iproxy: Running TCP services over UDP

Horms (Simon Horman)
horms@verge.net.au

January 2002

http://verge.net.au/linux/iproxy/

I've been to cities that never close down,
From New York to Rio and old London town,
But no matter how far or how wide I roam,
I still call Australia home.
Peter Allen — I Still Call Australia Home

Presented

linux.conf.au
6th – 9th February 2002
University of Queensland
Brisbane, Queensland, Australia

Thanks

Special thanks goes to: Tony Guntharp whose couch in San Francisco was party to much of the work on this project, along with significant cheese consumption. Ted T'so and Walt Drummond for being obsessed by Multicast. And last but not least, Andrew Tridgell for conceiving iproxy and inspiring me to code lots.

Abstract

iproxy comprises of a client-side proxy and a server-side proxy that allows arbitrary TCP/IP services to run over Broadcast, Multicast or Unicast UDP. It was originally conceived as a method to configure servers that had not been given an IP address on the LAN using an web-based interface.

This paper will focus the implementation issues in sending and receiving UDP traffic when nothing is known about the network the machine is attached to. It will illustrate how to create simple Multicast, Broadcast and Unicast UDP clients and servers. It will also discuss the problems and solutions realised when trying to carry TCP connections over UDP.

Contents

1 Introduction **1**

2 Implementation **1**

2.1 Encapsulation 2

2.2 Server Discovery 2

2.3 Opening a Connection 3

2.4 Connection Tracking 4

2.5 Closing a Connection 4

2.6 Acknowledgement 5

2.7 Timeouts 5

2.8 Keep Alive 5

2.9 Multiple Interfaces 6

3 Variations on a Theme **6**

3.1 Broadcast 7

3.2 Multicast 7

3.3 Unicast 9

4 Applications **9**

5 Conclusion **10**

1 Introduction

While working on a doomed NAS[1] project, it was thought that it would be nice to be able to configure the NAS boxen without having to configure their network settings using a console.

Having the NAS box set up to use DHCP[15] is an obvious and desirable solution. However, we were after a solution for networks that did not have a DHCP infrastructure in place. The idea of communicating using broadcast UDP was raised as it allows communications with nodes on the same subnet, regardless of their IP address. Central to the idea was that we should be able to use existing administrative interfaces provided via HTTP[1][8], HTTPS[16], Telnet[10] and SSH[19]. *iproxy* was born and a journey into tunnelling TCP[12] over UDP[13] began.

2 Implementation

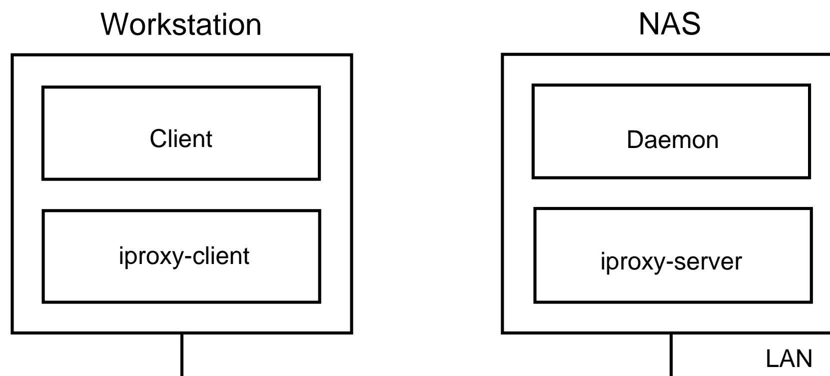

Figure 1: Basic *iproxy* Architecture

The iproxy architecture consists of two proxy servers, *iproxy-client* and *iproxy-server*. These daemons convert TCP connections into UDP datagrams and vice versa.

iproxy-client runs on the user's workstation. It listens for incoming TCP connections. This may be from a web browser, SSH or Telnet client, or a client for any TCP protocol. The iproxy-client then encapsulates this connection as UDP datagrams. *iproxy-server* runs on the NAS boxen on the network and listens for UDP datagrams sent by a iproxy-client. It then opens a connection to the local daemon, whether it be HTTP, SSH, Telnet

[1]NAS: Network Attached Storage

or another TCP protocol and relays the encapsulated information from the user's client. Replies from the daemon follow the reverse path.

