Transport Area
Internet-Draft
Intended status: Informational
Expires: June 18, 2014

P. Hurtig, Ed.
Karlstad University
S. Gjessing
M. Welzl
University of Oslo
M. Sustrik

December 15, 2013

Transport APIs
draft-hurtig-tsvwg-transport-apis-00

Abstract

Commonly used networking APIs are currently limited by the transport layer's inability to expose services instead of protocols. An API/ application/user is therefore forced to use exactly the services that are implemented by the selected transport. This document surveys networking APIs and discusses how they can be improved by a more expressive transport layer that hides and automatizes the choice of the transport protocol.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The intention of this document is to create an understanding of some commonly used network APIs and how the mechanisms they provide could possibly be enhanced via a richer set of transport services. A non-comprehensive list of APIs is given, along with a brief description and a discussion of how they relate to services provided by current transports.

To understand what tools a transport system could have available to better realize mechanisms that higher level APIs offer, the next section gives a high-level (and most certainly incomplete) overview

of services offered by transports that have been published by the IETF or are currently being proposed.

This overview is followed by two sections describing different types of transport APIs: general APIs and APIs exposing the underlying transport.

The general APIs can intuitively benefit from a richer set of transport services as they do not expose the underlying transport to the application. Section 3 describe a subset of these APIs and analyze how they can benefit from transport services. The complexity of these APIs range from providing simple transport interfaces to providing advanced communication libraries utilizing message-oriented middleware. API-wise there are two broad classes of such middleware: centralized solutions where a server manages the communication and decentralized ones where the endpoints communicate directly. Although there is no standard interface for these types of middleware the JMS API (see Section 3.4) can be thought of as the canonical API for centralized solutions and the BSD socket API, as implemented by nanomsg (see Section 3.2), for the decentralized.

APIs that expose the underlying transport, including e.g. BSD sockets, differ a lot from general APIs as they both require an explicit choice of transport, and then expose this choice. This is a significant limitation in the context of transport services, as an explicit choice of transport also limits the amount of services that can be used. It is, however, possible to enhance this type of APIs as some transports provide services that are not fully exposed to applications. Section 4 explains how such services can be used and provides descriptions of the most common APIs and how they can be enhanced.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Services Offered by IETF Transports

From [WJG11], TCP [RFC0793] [RFC5681], UDP [RFC0768], UDP-Lite [RFC3828], SCTP [RFC4960] and DCCP [RFC4340] offer various combinations of: TCP-like congestion control / "smooth" congestion control (which is expected to have less jitter); application PDU bundling (which is the mechanism called "Nagle" in TCP); error detection (using a checksum with full or partial payload coverage); reliability (yes/no); delivery order. The point of not always requiring full reliability and ordered delivery is that these

mechanisms can come at the cost of extra delay which is unnecessary if these properties of the data transmission are not needed. After the publication of [WJG11], some more features were defined, e.g. SCTP now also offers partial reliability using a timer.

MPTCP [RFC6824] and SCTP offer multihoming for improved robustness (as a backup in case a path fails), which is a mechanism that is listed in [WJG11] but could perhaps be hidden from an application. Similarly, it was shown in [WNG11] that the benefits of multi-streaming (mapping multiple application streams onto one connection, or "association" in SCTP terminology) can be exploited without exposing this functionality to an application. Because of this assumption, multi-streaming was not included as a service in [WJG11].

MPTCP and CMT-SCTP also use multiple paths to achieve better performance, at the possible cost of some extra delay and jitter; as discussed in Appendix A.2 of [RFC6897], an advanced MPTCP API could allow applications to provide high-level guidance about its requirements in terms of high bandwidth, low latency and jitter stability, or high reliability.

The newly proposed Minion [MINION] has a somewhat different way of translating some of the above mentioned lower-level transport mechanisms (e.g. multi-streaming or partial reliability) into application services. It provides message cancellation and has a notion of superseding messages, i.e. a later message rendering a prior one unnecessary. Ordered delivery is provided according to pre-specified message dependencies, and a request-reply communication model is offered (i.e. a message can be a reply to another message, i.e. address the original message's reply-handler).

