Small Scheme Stack: A Scheme TCP/IP Stack Targeting Small Embedded Applications

Vincent St-Amour
Université de Montréal
stamourv@iro.umontreal.ca

Lysiane Bouchard
Université de Montréal
boucharl@iro.umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

Interaction with embedded systems is usually achieved by hooking up these devices to a computer network. The TCP/IP stack of protocols has often been used to this end, requiring compact stacks to be implemented as regular ones are too large for embedded systems. Traditionally, compact stacks such as uIP [4] have been implemented in C. Here we report on our experience in implementing S^3 ("Small Scheme Stack"), a compact TCP/IP stack written in Scheme for microcontrollers with a few kilobytes of memory. This paper describes how we were able to minimize the code size and memory requirements by taking advantage of Scheme's power of abstraction and of a virtual machine. We also provide examples of the stack's use to write network applications.

1. Introduction

Increasingly, electronic devices such as cell phones, intelligent toys, security systems, and home appliances, are interfaced with their environment through networking. The Internet, in its wired and wireless forms, is a popular solution because of the extensive existing infrastructure and the wealth of accessible services. It allows these devices to be controlled remotely from most personal computers and to communicate with any similarly networked device on the internet. Moreover, an embedded system with a network interface can reduce costs by eliminating the need for dedicated peripherals, such as a keyboard, display, and disk storage. Equivalent functionality can be achieved by sharing the resources of another computer through the network and an appropriate protocol such as TELNET, HTTP, X11, and NFS. For example, a digital thermometer built from a microcontroller, a temperature sensor and a IEEE 802.11 wireless network chip could use NFS to log the temperature hourly in a file on a distant workstation and use HTTP to allow users to view the temperature log through a web browser and to configure a "temperature alarm" which sends email alarms using SMTP.

A TCP/IP protocol stack is a non-trivial piece of software for which an accordingly powerful microcontroller is needed to store the program code and data, and manage the communication in real-time. Our work aims to demonstrate that Scheme is well suited to build a reasonably featureful TCP/IP stack for inexpensive mid-level microcontrollers with on the order of 16 kB of total memory.

2008 Workshop on Scheme and Functional Programming

We achieve this goal using an optimizing Scheme compiler and a compact Scheme virtual machine (PICOBIT), and by taking advantage of some of Scheme's features to write compact code, including higher-order functions and garbage collection.

After discussing related work in the next section, we give an overview of the features of S^3 ("Small Scheme Stack"). Its application programming interface will then be described, with supporting examples, followed by the stack's implementation. We conclude with some experimental results.

2. Related Work

The TCP/IP stack of protocols has often been used in conjunction with embedded systems, both for their remote configuration and for remote access to their function. For this purpose, a number of TCP/IP stacks targeting embedded systems have been developed. For example, Adam Dunkels's uIP [4] and lwIP [3], both written in C, aim to provide full support for TCP [17], and UDP [18] in the case of lwIP, while keeping a low code size, on the order of 10 kB once compiled, and memory footprint, around 4 kB of RAM with a web server running. Having full protocol support gives much flexibility, both in terms of supported peer types (the stack can communicate with any device, not only workstations with a full-scale TCP/IP stack) and supported application types. The goals of S^3 are similar to uIP's: low code size and memory footprint while implementing the most commonly needed features of each protocol.

PowerNet [15], written in Forth, targets 32-bit embedded systems. Both uIP and S^3 primarily target 8-bit microcontrollers, but can be ported to other architectures. Like both uIP and S^3, PowerNet supports TCP and UDP. In addition, it includes drivers for Ethernet [10] and SLIP [21], as we do, and various ready-to-use applications such as a web server and a TELNET server. Since PowerNet is written in Forth, it is easy to interface to other Forth applications on the same chip. S^3 has a similar goal of being easy to use from Scheme applications. However, PowerNet is a closed-source product, which restricts our ability to study it further and draw comparisons with S^3.

Several stacks have been written in functional languages, such as FoxNet [1] written in Standard ML. The authors made significant use of functional programming and high-level language features, such as Standard ML's powerful module system, continuations and garbage collection. In particular, the authors measured that the use of a garbage collector did not impact the performance of their stack, while providing all the usual benefits of automatic memory management in terms of programmer effort and maintainability. Since FoxNet was designed for workstation-class machines, no particular efforts were made to keep the stack as compact as possible. Accordingly, this stack was implemented in around 50 kLOC of Standard

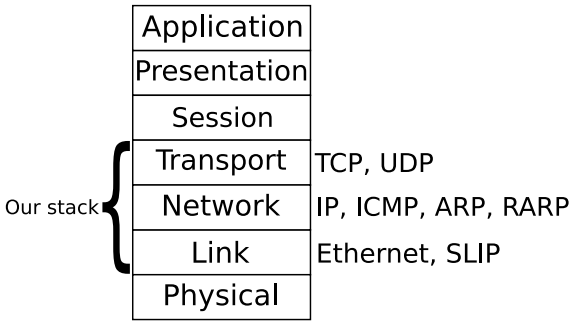

Figure 1. Layers of the OSI model and those implemented by S^3

ML, which means it likely is much too large to be used on small microcontrollers.

