How to replicate the fire: HA for netfilter based firewalls

Harald Welte
Netfilter Core Team + Astaro AG
laforge@gnumonks.org || laforge@astaro.com
http://www.gnumonks.org/

Abstract

With traditional, stateless firewalling (such as ipfwadm, ipchains) there is no need for special HA support in the firewalling subsystem. As long as all packet filtering rules and routing table entries are configured in exactly the same way, one can use any available tool for IP-Address takeover to accomplish the goal of failing over from one node to the other.

With Linux 2.4.x netfilter/iptables, the Linux firewalling code moves beyond traditional packet filtering. Netfilter provides a modular connection tracking susbsystem which can be employed for stateful firewalling. The connection tracking subsystem gathers information about the state of all current network flows (connections). Packet filtering decisions and NAT information is associated with this state information.

In a high availability scenario, this connection tracking state needs to be replicated from the currently active firewall node to all standby slave firewall nodes. Only when all connection tracking state is replicated, the slave node will have all necessarry state information at the time a failover event occurs.

The netfilter/iptables does currently not have any functionality for replicating connection tracking state accross multiple nodes. However, the author of this presentation, Harald Welte, has started a project for connection tracking state replication with netfilter/iptables.

The presentation will cover the architectural design and implementation of the connection tracking failover sytem. With respect to the date of the conference, it is to be expected that the project is still a work-in-progress at that time.

1 Failover of stateless firewalls

There are no special precautions when installing a highly available stateless packet filter. Since there is no state kept, all information needed for filtering is the ruleset and the individual, seperate packets.

Building a set of highly available stateless packet filters can thus be achieved by using any traditional means of IP-address takeover, such as Hartbeat or VRRPd.

The only remaining issue is to make sure the firewalling ruleset is exactly the same on both machines. This should be ensured by the firewall administrator every time he updates the ruleset.

If this is not applicable, because a very dynamic ruleset is employed, one can build a very easy solution using iptables-supplied tools iptables-save and iptables-restore. The output of iptables-save can be piped over ssh to iptables-restore on a different host.

Limitations

- no state tracking
- not possible in combination with NAT
- no counter consistency of per-rule packet/byte counters

2 Failover of stateful firewalls

Modern firewalls implement state tracking (aka connection tracking) in order to keep some state about the currently active sessions. The amount of per-connection state kept at the firewall depends on the particular implementation.

As soon as **any** state is kept at the packet filter, this state information needs to be replicated to the slave/backup nodes within the failover setup.

In Linux 2.4.x, all relevant state is kept within the *connection tracking subsystem*. In order to understand how this state could possibly be replicated, we need to understand the architecture of this conntrack subsystem.

2.1 Architecture of the Linux Connection Tracking Subsystem

Connection tracking within Linux is implemented as a netfilter module, called ip_conntrack.o.

Before describing the connection tracking subsystem, we need to describe a couple of definitions and primitives used throughout the conntrack code.

A connection is represented within the conntrack subsystem using *struct ip_conntrack*, also called *connection tracking entry*.

Connection tracking is utilizing *conntrack tuples*, which are tuples consisting out of (srcip, srcport, dstip, dstport, l4prot). A connection is uniquely identified by two tuples: The tuple in the original direction (IP_CT_DIR_ORIGINAL) and the tuple for the reply direction (IP_CT_DIR_REPLY).

Connection tracking itself does not drop packets[1] or impose any policy. It just associates every packet with a connection tracking entry, which in turn has a particular state. All other kernel code can use this state information[2].

2.1.1 Integration of conntrack with netfilter

If the ip_conntrack.o module is registered
with netfilter, it attaches to the
NF_IP_PRE_ROUTING,
NF_IP_POST_ROUTING,
NF_IP_LOCAL_IN and
NF_IP_LOCAL_OUT hooks.

Because forwarded packets are the most common case on firewalls, I will only describe how connection tracking works for forwarded packets. The two relevant hooks for forwarded packets are NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING.

Every time a packet arrives at the NF_IP_PRE_ROUTING hook, connection tracking creates a conntrack tuple from the packet. It then compares this tuple to the original and reply tuples of all already-seen

[1] well, in some rare cases in combination with NAT it needs to drop. But don't tell anyone, this is secret.

