: ubyte2,
                    remote_port: ubyte2}
  ...
  include PROTOCOL
   sharing type address = tcp_address
          ...
end
```

Figure 2: Derivation of Specific Signatures from the Generic Signature.

notion of the "remainder of a computation". Continuations are useful in implementing coroutines, events, timers, and other control abstractions.

Unsigned integers of specified size are generally useful in implementing networking protocols; the `SML int` type is inadequate in number of bits (31 bits on the MIPS), in being signed, and in the operations it supports. The new types support bit-wise logical, shift, and arithmetic operations, and specify the number of bits in a value independently of the machine's word size. We currently use these logical and shift operations also to access the bits in a word.

On most implementations of `SML`, normal arrays contain pointers to the objects stored in the array. Our project has added arrays of bytes stored in-line, giving `SML` programs direct (but safe) access to memory. These byte arrays can be used to memory-map devices or to communicate with the operating system, so that each byte array provides a limited segment of unsafe memory that can be used to store integers and integers of specified size. The Fox Project is currently studying the possibility of using structured flat types as an alternative to using byte arrays, but this work is not reported here.

The SML/NJ compiler [2] provides continuations and byte arrays; our project has extended it to support integers of specified size.

3 Structure of the Protocol Stack

The design of our protocol stack is modeled after that of the x-kernel [16], developed at the University of Arizona. In the x-kernel architecture each protocol presents essentially the same interface to the outside world. This makes it possible to combine protocols in new and useful ways, for instance by having an instance of TCP run directly over ethernet, without IP, or by replacing the monolithic TCP protocol with a suite of protocols which together perform the same functions.

```
structure Device = ...
structure Eth = Eth (structure Lower = Device ...)
structure Ip = Ip (structure Lower = Eth ...)
structure Standard_Tcp =
     Tcp (structure Lower = Ip
          val do_checksums = true ...)
structure Special_Tcp =
     Tcp (structure Lower = Eth
          val do_checksums = false ...)
```

Figure 3: Assembling the standard TCP/IP stack and a non-standard stack.

We have followed the x-kernel in having a single interface for all of our protocols; this common interface allows protocols to be composed arbitrarily, for example by running TCP directly over ethernet. Unlike the x-kernel, our interfaces are defined formally as signatures, and syntactic compliance of an implementation with the interface is checked by the compiler to prevent type errors and argument mismatch. Also unlike the x-kernel, the compiler can check that any composition of layers is proper and that all the functions required of "the layer below TCP", for example, are present as functor parameters before allowing the composition.

The general structure of our protocol stack, shown in Figure 2, has been described in detail elsewhere [3] and will only be summarized here. We have a signature `PROTOCOL` which is *generic* in that it is satisfied by all the modules implementing each of the protocols in the stack. From this signature we derive *specific* signatures for each of the protocols such as IP, TCP, and Ethernet by *including* the generic protocol, and thereby specifying all the objects specified by the generic signature, and also *sharing* any types from the generic signature that are to be specialized in the specific protocol. This style of derivation guarantees that all the objects required by the generic signature are also required by the specific signature, and therefore that any structure that matches the specific signature also matches the generic signature.

We define a protocol implementation as a structure that satisfies the corresponding protocol-specific signature. This structure is usually created by instantiating (applying) a functor on specific configuration parameters, such as the initial window size to be used for connections in TCP. One of the parameters to every composable protocol is the protocol below it, specified as a structure that satisfies the `PROTOCOL` signature. Since the `PROTOCOL` signature is satisfied by any protocol that satisfies a specific signature, any valid protocol implementation can be used as the protocol below any other protocol.

Figure 3 shows the assembly of some protocol stacks defined using this module structure. The standard stack has TCP over IP over Ethernet, and checksums are enabled as required by the standard. The special stack runs TCP directly over ethernet; this stack can only be used to communicate among machines on the same Ethernet, and we can therefore turn off checksums since the Ethernet's CRC[1]

[1] And the absence of routers which might corrupt the source address.