The House [9] operating system, written in Haskell, also includes a network stack. Like FoxNet, House was designed for workstations; therefore, most size issues affecting FoxNet are likely to apply to House's stack. It should be noted that this stack does not implement TCP and thus only offers UDP as transport-layer protocol.

To our knowledge, no stack targeting small embedded systems has ever been written in a functional programming language. In this paper, we show the feasibility of such a challenge through the use of Scheme.

3. TCP/IP Stack Overview

S^3 handles 3 layers of the OSI model [23] represented in Figure 1: the link, network and transport layers. On the link layer, Ethernet and Serial Line Internet Protocol (SLIP) encapsulation are supported. On the transport layer, S^3 offers support for both TCP and UDP. Most of the TCP standard has been implemented in S^3, enabling support for various types of applications, unlike many embedded TCP/IP stacks, which were written with only a particular application in mind, often a web server.

S^3 should be considered to be a minimal implementation. Some of the infrequently used features have been omitted. S^3 is easily extensible; support for the missing features can be added if need be.

The following sections explain how S^3 handles each layer of the OSI model.

3.1 Physical Layer

The lowest layer of the OSI model, the physical layer, handles the transmission of raw bits over a physical connection. TCP/IP stacks do not usually implement this layer, instead relying on an underlying device driver for the first two layers. S^3 is no exception and requires the use of a device driver. For our tests, we have used libpcap [11] as an abstract interface over device drivers, in our case for Ethernet.

3.2 Link Layer

The second layer of the OSI model, the link layer, is responsible for the transmission of data frames between two nodes of a network segment. Since S^3 targets small embedded systems, it will likely be used without the support of a sophisticated network device driver. Therefore, unlike most TCP/IP stacks, we offer support for the OSI link layer in addition to the network and transport layers that are traditionally implemented in TCP/IP stacks.

S^3 supports the use of different link-layer protocols. Currently, support for Ethernet and SLIP have been implemented. PPP support could be added easily. The support of multiple link-layer protocols makes it possible to deploy microcontrollers running S^3 in various environments. For example, using the Ethernet back-end, such a microcontroller could be a first-class citizen in any typical home network, opening many possibilities for home automation or ubiquitous programming, among others. For integration with other more traditional embedded systems, or interfacing to a workstation, the SLIP back-end could be used.

3.3 Network Layer

The network layer is responsible for the delivery of packets from their source to their destination. In the case these two nodes are not on the same network segment, the network layer is responsible for the routing of packets. On the network layer, S^3 supports the Internet Protocol [20] (IP), Internet Control Message Protocol [19] (ICMP), Address Resolution Protocol [16] (ARP) and Reverse Address Resolution Protocol [8] (RARP) protocols, making it suitable for a wide array of tasks.

To increase flexibility, S^3 supports the use of multiple IP addresses per interface. We can therefore easily simulate multiple hosts on the same stack. The dispatch between the different hosts is done at the application level, using the filter functions described in Section 4.3.

Some features of the IP standard were omitted in the current version. For example, no support for IP fragmentation, options or type of service is currently offered. While the omission of these features might make communicating with some hosts impossible, their use is infrequent enough that it is unlikely it would prove to be a problem.

In the case of ICMP, S^3 can report any error that occurs on the transport layer (protocol unreachable, port unreachable, and so on) as well as reply to echo requests and address mask requests. While the ICMP standard specifies other operations, S^3 implements enough operations to communicate properly on the network.

While ARP and RARP are not used to carry data as IP does, their role in address resolution makes them essential for routing on an Ethernet-based network. Both standards have been implemented fully. These protocols can only be used with the Ethernet back-end, since they depend on a notion of address that does not exist in SLIP. In order for RARP to work properly, a list of pairs of MAC addresses and their corresponding IP addresses has to be given to the stack. This list is specified as a Scheme association list in a configuration file. This file is detailed in Section 4.1.

3.4 Transport Layer

The fourth layer of the OSI model, the transport layer, handles the delivery of data to the appropriate applications. On the transport layer, S^3 supports the two protocols in widest use, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

As in the case of IP on the network layer, the TCP standard was partially implemented for the same reasons. A notable omission is the urgent data feature which, while useful when implementing some kinds of applications, is not absolutely necessary for most applications built over TCP.

As for UDP, all features specified in the standard have been implemented. The extreme simplicity of UDP made its implementation quite straightforward.

3.5 Higher Layers

The implementation of the other layers of the OSI model, the session, presentation and application layers, is left to the applications built over S^3. The application programming interface we provide (see next section) was created to assist in their implementation.

4. Application Programming Interface

The design of an API has a big influence on the design of the applications using it; therefore, much of the design effort went into the API of S^3. In this section, we describe the resulting interface and illustrate its use by implementing simple applications.

4.1 Configuration

Many aspects of S^3 can be configured using a set of Scheme definitions in a file. An advantage of this approach is that, since all the configuration is contained in S-expressions, it can be easily program-generated if need be. Since this configuration file is actually just another module of the program, it is then compiled and linked with the rest of the stack.