2.1 Encapsulation

Checksum	Version	Flag	Client Id	Server Id	Length	Offset

Figure 2: *iproxy* Packet Header

To enable a TCP connection to be tunnelled in UDP datagrams some identification of the packet is required. To achieve this iproxy packets have a header followed by up to 1048 bytes of data from the TCP connection. To maximise portability, each field in the header is a 32-bit unisigned integer in network byte order. The fields are as follows:

- *Checksum:* Rolling 8-bit checksum of the header fields other than the checksum and the data.
- *Version:* iproxy protocol version. Used to identify incompatible versions of the protocol in the future. Currently 1.
- *Flag:* Flags to specify special packets, such as Acknowledgements, Keep Alive and Finish Packets.
- *Client Id:* Unique identifier of the client.
- *Server Id:* Unique identifier of the server.
- *Length:* Total length of the packet including the header and data.
- *Offset:* Offset. Incremented by the length of the data for each successive packet. Used in Lieu of a sequence number.

2.2 Server Discovery

Given that the motivation for iproxy is to enable configuration of hosts running iproxy-server without prior knowledge of their setup, auto discovery is an important feature of iproxy. This is implemented by iproxy-client sending a discovery packet with its own Client Id and IPROXY_LIST_SID as the Server Id. All iproxy-servers that receive this discovery packet should reply with using the Client Id in the packet and their own Server Id.

At this stage this is done manually by running iproxy-client with the `-l` command-line option which causes it to issue a discovery packet, display any packets received from iproxy-servers.

```
# iproxy-client -l
[12841] Hello from 66!
[12841] Hello from 67!
[12841] Tired of waiting for Hello replies
```

Here we see that there are two iproxy-servers running on the network. They have Server Ids 66 and 67. 12841 is the process Id of the iproxy-client.

We can run an instance of iproxy-client to communicate with the iproxy-server with Server Id 67 using the `-s` command-line option.

```
# iproxy-client -s 67
```

2.3 Opening a Connection

To open a connection a client connects to iproxy-client which is listening on a known TCP port. On receipt of a new connection iproxy-client sends a UDP datagram with its Client Id, the Server Id that it has been configured to communicate with and no data.

When the iproxy-server receives this packet it will open a connection to the local daemon. If the daemon returns any data at this stage it is returned in the reply UDP packet, else an empty reply is sent.

For Example:

On the Workstation, configure iproxy-client to communicate with iproxy-server with Server Id 67, and listen for incoming TCP connections on port 7777. `-v` runs with verbose debugging.

```
iproxy-client -s 67 -p 7777 -v
```

On the Server configure iproxy-server to have Server Id 67 and proxy incoming UDP connections to the apache[2] daemon running locally on port 80.

[2] Apache HTTP Server: http://www.apache.org

```
iproxy-server -s 67 -d 80 -v
```

If a connection is now opened to port 7777 on the workstation a UDP packet will be sent from iproxy-client to iproxy-server. On the Server a connection to local port 80 will be opened and an ACK packet will be send back to iproxy-client.

In the case of an apache HTTP daemon, the daemon does not send any data at this stage so nothing further happens until the client that opened the connection to iproxy-client issues a request. In the case of a service where the server does issue some data upon connect, such a Telnet, this would be sent to iproxy-client by iproxy-server and an ACK would be returned by iproxy-client.

2.4 Connection Tracking

One of the key differences between TCP and UDP is that the former is connection oriented while the latter is not. This means that in order to tunnell TCP services over UDP there must be a way to keep track of connections. In particular, as iproxy-server may be receiving UDP datagrams from the same iproxy-server for different connections at the same time it must be able to differentiate which connection each datagram belongs to.

To do this the following information is used for each packet: source port, source address and client id. This is in addition to the destination port which is matched by UDP and discarding of all packets not addressed to the iproxy-server's server id.