When applying multi-streaming, priorities between streams become a mere scheduling decision. In the absence of multi-streaming, there is at least one congestion control method in an RFC that is more aggressive than standard Reno-like TCP (HighSpeed TCP [RFC3649]), and there is also the more recent LEDBAT [RFC6817] which is specifically designed for low-priority "scavenger" traffic. All in all, it is probably correct to say that IETF transports are likely to be able to honor priorities between data streams in one way or another.

3. General Networking APIs

This section introduces and provides an analysis of commonly used networking APIs in the context of transport services. That is, how are these APIs currently designed and how, if at all, can these APIs be simplified and/or enhanced given a transport API that exposes all services provided by the operating system.

Please note that the current list of APIs is incomplete and rather arbitrary. Feedback is very welcome!

3.1. ZeroMQ

3.1.1. Description

ZeroMQ is a messaging library that simplifies and improves the usage of sockets. It operates on messages, and has embedded support for a variety of communication styles including e.g. request/reply or pub/sub. What this means is that, for instance, a socket of type "request" can issue one request, and then a reply must arrive on that socket; any other sequence of communication will produce an error message. ZeroMQ tries to be transport agnostic and currently works on top of IPC, TCP and PGM.

Internally, ZeroMQ's functionality largely depends on buffering mechanisms. For instance, in contrast to native Berkeley sockets, a single server socket can be used to read and respond to requests from multiple clients. To achieve this, ZeroMQ must accept incoming requests and read their data as they arrive from multiple clients, buffer them, and upon the application's request hand the data over to the application using fair queuing.

3.1.2. Analysis

Like Minion, ZeroMQ introduces delimiters into a TCP stream to send frames of a given size using the ZeroMQ Message Transport Protocol [ZMTP]. Some form of multi-streaming is intended for the future: According to the FAQ [ZMQFAQ] page, having multiple sockets share a single TCP connection is being added to the next version of the ZMTP protocol. Today one can accomplish this "using a proxy that sits between the external TCP address, and your tasks".

Multi-streaming over standard TCP creates an RTT of HOL blocking delay for all out-of-order packets that arrive at the receiver's buffer. This problem also occurs with e.g. SPDY [SPDYWP] [SPDYID] over TCP; just like SPDY works better over QUIC [QUIC], ZeroMQ can be made to work better over a transport that natively supports multi-streaming.

Because ZeroMQ is implemented as a user space library, it cannot multiplex streams from multiple processes. This can be a significant drawback when many small stand-alone services are co-located on the same host. In contrast, in line with the way TCP and UDP are currently implemented, it is likely that broader transport services would be provided monolithically, e.g. in the system's kernel, thereby eliminating this problem.

The notion of request and reply sockets seems to be similar in Minion and in ZeroMQ. Hence, mapping such ZeroMQ sockets onto Minion is probably an efficient way to implement them. One may wonder where to draw the boundaries between a transport like Minion and a middleware or library like ZeroMQ, i.e. is it really more efficient to provide request-reply functionality in the transport layer? Conceptually, many of Minion's functions (e.g., message cancellation and superseding messages) relate to having direct access to the sender and receiver-side buffers, which is otherwise limited depending on the TCP implementation, and by standard TCP's in-order-delivery requirement. At the same time, ZeroMQ's functions have to do with controlling the sender and receiver-side buffers; it therefore seems natural that transports such as Minion could improve the performance of ZeroMQ.

Notably, some transports might turn out to be a poor match for ZeroMQ. For example, MPTCP requires a larger receiver buffer than standard TCP due to the larger expected reordering. However, if ZeroMQ's ZMTP protocol does or will (in accordance with the FAQ mentioned above) multiplex data from several sockets over a single TCP stream, this might create extra delay before the the receiver-side ZeroMQ instance can take the data from the buffer and hand it over to the application.