Using this configuration file it is possible to change the list of IP addresses of the stack, its MAC address, the maximum accepted packet size, the list of addresses used for RARP (as mentioned previously), the default IP datagram time to live (TTL), the size of the input/output buffers used for TCP, among others.

4.2 Polling

The well-known Berkeley socket [13] interface offered by most TCP/IP stacks provides blocking read and write operations. Such operations are made possible by the presence of underlying synchronization mechanisms, such as mutexes, condition variables and threads. For example, on a multithreaded system, if a process blocks on a read operation, the other processes can continue their execution independently. We did not want to impose on the target platform any special requirements regarding threading. Requiring a multithreaded operating system or implementing one with continuations might exclude some target platforms (for lack of an appropriate OS, lack of ROM or RAM, etc). Instead, the API uses polling and it is up to the application to orchestrate the execution of concurrent activities.

Consequently, we chose to provide non-blocking I/O operations and let applications poll the stack until the desired operation succeeds. Specifically, whenever an application does an I/O operation, it either returns with a success value if the operation was successful or with a special code if the operation could not be completed. For example, a special code is returned when an application tries to read data from a connection before any has arrived. When an application receives such a code, it can then either try to do some other operations or let another application, or the stack, execute. Later on, this application can try the desired operation again until it either succeeds or fails.

Since we're considering a single-thread system, application-switching has to be done explicitly. Therefore, if control is never given to the stack, no network operation can ever succeed. By calling the `stack-task` function, with no parameters, S^3 will handle the next incoming packet and send any pending output packet.

In an environment where synchronization mechanisms are available, it would be possible to simulate blocking operations by using the non-blocking operations S^3 offers in conjunction with the available synchronization mechanisms.

4.3 TCP

For the TCP portion of S^3 we chose an interface similar to the well-known Berkeley sockets, but adapted for Scheme and single-threaded environments.

To bind a server to a port, the `tcp-bind` (see Figure 2) function has to be called from the application, which must specify the port number to be bound (1 to 65535) and the maximum number of simultaneous connections to be accepted. In addition, the application must give the stack a filter function, which must return a true value when the application accepts the packet, and `#f` if it refuses.

```
(tcp-bind portnum max-conns tcp-filter tcp-recv)
(tcp-filter dest-ip source-ip source-portnum)
(tcp-recv connection)
```

Figure 2. Signatures of `tcp-bind` and its filter and reception functions

```
(tcp-read connection [length])
(tcp-write connection data)
```

Figure 3. Signatures of the TCP input/output functions

```
(tcp-close connection [abort?])
```

Figure 4. Signature of the TCP close function

This acceptance or refusal is determined using the IP address the connection is destined to, the source IP, along with the source port number and is decided on a per-packet basis. If the stack possesses multiple IP addresses, the dispatch is done with these filter functions. Finally, the application also has to give the stack a reception function which will be called each time a new connection destined to the application is established.

When a TCP connection is being established with the stack, if the original request passes the filter function, the reception function is called, with a newly created connection object as parameter. It is at that time that the connection initialization inside the application is typically done. Since reading data from and writing data to the connection is done using this object, the application typically stores the connection object for future use.

Once the connection is established, the application can read data from or write data to the connection at any time. As explained before, these operations are non-blocking. All these operations take the connection object given to the reception function as parameter (see Figure 3).

The `tcp-read` function takes an optional *length* parameter and returns a byte vector. If the *length* parameter is given, the stack will attempt to read *length* bytes of input. If enough bytes are available, they will be returned. If not, all available bytes will be returned, and the returned vector will be of the appropriate length instead of being *length* bytes long. If the *length* parameter is omitted, all immediately available input will be read.

The `tcp-write` function returns the number of bytes that were really written to the connection. If this number is less than the length of the vector given to *tcp-write*, the application knows that its output has only been partially sent, and can react accordingly.

To terminate a connection, the `tcp-close` function is provided (see Figure 4). The connection to close is the first parameter. The *abort?* parameter must be either set to `#f` or omitted if the connection has been successful and ends normally. A true value indicated that an error state has been reached and the connection must be aborted. In both cases, the peer will be notified of the cause of the connection's termination. Once this function is called (or if the stack's peer aborts or closes the connection), no more data can be written to the connection, but any remaining data can be read. Once the application is done with the connection, it should release the connection object and leave it to be garbage-collected, the stack having already dropped all references to the connection object.

