June 2015
Geoff Huston

Multipath TCP

The Transmission Control Protocol (TCP) is a core protocol of the Internet networking protocol suite. This protocol transforms the underlying unreliable datagram delivery service provided by the IP protocol into a reliable data stream protocol. This protocol was undoubtedly the single greatest transformative moment in the evolution of computer networks.

Prior to TCP computer network protocols assumed that computers wanted a lossless reliable service from the network, and worked hard to provide it. DDCMP in DECnet was a lossless data link control protocol. X.25 in the telesaur world provided reliable stream services to the attached computers. Indeed, I recall that Ethernet was criticized when it was introduced to the world because of its lack of a reliable acknowledgement mechanism. TCP changed all of that. TCP pushed all of the critical functionality supporting reliable data transmission right out of the network and into the shared state of the computers at each end of the TCP conversation. TCP embodies the end-to-end principle of the Internet architecture, where there is no benefit in replicating within the network functionality that can be provided by the end points of a conversation. What TCP required of the network was a far simpler service where packets were allowed to be delivered out of order, but packets could be dropped and TCP would detect and repair the problem and deliver to the far end application precisely the same bit stream that was passed into the TCP socket in the first place.

The TCP protocol is now some 40 years old, but that doesn't mean that it has been frozen over all these years.

TCP is not only a reliable data stream protocol, but also a protocol that uses adaptive rate control. TCP can operate in a mode that allows the protocol to push as much data through the network as it can. A common mode of operation is for an individual TCP session to constantly probe into the to see what the highest sustainable data rate is, interpreting packet loss as the signal to drop the sending rate and resume the probing. This aspect of TCP has been a constant field of study, and much work has been done in the area of flow control and we now have many variants of TCP that attempt to optimize the flow rates across various forms of networks.

Other work has looked at the TCP data acknowledgement process, attempting to improve the efficiency of the algorithm under a broad diversity of conditions. SACK allowed a receiver to send back more information to the sender in response to missing data. FACK addresses data loss issues during slow start.

One approach to trying to improve the relative outcome of a data transfer, as compared to other simultaneously open TCP sessions, is to split the data into multiple parts and send each part in its own TCP session. Effectively opening up a number of parallel TCP sessions. A variant of TCP, MulTCP, emulates the behavior of multiple parallel TCP sessions in a single TCP session. These behaviours assume the same endpoints for the parallel TCP sessions and assume the same end-to end path through the network. An evolution of TCP that uses multiple parallel sessions, but tries to spread these sessions across multiple paths through the network, is Mutipath TCP.

Multipath TCP had a brief moment of prominence when it was revealed that Apple's release of iOS 7 contained an implementation of Multipath TCP for their Siri application, but it has the potential play a bigger role in the mobile Internet. In this article I would like to explore this TCP option in a little more detail, and see how it works and how it may prove to be useful in today's mobile networks.

Multi-Addressing in IP

First we need to return to one of the basic concepts of networking, that of addressing and addresses. Addresses in the Internet Protocol were subtly different from many other computer communications protocols that were commonly in use in the 1970's and 1980's. While many other protocols used the communications protocol level address as the address of the host computer, the Internet Protocol was careful to associate an IP address with the interface to a network. This was a relatively unimportant distinction in most cases as computers usually only had a single network attachment interface. But it was a critical distinction when the computer had two or more interfaces to two or more networks. An IP host with two network interfaces has two IP protocol addresses, one for each interface. In IP it is the interface between the device and the network that is the addressed endpoint in a communication. An IP host accepts an IP packet as being addressed to itself if the IP address in the packet matches the IP addresses of the network interface that received the packet, and when sending a packet, the source address in the outgoing packet is the IP address of the network interface that was used to pass the packet from the host into the network.

As simple as this model of network addressing may be, it does present some operational issues. One implication of this form of addressing is that when a host has multiple interfaces, the application level conversations using the TCP protocol are "sticky". If, for example, a TCP session was opened on one network interface, the network stack in the host cannot quietly migrate this active session to another interface while maintaining the common session state. An attempt by one "end" of a TCP conversation to change the IP address for an active session would not normally be recognized at the other end of the conversation as being part of the original session. So multiple interfaces and multiple addresses does not create additional resiliency of TCP connections.

