Raising the Datagram API to Support Transport Protocol Evolution

Tom Jones, Gorry Fairhurst – University of Aberdeen
Colin Perkins – University of Glasgow

Presentation given at the IFIP Networking 2017 Workshop on Future of Internet Transport, 12 June 2017
Problem: Transport Ossification

- Existing transport protocols are globally deployed:
  - Can’t expect quick evolution of a network used by billions
  - TCP and UDP will be strongly conserved

- But – desire to change transport, to better meet application needs
The TCP Straightjacket

• Compatible evolution of TCP significantly constrained – too much infrastructure “understands” TCP protocol to permit changes

• UDP provides minimal services – but offers few constraints
• Alternative protocols not deployable

• For the transport to evolve in ways that differ from the TCP model, must tunnel over UDP
Enabling a UDP Substrate

- Layering new transport sub-layer over UDP is conceptually straight-forward
- Complexity is not in the layering, it’s in defining the new transport protocols; enabling flexible composition of transport services under a coherent API
Enabling Future Transport Services (1)

• Goal → raise the datagram API to support transport service composition and reduce implementation complexity

• What transport services to support in future protocols?
  • End-to-end security – while maintaining ease of management
  • NAT traversal and connection racing → as a generic service
  • Lower latency, avoiding HoL blocking
    • Alternative congestion control algorithms and ECN
    • Quality of service, active queue management, partial reliability
Enabling Future Transport Services (2)

• Goal → raise the datagram API to support transport service composition and reduce implementation complexity

• How to compose services and develop new systems?
  • Correctness of implementation → security and robustness
  • Ease of transport service composition, validation against specification, clean specification of policy
  • Integrate with higher-level systems languages → Go, Rust, Swift, …
What does the Sockets API do?

- Datagram API in Berkeley sockets:
  - Create socket
  - Bind to local port; “connect” remote
    - No on-the-wire effect from connection – locks destination address and enables receipt of ICMP responses on the socket
  - Send and receive datagrams
  - Set and get options
  - Resolve DNS names to IP addresses

- Limited consistency in option usage between different systems
  - Ad-hoc addition of features – different options to enable the same feature
  - Inconsistent feature implementation
  - Use of options to trigger actions
    setsockopt(socket, IPPROTO_IP, IP_ADD_MEMBERSHIP, ...)
What Transport Services are Missing?

Datagram API lacks critical features needed for new protocols

- Establishing connectivity
- Support for multiple interfaces
- Control over QoS and reliability
- Congestion control
What Transport Services are Missing?

Datagram API lacks critical features needed for new protocols

- Establishing connectivity
- Support for multiple interfaces
- Control over QoS and reliability
- Congestion control

connect(), listen(), accept() suitable for connection-oriented client-server protocols

- Unsuitable for peer-to-peer NAT traversal
  - No support for probing connectivity via STUN, TURN, ICE
- No connection racing
  - No support happy eyeballs IPv6 transition strategy
- Generically, no path layer features
  - No help discovering, probing, and gaining consent for use of path(s) from source to receiver
Datagram API lacks critical features needed for new protocols

• Establishing connectivity
• Support for multiple interfaces
• Control over QoS and reliability
• Congestion control

Interfaces can have radically different properties
• Present API doesn’t make these easy to discover; applications must probe to determine what works
• No way to determine if information gathered on an interface is valid on any other interface (e.g., DNS lookup results)

• Hard to portably determine valid interfaces, and changes to interface availability
• Complicates NAT traversal, connection racing
• No systems support for migrating traffic flows between network interfaces
What Transport Services are Missing?

Datagram API lacks critical features needed for new protocols

- Establishing connectivity
- Support for multiple interfaces
- **Control over QoS and reliability**
- Congestion control

TCP provides a reliable, ordered, byte-stream subject to HoL blocking -- no flexibility in API

Datagram API exposes best effort IP service, but no help with (partial) reliability, ordering, or framing

- Limited support for ECN use with datagrams
- Limited support for QoS use with datagrams – how to determine what code points work?
What Transport Services are Missing?