Despite its simplicity, this interface remains expressive enough to be used to implement various applications. We wrote a minimalist web server to test our API (see Figure 5).

```
(define connections '()) ;; (conn-object . data)

(define (visit-all cs)
  (let ((c (car cs))))
    (if (visit c)
        (cons c (visit-all (cdr cs)))
        (begin
          (tcp-close c #t) ;; abort
          (visit-all (cdr cs)))))

;; (visit conn) visits one connection and
;; returns false when the connection is over.
(define (visit conn)
  (let ((new-data (tcp-read (car conn))))
    (cond ((not new-data)
           ;; we received nothing, try again later
           #t)
          ((equal? new-data 'end-of-input)
           ;; no more input, give up
           #f)
          (else
           ;; save the new input and try to answer
           (set-cdr! conn (update-data (cdr conn)
                                       new-data))
           (answer conn)))))

;; (update-data orig new) appends the new data to
;; what has already been received
(define (update-data orig new) ...)

(define (answer conn)
  (let* ((data (cdr conn))
         (len (string-length data))
         (spc (find-first #\space data 0 len))
         (spc2 (find-first #\space data spc len)))
    (cond ((not spc2)
           ;; we didn't receive the target yet
           #t)
          (else
           ;; we have received the target, answer
           (serve (car conn) ...)
           (stack-task)
           (tcp-close (car conn))
           #f)))) ;; done

(define (serve conn target) (tcp-write conn ...))

(define (find-first target data start end) ...)

(define (tcp-filter dst-ip src-ip src-port)
  (equal? dst-ip my-ip))

(define (tcp-recv conn)
  (cons (cons conn "") connections))

(define (main-loop)
  (stack-task)
  (set! connections (visit-all connections))
  (main-loop))

(tcp-bind 80 20 tcp-filter tcp-recv)

(main-loop)
```

Figure 5. The outline of a minimalist web server built using S^3

```
(udp-bind portnum udp-filter udp-recv)
(udp-filter dest-ip source-ip source-portnum)
(udp-recv source-ip source-portnum data)
```

Figure 6. Signatures of `udp-bind` and its filter and reception functions

```
(udp-write dest-ip source-port dest-port data)
```

Figure 7. Signature of the UDP output function

```
(define (echo source-ip source-port data)
  (udp-write source-ip 7 source-port data)
  (stack-task))

(define (main-loop)
  (stack-task)
  (main-loop))

(udp-bind 7
          (lambda (dest-ip source-ip source-port)
            (equal? dest-ip my-ip))
          echo)

(main-loop)
```

Figure 8. A simple UDP echo server

4.4 UDP

UDP is a much simpler protocol than TCP. UDP support for S^3 was implemented in only 50 lines of Scheme code.

Binding a UDP port is similar to a TCP port. First, the `udp-bind` function is called with the port number to be bound, a filter function and a reception function (whose signature is given in Figure 6). Due to the connectionless nature of UDP, the reception function is called every time a UDP packet destined to the appropriate port is received and passes the filter function.

The signature of the receiving function has been changed accordingly: instead of taking a connection structure as parameter, it takes the IP address of the source of the datagram along with the port number used by the source, and a byte vector containing the data contained within the datagram. Since the IP address of the source is given to the application, it can do some internal dispatch between peers as need be. Since the logical data unit for UDP is the datagram, it makes more sense to pass all the data contained in the datagram at once rather than keeping it in a buffer to give the illusion of a stream, as is the case with TCP.

Since all data is handed to the application as soon as it is received by the stack, it is inappropriate to have functions for reading data in smaller chunks. However, an output function is still necessary. Since UDP datagrams can be sent in any circumstance, not only within the confines of a connection as is the case with TCP, more information is needed to do output. In our case, to produce any output the `udp-write` function needs the IP address of the destination as well as both the source and destination ports, along with the data to be sent.

Again due to the connectionless nature of UDP, no connection control functions are necessary.

To illustrate the use of our UDP API, an implementation of the standard UDP echo server listening on port 7 is given in Figure 8. The server's logic is simple. The `echo` function sends an UDP datagram to the client containing the data that was received. The setup of the server is also simple. A call to the `udp-bind` function

links our echo server to UDP port 7 with a filter function that accepts any packet. Any UDP datagram received on port 7 would now be handed to the echo server.

5. Implementation

Our goals of low code size and low memory footprint had an important impact on the design of S^3. In this section, we will elaborate on design decisions that were made and language features we leveraged in order to meet our goals.

5.1 Packet Limit

Only one packet is present in the stack at a time. Unlike many TCP/IP stacks, S^3 does not have a buffer to store packets received while processing another. Instead, any packet arriving while the stack is already processing a request is ignored. The absence of such a buffer can save a lot of memory, if we consider 200 bytes to be the average packet size (which amounts to a TCP packet carried over Ethernet containing 146 bytes of data). We also avoid the costs related to the upkeep of such a data structure, both in code complexity (therefore, in code size, which we try to keep as low as possible) and in processing time, which is especially important considering the humble speed of the processors present in most embedded systems.

Due to the nature of the protocols we support, ignoring packets is not a threat to communication integrity. In the case of TCP, packets for which a reception acknowledgement was not received (such as those that were ignored by the stack) will be resent, with a fixed number of retries to avoid congestion or neverending attempts to communicate with an unreachable host. In the case of UDP, the protocol does not guarantee the reception of datagrams, it is up to the applications to ensure the integrity of the communication. Therefore, ignoring packets might induce delays in the communication, but it should not compromise its integrity. As for the underlying protocols (IP and ICMP), no guarantee is made by the protocols either.

This approach should not result in a performance much different from that of a buffered stack. Consider the following cases: if the stack processes packets at a speed faster than the speed at which they arrive, the stack never has to ignore any packets, giving us a better performance than a buffered stack, since we save the upkeep costs of the queue, which would never be needed in this case.