[2] state information is internally represented via the *struct sk_buff.nfct* structure member of a packet.

connections[3] to find out if this just-arrived packet belongs to any existing connection. If there is no match, a new conntrack table entry (struct ip_conntrack) is created.

Let's assume the case where we have already existing connections but are starting from scratch.

The first packet comes in, we derive the tuple from the packet headers, look up the conntrack hash table, don't find any matching entry. As a result, we create a new struct ip_conntrack. This struct ip_conntrack is filled with all necessarry data, like the original and reply tuple of the connection. How do we know the reply tuple? By inverting the source and destination parts of the original tuple.[4] Please note that this new struct ip_conntrack is **not** yet placed into the conntrack hash table.

The packet is now passed on to other callback functions which have registered with a lower priority at NF_IP_PRE_ROUTING. It then continues traversal of the network stack as usual, including all respective netfilter hooks.

If the packet survives (i.e. is not dropped by the routing code, network stack, firewall ruleset, ...), it re-appears at NF_IP_POST_ROUTING. In this case, we can now safely assume that this packet will be sent off on the outgoing interface, and thus put the connection tracking entry which we created at NF_IP_PRE_ROUTING into the conntrack hash table. This process is called *confirming the conntrack.*

The connection tracking code itself is not monolithic, but consists out of a couple of seperate modules[5]. Besides the conntrack core, there are two important kind of modules: Protocol helpers and application helpers.

Protocol helpers implement the layer-4-protocol specific parts. They currently exist for TCP, UDP and ICMP (an experimental helper for GRE exists).

2.1.2 TCP connection tracking

As TCP is a connection oriented protocol, it is not very difficult to imagine how conntection tracking for this protocol could work. There are well-defined state transitions possible, and conntrack can decide which state transitions are valid within the TCP specification. In reality it's not all that easy, since we cannot assume that all packets that pass the packet filter actually arrive at the receiving end, ...

It is noteworthy that the standard connection tracking code does **not** do TCP sequence number and window tracking. A well-maintained patch to add this feature exists almost as long as connection tracking itself. It will be integrated with the 2.5.x kernel. The problem with window tracking is its bad interaction with connection pickup. The TCP conntrack code is able to pick up already existing connections, e.g. in case your firewall was rebooted. However, connection pickup is conflicting with TCP window tracking: The TCP window scaling option is only transferred at connection setup time, and we don't know about it in case of pickup ...

2.1.3 ICMP tracking

ICMP is not really a connection oriented protocol. So how is it possible to do connection tracking for ICMP?

[3] Of course this is not implemented as a linear search over all existing connections.

[4] So why do we need two tuples, if they can be derived from each other? Wait until we discuss NAT.

[5] They don't actually have to be seperate kernel modules; e.g. TCP, UDP and ICMP tracking modules are all part of the linux kernel module ip_conntrack.o

The ICMP protocol can be split in two groups of messages

- ICMP error messages, which sort-of belong to a different connection ICMP error messages are associated *RELATED* to a different connection. (ICMP_DEST_UNREACH, ICMP_SOURCE_QUENCH, ICMP_TIME_EXCEEDED, ICMP_PARAMETERPROB, ICMP_REDIRECT).
- ICMP queries, which have a request->reply character. So what the conntrack code does, is let the request have a state of *NEW*, and the reply *ESTABLISHED*. The reply closes the connection immediately. (ICMP_ECHO, ICMP_TIMESTAMP, ICMP_INFO_REQUEST, ICMP_ADDRESS)

2.1.4 UDP connection tracking

UDP is designed as a connectionless datagram protocol. But most common protocols using UDP as layer 4 protocol have bi-directional UDP communication. Imagine a DNS query, where the client sends an UDP frame to port 53 of the nameserver, and the nameserver sends back a DNS reply packet from its UDP port 53 to the client.

Netfilter trats this as a connection. The first packet (the DNS request) is assigned a state of *NEW*, because the packet is expected to create a new 'connection'. The dns servers' reply packet is marked as *ESTABLISHED*.

2.1.5 conntrack application helpers

More complex application protocols involving multiple connections need special support by a so-called "conntrack application helper module". Modules in the stock kernel come for FTP and IRC(DCC). Netfilter CVS currently contains patches for PPTP, H.323, Eggdrop botnet, tftp ald talk. We're still lacking a lot of protocols (e.g. SIP, SMB/CIFS) - but they are unlikely to appear until somebody really needs them and either develops them on his own or funds development.

