Implementing Protocols

Dominik Pantůček <dominik.pantucek@trustica.cz>

Trustica, Teesside University

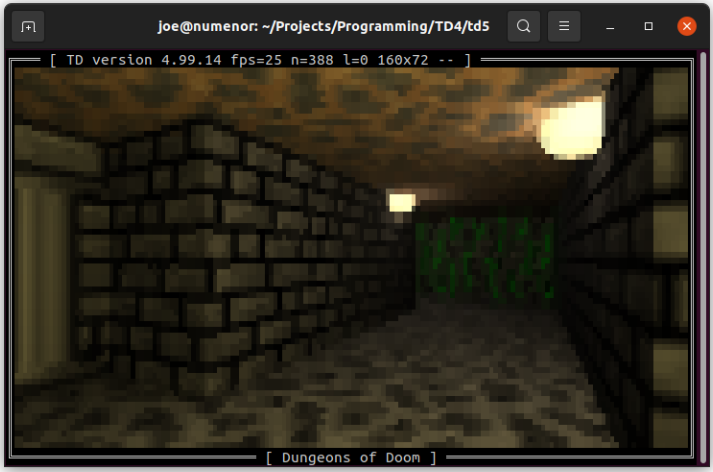
18. 3. 2023

Agenda

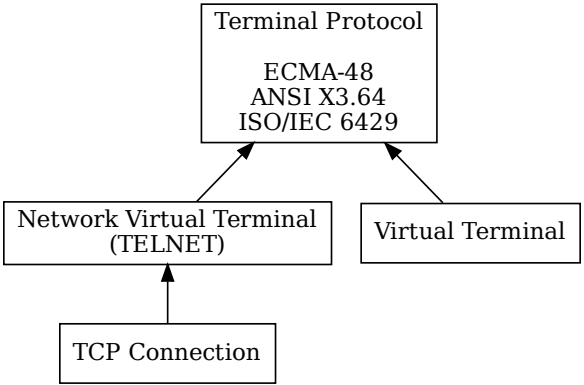
- Motivation
- Protocol Layers
- Input Ports
- Implementation
- Conclusion

Motivation

- Terminal output
- Semigraphics
- Keyboard input
- Window size
- Mouse input



Protocol Layers



Input Ports

- Reading S-expressions

```
(read)
```

- Waiting for data

```
(sync (current-input-port))
```

- Polling for data availability

```
(when (sync/timeout 0 (current-input-port))
  ...)
```

Implementation: The Basics

- Custom input port

```
(make-input-port name read-in peek close)
```

  - `name` – i.e. `'port-name`
  - Used in error reporting and similar contexts.
- Reading bytes: `read-in`

```
(define (read-in bytes) 0)
```

  - Never reads any data.
  - Always returns 0 bytes read.
- Peeking data: `peek`
  - Should return > 0 if some data is available.'
- Closing the port: `close`
  - May be handled elsewhere, `void` can be used.

Implementation: Changing Read Semantics

- Queueing data

```
(require data/queue)
(define the-queue (make-queue))
(enqueue! the-queue 'token)
```

- Reading tokens

```
(define token (dequeue! the-queue))
```

- Polling data availability – peeking

```
(when (non-empty-queue? the-queue)
  ...)
```

Implementation: Wrapping in a Struct

- Wrapping port in a struct.
- A struct property `prop:input-port`

```
(struct queue-port (port)
  #:property prop:input-port 0)
```

  - Struct field index 0 is used when the struct is used as input-port.
- Combining struct-based port with `data/queue`

```
(struct queue-port (port queue qsema)
  #:property prop:input-port 0)
```

  - `queue` – using `(make-queue)`
  - `qsema` – for protecting access to the queue

- Implementing `peek`

```
(define (peek bstr skip evt)
  (if (non-empty-queue? the-queue)
      1
      0))
```

Implementation: Wrapping Polling Data Availability

- Wrapping the struct constructor and peek procedure inside a closure

```
(define (make-queue-port (name 'queue-port))
  (define the-queue (make-queue))
  (define queue-sema (make-semaphore 1))
  (define (peek bstr skip evt)
    ...)
  (queue-port
    (make-input-port
      name
      read-in
      peek)
    the-queue
    queue-sema))
```

Polling Data Availability: Caveats

- Let's test it:

```
(define the-port (make-queue-port))
(sync/timeout 10 the-port)
```

- Works, but ...

```
$ time racket queue-port-test.rkt
#f

real 0m10,289s
user 0m10,201s
sys 0m0,084s
$
```

- It performs busy-wait loop!

