Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

`ct_sync`: state replication of `ip_conntrack`

Harald Welte
netfilter core team / Astaro AG / hmw-consulting.de
laforge@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 2.4/2.6 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 necessary state information at the time a failover event occurs.

Due to funding by Astaro AG, the netfilter/iptables project now offers a `ct_sync` kernel module for replicating connection tracking state accross multiple nodes. The presentation will cover the architectural design and implementation of the connection tracking failover sytem.

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, separate 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 Heartbeat 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 and can be optionally managed by some scripts utilizing scp or rsync.

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 iptables stateful NAT

- no counter consistency of per-rule packet/byte counters

2 Failover of stateful firewalls

Modern firewalls implement state tracking (a.k.a. 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 configuration and networking protocols used.

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.

Since 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` (`ip_conntrack.ko` in 2.6.x kernels).

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 of

- source IP address
- source port (or icmp type/code, gre key, ...)
- destination IP address
- destination port
- layer 4 protocol number

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.[k]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 re-

[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 referenced via the `struct sk_buff.nfct` structure member of a packet.

ply tuples of all already-seen 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 of a couple of separate 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 has existed for 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...

[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 separate kernel modules; e.g. TCP, UDP, and ICMP tracking modules are all part of the linux kernel module `ip_conntrack.o`.

2.1.3 ICMP tracking

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

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 treats 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 server's 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, IRC (DCC), TFTP, and Amanda. Netfilter CVS currently contains patches for PPTP, H.323, Eggdrop botnet, mms, DirectX, RTSP, and talk/ntalk. 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 state 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, however, a very different approach to replication: concurrent state tracking on all firewalling nodes.

While this scheme has not been implemented within `ct_sync`, the author still thinks it is worth an explanation in this paper.

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, NICs) 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 proxies, 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, there are remaining issues:

- *No resynchronization after reboot*. If a node is rebooted (because of a hardware fault, software bug, software update, etc.) it will lose 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 effects 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 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*
  - `ip_conntrack` analyzes a forwarded packet
  - `ip_conntrack` gathers some new state information
  - `ip_conntrack` updates conntrack hash table
  - `ip_conntrack` calls event API
  - function registered to event API builds and enqueues message to send ring
- *on active node, inside the conntrack-sync sender kernel thread*
  - `ct_sync_send` aggregates multiple messages into one packet
  - `ct_sync_send` dequeues packet from ring
  - `ct_sync_send` sends packet via in-kernel sockets API
- *on slave node(s), inside network RX softirq*
  - `ip_conntrack` ignores packets coming from the `ct_sync` interface via NOTRACK mechanism
  - UDP stack appends packet to socket receive queue of `ct_sync_recv` kernel thread
- *on slave node(s), inside conntrack-sync receive kernel thread*
  - `ct_sync_recv` thread receives state replication packet
  - `ct_sync_recv` thread parses packet into individual messages
  - `ct_sync_recv` thread creates/updates local `ip_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 principal design criteria of this protocol are:

- **reliable against data loss**, as the underlying UDP layer only provides 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 third party.

To achieve the reliability against data loss, an easy sequence numbering scheme is used. All protocol messages are prefixed by a sequence 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[6] backlog of recently-sent packets in order to be able to fulfill retransmission requests.

[6] *reasonable size* must be large enough for the round-trip time between master and slowest slave.

The different state replication protocol packet types are:

- **`CT_SYNC_PKT_MASTER_ANNOUNCE`**: A new master announces itself. Any still existing master will downgrade itself to slave upon reception of this packet.
- **`CT_SYNC_PKT_SLAVE_INITSYNC`**: A slave requests initial synchronization from the master (after reboot or loss of sync).
- **`CT_SYNC_PKT_SYNC`**: A packet containing synchronization data from master to slaves
- **`CT_SYNC_PKT_NACK`**: A slave indicates packet loss of a particular sequence number

The messages within a `CT_SYNC_PKT_SYNC` packet always refer to a particular *resource* (currently `CT_SYNC_RES_CONNTRACK` and `CT_SYNC_RES_EXPECT`, although support for the latter has not been fully implemented yet).