Datagram API lacks critical features needed for new protocols

- Establishing connectivity
- Support for multiple interfaces
- Control over QoS and reliability
- Congestion control

TCP assumes congestion control occurs below API, and doesn’t expose behaviour or offer any controls

- Datagram service requires congestion control be implemented above the API, with no support and no visibility into send/receive queues
- No support for cooperation between congestion controller, transport, and application – needed to ensure low-latency
What Transport Services are Missing?

Datagram API lacks critical features needed for new protocols

- Establishing connectivity
- Support for multiple interfaces
- Control over QoS and reliability
- Congestion control

Clear the current API is too low level – doesn’t meet needs of applications or help implementors of new transport protocol layers
Principles for Raising the Datagram API

• Follow four principles when revising the API:
  • An application using the new API that does nothing new should receive similar service to that of the Sockets API
  • Commonly needed functions should be placed below the API when these can be automated – do not require application decisions
  • Functions where the preference can be expressed as a policy can also be placed below the API
  • Functions that rely on application algorithms or detailed knowledge of trade-offs related to data should be implemented above the API

• Ensures continuity of behaviour, avoids surprises, while allowing transport evolution
Below the Datagram API – Policies and State

• Higher-layers pass abstract policy information through the API – map onto transport services rather than concrete protocol features

• Per-interface information base:
  • MTU, line rate, IP address, DNS name cache, supported QoS features, …

• Per-path information base:
  • Credentials for crypto session resumption, key continuity, opportunistic encryption, …
  • Last achieved congestion control state
  • Destination feature support

• Policy specifies what is needed, not how accomplished

![Listing 2. Example JSON file describing a NEAT Abstract Policy](example_json_file)

```json
{
  "transport": [  
    {  
      "value": "Datagram", "precedence": 1
    },
  ],
  "qos": [  
    {  
      "value": "Interactive Video", "precedence": 1
    },
    {  
      "value": "Live Video", "precedence": 2
    }
  ],
  "network": [  
    {  
      "value": "cost", "precedence": 1
    },
    {  
      "value": "capacity", "precedence": 2
    }
  ]
}
```
Below the Datagram API – Mechanisms

- Policies bind to concrete protocol features and transports
  - DSCP, ECN, IP addresses, network interfaces, congestion controllers, etc.
  - TCP, UDP, SCTP, ...

- Push new transport implementations below the API – UDP as transport demultiplex
  - Congestion control algorithms
  - Reliability – retransmission, FEC
  - Reordering
  - PDU parsing and serialisation, framing
  - Connection racing, probing for NAT traversal – driven by high level policy

- Asynchronous and event driven
Implementation Approach

- Asynchronous, to match network behaviour
- Rich sharing of data across the API
  - Application policies and preferences
  - Queries of interface/path management data
  - Jointly managed send and receive queues

- A higher-level API for applications… and, below that, a richer API framework for transport services
Example: The NEAT API

- An example of the application API:
  - JSON policy specification – policy manager component below API
  - Asynchronous event loop – callback driven
  - Allows transport behaviour to be chosen by the stack – e.g., happy eyeballs connection racing

```
static struct neat_flow_operations ops;
static struct neat_ctx *ctx = NULL;
static struct neat_flow *flow = NULL;
ctx = neat_init_ctx();
flow = neat_new_flow(ctx)
prop = "(see Listing 2)";
neat_set_property(ctx, flow, &prop)
ops.on_writable = on_writable;
ops.on_readable = on_readable;
ops.on_error = on_error;
neat_set_operations(ctx, flow, &ops)
neat_open(ctx, flow, hostname, port)
neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);
static neat_error_code
on_writable( struct neat_flow_operations *opCB)
{
    neat_write(opCB->ctx, opCB->flow, buf)
    return NEAT_OK;
}

static neat_error_code
on_readable( struct neat_flow_operations *opCB)
{
    neat_read(opCB->ctx, opCB->flow, buf)
    return NEAT_OK;
}
```

Listing 3. NEAT Example Application listing
Raising the Datagram API to Support Protocol Evolution

• We propose raising the datagram API to allow specification of policy and transport services

• Give the protocol stack flexibility to fulfil application needs via different transports

• Post Sockets APIs must raise the level of abstraction and enable composition of transport services – raising the datagram API is but a first essential step