On the other hand, if the stack processes packets at a speed slower than the speed at which they arrive, the stack has to ignore some of the incoming packets, resulting in delays. Now, let us consider a buffered stack in the same situation. Since the packets arrive faster than they are processed, the buffer would gradually fill up, and eventually be full. Once the buffer is full, the stack is forced to ignore any incoming packet, just like it would if it did not have buffering. Of course, the buffering stack would last a little bit longer before ignoring packets, maybe long enough to reach a lull in the reception and catch up on the accumulated packets, whereas S^3 would have to cope with the reception of the retransmissions of the original packets in addition to the new ones. Therefore, if traffic follows a pattern of bursts of heavy traffic interleaved with quiet periods, a buffered stack would probably fare much better than a non-buffered one, whereas if traffic is of about constant intensity, both stacks would end up ignoring packets, with the non-buffered one doing slightly better since it would be processing packets faster because of the savings due to the absence of buffer upkeep time.

However, keeping only one packet in the system at once also has some drawbacks. First, since ignored packets are likely to get resent (especially in the case of TCP), the risk of network congestion might be higher than with a buffered stack. Second, since we have to wait until an ignored packet is resent to be able to respond to it (whereas a buffered stack would have kept it in its buffer), delays can be introduced in the communication, as we noted above. In extreme cases, this might even cause the connection to be dropped by our peer. If these drawbacks end up being major, and memory is not too much of an issue, adding a packet queue to S^3 would require only minor changes, thanks to its modular design.

It should also be noted that these drawbacks are unlikely to matter since most networking hardware already does a certain amount of buffering.

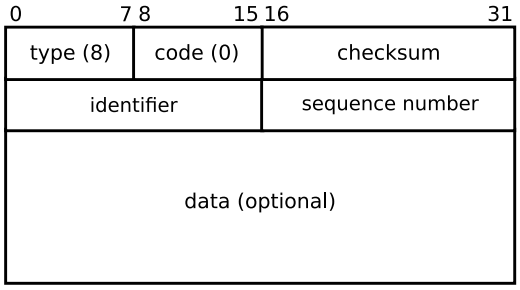

Figure 9. An ICMP echo request

```
;; change the type to icmp echo reply
(pkt-set! icmp-type 0)

;; clear the checksum
(integer->pkt-2 0 icmp-checksum)

;; calculate the new checksum
(integer->pkt-2
  (reverse-checksum (compute-icmp-checksum))
  icmp-checksum)
```

Figure 10. The code used to reply to an ICMP echo request

5.2 Reply Generation

The reply packets sent by S^3 are created by mutating the original packet received by the stack. Since a large part of the data is common between a request and its response, albeit not necessarily in the same location (for example the source and target IP addresses in the IP headers, which are obviously swapped between request and response), we save memory by not storing it twice.

Also, since packet structure is virtually unchanged between the request and the response, only minimal changes to the headers are necessary most of the time while creating a response, the only major modifications coming from the changes in data. Some of the information present in the headers being exactly the same between the request and the response, we also save some time by not having to copy it.

In the case of an ICMP echo request, most often used by end users through the `ping` utility (see Figure 9), S^3 would only have to change the `type` byte (from 8 to 0) and recalculate the checksum, the `code` byte being 0 on both the request and the response, and the identifier, sequence number and data being simply echoed by the server. Therefore, only three bytes are changed when creating an ICMP echo reply. Considering that the default data size for an ICMP echo request on GNU/Linux is 56 bytes, giving a total of 64 bytes for the whole ICMP message, less than 5% of the packet has to be modified, which is quite an improvement over having to build the whole response from scratch. The code used for responding to an ICMP echo request is presented in Figure 10.

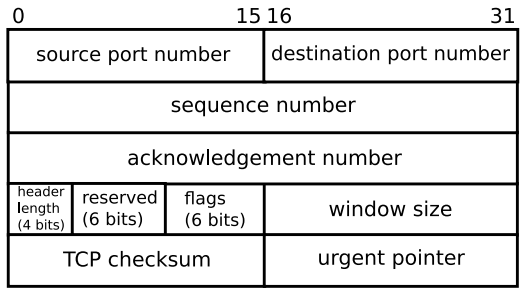

Figure 11. A TCP header

Of course, the stack could also store packet templates to be filled with the remaining data, as does Miniweb [5], but storing these packets requires memory, which is in short supply in our case.

The savings brought by this approach are not always as impressive. In the case of an ordinary TCP header, as represented in Figure 11, if we consider the worst case, only the 2 bytes containing the window size and the 2 containing the urgent pointer (which never change, since we do not support urgent data) would stay the same between the request and the response. We would save 4 bytes over a total of 20 bytes for the whole header, if we suppose no options, which is not as large a savings.

This approach for the generation of responses assumes that each field is in the same location for both the request and the reply. Since S^3 does not support options in the IP headers, the header of any protocol we support is guaranteed to always be the same size. Therefore, since all the protocols we support can only be used with a single underlying protocol at a time, information within the headers is guaranteed to always be in the same location.

