S^3: the Small Scheme Stack
A Scheme TCP/IP Stack Targeting Small Embedded Applications

Vincent St-Amour
Université de Montréal
Joint work with Lysiane Bouchard and Marc Feeley

Scheme and Functional Programming Workshop
September 20, 2008

Outline

- Motivation for a Scheme network stack
- Protocols supported by S^3
- Application program interface
- Implementation
- Related work
- Experimental results

Small embedded systems

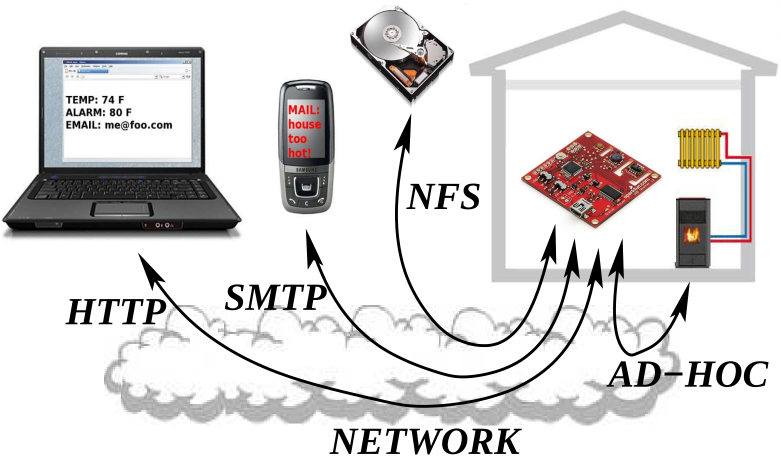
- High volume
- Low cost
- Low memory
- Low computational power
- Need to interact with
  - user (configuration, control)
  - storage device (to keep logs)
  - other embedded systems (automation, distribution)

Why use TCP/IP in embedded systems

Networking infrastructure

- is ubiquitous ($\Uparrow$ accessibility)
- gives access to many services ($\Uparrow$ features)
- eliminates need for I/O periperals ($\Downarrow$ cost)

Sample application: house temperature monitor



Goals

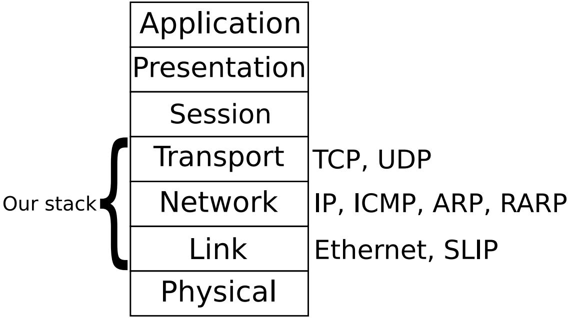
- Show that a network stack can be implemented in Scheme
- Why use Scheme?
    - Why not?
    - Scheme's high-level features $\longrightarrow$ compact code
- Portability to different Scheme implementations

Outline

- Motivation for a Scheme network stack
- Protocols supported by S^3
- Application program interface
- Implementation
- Related work
- Experimental results

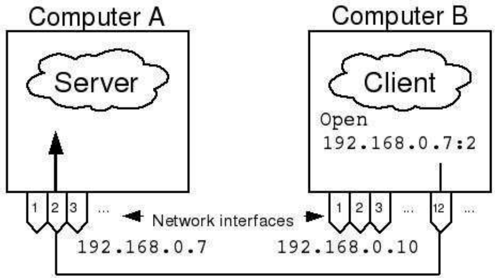
Protocols supported by S^3

OSI Model Layers



TCP

- Most complex protocol implemented in S^3
- Connection-based
  - Server listens for connections on a port number
  - Client connects to that port



  - Stream paradigm
  - Packets are acknowledged by receiver
  - Sender retransmits after timeout

Highlights of our approach

- We target very small systems (2 kB data, 32 kB program, $< \$5$)
- Discard seldom-used features of the protocols
- Minimal buffering
- Polling-based API
- Scheme-specific
  - PICOBIT Scheme virtual machine
  - Higher-order functions

Outline

- Motivation for a Scheme network stack
- Protocols supported by S^3
- Application program interface
- Implementation
- Related work
- Experimental results

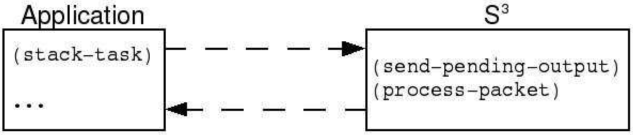
S^3 Configuration

- Packet limits and MAC address
- Contained in S-expressions (could be program-generated)
- Compiled and linked with the stack
- Example :

```
(define pkt-allocated-length 590)
(define my-MAC '#u8(#x00 #x20 #xfc #x20 #x0d #x64))
(define my-IP1 '#u8(10 223 151 101))
(define my-IP2 '#u8(10 223 151 99))
```