The simplicity of giving each network interface a unique IP address does not suit every possible use case, and it was not all that long before the concept of "secondary addresses" came into use. This was a way of using multiple addresses to refer to a host by allowing a network interface to be configured with multiple IP addresses. In this scenario, an interface will receive packets addressed to any of the IP addresses associated with the interface. Outgoing packet handling allows the transport layer to specify the source IP address, which would override the default action of using the primary address of the interface on outgoing packets. Secondary addresses have their uses, particularly when you are trying to achieve the appearance of multiple application-level "personas" on a single common platform, but in IPv4 they were perhaps more of an specialized solution to a particular family of requirements, rather than a commonly used approach. Applications using TCP were still "sticky" with IP addresses that were used in the initial TCP handshake and could not switch the session between secondary IP addresses on the same interface.

IPv6 addressing is somewhat different. The protocol allows from the outset for an individual interface to be assigned multiple IPv6 unicast addresses without the notion of "primary" and "secondary" addresses. The IPv6 protocol introduces the concept of an address "scope", so an address may be assuredly unique in the context of the local link-layer network, or it may have a global scope, for example. Privacy considerations have also introduced the concept of permanent and temporary addresses, and the efforts to support a certain form of mobility have introduced the concepts of "home addresses" and "care-of" addresses.

However, to some extent these IPv6 changes are cosmetic modifications to the original IPv4 address model. If an IPv6 host has multiple interfaces, each of these interfaces will have its own set of IPv6 addresses, and when a TCP session is started using one address pair TCP does not admit the ability to

shift to a different address pair in the life to the TCP session. A TCP conversation that started over one network interface is stuck with the network interface for the life of the conversation whether its IPv4 or IPv6.

The Internet has changed significantly with the introduction of the mobile Internet, and the topic of multi-addresses is central to many of the issues with mobility. Mobile devices are adorned with many IP addresses. The cellular radio interface has its collection of IP addresses. Most of these "smart" devices also have a WiFi interface which also has its set of IPv4 and possibly IPv6 addresses. And there may be a Bluetooth network interface with IP addresses, and perhaps some USB network interface as well. When active each of these network interfaces require their own local IP address. We now are in an Internet where devices with multiple active interfaces and multiple useable IP addresses are relatively commonplace. But how can we use these multiple addresses?

For many scenarios there is little value in being able to use multiple addresses. The conventional behavior is where each new session is directed to a particular interface, and the session is given an outbound address as determined by local policies. However, when we start to consider applications where the binding of location and identity is more fluid, and where network connections are transient, and the cost and capacity of connections differ, as is often the case in todays mobile cellular radio services and in WiFi roaming services, then having a session that has a certain amount of agility to switch across networks can be a significant factor.

If individual end-to-end sessions were able to use multiple addresses, and by inference able to use multiple interfaces, then an application could perform a seamless handoff between cellular data and WiFi, or even use both at once. Given that the TCP interface to IPv4 and IPv6 is identical it is even quite feasible to contemplate a seamless handoff between the two IP protocols. The decision as to which carriage service to use at any time would no longer be a decision of the mobile carrier or that of the WiFi carrier, or that of the device, or that of its host operating system. If applications were able to utilize multiple addresses, multiple protocols and multiple interfaces, then the decision could be left to the application itself to determine how best to meet its needs as connections options become available or as they shut down. At the same time as the debate between traditional mobile operators in the licensed spectrum space and the WiFi operators in the unlicensed spectrum space heats up over access to the unlicensed spectrum, the very nature of how "WiFi handoff" is implemented by devices and by applications is changing. Who is in control of this handoff function is changing as a result. Multi-Addressing and Multipath TCP is an interesting response to this situation by allowing individual applications to determine how they want to operate in a multi-connected environment.

SHIM6

One of the first attempts to make use of multiple addresses in IP was the SHIM6 effort in IPv6.

In this case the motivation was end site resilience in an environment of multiple external connections, and the constraint was to avoid the use of an independently routed IPv6 address prefix for the site. So this was an effort to support site multi-homing without routing fragmentation. To understand the SHIM6 model we need to start with an end site that does not have its own provider independent IPv6 address prefix, yet is connected to two or more upstream transit providers who each provide addresses to the end site. In IPv4 its common to see this scenario approached with Network Address Translators (NATs). In IPv4 the site is internally addressed using a private address prefix, and the interface to each upstream provider is provisioned with a NAT. Outbound packets have their source address rewritten to use an address that is part of the provider's prefix as it transits the NAT. Which provider is used is a case of internal routing policies towards each of the NATs. While it is possible to configure a similar setup in IPv6 using an IPv6 ULA prefix as the internal address and NAT IPv6-to-IPv6 devices to each upstream service provider, one of the concepts behind IPv6 and its massive increase in address space was the elimination of NATs. So how can an IPv6 end site be homed into multiple upstream service

providers, yet do so without needing to advertise a more specific routing entry in the inter domain routing tables and avoiding the use of any form of network address translation?