A table of active connections is kept on the iproxy-server. If a packet matches one of these connections then it is considered part of that connection, else is it considered to be the first packet in a new connection.

2.5 Closing a Connection

A connection may be closed by sending a packet with an empty data section and the flag entry in the iproxy header set to IPROXY_FIN_FLAG, a FIN packet.

On receipt of a FIN packet a FIN packet is sent back. This is an acknowledgement that the FIN was received. iproxy-client will then exit the process that was forked to handle this connection. iproxy-server on the other hand does not fork and will mark the connection as closed and and ignore any other

packets for this connection. After the expire timeout, EXPIRE_TIMEOUT seconds the connection will be purged, and thus any subsequent packets will be assumed to be packets for a new connection, which will most likely time out.

Connections that are marked as closed may be purged before the expire timeout if the connection table, which is fixed in size, runs out of free connections and a packet for a new connection is received. If this occurs, the closed connection closest to its expire timeout will be used.

2.6 Acknowledgement

For each UDP datagram that is received an acknowledgement packet is returned. This is a packet with an empty data section and the flag entry in the iproxy header set to IPROXY_ACK_FLAG. Until a packet is Acknowledged, no additional packets will be sent, that is no additional data will be read from the associated TCP connections.

Though this scheme is simple and inefficient it all but eliminates the possibility of out of order packets. Thus if any out of order packets are received they are simply dropped.

2.7 Timeouts

The expire timeout is also used to close idle connections. Each time a packet is received by a connection its expire timeout is reset. If the expire timeout passes without the receipt of another packet a FIN packet is sent. In the case of iproxy-client the forked process will exit. In the case of iproxy-server the packet will be marked as closed.

There is also a resend timeout, set to RESEND_TIMEOUT seconds each time a packet is received. The purpose of this is to resend packets when if this timeout expires before an Acknowledgement packet is received.

It is of note that the expire timeout should generally be longer than the resend timeout, else a resend will never occur as the packet will expire and the connection will be closed first.

2.8 Keep Alive

In order to prevent protocols that may have be idle for a time, such as Telnet or SSH, from causing the connection to timeout keep alive packets are sent.

A keep alive packet is a packet with an empty data section and the flag entry in the iproxy header set to IPROXY_KEEPALIVE_FLAG. These are sent by iproxy-client if a connection is idle for KEEPALIVE_TIME seconds. This should be less than EXPIRE_TIMEOUT, else iproxy-server will expire connections before a keep alive is sent.

On receipt of a keep alive packet, iproxy-server will reset the resend and expire timeouts for the connection. Keep Alive packets are ignored if the connection is awaiting an Acknowledgement.

2.9 Multiple Interfaces

Given that the motivation for iproxy is to communicate with machines whose network configuration is not known, it is important that packets are sent out all available interfaces. Otherwise the packets may be sent out an interface which is not connected to the network, while the interface that has is connected lies idle. Again, this is only an issue because nothing is known about the network configuration.

To achieve this iproxy has code to find the base IP address of each active interface on the host. By binding a socket to each of these addresses and sending each packet to each of these sockets, packets are sent out each and every interface.

As a result of this is is possible to receive duplicate packets. The connection tracking code handles this by rejecting packets who match an existing connection, other than that the source IP address differs. In addition, checks on the offset field in the iproxy header guard against out of order and duplicate packets.

3 Variations on a Theme

iproxy was originally conceived to run over broadcast UDP. The idea being that this can be used to communicate with hosts whose IP address is unknown. However, there is the restriction that the hosts be on the same physical subnet. A suggestion was made that using Multicast would enable communication with nodes on different subnets. For the sake of completeness Unicast support was also added. The line was drawn at Anycast[11].

The main difference between the Broadcast, Multicast and Unicast implementation lies in the handling of sockets. The code used below is for IPv4[14]. Analogous code for IPv6[5] may be written but is not currently

part of iproxy. This section could not have been written without the assistance of UNIX Network Programming[17].

3.1 Broadcast

When using broadcast UDP, packets sent onto the network are received by each host on the same physical subnet. As mentioned previously it is desirable for iproxy to send packets out each and every interface, thus multiple sockets are opened.