```
functor Tcp
    (structure Lower: PROTOCOL
     structure Aux: IP_AUX
     sharing type Lower.outgoing_message = Send_Packet.T
         and type Lower.address = Aux.address
         and type Lower.address_pattern =
                     Aux.address_pattern
         and type Lower.incoming_message =
                     Aux.incoming_message
         and type Lower.connection = Aux.connection
     val initial_window: int
     val compute_checksums: bool
     val abort_unknown_connections: bool
     val user_timeout: int
     structure Scheduler: COROUTINE
     structure B: FOX_BASIS (* our utilities *)
     val do_prints: bool
     val do_traces: bool): TCP_PROTOCOL =
```

Figure 4: The Functor Parameters for TCP.

makes the weaker TCP checksums redundant.[2]

All this can be done with the x-kernel. In the x-kernel, however, the stack is defined using not the C language, but a specialized language with its specialized compiler. In our system, stack composition is defined entirely in SML.

The actual functor implementing TCP takes several parameters, as shown in Figure 4. The first is the lower protocol; the second is an auxiliary structure which provides functions that TCP requires from the lower layer, but that are not present in the PROTOCOL signature. The signature, shown in Figure 5, is called IP_AUX to reflect the fact that these functions are traditionally either supplied by the IP layer or dependent on the type of the IP address, and are required due to TCP's dependence on values carried in the IP header. A structure satisfying this signature must be supplied as a parameter to the UDP functor as well. Note that with this structure, any change in the definition of IP (for example, from IP version 4 to version 7) will affect the IP implementation and the Auxiliary structure, but not TCP.

The remaining functor parameters can be used to customize TCP. The parameter abort_unknown_connection is set to false when we wish to run our implementation for brief periods of time on a workstation without disturbing connections that were set up by the resident operating system. The user timeout specifies the length of time before hung operations fail. The scheduler structure provides the thread creation and sleep operations needed to correctly implement TCP; as the signature name, COROUTINE, suggests, this scheduler is non-preemptive, so thread switch will only occur when a scheduler function is invoked and data structure locks are therefore not necessary[3]. In the current implementation, the scheduler is implemented entirely in SML using continuations[4], and the cost of a thread switch is the cost of only a few function calls: on a DECstation 5000/125, the cost of a call to an empty function is about $1.2\mu s$, whereas the time required by our scheduler to create a thread, terminate the current thread, and switch to the new thread is approximately $30\mu s$. This time includes the time to update the scheduler data structures.

```
signature IP_AUX =
 sig
  type connection
  type address
  type address_pattern
  type incoming_message
  val hash: address -> int
  val eq: address * address -> bool
  val makestring: address -> string
  val default_pattern: address_pattern
  val info: incoming_message
          -> {src: address, checksum: ubyte2,
              data: Receive_Packet.T}
(* check computes the pseudo-header checksum *)
  val check: address -> ubyte2
  val mtu: connection -> int
 end (* sig *)
```

Figure 5: The Auxiliary Signature used by TCP and UDP.

4 Structure of the TCP Implementation

We designed the TCP implementation to have the same structure as the TCP standard. This structure has simplified the initial implementation and it is our expectation that it will simplify subsequent maintenance as well. For example, the receive procedure is described in the standard as a procedure with branch points and merge points, but no loops (a directed acyclic graph). We have implemented the receive code by implementing exactly the branches specified in the standard, using functions as labels for the merge points. Not only does this structure make it easier to get the code to conform to the standards, it also makes it easier to refer to the standard when modifying the structure to make sure that the code still implements the correct procedure.

Since, as discussed by Clark and others [7], many of the costs of conventional TCP processing are related to buffering, checking the data, and copying between layers, and are therefore unrelated to performing protocol-specific tasks, we expect any performance penalty due to the clean structure to be small compared to the total costs of the protocol implementation.

We have however implemented fast-path receive and send routines which handle the normal cases quickly, and defer to the full code for the less common cases.

The standard description of the TCP protocol [8, 15] uses a connection state as well as a data structure, the latter being referred to as a *Transmission Control Block* (TCB). We designed a module, called Tcb, to define the types with which these data structures are represented and some basic operations on values of these types. Each connection value returned by an open call has a field to store the connection state, with a component of this variable to store the TCB

[2] A reviewer pointed out that widely available Ethernet implementations do not correctly implement the CRC. In light of this, the TCP described here is only useful if there is specific knowledge that the Ethernet implementation implements the CRC correctly. We have chosen to retain the example for illustration.