This in-place creation of response packets is made more efficient by the use of a vector to represent the packet currently in the stack. Since the order of the modifications is not necessarily sequential, having efficient random access in the packets can be especially useful. It is also worth noting that such in-place edition is possible because Scheme supports destructive updates. This approach would not have been possible, or at least would have been more complex, using Haskell or another purely functional language.

Since Scheme vectors are of fixed size and a packet can trigger a response longer than itself, we need to manipulate our packet in a preallocated vector with a length considered sufficient. The default value for this length is 590 bytes (the smallest maximum IP datagram size we must support according to the standard is 576 bytes and the length of an Ethernet header is 14 bytes). This can be changed in the configuration file, as described in section 4.1. Any packets larger than the size of the buffer are ignored by S^3.

Keeping a deliberately large vector in the stack might be considered wasteful, especially since memory is usually in short supply on our target platform. However, let's consider the alternative: storing each incoming packet in a vector of the right size. Since the response to each packet might be longer than the original packet, we have to store outgoing packets separately and lose all the benefits of creating them in-place. Moreover, the sum of the lengths of the incoming and outgoing packet currently in the system may very well be larger than the size of the original too large vector, which would end up wasting even more space than our approach. If we add the costs of constantly allocating new vectors and the costs in space of the old vectors before they get garbage-collected, this alternative solution does not seem appropriate for limited memory systems.

5.3 Virtual Machine

To minimize the code size of S^3, we use a bytecode interpreter-based approach to run the TCP/IP stack. We chose the PICOBIT Scheme system which was designed for use on mid-level PIC [14] microcontrollers. Because of the limitations of the target environment, especially the small amount of memory, PICOBIT itself has several unique limitations that are not present in other Scheme systems.

The PICOBIT runtime has to distinguish between values that are stored in random-access memory (RAM) and those that are stored in read-only memory (ROM). Due to constraints of both time and space, the PICOBIT compiler stores in ROM all values that are known at compile-time. Since anything stored in ROM is, by definition, read-only, special care has to be taken to avoid storing in ROM a vector that would need to be mutated.

Another limitation we had to keep in mind while building S^3 is that PICOBIT offers support for integers up to 24 bits wide. Therefore, any value wider than 24 bits would have to be stored and passed around as a vector of bytes. In order to support systems that only have support for even smaller integers, such as PICBIT [7] which supports 16-bit integers, S^3 only uses integers up to 16 bits wide.

Despite the previously mentioned limitations, the use of a virtual machine remains an interesting approach. Bytecode being at a higher level of abstraction than machine instructions, we were able to gain a significant reduction of the code size on the target microcontroller. On the other hand, the virtual machine necessary for such an approach takes some space too, reducing our gains somewhat. This is discussed in more detail in Section 6.

5.4 First-Class Procedures

First-class procedures are a powerful means of abstraction that provide many ways of combining functions. This had an interesting impact on the reduction of the code size of S^3.

During its lifetime a TCP connection will pass through different states (listening, established, waiting for an acknowledgement, and so on). In each of these states the tasks a stack has to accomplish for a given connection are different. To keep track of which tasks need to be done for each connection we chose to store a TCP state function along with each connection. When executed, this function accomplishes the tasks related to the current state of the connection and creates a new state function that reflects the new state of the connection. This new function then replaces the original one in the connection objects, and will be called the next time this connection is visited. This is in essence a continuation-based coroutine mechanism.

Since the number of states of a TCP connection is prohibitively large, we chose not to implement all of them statically. Instead, we create state functions dynamically as needed. Since some tasks have to be accomplished in many different states, for example enqueuing newly received data to the current stream, state functions are mostly created by combining existing functions representing different tasks. The code reuse brought by this combination of common tasks helped us achieve a significant reduction in code size. Similar results could have been achieved in C using function pointers, but such an approach would have been more error-prone and the resulting code would have been both longer and more complex, whereas in Scheme, thanks to its support for closures, the code is actually shorter and more straightforward than it would have been if we had chosen to define each state function statically.

Another interesting application of first-class procedures in S^3 is the use of filter and reception functions, as detailed in Section 4. The use of these functions allows application authors to handle packets destined to their applications as they wish, something much more complex to do with a traditional socket interface.

5.5 Garbage Collection

In addition to all the usual benefits of automatic memory management on programmer productivity and code simplicity, we also observed that the presence of a garbage collector in the PICOBIT runtime brought us some specific benefits.

First of all, consider the case where a piece of data has to be accessed by two different parts of a program. For example, the objects used to represent TCP connections have to be accessed both by S^3 and by the application to which the connection is destined. If we managed memory manually, the data would have to be owned by one of the parts, which would free it at an appropriate time. Of course, if the data is freed by its owner and another part of the program still needs to access it, we get unpredictable behavior. With automatic memory management, it is not necessary to assign the data to a particular owner, we therefore avoid dangling pointer bugs. Similar results could be achieved in C with the use of reference-counting, but this reference counting would have to be implemented in the TCP/IP stack, whereas in our case, the Scheme runtime relieves the programmer from that burden.