A socket is opened to listen for incoming datagrams, this is bound to `0.0.0.0` for the configured port, and thus will accept packets for any local or broadcast address. A socket is also opened to send outgoing datagrams on each interface. To do this a socket is opened for each interface and bound to the address of that interface. Each of these outgoing sockets has the SO_BROADCAST socket option set to enable them to send broadcast datagrams.

```
int s;
int one=1;
/* s is an open socket bound to a local interface ... */
setsockopt(s, SOL_SOCKET, SO_BROADCAST,
        (char *)&one, sizeof(one));
```

Figure 3: Setting the SO_BROADCAST Option for a Socket

3.2 Multicast

Multicast, like broadcast allows packets to be received by multiple hosts, and thus is useful in accessing hosts of unknown IP address setup. As long as iproxy-server is a member of a known multicast group it can be addressed without any additional information of its current network setup.

The key advantage of multicast over broadcast for iproxy is that multicast traffic may be routed between physical subnets, providing that a multicast-routing infrastructure is in place. This is in contrast to broadcast traffic which is not routed between subnets[3].

The key difference programatically between broadcast and multicast is that the SO_BROADCAST socket option is not set and that the multicast time

[3] UDP broadcast traffic is not routed between subnets, with the notable exception of the dubious `ip helper-address` command[4] in Cisco's IOS.

to live (TTL) must be set and a multicast group must be joined.

The Multicast TTL is set using the the IP_MULTICAST_TTL socket option. This sets the number of hops that a multicast packet will survive. At each routing hop the TTL is decremented and when it reaches zero the packet is dropped. The default is one, meaning that the packet will be dropped at the first router it encounters, the same behaviour as broadcast.

```
int s;
int mcast_ttl=1; /* TTL of multicast packets in hops.
                  * 1 is the default */
/* s is an open socket bound to a local interface ... */
setsockopt(iface_fd[i], IPPROTO_IP, IP_MULTICAST_TTL,
        (char *)&mcast_ttl, sizeof(mcast_ttl));
```

Figure 4: Setting the Multicast TTL for a Socket

To receive multicast packets a socket must be the member of a multicast group. A multicast group is simply a Class D address[4]. A socket joins a multicast group using the IP_ADD_MEMBERSHIP socket option. This causes an IGMP[5] message to be sent, announcing to local multicast routers that the node has joined the multicast group in question.

```
int s;
struct ip_mreq mreq;
struct in_addr addr;
struct iface_struct iface;
/* s is an open socket bound to a local interface
 * addr is the IP address of the multicast group
 * iface is a portable structure provided by iproxy,
 *       borrowed from samba containing the interface's
 *       address */
mreq.imr_multiaddr.s_addr=mcast_addr_bin.s_addr;
mreq.imr_interface.s_addr=iface.ip.s_addr;
setsockopt(fd_out, IPPROTO_IP, IP_ADD_MEMBERSHIP,
        (void *) &mreq, sizeof(mreq));
```

Figure 5: Joining a Multicast Group

[4]Class D: 224.0.0.0/4, addresses reserved for multicast use.

[5]IGMP: Internet Group Management Protocol. This is the control protocol used for multicast routing. It is somewhat analogous to the way that ICMP is used for Unicast routing

3.3 Unicast

Unicast support was added to iproxy for the sake of completeness. The key advantage of unicast is that it is universally routed across the Internet, thus allowing TCP over UDP tunnell to run between two known endpoints. However, unicast does require prior knowledge of the network configuration of both endpoints and, hence, is of no use for the configuration task for which iproxy was originally conceived.

Programatically, unicast was much simpler to implement as only a single socket needs to be opened which is used for incoming and outgoing datagrams. No special socket options are required for unicast.

4 Applications

One of the most frequent reactions to tunnelling TCP over UDP is, "Why?". As outlined earlier the original motivation for iproxy was to provide a method to configure network devices without the need for an existing DHCP infrastructure or any sort of console or key-pad access. Iproxy is able to do this, though sadly the project it was originally intended for has since been disbanded. Given iproxy's unicast support two esoteric applications spring to mind owing to UDP traffic being largely ignored by network administrators.