[3] Synchronization is required in particular cases, such as to insure that no data is delivered on a connection until after the corresponding open returns to the caller.

[4] Except for the code to read the system clock, which sometimes calls Mach.

```
signature TCP_TCB =
 sig
  ...
  structure Q: FIFO
  structure D: DEQ
  datatype tcp_tcb =
      Tcb of {iss: ubyte4,
              snd_una: ubyte4 ref,
              ...
              queued: Send_Packet.T D.T ref,
              out_of_order: tcp_in Q.T ref,
   (* to_do contains the actions to perform *)
              to_do: tcp_action Q.T ref}
  datatype tcp_state =
      Closed of tcp_action Q.T ref
    | Listen of tcp_tcb * int
    | Syn_Sent of tcp_tcb * int
  (* Syn_Active: active open Syn_Received *)
    | Syn_Active of tcp_tcb
  (* Syn_Passive: passive open Syn_Received *)
    | Syn_Passive of tcp_tcb * int
    | Estab of tcp_tcb
    | Fin_Wait_1 of tcp_tcb * bool
    | Fin_Wait_2 of tcp_tcb
    | Close_Wait of tcp_tcb
    | Closing of tcp_tcb
    | Last_Ack of tcp_tcb
    | Time_Wait of tcp_tcb
  ...
 end (* sig *)
```

Figure 6: Excerpts from the TCB module signature.

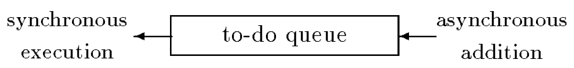

Figure 7: Quasi-Synchronous Control Structure.

(except in state Closed). Parts of the signature of the Tcb module are shown in Figure 6.

Note in Figure 6 that the TCB data structure provides a field, to_do, which stores the actions that must be done on behalf of this TCP connection. One of the early design decisions of this implementation was that the behavior of the implementation should be, whenever possible, a time-independent sequence of discrete events. Timer expirations and message receptions are asynchronous, but when these events occur they are synchronized by placing a corresponding action in the to_do queue for the connection. Since this control structure is synchronous except for the exact time and sequence in which actions are placed on the to_do queue, we call the control structure *quasi-synchronous*. This structure is illustrated in Figure 7. The actions placed on the to_do queue have type tcp_action, defined in the Tcb module and shown in Figure 8.

The control structure of our TCP is therefore very simple: executing an operation computes the corresponding actions and queues them onto the connection's to_do queue.

Which thread executes the actions on the to_do queue is a design decision; in the current implementation, the thread executing an operation then executes actions, one at a time, until at least those actions it placed on the queue have completed execution. A different choice might lead to a dedicated thread executing all of the actions for a connection. Actions are designed not to wait; instead, they can start timers or queue other actions for later execution if other actions must be performed at a later time.

```
datatype tcp_action =
    User_Error of string
  | Send_Segment of tcp_out
  | Process_Data of tcp_in
  | User_Data of Receive_Packet.T
  | Set_Timer of tcp_timer * int
  | Clear_Timer of tcp_timer
  | Timer_Expiration of tcp_timer
  | Close_After_Sends
  | Complete_Open of bool
  | Complete_Close of bool
  | Complete_Send of ubyte4
  | Probe_Window
  | Delete_Tcb
```

Figure 8: TCP Actions.

With the quasi-synchronous control structure, only the exact sequence in which actions such as Process_Data or Timer_Expiration are added to the queue is undefined; once the actions have been placed on the queue the behavior of TCP is completely deterministic and testable.

The quasi-synchronous control structure and other top-level TCP operations are implemented by the module Main.

Based on this control structure and the Tcb module, we have defined several modules to implement the TCP state machine and the actions to be taken on receipt of a packet. The module State performs the main state manipulations required on connection open, close, or abort, and also when a timer expires.

The module Receive processes incoming segments and performs any necessary state manipulations. Receive uses Resend to implement the round-trip time computations developed by Karn and Jacobsen [15], and to remove acknowledged segments from the retransmit queue.