The conventional IPv6 architecture has the site receiving an end-site prefix delegation from each of its upstream service providers, and the interface routers would each advertise its end site prefix into the site. Hosts within the site would see both router advertisements, and would configure their interface with multiple IPv6 addresses, one for each site prefix. Presumably, the end site has chosen to multi-home in order to benefit from the additional resiliency that such a configuration should offer. When the link to one provider is down, there is a good chance that the other link will remain up, particularly if the site has been careful to engineer the multi-homed configuration using discrete components at every level. It would be even better if it were the case that even when the link to the upstream provider is up, if that provider is unable to reach a specific destination, then another of the site's upstream providers would be able to continue to carry on and support all active end-to-end conversations without interruption, in exactly the same manner as happens when this functionality is implemented in the routing system.

What SHIM6 attempted was a host-based approach to use the additional local IPv6 addresses in the host as indicators of potential backup paths to a destination. If a communication with a remote counterpart were fail (i.e. the flow of incoming packets from the remote host stopped) then idea was that the IP level shim in the local host would switch to use a different source/destination address pair. To prevent the upper level transport protocol from being fatally confused by these address changes in the middle of one or more active sessions, the local SHIM module also included a network address translation function, so that while the address pair on the wire may have changed, the address pair presented to the upper layer by the shim would remain constant, and the path change would not be directly visible at the transport layer of the protocol stack.

This approach essentially folds the NAT function into the host IP protocol stack. In terms of design it avoided altering either TCP or UDP, and endeavored to preserve the IP addresses used by active transport sessions. What this implied was that if you wanted to change the routing path, but not change the IP addresses used by transport, then address translation was an inevitable consequence. Network-based NATs was the response in IPv4, and to avoid this in IPv6 the SHIM6 effort attempted to push the NAT functionality further "back", implementing a NAT in each host.

SHIM6 was a less than entirely satisfactory approach.

Network operators expressed deep distrust that pushed decision making functionality back into individual hosts (an distrust that network operators continue to hold when the same issue arises with WiFi handoff). The Network operators wanted to control the connectivity structure for the hosts in their network, in precisely the same manner as the routing system provided network level control over traffic flows. So while these network operators had some sympathy with the SHIM6 objective of avoiding further bloat in the routing table, which reduced the "independence" of attached end sites by using IPv6 address prefixes drawn from the upstream's address block, they were unsupportive of an approach that pushed connectivity choice and control back to individual end host systems.

Outside of this issue of control over the end host there was another multi-homing problem that SHIM6 did not address. While the provision of backup paths in the case of failure of the primary path is useful, what is even more useful is to be able to use the backup paths in some form of load sharing configuration. However, at this point the SHIM6 approach runs into problems. Because SHIM6 operates at the IP layer it is not directly aware of packet sequencing. When a SHIM unit at one end of a conversation splays a sequence of packets across multiple paths, the corresponding SHIM unit at the remote end will pass the packets into the upper transport layer in the order of their arrival, not in the original order. This out of order delivery can prove to be a significant problem for TCP if multiple paths are left open by SHIM6. The best SHIM6 can provide is a primary / backup model for individual sessions, where at any time all data traffic for a session is passed along the primary path.

Inexorably, we are drawn to the conclusion that the most effective place to insert functionality that allows a data flow to utilize multiple potential paths across the network is in the transport layer itself, and ne weed to jack ourselves further up the protocol stack from the IP level approach of SHIM6 and re-examine the space from the perspective of TCP.

Multipath TCP

The approach of incorporating multiple IP addresses in the transport protocol is comparable to SHIM6's efforts one level further down in the protocol stack, in so far as this is an end-to-end mechanism with a shared multiplex state maintained in the two end hosts, and no state whatsoever in the network.