Implementation: Better Polling

- Add current queue size explicitly – a counter.
- Counter as event?
- Counting semaphore!

```
(struct queue-port (port queue qsema csema)
  #:property prop:input-port 0)
```

- Now waiting for data is a simple sync.

```
(sync (queue-port-csema the-port))
```

- Using `sync` directly means decrementing the counter!

Implementation: Better Polling

- Use event that becomes ready for synchronization when the semaphore is.
- New peek procedure:

```
(define (make-queue-port (name 'queue-port))
  ...
  (define count-sema (make-semaphore 0))
  (define (peek bstr skip evt)
    (if (non-empty-queue? the-queue)
        1
        (semaphore-peek-evt count-sema)))
  (queue-port
    ...
    count-sema))
```

- Queueing data

```
(enqueue! the-queue 'token)
(seaphore-post count-sema)
```

- Reading tokens

```
(semaphore-wait count-sema)
(define token (dequeue! the-queue))
```

- Skipping the reader: `port-read-handler`

```
(define the-port (make-input-port ...))
(port-read-handler
   the-port
   (case-lambda
     ((port)
      (semaphore-wait count-sema)
      (call-with-semaphore
       queue-sema
       (thunk
        (dequeue! the-queue))))
     ((port source)
      (error 'read-syntax "Not implemented"))))
(queue-port the-port ...)
```

Implementation: Putting it All Together

```
(define (make-queue-port
          (name 'queue-port))
  (define queue-sema
    (make-semaphore 1))
  (define the-queue (make-queue))
  (define count-sema
    (make-semaphore 0))
  (define the-port
    (make-input-port name
     (λ (bytes) 0)
     (λ (bytes skip evt)
       (call-with-semaphore queue-sema
        (thunk
          (if (non-empty-queue?
               the-queue)
              1
              (semaphore-peek-evt
                count-sema)))))
     void))
```

```
(port-read-handler the-port
  (case-lambda
    ((port)
     (semaphore-wait count-sema)
     (call-with-semaphore
      queue-sema
      (thunk
       (dequeue! the-queue))))
    ((port source)
     (error 'read-syntax
            "Not implemented"))))
 (queue-port the-port the-queue
     queue-sema count-sema))
```

- The implementation is trivial at this point:

```
(define (queue-port-post! qp v)
  (call-with-semaphore
   (queue-port-qsema qp)
   (thunk
    (enqueue! (queue-port-queue qp) v)
    (semaphore-post (queue-port-csema qp)))))
```

TTY

- Get `termios` of the input port
- Use `cfmakeraw` (linked)
- Update `termios` of the input port
- Setup periodic checking of `TIOCGWINSZ` ioctl (no UNIX signals needed)
- Disable output buffering

NVT

- Negotiate NVT options
- Request and enable LINEMODE configuration
- Implement input port IAC (255) processing:
  - Escaped values 255
  - NVT control codes

- Pass input and output ports to the next layer.

Implementation: Protocol Layer

- Input port handles transport protocol parsing and filtering.
- Output port is properly configured.
  - No output filtering by default!
  - If the underlying terminal implementation is NVT, byte 255 is **not** escaped!
  - Commonly used UTF-8 encoding does not produce byte values of 255.
- ANSI protocol layer on incoming input port:
  - Key decoding: arrow keys, function keys ...
  - Mouse protocol decoding: position, movement, button presses and releases
- ANSI protocol setup using ANSI Control Sequences.
  - No output filtering (again)!
  - Sending byte 27 (ESC) has no special handling!
  - Commonly used UTF-8 encoding does not produce byte values of 27.
- Procedures wrapping generic control sequences: cursor movement, colors ...

TTY

```
(define tty (make-tty))
```

NVT

```
(define listener
  (tcp-listen 1234 4 \#t))

(define tty
  (nvt-tcp-accept listener))
```

```
(with-term tty
  (define key (read))
  (displayln (format "Pressed ~v." key))
  (for ((i (in-range 10)))
    (displayln "Tick!")
    (when (sync/timeout 1 (current-input-port))
      (displayln (format "Pressed ~v." key)))))
```

Conclusion

- Racket is a framework not only for language-oriented development.
- Racket ports allow:
  - composition (like in function composition)
  - layered architecture (bytes, characters, tokens, ...)
  - unidirectional and bidirectional semantics (as needed)
- Racket ports are good for protocol-oriented development.

Questions...

... and answers.

Thank you.
See you in the afternoon!