It is of note that many network administrators deploy state-less packet filters to protect their networks from unwanted nasties coming in and in some cases to restrict the external access for users of the network. It is beyond the scope of this paper to discuss the relative merits of state-less packet filtering, state-full packet filtering and proxy-based firewalls. However, it is of note that if UDP traffic is allowed to pass through a packet-filter without inspection then iproxy may be used to tunnell arbitrary TCP services through the packet-filter, thus circumventing the local security policy. While existing solutions exist to do this by tunnelling traffic via DNS[9], HTTP[2], HTTPS[3], ICMP[6] and even SMTP[7], iproxy potentially offers vast performance advantages over their solutions.

Continuing the theme of unnoticed UDP traffic it is of particular note that Internet Packet Quota (IPQ)[18] does not attempt to count UDP traffic. This partially an implementation issue relating to identify the user that a particular datagram belongs and partly because UDP traffic was not seen as something of particular interest in terms of percentage of network utilisation when IPQ was being developed A creative user with iproxy in hand could easily change this situation.

5 Conclusion

iproxy enables arbitrary TCP services to be tunnelled over UDP. The implementation currently supports IPv4 and is able to use Multicast Unicast and Broadcast. This may be used to communicate with hosts whose network configuration is unknown, in particular to configure network devices. It may also be used to get TCP traffic past unsuspecting network administrators.

References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk. Rfc1945: Hypertext transfer protocol – http/1.0. Internet Engineering Task Force: http://www.ietf.org/, May 1996.

[2] Lars Brinkoff. httptunnel. http://www.nocrew.org/software/httptunnel.html, 2001.

[3] Chris Chiappa. ssh-https-tunnel. http://www.snurgle.org/ griffon/ssh-https-tunnel, March 2000.

[4] Cisco Systems, Inc. *Cisco IOS IP Configuration Guide*, cisco ios 12.2 edition, 2001. http://www.cisco.com/.

[5] S. Deering and R. Hinden. Rfc1883: Internet protocol, version 6 (ipv6) specification. Internet Engineering Task Force: http://www.ietf.org/, December 1995.

[6] Magnus Lundström et al. icmptunnel. http://www.detached.net/icmptunnel/, 1999.

[7] Magnus Lundström et al. mailtunnel. http://www.detached.net/mailtunnel/, 1999.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Rfc2068: Hypertext transfer protocol – http/1.1. Internet Engineering Task Force: http://www.ietf.org/, January 1997.

[9] Florian Heinz and Julien Oster. Ip tunneling through nameservers, September 2000. http://www.slashdot.org/.

[10] A. McKenzie. Rfc405: Telnet protocol specification. Internet Engineering Task Force: http://www.ietf.org/, May 1973.

[11] C. Partridge, T. Mendez, and W. Milliken. Rfc1546: Host anycasting service. Internet Engineering Task Force: http://www.ietf.org/, November 1993.

[12] J. Postel. Rfc761: Dod standard: Transmission control protocol. Internet Engineering Task Force: http://www.ietf.org/, January 1980.

[13] J. Postel. Rfc768: User datagram protocol. Internet Engineering Task Force: http://www.ietf.org/, August 1980.

[14] J. Postel. Rfc791: Darpa internet program: Protocol specification. Internet Engineering Task Force: http://www.ietf.org/, September 1981.

[15] R.Droms. Rfc2131: Dynamic host configuration protocol. Internet Engineering Task Force: http://www.ietf.org/, March 1997.

[16] E. Rescorla. Rfc2818: Http over tls. Internet Engineering Task Force: http://www.ietf.org/, May 2000.

[17] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall PTR, 2 edition, 1997.

[18] University Of New South Wales. Ipq: Internet protocol quota. http://www.cse.unsw.edu.au/ ipq/doc/, 2000.

[19] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. Internet-draft: Ssh connection protocol. Internet Engineering Task Force: http://www.ietf.org/, November 2001.