As for memory usage, the use of garbage collection prevents some kinds of memory leaks from happening, leaks which, given the limited amount of memory of our target platform, could be disastrous. Since the PICOBIT runtime uses a mark-and-sweep garbage collector [12], the overall memory usage of S^3 is not much higher than it would be with manual memory management, only one or two tag bits by object being necessary, depending on its type. It should be noted that a stop-and-copy garbage collector was rejected, as using twice as much memory as is really necessary (once to store the actual objects, and then as much to copy them) is unacceptable on low-memory systems such as microcontrollers.

As with the costs of the virtual machine, the garbage collection costs are amortized over all Scheme programs. Once again, since a TCP/IP stack is seldom used without other applications, this can have an important impact. As for the costs in terms of code size, the garbage collector represents about 11% of the code size of PICOBIT's virtual machine, which is acceptably low.

5.6 Portability

Another major advantage of using Scheme is its machine-independent semantics. While S^3 was built with 8-bit PIC microcontrollers in mind, it would run just as well on a 16-bit or 32-bit system, as long as the underlying Scheme system supports R^4RS [2] as well as a few extensions: byte vectors (as defined in SRFI 4 [6]), bitwise OR and XOR and access to a timing mechanism. For example, the exact same stack that compiles to 8-bit PIC microcontrollers was successfully used as a user-space TCP/IP stack on a workstation. More details are available in Section 6.

5.7 Integration

The fact that S^3 is implemented in Scheme also simplifies integration with applications written in Scheme. No foreign function interface is needed, and no interface to an underlying stack has to be added to the Scheme system. In addition, if the underlying TCP/IP stack's API was based on blocking operations, then these would block the whole Scheme system rather than only the applications requesting them.

6. Experimental Results

One of our goals is to demonstrate that with Scheme a relatively complete TCP/IP stack can be implemented in a small amount of code. The whole Scheme code for the S^3 stack, with a device driver and a simple web server included, is 1497 lines of Scheme code. For comparison purposes, the uIP stack is 7725 lines of C code, without any device drivers or sample applications. In addition to being shorter than uIP overall, S^3 implements equivalent features in much fewer lines of code. For example, we mentioned that our implementation of UDP was written in 50 lines of Scheme code and implemented the whole protocol. uIP's UDP implementation implements the whole protocol as well, but does it in around 175 lines of C code. In this particular case, Scheme wins over C by a factor of more than 3 in number of lines of code needed.

CPU	VM size
i386	17.0 kB
MSP430	10.4 kB
PIC18	10.7 kB
PPC604	17.7 kB

Figure 12. Machine code size, for various processors, of the PICOBIT virtual machine with 13-bit object encoding

Using the PICOBIT compiler, we were able to compile the whole stack down to 5 kB of bytecode. The amount of machine code space occupied by the PICOBIT virtual machine, which is written in C, depends on many factors including the target processor and the size of heap supported. For a small heap storing at most 128 Scheme objects in RAM, an 8-bit object encoding can be used and the virtual machine compiles to compact code on the 8-bit PIC microcontroller, a mere 5.5 kB. Such a heap is too small for executing the S^3 stack so a 13-bit object encoding was used. The predominance of operations on larger than 8-bit values causes a significant increase in the virtual machine's code size. Figure 12 gives the virtual machine's code size for a few processors popular in embedded applications. We can see that the code size is between 10 kB and 18 kB.

When the size of the virtual machine is accounted for, the total code space required by S^3 on the PIC is 15.7 kB. This appears to be more code space than the 10 kB required by uIP. The S^3 approach will be more compact when the cost of the virtual machine is amortized on the rest of the application. To make a simple analysis, let's assume that, like the TCP/IP stack, application code written in C will compile to A bytes of machine code and the bytecode generated by PICOBIT will generate $A/2$ bytes of bytecode for equivalent Scheme code. Then the break-even point is $A =$11.4 kB. Larger programs (with larger values of A) will have a smaller code size with the Scheme approach.

It is also worth noting that the more features we add to the stack, the more we can save on code size overall, since the cost of the virtual machine is independent of the size of the program it runs (within reasonable limits).

To demonstrate the machine-independent nature of S^3, in addition to compiling to 8-bit PIC microcontrollers, we tested the stack as a user-space process on a workstation, running the Fedora Core 7 flavor of GNU/Linux. No modifications were necessary; the exact same Scheme code was used both on PIC, using the PICOBIT Scheme system, and on GNU/Linux, using the Gambit Scheme system. A simple compatibility layer could be necessary to use S^3 with other Scheme implementations, as long as they offer the necessary features mentionned in the previous section (e.g. R4RS, byte vectors, bitwise OR and XOR and timing) These features exist in most Scheme systems.

Since having direct access to drivers from a user-space program is not feasible, we simulated a device driver using libpcap [11] This shows the independence of S^3 from the particular network device used.

