Falling Back!

... and: a Functional Decomposition of Post-Sockets

Michael Welzl

(with help from many NEATers)
Who can we talk to?

- New App
- New API
- Several Protocols

- New App
- New API
- New Protocol X

- New App
- New API
- TCP

- New App
- New API
- UDP

- Current App
  - Sockets
  - TCP

- Current App
  - Sockets
  - UDP
**Application-Framed (AFra-)Bytestream**

- We know that messages are cool
  - Can send them unordered, unreliable, ..

- But: TCP operates on streams
  - “Don’t care, I want to do something new?” Please wait until the end…

- Telling an application “sorry, you only get a stream here” is not much different than saying “sorry, use TCP instead”
  - How to reduce # hoops an app developer has to jump through for fall-back?

- Message-oriented TCP apps already frame their data
  - Unnecessary to repeat this in transport layer: hence Application-Framed
  - Requirement to tell receiver app “here is your complete message” creates a major limitation, but it’s often unnecessary
Example implementation

- Normal TCP-like bytestream API
  - Optional: some additional information exchanged

- **Sender app:** hands over a stream of bytes, informs transport about frame boundaries and requirements (order, reliability, ..)
  - Delimited **frames** stay intact, in order
  - More relaxed rules possible **between** frames
  - Delimiters assumed to be known by application

- **Receiver app:** receives stream of bytes
  - App-level delimiters turn it into messages

- **TCP = special case:** no delimiters used
  - Can talk to “normal” TCP applications
Unordered message delivery: SCTP

Sender app:
- Msg 3
- Msg 2
- Msg 1

App-defined header. Could also be e.g. implicit knowledge about size
- Inform where frame begins
- Inform where frame ends
- Configure: “unordered”

Receiver app:
- Msg 1
- Msg 3
- Msg 2

App knows how to identify messages
- Just a byte stream!

API

Block 1
Block 2
Block 3
Unordered message delivery: **TCP**

**Sender app**
- Msg 3
- Msg 2
- Msg 1

**Receiver app**
- Msg 1
- Msg 2
- Msg 3

- Inform where frame begins
- Inform where frame ends
- Configure: “unordered”
  …TCP just ignores this!

App knows how to identify messages

Just a byte stream!
Unreliable unordered msg delivery: SCTP

Sender app

- Inform where frame begins
- Inform where frame ends
- Configure: “unreliable, unordered”

Receiver app

- App knows how to identify messages
  - Just a byte stream!

API

Block 1
Block 2
Block 3

API

Block 1
Block 2
Block 3
Unreliable unordered msg delivery: TCP

Sender app

- Msg 1
- Msg 2
- Msg 3

API

Block 1 Block 2 Block 3

Receiver app

- Msg 1
- Msg 2
- Msg 3

API

Block 1

• Inform where frame ends
• Inform where frame begins
• Configure: “unreliable, unordered”
  ... TCP just ignores this!

App knows how to identify messages

Just a byte stream!

Block 1 Block 2 Block 3

neat
Unreliable message delivery: SCTP, large messages

• Inform where frame ends
• Inform where block begins
• Configure: “Unreliable”
Unreliable message delivery:
SCTP, large messages

Sender app

Receiver app

App knows how to identify messages

Just a byte stream!

Packets

SCTP
Some SCTP code consequences

• Sender side:
  – No need to know frame length at the beginning

• Receiver side:
  – Do not expose message semantics
    => partial message flag unnecessary
    (can hand over data as it arrives)
  – No need to support huge messages
    => always (try to) use interleaving
What is a flow?

- From minset TAPS draft:
  No application-specific knowledge involved in decision to use multiple connections or multiple streams of an association
  - To the app, “Multi-streaming” is only a flow grouping concept

- Suggest to only expose “flows”
  - 3 flows can e.g. be 3 streams of one association or 3 TCP connections
  - Flows have properties: flow group number, flow priority

- Note: this affects establishment / teardown semantics
  - E.g., streams of an association are always there, just begin to send
    => connect() without data not guaranteed to do anything on the wire
Where does this lead?

- Simple traditional TCP-like stream API with:
  1) Protocol-specific extras removed
  2) Optional sender-side app-transport info. exchange added
  3) Slightly changed connection setup/teardown semantics

...Enables:
- Downward compatibility
- Using unordered and partially reliable messages
- Using multi-streaming

- If used as an element of a post-sockets system, this...
  - Can enable falling back to TCP
  - Minimizes kernel API changes
A functional decomposition of post-sockets

“...don’t need to be told where my messages begin and end”

Unordered / partially reliable messages, multi-streaming

Sender

Receiver

Messaging App

TCP App

AFra-Bytestream

TCP Socket

Various protocols

TCP

Minion

Hollywood

Kernel

neat
A functional decomposition of post-sockets

Sender

**Messaging App**

**Framing layer**

**AFra-Bytestream**

**Various protocols**

**TCP**

**Minion**

**Hollywood**

Receiver

**Messaging App**

**Framing layer**

**TCP Socket**

**Minion**

**Hollywood**

“**I need to be told where my messages begin and end**”

Here’s your atomic object

e.g. COBS

Enable unordered delivery even with normal TCP sender
A functional decomposition of post-sockets

Post-sockets App

Post-sockets

Framing layer

AFra-Bytestream

Various protocols

TCP

Minion

Hollywood

Sender

Post-sockets App

Post-sockets

Framing layer

e.g. COBS

Minion

Hollywood

Receiver

Hold data in user space (TCP_NOTSENT_LOWAT / similar) or do it in the kernel

TCP Socket

Nice-ness, object dependencies, ...

TCP

Kerne
A functional decomposition of post-sockets

Sender

Post-sockets App

Post-sockets

Framing layer

AFra-Bytestream

Various protocols

Kernel

e.g. COBS

Receiver

Post-sockets App

Post-sockets

Framing layer

TCP Socket

Messaging App

TCP App

Kernel

Various protocols

Sender

Post-sockets App

Post-sockets

Framing layer

AFra-Bytestream

Various protocols

Kernel

e.g. COBS

Receiver

Post-sockets App

Post-sockets

Framing layer

TCP Socket

Messaging App

TCP App

Kernel
Questions?