The module Send segments outgoing data and places corresponding Send_Segment actions onto the to_do queue. Note that this subdivision is similar to that in the main specification for TCP, RFC 793.

Thanks to their deterministic nature, each of these modules can be tested in isolation. For each module we have written test code that is run whenever the code is modified; though the test code does not offer complete coverage, it helps point out implementation defects by comparing the TCB produced by the operation with the TCB expected in accordance with the standard. These test programs can be extended to test for specific bugs that might be hard to reproduce in a complete implementation. With these test modules we have been able to easily locate several minor bugs that would have been time-consuming to locate with conventional techniques. It is not unusual for us to make a significant change in the protocol implementation, let the compiler locate the syntax and type errors and correct them, let the test modules locate the minor inconsistencies and correct them, then run the protocol and watch it perform flawlessly.

The time-dependent operations are implemented by the module `Action`, which implements timers and segment externalization and internalization. Figure 9 shows the complete module structure of the implementation.

For an example of how our module and control structure work, consider receiving a non-empty packet from the network. The packet is delivered to TCP's handler, which calls the receive function provided by the `Action` module. The receive function computes the checksum and decodes the packet header, then places a `Process_Data` action containing the internalized segment onto the `to_do` queue. When this action reaches the front of the queue, it is given to the `Receive` module which processes the segment and, if the sequence number is the expected one, places onto the `to_do` queue a `User_Data` action with the segment data (if the sequence number is not the desired one, the packet is queued for later use, or discarded if a duplicate). The receive operation may also put onto the `to_do` queue a `Send_Segment` containing an acknowledgement and a `Clear_Timer` for the ack timer, or else a `Set_Timer` for the ack timer if the ack is to be delayed. The `User_Data` action delivers the data to the user when it reaches the front of the queue.

With the quasi-synchronous scheme, we have actually built a custom scheduler as part of the TCP implementation. One advantage of this is that we can easily redefine the scheduling priorities. As currently defined, this scheduler is round-robin and only supports a single priority. By replacing the current FIFO with a priority queue, we could specify that particular actions, e.g., actions which affect the packet latency, be executed with higher priority than others.

5 Performance and Discussion

Since our architecture is inspired by the x-kernel architecture, we have chosen to compare the performance of our protocol stack implementation with that of the x-kernel. In this way, both the advantages and the disadvantages of running in user mode on top of the Mach 3.0 microkernel are factored out, and only the differences due to the programming language used and to the details of the implementation remain.

Other studies [7, 5, 9] have shown that the main obstacles to high throughput in real implementations and for present-day systems are data-touching operations such as copy and checksum. We have optimized our code to copy data as little as possible. Our protocols copy data only once, when delivering a segment to the micro-kernel or when delivering data to the user process[5], though the latter is not reflected in the benchmarks. The Mach 3.0 micro-kernel also copies the data to and from the device, so altogether sending or receiving data causes it to be copied two times.

The copy routines are written in `SML` and require approximately 300μs per kilobyte for word-aligned copies on a DECstation 5000/125; this is approximately one fifth the speed of the library-supplied `bcopy` (clocked at 61μs per kilobyte) and may improve as the code is refined and the compiler code generation is improved: the current compiler fails to optimize accesses to successive elements of arrays and thus checks array bounds on every access and recomputes pointers on every access.

[5] Additional copies might be required when using IP fragmentation or when the protocol above TCP (if any) is using headers which will not fit in a segment. We have not optimized for these cases.

```
fun word_check (n, accumulator, limit) =
     if n >= limit then
      ... (* check odd bytes, renormalize *)
     else       (* the common path *)
      let val byte4 = Byte4.sub (b, n)
          val low = Byte4.&& (byte4, 4uxffff)
          val high = Byte4.>> (byte4, 16)
          val res1 = Byte4.+ (high, low)
          val sum = Byte4.+ (res1, partial)
 (* tail recursive loop. *)
      in word_check (n + 4, sum, limit)
      end
```

Figure 10: Main loop of Checksum Computation.