3.2. nanomsg

3.2.1. Description

3.2.2. Analysis

3.3. enet

3.3.1. Description

enet started out as a networking layer for a first-person shooter where low latency communication with very frequent data transmission was needed. It is a lightweight library that is entirely based on UDP, which it extends with a set of optional features such as reliability and in-order packet delivery.

Its features include connection management (monitoring of a connection with frequent pings), optional reliability, sequencing (mandatory for reliable transmission), fragmentation and reassembly, aggregation, flow control. It gives its user control over the packet size (a function call allows a packet to be resized), and sequential delivery is enforced.

Reliability in enet is a binary choice; it does not allow providing a deadline or maximum number of retransmissions per packet; if a per-host-configurable number of retries is exceeded, the host is disconnected.

Because HOL blocking delay can arise when guaranteeing sequential delivery, enet also has a form of multi-streaming (called "channels").

enet provides window-based flow control for reliable packets and a dynamic throttle that drops packets from the send buffer if the network is congested based on a given probability. This probability is based on measuring the RTT to a peer; if the current RTT is significantly greater than the mean RTT, the probability is increased up to a configurable maximum value. Each host's bandwidth limits are taken into account as an upper bound for the bandwidth used by enet.

A broadcast function can be used to send a packet to all currently connected peers on a host.

3.3.2. Analysis

Many of the functions in enet resemble functions found in SCTP and Minion -- e.g., control over the packet size, optional reliability, multi-streaming. Since enet intends to be "thin", simply using these protocols instead probably would not make it better. However, enet's goal being low latency, it could benefit from other functions such as SCTP's and MPTCP's multi-path capability (picking the lower latency path). The congestion control also appears to be rather rudimentary -- there are known issues with using the RTT as a congestion signal (for one, it is incapable of distinguishing between congestion on the forward and backward path). Probably, using the congestion control embedded in an IETF-standardized protocol could improve enet's performance under certain situations. Finally, the "broadcast" functionality could benefit from multicast.

3.4. Java Message Service

3.4.1. Description

3.4.2. Analysis

3.5. Chrome Network Stack

3.5.1. Description

3.5.2. Analysis

3.6. CFNetwork

3.6.1. Description

3.6.2. Analysis

3.7. Apache Portable Runtime

3.7.1. Description

3.7.2. Analysis

3.8. VirtIO

3.8.1. Description

3.8.2. Analysis

4. Networking APIs with Exposed Transport

Much of the motivation behind the transport services concept comes from the limitations posed by networking APIs that require the user to explicitly chose a transport, and thus confine itself to a certain number of "services". It is, however, possible to include such APIs in the transport services concept if mechanisms can be hidden from the application [WNG11].

This section describes a number of commonly used APIs that expose the underlying transport and analyzes how these particular APIs could be improved with transport services.

4.1. Berkeley Sockets

4.1.1. Description

4.1.2. Analysis

4.2. Java Libraries

4.2.1. Description

The Java library has classes to handle TCP and UDP sockets. There is also a separate library, not included with the regular Java distribution, that interfaces SCTP.

The java.net library contains the two classes Socket and ServerSocket that handle TCP sockets. These sockets write a message at a time, but read character streams. A ServerSocket contains a method called

"accept", that waits for a connection request from a client. The class DatagramSocket handles UDP-sockets. It "receive"s and "send"s objects of the class DatagramPacket that contain characters. The "close" method closes the connection. Finally the library contains a class called NetworkInterface that can be used to query the operating system about available network interfaces.

The separate Java library that handle SCTP a is called com.sun.nio.sctp. Similar to the TCP-sockets there are classes called SctpChannel and SctpServerChannel. An instance of the former can control a single association only, while an instance of the latter can control multiple associations. Instances of the class SctpMultiChannel can also control multiple associations.

4.2.2. Analysis

The Java socket api is very similar to the Berkeley socket api. A main difference is that the transport to be used is defined as a parameter to the socket() call in the Berkeley socket api, while in Java different classes is used for the different protocols. There is no well known support for DCCP in Java.