The basic mechanisms for Multipath TCP (MTCP) are also similar to that of SHIM6, with an initial capability exchange to confirm that both parties support the mechanism, allowing the parties to then open up additional paths, or channels. But at this point the functionality diverges. In SHIM6 these alternate paths are provisioned as backup paths in the event of failure of the primary path, while in the case of MPTCP these additional paths can be used immediately to spread the load of the communication across these paths, if so desired by the application.

One of the most critical assumptions of MPTCP was drawn from SHIM6, in that the existence of multiple addresses in a host is sufficient to indicate the existence of multiple diverse paths within the network. Whether this is in fact the case of not is perhaps not that critical, in that even in the case where the addresses are on the same path from end-to-end the end result is roughly equivalent to running multiple parallel sessions of TCP.

The basic approach to Multipath TCP is the division of the application's single outbound flow into multiple sub-flows, each operating its own end-to-end TCP session, and the re-joining of multiple input sub flows into a single flow to present to the remote counterpart application. This is shown in Figure 1.

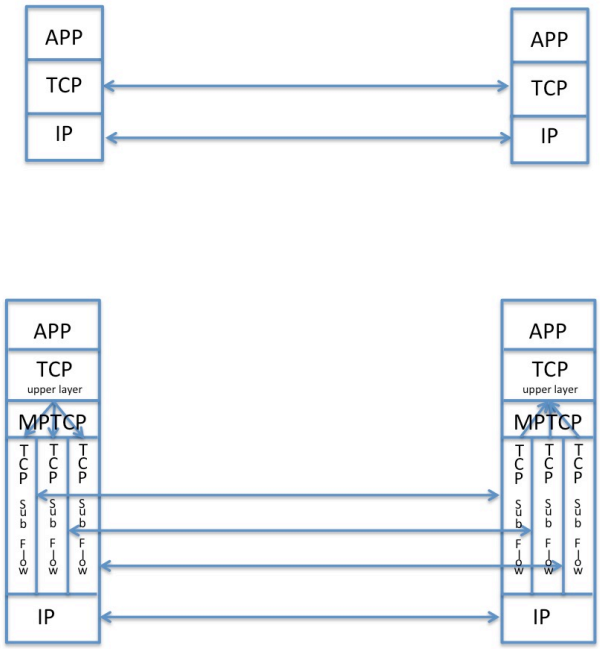

Figure 1: Comparison of Standard TCP and MPTCP Protocol Stacks

This is essentially a "shim" inserted in the TCP module. To the upper level application MPTCP can operate in a manner that is entirely consistent with TCP, so that the opening up of subflows and the manner in which data is assigned to particular subflows is intentionally opaque to the upper level

application. The envisaged API allows the application to add and remove addresses from the local multipath pool but the remainder of the operation of the MPTCP shim is not envisaged to be directly managed by the application. MPTCP also leaves the lower level components of TCP essentially untouched, in so far as each MPCTP subflow is a conventional TCP flow. On the data sender's side the MPTCP shim essentially splits the received stream from the application into blocks and directs individual blocks into separate TCP subflows. On the receiver's side the MPTCP shim assembles the blocks from each TCP subflow and reassembles the original data stream to pass to the local application.

Operation of MPTCP

TCP has the ability to include 40 bytes of TCP options in the TCP header, indicated by the Data Offset value. If the Data Offset value is greater than 5, then the space between the final 32 bit word of the TCP header (checksum and Urgent Pointer) ands the first octet of the data can be used for options. MPTCP uses the Option Kind value of 30 to denote MPTCP options. All MPTCP signaling is contained in this TCP header options field.

The MPTCP operation starts with the initiating host passing a MP_CAPABLE capability message in the MPTCP options field to the remote host as part of the initial TCP SYN message when opening the TCP session. The SYN+ACK response contains a MP_CAPABLE flag in its MPTCP options field of the SYN+ACK response if the other end is also MPTCP capable. The combined TCP and MTCP handshake concludes with the ACK and MP_CAPABLE flag, confirming that both ends now have each other's MPTCP session data. This capability negotiation exchanges 64 bit keys for the session, and each party generates a 32 bit hash of the session keys which are subsequently used as a shared secret between the two hosts for this particular session to identify subsequent subjoin connection attempts.