7. Future Work

S^3 was designed to be able to interact with a large variety of peers, as long as they use the same standard protocols. However, in the case where we have several embedded systems, all running S^3 along with Scheme applications, having these systems communicate over TCP or UDP might not be necessary. Since these systems would all run Scheme, it might be more efficient to avoid the overhead of these protocols by transmitting Scheme data directly over the link layer. A simple protocol could be devised to this end, but in any case, it would likely be much simpler than the whole TCP/IP stack of protocols, albeit more limited.

While S^3 can already be used in various application domains, the omission of certain protocol features would complicate or prevent its use for some applications. For example, TCP urgent data and IP fragmentation, omitted as of now, could be added in future versions. These features could probably be implemented with minor changes to the structure of S^3, so, while they might have an impact on performance, and likely on code size, most advantages exposed in this paper would remain.

In addition to the currently supported link layer protocols, Ethernet and SLIP, the Point-to-Point Protocol (PPP) could be added to S^3 to expand its possibilities.

It would also be interesting to be able to easily include or exclude protocols from S^3. This would enable S^3 to run on devices which are too limited either in data memory or in code space to run a complete TCP/IP stack. Right now, this can be done by removing or commenting out specific modules of the stack, but no simple way of acheiving this is currently available. This could eventually be done though our configuration file, by using compilation options or by some other means.

We intend to explore various ways of reducing the code size of the PICOBIT virtual machine. One straightforward but time consuming approach would be to rewrite critical parts of the virtual machine in assembler. A more interesting avenue is to design for the PIC a compiler for a subset of C which handles well the rather idiosyncratic programming style used in the PICOBIT virtual machine.

Finally, we are considering the use of S^3 and the JSS Scheme to Javascript compiler to build a browser-based operating system similar to the Lively Kernel [22], but written in Scheme.

8. Conclusion

Important reductions can be achieved both in terms of code size and data memory usage by the use of Scheme's power of abstraction and of the PICOBIT Scheme system for mid-level PIC microcontrollers. Through the use of these reductions, we managed to compile S^3, a simple TCP/IP stack, down to 5 kB of bytecode, which would make it suitable for use on embedded systems with only a few kilobytes of memory and other such limited environments.

It is worth noting that the uIP TCP/IP stack, that also targets embedded systems but is written in C, compiles down to a 10 kB binary. This shows that a TCP/IP stack written in Scheme is suitable for use on small embedded systems and also that Scheme can compete with C in terms of both code size and memory usage, while still providing high-level features such as higher-order functions and garbage collection that simplifies the writing non-trivial applications.

References

[1] E. Biagioni. A structured TCP in Standard ML. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*, pages 36–45, London, England, 1994.

[2] W. Clinger and J. Rees. Revised^4 report on the algorithmic language Scheme. Technical report, 1991.

[3] A. Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001:20, SICS – Swedish Institute of Computer Science, Feb. 2001. Master's thesis.

[4] A. Dunkels. Full TCP/IP for 8-bit architectures. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM.

[5] A. Dunkels. Miniweb. http://www.sics.se/~adam/miniweb/, 2005.

[6] M. Feeley. SRFI 4: Homogeneous numeric vector datatypes. http://srfi.schemers.org/srfi-4/srfi-4.html, 1999.

[7] M. Feeley and D. Dubé. PICBIT: A Scheme system for the PIC microcontroller. In *Scheme Workshop 2003*, November 2003.

[8] R. Finlayson, T. Mann, J. Mogul, and M. Theimer. A Reverse Address Resolution Protocol. RFC 903 (Standard), June 1984.

[9] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, Tallinn, Estonia, 2005.

[10] IEEE. *802.3i-1990 IEEE Supplement to Carrier Sense Multiple Access with Collision Detection CSMA/CD Access Method and Physical Layer Specifications: System Considerations for Multisegment 10 Mb/s Baseband Networks (Section 13) and Twisted-Pair Medium Attachment Unit (MAU) and Baseband Medium, Type 10BASE-T (Section 14)*. 1990. IEEE product number SH13763.

[11] V. Jacobson, C. Leres, and S. McCanne. libpcap. http://www.tcpdump.com.

[12] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996.

[13] S. Leffler. Networking implementation notes 4.3BSD edition, 1986.

[14] Microchip Technology Inc. PIC microcontrollers. http://www.microchip.com.

[15] MicroProcessing Engineering, Limited. PowerNet TCP/IP stack. http://www.mpeforth.com/powernet.htm.

[16] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48-bit Ethernet Addresses for Transmission on Ethernet Hardware. RFC 826 (Standard), Nov. 1982.

[17] J. Postel. DoD standard Transmission Control Protocol. RFC 761, Jan. 1980.

[18] J. Postel. User Datagram Protocol. RFC 768 (Standard), Aug. 1980.

[19] J. Postel. Internet Control Message Protocol. RFC 792 (Standard), Sept. 1981.

[20] J. Postel. Internet Protocol. RFC 791 (Standard), Sept. 1981.

[21] J. Romkey. Nonstandard for transmission of IP datagrams over serial lines: SLIP. RFC 1055 (Standard), June 1988.

[22] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform: The lively kernel experience, Jan. 2008.

[23] H. Zimmermann. OSI reference model—the ISO model of architecture for open systems interconnection. 1988.