Polling

- Avoid synchronization mechanisms (mutexes, condition variables, ...) so that S^3 can be integrated easily to other Scheme systems
- Single-thread system
- Cooperative multitasking
- Non-blocking operations + polling
- Explicit task switching with the call `(stack-task)`



TCP

- (tcp-bind *portnum* *max-conns* *tcp-filter* *tcp-recv*)
- (*tcp-filter* *dest-ip* *source-ip* *source-portnum*)
- (*tcp-recv* *connection*)
- (tcp-read *connection* [*length*])
- (tcp-write *connection* *data*)
- (tcp-close *connection* [*abort?*])

TCP counter server

```
(define counter 0) ;; current count
;; connections that need to be serviced
(define connections '())

(define (main-loop)
  (stack-task)
  (for-each (lambda (c)
              (tcp-write c (u8vector counter))
              (set! counter (+ counter 1))
              (tcp-close c))
            connections)
  (set! connections '())
  (main-loop))

(tcp-bind 24 10
          (lambda (dest-ip source-ip source-port)
            (equal? dest-ip my-ip))
          (lambda (c)
            (set! connections (cons c connections))))

(main-loop)
```

UDP

- (udp-bind *portnum* *udp-filter* *udp-recv*)
- (*udp-filter* *dest-ip* *source-ip* *source-portnum*)
- (*udp-recv* *source-ip* *source-portnum* *data*)
- (udp-write *dest-ip* *source-port* *dest-port* *data*)

UDP echo server

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

(udp-bind 7
          (lambda (dest-ip source-ip source-port)
            (equal? dest-ip my-ip))
          (lambda (source-ip source-port data)
            (udp-write source-ip 7 source-port data)
            (stack-task)))

(main-loop)
```

Outline

- Motivation for a Scheme network stack
- Protocols supported by S^3
- Application program interface
- Implementation
- Related work
- Experimental results

Virtual machine

PICOBIT Scheme system :

- Compiler written in Scheme
- Virtual machine written in C
- Bytecode more abstract than machine instructions

Notable constraints :

- Objects in ROM are immutable
- 24-bit integers
- Small number of object encodings (either 256 or 8192)

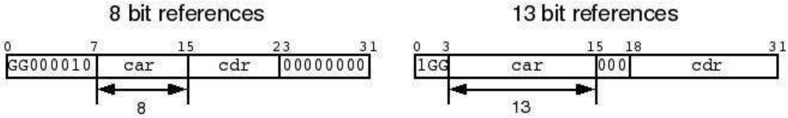
PICOBIT optimizations

- Constant propagation
- Tree-shaker (eliminates dead globals and functions)
- Function inlining for single calls
- Jump cascade tightening
- Specialized instruction set for S^3
- Functions only used in calls are not allocated an object encoding

PICOBIT object representation

Regular objects

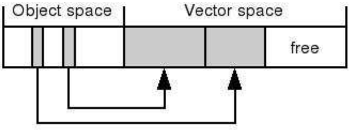
- Pairs, numbers, symbols, closures, continuations
- Always 4 bytes long
- Simple garbage collector (mark-and-sweep)
- Configurable encoding (8 or 13 bit references)



PICOBIT object representation

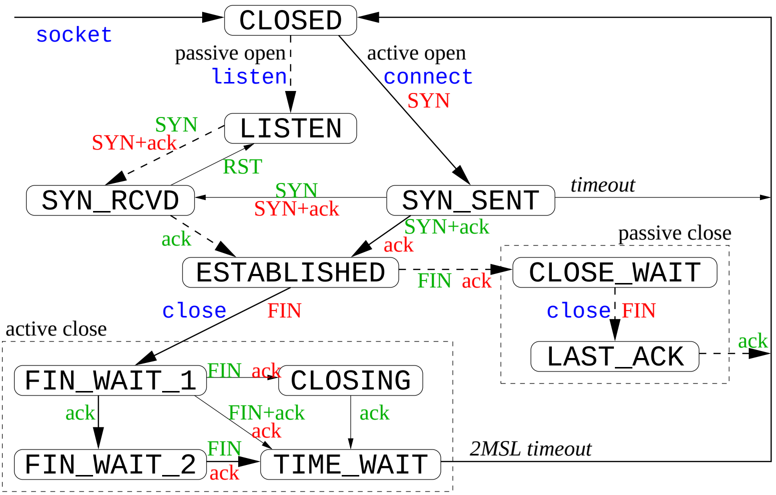
Byte vectors

- Omnipresent in S^3
- Stored separately from regular objects
  - Header stored in object space



  - Data stored in a contiguous block in vector space
  - Data space reclaimed when header gets garbage-collected
- Byte vector copy and equality implemented as virtual machine instructions

TCP connection automaton



First-class procedures

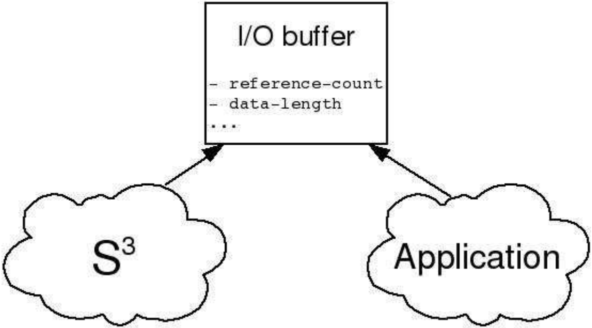
- TCP state functions :
  - Tasks stored as state functions within the connection