The checksum code is also written in `SML` and is optimized using the techniques described by Braden, Borman, and Partridge [4]. The current implementation requires as little as 343μs per kilobyte for data aligned on byte-2 (short) boundaries; for comparison, the x-kernel's checksum routine, which uses a slower algorithm, has been clocked at 375μs per kilobyte.

The main loop of our checksum code is shown in Figure 10. The code accumulates up to 16 bits of overflows in the top part of the 4-byte sum; code outside the loop insures that no more than 2^{16} 2-byte quantities are summed, and that $n \bmod 4 = 0$ and $limit \bmod 4 = 0$. The result is renormalized at the end of the loop. The input byte array, `b`, is declared in the scope of the `word_check` function, and need not be passed in explicitly. The arguments to `word_check` are the current index of the computation, the partial result so far, and the limit. The caller of this function must insure that $n \bmod 4 = 0$ and $limit \bmod 4 = 0$; any other values will lead to either an exception being raised or an incorrect result. The notation `4ux` is used to introduce a hexadecimal 4-byte constant value, and like all other `Byte4` operations, is part of the extensions to SML made within the scope of the Fox project.

Because this function is tail recursive, that is, the result returned is the result of the recursive call, the compiler is able to implement this function using loops and keeping the arguments in registers.

Another element that has proven to be slow in many implementations of TCP is the timer code [7]. We have been able to write our own timer code in `SML`, and it is simple and fast. The entire code for starting a timer, clearing a timer, and timer expiration is shown in Figure 11.

The timer uses the `Scheduler` structure (module) mentioned above. This structure is part of the Fox Net implementation. The fork operation and the implicit exit at the end of the thread each take unit time and are quite fast; the sleep queue, a priority queue implemented as a heap, is also

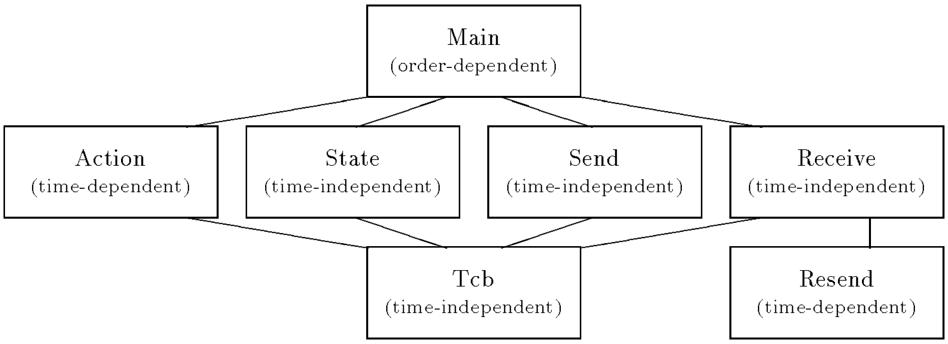

Figure 9: The Module Structure of TCP.

```
fun start (handler, ms) =
     let val cleared = ref false
         fun sleep () =
              (Scheduler.sleep (ms);
               if ! cleared then ()
               else handler ())
         val thread = Scheduler.Normal sleep
     in Scheduler.fork (thread);
        cleared
     end
fun clear cleared = cleared := true
```

Figure 11: Implementation of Timers.

quite fast.

The implementation of `start` allocates from the heap a new updatable boolean cell and creates a new closure for the function `sleep`. Creating such a closure is equivalent to a creating a new `sleep` function with built-in pointers to the updatable cell and to the arguments to the function; the ability to create such functions is part of the first-class functions provided by SML. The newly created boolean is returned to the caller and can be changed to clear the timer. When the forked thread gets to run, it sleeps for the specified time (`Normal` identifies the kind of thread), then only calls the handler if the shared state has not been cleared.

The `clear` function is not pure, that is, works by changing the value of a variable.

A simple timer implementation such as this one depends for performance on having both fast thread creation and switching, and fast heap allocation of the shared state.