When a socket object is created it can either be connected immediately, or the "connect" method can be called later. If not already bound, a socket is bound to a local address by calling the method "bind". To shut down the connection, "close" is called. If an application calls "receive" on a datagram socket, the method call will block the application until a packet is received, which may never happen using an unreliable transfer. When operations on Sockets fail, an exception is thrown.

The SCTP interface is event driven. When the SCTP stack wants to notify the applications, it generates a Notification object. This object is passed as parameter to the method "handleNotification" in an instance of the class NotificationHandler. An association will be implicitly set up by a send or receive method call if there is no current association. The SCTP library is only supporter at run time by Linux and Solaris.

4.3. Netscape Portable Runtime

4.3.1. Description

4.3.2. Analysis

4.4. Infiniband Verbs

4.4.1. Description

4.4.2. Analysis

4.5. Input/Output Completion Port

4.5.1. Description

4.5.2. Analysis

5. Security Considerations

TBD

6. IANA Considerations

At this point, the memo includes no request to IANA.

7. Acknowledgments

Hurtig, Gjessing, and Welzl are supported by RITE, a research project (ICT-317700) funded by the European Community under its Seventh Framework Program. The views expressed here are those of the author(s) only. The European Commission is not liable for any use that may be made of the information in this document.

8. Comments Solicited

To be removed by RFC Editor: This draft is a part of the first steps towards an IETF BoF on Transport Services. Comments and questions are encouraged and very welcome. They can be addressed to the current mailing list <transport-services@ifi.uio.no> and/or to the authors.

9. References

9.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3208] Speakman, T., Crowcroft, J., Gemmell, J., Farinacci, D., Lin, S., Leshchiner, D., Luby, M., Montgomery, T., Rizzo, L., Tweedly, A., Bhaskar, N., Edmonstone, R., Sumanasekera, R., and L. Vicisano, "PGM Reliable Transport Protocol Specification", RFC 3208, December 2001.

[RFC3649] Floyd, S., "HighSpeed TCP for Large Congestion Windows", RFC 3649, December 2003.

[RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., and G. Fairhurst, "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, July 2004.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, March 2006.

[RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, December 2012.

[RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, January 2013.

[RFC6897] Scharf, M. and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations", RFC 6897, March 2013.

9.2. Informative References

[MINION] Iyengar, J., Cheshire, S., and J. Graessley, "Minion - Service Model and Conceptual API", draft-iyengar-minion-concept-02.txt (work in progress), October 2013.

[QUIC] Roskind, J., "QUIC: Design Document and Specification Rational", April 2012, <https://bitly.com/Hm0DyX>.

[SPDYID] Belshe, M. and R. Peon, "SPDY Protocol", draft-mbelshe-httpbis-spdy-00.txt (work in progress), February 2012.

[SPDYWP] Belshe, M., "SPDY: An Experimental Protocol for a Faster Web", April 2012, <http://www.chromium.org/spdy/spdy-whitepaper>.

[WJG11] Welzl, M., Jorer, S., and S. Gjessing, "Towards a Protocol-Independent Internet Transport API", IEEE ICC 2011., June 2011.

[WNG11] Welzl, M., Niederbacher, F., and S. Gjessing, "Beneficial Transparent Deployment of SCTP: the Missing Pieces", IEEE GLOBECOM 2011, December 2011.

[ZMQFAQ] Sustrik, M., "Frequently Asked Questions - zeromq", July 2008, <http://zeromq.org/area:faq>.

[ZMTP] Hintjens, P., Hurton, M., and I. Barber, "ZMTP - ZeroMQ Message Transport Protocol", June 2013, <http://rfc.zeromq.org/spec:23>.

Authors' Addresses

Per Hurtig (editor)
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

Stein Gjessing
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 44
Email: stein.gjessing@ifi.uio.no

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Martin Sustrik

Phone: +421 908 714 885
Email: sustrik@250bpm.com