2.1.6 Integration of connection tracking with iptables

As stated earlier, conntrack doesn't impose any policy on packets. It just determines the relation of a packet to already existing connections. To base packet filtering decision on this sate information, the iptables *state* match can be used. Every packet is within one of the following categories:

- **NEW**: packet would create a new connection, if it survives
- **ESTABLISHED**: packet is part of an already established connection (either direction)
- **RELATED**: packet is in some way related to an already established connection, e.g. ICMP errors or FTP data sessions
- **INVALID**: conntrack is unable to derive conntrack information from this packet. Please note that all multicast or broadcast packets fall in this category.

2.2 Poor man's conntrack failover

When thinking about failover of stateful firewalls, one usually thinks about replication of state. This presumes that the state is gathered at one firewalling node (the currently active node), and replicated to several other passive standby nodes. There is, howeve, a very different approach to replication: concurrent state tracking on all firewalling nodes.

The basic assumption of this approach is: In a setup where all firewalling nodes receive exactly the same traffic, all nodes will deduct the same state information.

The implementability of this approach is totally dependent on fulfillment of this assumption.

- *All packets need to be seen by all nodes*. This is not always true, but can be achieved by using shared media like traditional ethernet (no switches!!) and promiscuous mode on all ethernet interfaces.
- *All nodes need to be able to process all packets*. This cannot be universally guaranteed. Even if the hardware (CPU, RAM, Chipset, NIC's) and software (Linux kernel) are exactly the same, they might behave different, especially under high load. To avoid those effects, the hardware should be able to deal with way more traffic than seen during operation. Also, there should be no userspace processes (like proxes, etc.) running on the firewalling nodes at all. WARNING: Nobody guarantees this behaviour. However, the poor man is usually not interested in scientific proof but in usability in his particular practical setup.

However, even if those conditions are fulfilled, ther are remaining issues:

- *No resynchronization after reboot*. If a node is rebooted (because of a hardware fault, software bug, software update, ..) it will loose all state information until the event of the reboot. This means, the state information of this node after reboot will not contain any old state, gathered before the reboot. The effect depend on the traffic. Generally, it is only assured that state information about all connections initiated after the reboot will be present. If there are short-lived connections (like http), the state information on the just rebooted node will approximate the state information of an older node. Only after all sessions active at the time of reboot have terminated, state information is guaranteed to be resynchronized.
- *Only possible with shared medium*. The practical implication is that no switched ethernet (and thus no full duplex) can be used.

The major advantage of the poor man's approach is implementation simplicity. No state transfer mechanism needs to be developed. Only very little changes to the existing conntrack code would be needed in order to be able to do tracking based on packets received from promiscuous interfaces. The active node would have packet forwarding turned on, the passive nodes off.

I'm not proposing this as a real solution to the failover problem. It's hackish, buggy and likely to break very easily. But considering it can be implemented in very little programming time, it could be an option for very small installations with low reliability criteria.

2.3 Conntrack state replication

The preferred solution to the failover problem is, without any doubt, replication of the connection tracking state.

The proposed conntrack state replication soltution consists out of several parts:

- A connection tracking state replication protocol
- An event interface generating event messages as soon as state information changes on the active node
- An interface for explicit generation of connection tracking table entries on the standby slaves
- Some code (preferrably a kernel thread) running on the active node, receiving state updates by the event interface and generating conntrack state replication protocol messages
- Some code (preferrably a kernel thread) running on the slave node(s), receiving conntrack state replication protocol messages and updating the local conntrack table accordingly

Flow of events in chronological order:

- *on active node, inside the network RX softirq*
  - connection tracking code is analyzing a forwarded packet
  - connection tracking gathers some new state information
  - connection tracking updates local connection tracking database
  - connection tracking sends event message to event API
- *on active node, inside the conntrack-sync kernel thread*
  - conntrack sync daemon receives event through event API
  - conntrack sync daemon aggregates multiple event messages into a state replication protocol message, removing possible redundancy
  - conntrack sync daemon generates state replication protocol message
  - conntrack sync daemon sends state replication protocol message (private network between firewall nodes)
- *on slave node(s), inside network RX softirq*
  - connection tracking code ignores packets coming from the interface attached to the private conntrac sync network
  - state replication protocol messages is appended to socket receive queue of conntrack-sync kernel thread
- *on slave node(s), inside conntrack-sync kernel thread*
  - conntrack sync daemon receives state replication message
  - conntrack sync daemon creates/updates conntrack entry

2.3.1 Connection tracking state replication protocol

In order to be able to replicate the state between two or more firewalls, a state replication protocol is needed. This protocol is used over a private network segment shared by all nodes for state replication. It is designed to work over IP unicast and IP multicast transport. IP unicast will be used for direct point-to-point communication between one active firewall and one standby firewall. IP multicast will be used when the state needs to be replicated to more than one standby firewall.