To benchmark the throughput of the protocol stack, we have written a program which tries to send large amounts of data in one direction as fast as possible, letting TCP's flow control mechanisms regulate the speed at which data is delivered. We standardize the TCP window size to 4096 bytes, which is a size used by many implementations, and leave the Mach buffer space at its standard 24K bytes[6]. The test consists of sending 10^6 bytes of data between a designated sender and a designated receiver on an isolated 10Mb/s ethernet. The receiver starts a timer, sends the designated sender a small packet specifying the amount of data desired, and stops the timer after all the specified data has been received. The received data is discarded when it is received at the application level.

[6] This is the value currently used by default for installations of Mach 3.0 at our site.

	Fox Net	x-kernel	ratio
Throughput (Mb/s)	0.6	2.5	0.24
Round-Trip (ms)	36	4.9	9.4

Table 1: Speed Comparison of TCP Implementations.

The results are summarized in Table 1, with the x-kernel numbers reflecting the performance of x-kernel version 3.2. All numbers show the performance on 64MB DECstation 5000/125s running the Mach 3.0 microkernel. It should be noted that top-performing TCP implementations can substantially outperform both the Fox Net and the x-kernel implementations [10].

Although SML/NJ has a sampling profiler, it depends on features of Unix that are not supported under Mach 3.0. Instead of using this profiler, we have installed hardware devices containing free-running counters that can be mapped into the address space of the SML task and then read directly from SML programs. One call each to the two functions which start and stop a timing counter require $15\mu s$ altogether to record the initial value of the counter, compute the difference between the initial and the final value, and increase a counter. We call these functions to record the time taken in specific parts of the TCP/IP stack.

A summary of the results of profiling is shown in Table 2. The time spent updating counters is estimated by multiplying the number of times the counters were updated by the $15\mu s$ typically required to update a counter. The miscellaneous figure includes buffer allocation and deallocation, calling Mach to read the system clock, and other utilities. Some of the other figures may include the time spent collecting garbage, since running counter(s) keep running while garbage is being collected; there may also be other overlaps

		Sender	Receiver
protocol	TCP	29.0	27.5
	IP	7.8	9.7
	eth, Mach interf.	11.2	11.9
data touch	copy	10.5	6.3
	checksum	5.1	5.6
Mach	Mach send	7.5	6.0
	packet wait	15.8	9.3
other	g. c.	3.4	5.0
	misc.	4.7	7.3
	counters (est.)	5.2	5.4
	total	100.2	94.0

Table 2: Execution Profile (Percent of Total Time) of the TCP/IP stack.

in the measurements. The major component that is not profiled is the scheduler, since the $15\mu s$ counter update time would significantly affect our current performance of $30\mu s$ to create and switch to a thread.

It is interesting that garbage collection is not a very significant component of the cost. While we have not measured the duration of individual garbage collections, we do know that the numbers shown above reflect time spent doing minor collections. SML/NJ has a generational garbage collector which can either collect objects in the nursery, or trace, collect, and compact all known objects; a minor collection is a collection of the nursery. Since most of the nursery is garbage, minor collections are usually quite fast; measurements by Nettles and O'Toole [14] on a faster system (Decstation 5000/200) show pauses of under a hundred milliseconds on average for the SML/NJ collector. The minor collection pauses are considerably shorter than the pauses imposed by the operating system when running in user mode on top of Mach 3.0. Traces of the running TCP show that these pauses do not substantially affect the performance of TCP.

Runs of over 5 MB often require at least one major garbage collection, which takes substantially longer than the minor collection. We have observed, however, that the overall throughput on the longer runs is the same or faster than on the shorter runs; our interpretation is that the amortization of startup costs over a longer run more than compensates for the additional time and for the slight disruption of doing a major garbage collection.

These profiling numbers, and the more detailed ones we use in optimizing, show that the time spent in non-data touching protocol processing for TCP and for the other protocols is still quite substantial, and very noticeably greater than either the copy or the checksum times. Since our optimization of TCP is far from complete, this result is not unexpected. By comparison, the implementation of TCP in the x-kernel is derived from the Berkeley code, which is highly optimized.

In implementing copy in `SML`, we have been hampered by poor compiler support for tight, unrolled, bytearray-accessing loops, and are substantially slower than comparable `C` code. For checksum on the other hand our implementation uses a better algorithm to overcome these limitations and surpass the speed of the more basic x-kernel implementation. It is therefore not unreasonable to expect that after optimization the speed of the Fox Net TCP will be somewhere between these extremes, and perhaps close to that of the x-kernel TCP. In this context it is reassuring to note that the throughput of communication using only the Fox Net IP is comparable to, and sometimes greater than, that of the x-kernel implementation of IP [3].