```
(define (tcp-visit conn)
  (set-state-function! conn ((state-function conn)))
  (set-info! conn tcp-attempts-count 0)
  (set-timestamp! conn))
```

  - Continuation-based coroutine mechanism
  - Dynamic creation of state functions
- Filter / reception functions

Garbage collection

- Objects with multiple owners



No need to implement reference counting in the stack

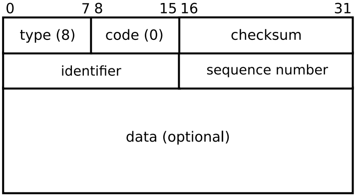
- Dangling pointer bugs and memory leaks eliminated
- Benefits on programmer productivity and code simplicity
- Mark-and-sweep

Packet limit

- S^3 constraints
  - One packet in the stack at a time
  - No buffering
  - Low amount of traffic for expected applications
- Pros
  - No costs related to the upkeep of a packet queue (code, space, time)
  - Not a threat to communication integrity
- Cons
  - Higher risk of congestion
  - Dropping packets might cause delays
- Most networking hardware already does a certain amount of buffering !

Reply generation

- Generated in-place
- Information stored only once
- Minimal changes to the headers
- Possible thanks to Scheme's mutable vectors
- Example :



Preallocated length packets

Approach :

- Fixed size vectors
- A packet might trigger a response longer than itself
- Packets are stored in a preallocated vector of length considered sufficient

Pros :

- In-place response generation
- No allocation / deallocation costs
- Two vectors of the right size may be larger than a single preallocated one

Cons :

- May waste space

Integration

- Easy integration with Scheme applications
- No FFI needed between applications and S^3
- Hardware access done within the virtual machine (`libpcap` linked with the PICOBIT virtual machine for workstation tests)

Outline

- Motivation for a Scheme network stack
- Protocols supported by S^3
- Application program interface
- Implementation
- Related work
- Experimental results

Related work

Embedded stacks :

- uIP (C, TCP only, real world use)
- lwIP (C, TCP & UDP)
- PowerNet (Forth, commercial product)

Functional stacks :

- FoxNet (SML, big, aims speed)
- House (Haskell, only UDP)

Outline

- Motivation for a Scheme network stack
- Protocols supported by S^3
- Application program interface
- Implementation
- Related work
- Experimental results

Goal

- $MIN \quad size(system) = size(stack) + size(application) + [size(VM)]$
- Comparison with uIP
  - Similar feature set
  - Similar design choices
    - No buffering
    - Application program interface

Space usage

Source code :

- S^3 : 1113 lines of Scheme
- uIP : 7725 lines of C

Binary :

- S^3 (bytecode) :

Full S^3	5.1 kB
TCP only	4.7 kB
UDP only	2.0 kB
RARP only	1.0 kB

  - Naïve commenting out of protocols
  - Other combinations possible to adapt to the target system

- uIP : 10 kB of machine code on PIC18

Virtual machine size

- $size(VM)$:

CPU	13 bit references	8 bit references
i386	17.0 kB	-
MSP430	10.4 kB	-
PIC18	10.7 kB	4.8 kB
PPC604	17.7 kB	-

- $size(stack) + size(VM)$ on PIC18 :

S^3version	References	Total size	Break-even point ($size(application)$)
Full S^3	13 bit	15.8 kB	11.4 kB
TCP only	13 bit	15.4 kB	10.8 kB
UDP only	8 bit	6.8 kB	S^3 always smaller
RARP only	8 bit	5.8 kB	S^3 always smaller

- Expected break-even point calculated using $size(application_C) = 2\ size(application_{Scheme})$

Future work

- Support for more protocols (PPP)
- Easy inclusion / exclusion of protocols using the PICOBIT tree-shaker
- Reducing VM size further with a specialized C compiler for (PICOBIT) virtual machines
- Browser-based operating system with JSS

Conclusion

- Abstraction can bring savings
- Bytecode-based approach
- Suitable for complex applications on small embedded systems
- Competitive with C