The principle design criteria of this protocol are:

- **reliable against data loss**, as the underlying UDP layer does only provide checksumming against data corruption, but doesn't employ any means against data loss
- **lightweight**, since generating the state update messages is already a very expensive process for the sender, eating additional CPU, memory and IO bandwith.
- **easy to parse**, to minimize overhead at the receiver(s)

The protocol does not employ any security mechanism like encryption, authentication or reliability against spoofing attacks. It is assumed that the private conntrack sync network is a secure communications channel, not accessible to any malicious 3rd party.

To achieve the reliability against data loss, an easy sequence numbering scheme is used. All protocol messages are prefixed by a seuqence number, determined by the sender. If the slave detects packet loss by discontinuous sequence numbers, it can request the retransmission of the missing packets by stating the missing sequence number(s). Since there is no acknowledgement for sucessfully received packets, the sender has to keep a reasonably-sized backlog of recently-sent packets in order to be able to fulfill retransmission requests.

The different state replication protocol messages types are:

- **NF_CTSRP_NEW**: New conntrack entry has been created (and confirmed[6])
- **NF_CTSRP_UPDATE**: State information of existing conntrack entry has changed
- **NF_CTSRP_EXPIRE**: Existing conntrack entry has been expired

To uniquely identify (and later reference) a conntrack entry, a *conntrack_id* is assigned to every conntrack entry transferred using a NF_CTSRP_NEW message. This conntrack_id must be saved at the receiver(s) together with the conntrack entry, since it is used by the sender for subsequent NF_CTSRP_UPDATE and NF_CTSRP_EXPIRE messages.

The protocol itself does not care about the source of this conntrack_id, but since the current netfilter connection tracking implementation does never change the addres of a conntrack entry, the memory address of the entry can be used, since it comes for free.

2.3.2 Connection tracking state syncronization sender

Maximum care needs to be taken for the implementation of the ctsyncd sender.

The normal workload of the active firewall node is likely to be already very high, so generating and sending the conntrack state replication messages needs to be highly efficient.

- **NF_CTSRP_NEW** will be generated at the NF_IP_POST_ROUTING hook, at the time ip_conntrack_confirm() is called. Delaying this message until conntrack confirmation happens saves us from replicating otherwise unneeded state information.
- **NF_CTSRP_UPDATE** need to be created automagically by the conntrack core. It is not possible to have any failover-specific code within conntrack protocol and/or application helpers. The easiest

[6] See the above description of the conntrack code for what is meant by *confirming* a conntrack entry

way involving the least changes to the conntrack core code is to copy parts of the conntrack entry before calling any helper functions, and then use memcmp() to find out if the helper has changed any information.

- **NF_CTSRP_EXPIRE** can be added very easily to the existing conntrack destroy function.

2.3.3 Connection tracking state syncronization receiver

Impmentation of the receiver is very straightforward.

Apart from dealing with lost CTSRP packets, it just needs to call the respective conntrack add/modify/delete functions offered by the core.

2.3.4 Necessary changes within netfilter conntrack core

To be able to implement the described conntrack state replication mechanism, the following changes to the conntrack core are needed:

- Ability to exclude certain packets from being tracked. This is a long-wanted feature on the TODO list of the netfilter project and will be implemented by having a “prestate” table in combination with a “NOTRACK” target.

- Ability to register callback functions to be called every time a new conntrack entry is created or an existing entry modified.

- Export an API to add externally add, modify and remove conntrack entries. Since the needed ip_conntrack_lock is exported, implementation could even reside outside the conntrack core code.

Since the number of changes is very low, it is very likely that the modifications will go into the mainstream kernel without any big hassle.

Proceedings of the
Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*