6 Related Work

The x-kernel project at the University of Arizona [16] has been mentioned in the preceding sections as the model for the design of our implementation. In the x-kernel architecture each protocol presents essentially the same interface to the outside world. The x-kernel has also developed ideas that we have not (yet) made use of, such as virtual protocols.

Another feature of the x-kernel which we have used in our design is reversing the direction of the receive call, so that received data is delivered to a higher layer by calling the higher layer's receive handler. This has been described as *upcalls* by Clark [6]. With `SML` we can improve on this idea by specializing on the connection information the handler supplied to the open call; the resulting connection-specific handler can encapsulate pre-computed information about the connection.

Christian Tschudin [20] has presented an approach to dynamic protocol composition based on a protocol stack environment. Interestingly, Tschudin uses an extension of Pascal for his work. However, we believe that some of the features of `SML` which are not present in Pascal, such as automatic type inference, polymorphism, automatic storage management, higher order functions, and especially modules, are fundamental to our work. We also believe strongly that by producing a complete implementation of a standard protocol stack we are able to demonstrate the usefulness of our approach more convincingly than by using Tschudin's "toy environment".

The architecture of the MIT/Symbolics Lisp machine [13] is designed to execute Lisp efficiently, and the entire operating system is written in Lisp. Unlike Lisp, `SML` has a type system that insures that type errors are discovered at runtime, and this property is supported by the module system; we believe the module system has been helpful in designing, building, and maintaining our code. Unlike the Lisp machine, the Fox Net runs on stock hardware and, for the time being, on top of a stock operating system.

The Erlang language [17] was developed at Ericsson and has been used in several telephony applications, demonstrating that functional languages can be used for real-time systems, with substantial resulting benefits in development time and overall maintainability.

We based our implementation on RFC 793 and RFC 1122 [8, 15]. We have also relied on the work of others to identify potentially expensive parts of the TCP specification. In their analysis of the costs of TCP, Clark, van Jacobson, Romkey, and Salwen [7] argue that protocol-specific processing is only a minor contributor to the overall cost of a protocol when the protocol is processing data as fast as possible. This result is supported and extended by other studies; specifically, Kay and Pasquale [9] show that on average significantly less than half the cost of TCP processing is in TCP-specific protocol processing, even for short packets. Though (as pointed out to us by a reviewer) the results of this study may not apply to the latest version of Berkeley TCP, the consensus is that for large packets, which are important for high throughput, significant costs are instead data copying and buffering, checksumming, timer execution, and system overheads; all these overheads can be made substantially smaller through proper design. We believe good designs are more easily achieved by using languages that support structured implementations than by using languages that permit violation of abstractions.

Our implementation is built on top of the Mach 3.0 microkernel [18], and uses the Mach Interprocess Communication mechanism to send and receive packets. Maeda and Bershad [10] have implemented a high-performance TCP on top of the Mach 3.0, and their performance is substantially higher than ours. Their implementation is hardly comparable to ours, however, since they used techniques, such as increasing window and kernel buffer sizes and adding special kernel traps (a "fast path" to the device), that we have not (yet) used in our system.

The Standard ML language is defined by Milner, Tofte and Harper [12, 11]. Work on the language is actively continuing. Our project uses a modified version of the Standard ML of New Jersey (SML/NJ) compiler from AT&T Bell Laboratories [2].

In the current implementation of TCP, higher-order functions are the only "functional" feature of SML we have used. We have purposely tried to construct a conventional implementation of TCP/IP that could be compared to other existing implementations. In the future, we may want to provide different implementations and different protocols that may be better suited for use by functional programmers. One example that we may want to imitate or re-implement is CML (Concurrent ML), described by Reppy [19]. CML provides typed channels and lightweight threads integrated into a parallel programming environment.