Further TCP subflows can be added to the MPTCP session by an a conventional TCP SYN exchange with the MPTCP option included. In this case the exchange contains the MP_JOIN values in the MPTCP options field. The values in the MP_JOIN exchange includes the hash of the original receiver's session key and includes the token value from the initial session, so that both ends can associated the new TCP session with the existing session, as well as a random value intended to prevent replay attacks. The MP_JOIN option also includes the sender's address index value to allow both ends of the conversation to reference a particular address even when NATs on the path perform address transforms. MPTCP allows these MP_JOINs to be established on any port number, and by either end of the connection This means that while a MPTCP web session may start using a port 80 service on the server, but subsequent subflows may be established on any port pair and it is not necessary for the server to have a LISTEN open on the new port. The MPTCP session token allows the 5-tuple of the new subflow (Protocol number, source and destination addresses, source and destination port numbers) to be associated with the originally established MPTCP flow. Two hosts can also inform each other of new local addresses without opening a new session by sending ADD_ADDR messages, and remove them with the complementary REMOVE_ADDR message.

Individual subflows use conventional TCP signaling. However, MPTCP adds a Data Sequence Signal (DSS) to the connection that describes the overall state of the data flow across the aggregate of all of the TCP sub flows that are part of this MPTCP session. The sender sequence numbers include the overall data sequence number and the subflow sequence number that is used for the mapping of this data segment into a particular subflow. The DSS Data ACK sequence number is the aggregate acknowledgement of the highest in-order data received by the receiver. MPTCP does not use SACK, as this is left to the individual subflows.

To prevent data loss causing blockage on an individual subflow, a sender can retransmit data on additional subflows. Each subflow is using a conventional TCP sequencing algorithm, so an unreliable connection will cause that subflow to stall. In this case MPTCP can use a different subflow to resend the data, and if the stalled condition is persistent it can reset the stalled subflow with a TCP RST within the context of the subflow.

Individual subflows are stopped by a conventional TCP exchange of FIN messages, or through the TCP RST message. The shutting down of the MP-TCP session is indicated by a data FIN message which is part of the data sequencing signaling within the MPTCP option space.

Congestion control appears still to be an open issue for MPTCP. An experimental approach is to couple the congestion windows of each of the subflows, increasing the sum of the total window sizes at a linear rate per RTT interval, and applying the greatest increase to the subflows with the largest existing window. In this way the aggregate flow is no worse than a single TCP session on the best available path, and the individual subflows take up a fair share of each of the paths it uses. Other approaches are being considered that may reduce the level of coupling of the individual subflows.

MPTCP and Middleware

Today's Internet is not the Internet of old. Today's Internet is replete with various forms of middleware that includes NATs, load balancers, traffic shapers, proxies, filters and firewalls. The implication of this is that any deviation from the most basic forms of use of IP will run into various issues with various forms of middleware.

For MPTCP the most obvious problem is that of middleware that strips out unknown TCP options.

However, more insidious issues come with the ADD_ADDR messages and NATs on the path. Sending IP addresses within the data payload of a NATTed connection is always a failure-prone option, and MPTCP is no exception here. MPTCP contains no inbuilt NAT detection functions, and no way to determine the directionality of the NAT. A host can communicate to the remote end it's own IP address of additional available addresses, but if there is a NAT translating the local host outbound connections then the actual address will be unavailable for use until the host actually starts a TCP session using this local address as the source.

A simple approach that is effective where there are NATs in place is to leave the role of initiation of new Subflows to the host that started the connection in the first place. In a Client / Server environment this would imply that the role of setting up new Subflows is one that is best left to the client in such cases. However, no such constraints exist when there are no NATs, and in that case either end can initiate new subflows, and the ADD_ADDR messages can keep the other end informed about potential new parallel paths between the two hosts. Logically it makes little sense for MPTCP itself to define a NAT-sensing probe behavior, but it makes a lot of sense for the application using MPTCP to undertake such a test.

The Implications of MPTCP

MPTCP admits considerable flexibility in the way an application can operate when there is a richness of connection options available.

All TCP subflows carry the MPTCP option, so that the MPTCP shared state is shared across all active TCP subflows. No single subflow is the "master" in the MCTCP sense. Subflows can be created when interfaces come up, and removed when they go down. Subflows are also IP protocol agnostic: they can use a collection of IPv4 and IPv6 connections simultaneously. Subflows can be used to load share across multiple network paths, or operate in a primary/backup configuration depending on the application and the flexibility offered in the API in particular implementations of MPTCP.