For every resource, there are several message types. So far, only `CT_SYNC_MSG_UPDATE` and `CT_SYNC_MSG_DELETE` have been implemented. This means a new connection as well as state changes to an existing connection will always be encapsulated in a `CT_SYNC_MSG_UDPATE` message and therefore contain the full conntrack entry.

To uniquely identify (and later reference) a conntrack entry, the only unique criteria is used: `ip_conntrack_tuple`.

2.3.2 `ct_sync` sender thread

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.

It was therefore decided to use a pre-allocated ringbuffer for outbound `ct_sync` packets. New messages are appended to individual buffers in this ring, and pointers into this ring are passed to the in-kernel sockets API to ensure a minimum number of copies and memory allocations.

2.3.3 `ct_sync` initsync sender thread

In order to facilitate ongoing state synchronization at the same time as responding to initial sync requests of an individual slave, the sender has a separate kernel thread for initial state synchronization (and `ct_sync_initsync`).

At the moment it iterates over the state table and transmits packets with a fixed rate of about 1000 packets per second, resulting in about 4000 connections per second, averaging to about 1.5 Mbps of bandwith consumed.

The speed of this initial sync should be configurable by the system administrator, especially since there is no flow control mechanism, and the slave node(s) will have to deal with the packets or otherwise lose sync again.

This is certainly an area of future improvement and development—but first we want to see practical problems with this primitive scheme.

2.3.4 `ct_sync` receiver thread

Implementation of the receiver is very straightforward.

For performance reasons, and to facilitate code-reuse, the receiver uses the same pre-allocated ring buffer structure as the sender. Incoming packets are written into ring members and then successively parsed into their individual messages.

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

2.3.5 Necessary changes within netfilter conntrack core

To be able to achieve the described conntrack state replication mechanism, the following changes to the conntrack core were implemented:

- Ability to exclude certain packets from being tracked. This was a long-wanted feature on the TODO list of the netfilter project and is implemented by having a “raw” 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. This is part of the nfnetlink-ctnetlink patch, since the ctnetlink event interface also uses this API.
- Export an API to externally add, modify, and remove conntrack entries.

Since the number of changes is very low, their inclusion into the mainline kernel is not a problem and can happen during the 2.6.x stable kernel series.

2.3.6 Layer 2 dropping and `ct_sync`

In most cases, netfilter/iptables-based firewalls will not only function as packet filter but also

run local processes such as proxies, dns relays, smtp relays, etc.

In order to minimize failover time, it is helpful if the full startup and configuration of all network interfaces and all of those userspace processes can happen at system bootup time rather then in the instance of a failover.

l2drop provides a convenient way for this goal: It hooks into layer 2 netfilter hooks (immediately attached to `netif_rx()` and `dev_queue_xmit`) and blocks all incoming and outgoing network packets at this very low layer. Even kernel-generated messages such as ARP replies, IPv6 neighbour discovery, IGMP, ... are blocked this way.

Of course there has to be an exemption for the state synchronization messages themselves. In order to still facilitate remote administration via SSH and other communication between the cluster nodes, the whole network interface used for synchronization is subject to this exemption from l2drop.

As soon as a node is propagated to master state, l2drop is disabled and the system becomes visible to the network.

2.3.7 Configuration

All configuration happens via module parameters.

- `syncdev`: Name of the multicast-capable network device used for state synchronization among the nodes
- `state`: Initial state of the node (0=slave, 1=master)
- `id`: Unique Node ID (0..255)
- `l2drop`: Enable (1) or disable (0) the l2drop functionality

2.3.8 Interfacing with the cluster manager

As indicated in the beginning of this paper, `ct_sync` itself does not provide any mechanism to determine outage of the master node within a cluster. This job is left to a cluster manager software running in userspace.

Once an outage of the master is detected, the cluster manager needs to elect one of the remaining (slave) nodes to become new master. On this elected node, the cluster manager will write the ascii character `1` into the `/proc/net/ct_sync` file. Reading from this file will return the current state of the local node.

3 Acknowledgements

The author would like to thank his fellow netfilter developers for their help. Particularly important to `ct_sync` is Krisztian KOVACS `<hidden@balabit.hu>`, who did a proof-of-concept implementation based on my first paper on `ct_sync` at OLS2002.

Without the financial support of Astaro AG, I would not have been able to spend any time on `ct_sync` at all.