7 Summary and Future Work

We have found that `SML` is suitable for systems programming. Perhaps the greatest strength of `SML` is its support for and enforcement of the concrete expression of abstractions that other programming languages leave to programmer convention. We have seen how higher-order functions can be used to express upcalls and staging of computation as well as encapsulation of state. Typing, module signatures, and functor headers let us describe the behavior of modules abstractly without having to describe the implementation, and are very useful for detecting inconsistencies when integrating software developed by a team of programmers. Automatic storage management has been invaluable in avoiding time-consuming storage errors, and we have been able to verify that even the SML/NJ stop-and-copy garbage collector does not unduly interfere with performance. Finally, continuations let programmers create their own specialized schedulers, and an implementation of a specialized scheduler which can be faster than even a highly optimized implementation of a more general scheduler.

The extensions to `SML` we found necessary include continuations, primitive types for 1-byte, 2-byte, and 4-byte quantities, and byte arrays; continuations have been especially useful for cleanly expressing fundamental ideas such as coroutines. Other people on our project are actively studying further extensions to the language that may allow us to dispense with byte arrays.

Our implementation of TCP is structured along three orthogonal dimensions: the module structure, the control structure, and the test structure. These structures have permitted one programmer to design and implement a complete, standard, and correct implementation of TCP in under three months. We believe the combination of structured design and advanced languages can contribute substantially to the software development process for networking protocols, and can lead to high-quality and high-performance implementations.

While the speed of our implementation is not yet satisfactory, and in particular is slower than the x-kernel which is itself not a particularly fast implementation, we are still optimizing the code and fully expect the speed to improve in the future. It is possible that once we have optimized the code sufficiently, the garbage collector will disrupt the TCP performance, in which case we will implement and use an incremental garbage collector with bounded pauses.

We also look forward to using our experience implementing TCP/IP to design and implement new protocols, some along the lines of existing protocols, others using ideas from the functional programming world to design interesting and useful network protocols.

8 Acknowledgements

The author would like to thank Robert Harper and Peter Lee for leading the Fox project and for providing feedback on this paper, Kenneth Cline, Nicholas Haines, Brian Milnes, and J. Gregory Morrisett for their indispensable contributions to the Fox project, and Elmootazbellah Elnozahy, Matthias Felleisen, and Eliot Moss for providing valuable input to the project. The author would also like to thank the anonymous reviewers, who have contributed very instructive comments.

References

[1] Andrew W. Appel. A runtime system. *Journal of Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third International Symposium on Programming Languages Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[3] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack – a sys-

tems application of Standard ML. In *1994 ACM Conference on Lisp and Functional Programming*, Orlando, FGV, June 27-29 1994. (to appear).

[4] R. Braden, D. Borman, and C. Partridge. Computing the internet checksum. RFC 1071, September 1988.

[5] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM 90*, September 1990.

[6] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th SOSP*, pages 171–180, Orcas Island, Wash., December 1985.

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

[8] USC Information Sciences Institute. Transmission control protocol. RFC 793, September 1981.

[9] Jonathan Kay and Joseph Pasquale. The importance of non-data touching processing overheads in tcp/ip. In *ACM SIGCOMM '93*, 1993.

[10] Chris Maeda and Brian N. Bershad. Protocol service decomposition for high-performance networking. In *14th ACM Symposium on Operating Systems Principles*, December 5–8 1993.

[11] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, 1990.

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[13] D. A. Moon. Symbolics architecture. *IEEE Computer*, 20(1), January 1987.

[14] Scott M. Nettles and James W. O'Toole. Real-Time Replication Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 217–226. ACM, June 1993.

[15] IETF Network Working Group. Requirements for internet hosts – communication layers. RFC 1122, October 1989.

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

[17] M. Persson, K. Odling, and D. Eriksson. A switching software architecture prototype using real time declarative language. In *International Switching Symposium*, Yokohama, 25 - 30 October 1992.

[18] 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*, Sept 1989.

[19] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[20] Christian Tschudin. Flexible protocol stacks. In *ACM SIGCOMM 91*, 1991.

[21] B. Zorn and P. Hilfinger. A memory allocation profiler for C and Lisp programs. In *Proceedings of the Summer 1988 USENIX Conference*, San Francisco, June 1988.