When applied to mobile devices this behavior can lead to unexpected results. I always assumed that my device was incapable of "active handoff". Any connections that were initiated across the cellular radio interface had to stay on that interface, and any connections established over the WiFi interface would also stay on that WiFi network. I always understood that active sessions could not be handed off to a different network. While it was never an explicitly documented feature, or if it was I 've never seen it, I

had also assumed that when my mobile device was in an area with an active WiFi connection, then the WiFi would take precedence over its 4G connection for all new connections. This assumption matched the factor of typical data tariffs, where the marginal cost of data over 4G is typically somewhere between 10 and 1,000 times higher than the marginal cost of the same data volume over the WiFi connection. But if applications use MPTCP instead of TCP then how will they balance their network use across the various networks? The way MPTCP is defined it appears that is applications simply open subflows on all available local interfaces then the fastest network will take on the greatest volume of traffic, rather than the cheapest.

But, as usual, it can always get more complicated. What if the WiFi network is a corporate service, with NATs, split horizon VPNs and various secure servers? If my device starts to perform MPTCP in such contexts, then to what extent are the properties of my WiFi connection preserved in the cellular data connection? Have I exposed new vulnerabilities through doing this? How can a virtual interface, such as a VPN, inform a MPTCP-aware application that other interfaces are not in the same security domain as the VPN interface?

However it does appear that MPTCP has a role to play in the area of seamless WiFi handoff. With MPTCP is it possible to a mobile handset to enter a WiFi serviced area and include a WiFi subflow into the existing data transfer without stopping and restarting the data flow. The application may even shutdown the cellular radio subflow once the WiFi subflow is active. This functionality is under the control of the application using MPTCP, rather than being under the control of the host operating system of the carrier.

Going Up the Stack

Of course it does not stop at the transport layer and with the use of MPTCP. Customised applications can do this themselves.

For example, the "mosh" application is an example of a serial form of address agility, where the session state is a shared secret, and the server will accept a reconnection from any client's IP address, as long as the client can demonstrate its knowledge of the shared secret.

Extending the TCP data transfer model to enlist multiple active TCP sessions at the application level in a load balancing configuration is also possible, in a manner not all that different from MPTCP.

Of course one could take this further and rather than use multiple TCP sessions between the same two endpoints you could instead share the same server's data across multiple endpoints, and use multiple TCP sessions to these multiple servers. At this point you have something that looks remarkably like the peer-to-peer data distribution architecture.

Another approach is to format the data stream into "messages", and permit multiple messages to be sent across diverse paths between the two communicating systems. This approach, SCTP, is similar to MPTCP in that it can take advantage of multiple addresses to support multiple paths. It combines the message transaction qualities of UDP with the reliable in-sequenced transport services of TCP. The problem of course in today's network is that because it is neither TCP nor UDP many forms of middleware, including NATs, are often hostile to SCTP and drop SCTP packets. One more cost of the escalation of middleware in today's Internet. These days innovation in protocol models is limited by the rather narrow rules applied by network middleware, and the approximate rule of thumb in today's Internet is that its TCP, UDP or middleware fodder!

It has been observed a number of times that the abstraction of a network protocol stack is somewhat arbitrary, and its possible to address exactly the same set of requirements at many different levels in the reference stack. In the work on multi-path support in the Internet we've seen approaches that exploit parallel data streams at the data link layer, the IP layer, within routing, in the transport layer and in the application layer. Each have their respective strengths and weaknesses. But what worries me is what

happens if you inadvertently encounter a situation where you have all of these approaches active at the same time? Is the outcome one of amazing efficiency, or paralysing complexity?

Further Reading

RFC5533 "Shim6: Level 3 Multihoming Shim Protocol for IPv6", E. Nordmark, M. Bagnulo, June 2009.

RFC 6182 "Architectural Guidelines for Multipath TCP Development", A. Ford et. al., March 2011.

RFC 6824 "TCP Extensions for Multipath Operation with Multiple Addresses", A. Ford et.al., January 2013.

Bit Torrent: https://wiki.theory.org/BitTorrentSpecification

Author

Geoff Huston B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. He has been closely involved with the development of the Internet for many years, particularly within Australia, where he was responsible for building the Internet within the Australian academic and research sector in the early 1990's. He is author of a number of Internet-related books, and was a member of the Internet Architecture Board from 1999 until 2005, and served on the Board of Trustees of the Internet Society from 1992 until 2001. He has worked as a an Internet researcher, as a ISP systems architect and a network operator at various times.

www.potaroo.net

Disclaimer