# Condor Network Integration Implementation

## Introduction

This document outlines a mechanism for integrating the Condor starter directly with the Linux network layer. We do this by creating a special network device per job, and placing the job in a separate *network namespace* such that all job traffic must go through this network device.

The entire mechanism is outlined here: <http://osgtech.blogspot.com/2011/12/network-accounting-for-condor.html>. This document is meant to describe the implementation.

This document covers four major pieces of code:

* High-level NetworkNamespaceManager object, which orchestrates the interactions with the network.
* Integration of network accounting with the Condor system.
* Low-level code for interacting with the kernel via netlink.
* Low-level code for interacting with the kernel via netfilter.

All code discussed here is posted on github (<http://github.com/bbockelm/condor>) in the network\_namespaces branch.

## Network Namespace Manager

**Files in question**: src/condor\_utils/network\_namespaces.h, src/condor\_utils/network\_namespaces.cpp

All logic centers around the NetworkNamespaceManager object (src/condor\_utils/network\_namespaces.cpp). This object is invoked by the condor\_starter at several points when spawning a new job:

* **CreateNamespace** – Called by the starter object prior to asking DaemonCore to create the job process. Initiates communication with the kernel, creates the pipe Ethernet devices, and invokes the sysadmin-provided callout script to configure the devices and routing.
* **PreClone** – Called immediately prior to forking the job. While it can be used prior to calling Linux’s clone(2), the network manager is not compatible with the shared-memory clone (CLONE\_VM must be called).
* **PostCloneParent** – Called immediately after the clone in the parent process. The parent waits until the child has finished creating a namespace, then moves one end of the network pipe into the child’s namespace. Finally, the parent waits until the child exec’s or sends an error.
* **PostCloneChild** – Called immediately after the clone in the child process. The child first creates its own network namespace and informs the parent of the result. It then blocks until the parent indicates it is OK to proceed. The child places itself in a separate namespace, adds a network address to the internal device, enables the device, and sets up the default and local route.
* **PerformAccounting** - given a ClassAd that is ready to be sent to the startd, check

## Low-level Netlink Code

**Files in question**: src/condor\_utils/network\_netlink.cpp, src/condor\_utils/network\_manipulation.h

Netlink is a communication domain (similar to Unix domain sockets, IPv4 sockets, or IPv6 sockets) for inter-process-communication on Linux. Primarily, we will use it for communicating with the kernel. In particular, we will be sending specially-formatted messages to manipulate the kernel’s network stack.

As far as we can tell, the message formats are extremely stable (cannot find any cases of breaking the ABI since the 2.4 to 2.6 switch), but are undocumented. We have attempted to comment one of the simpler calls (add\_address) in detail in the code. Those wanting to know how these network interactions work should start there.

Almost all functions exported are blocking routines that do a complete interaction with the kernel (i.e., the action being performed can be done with a single call to the function). “add\_address”, for example, takes socket, an IPv4 address string and an Ethernet device name, and adds the IPv4 address to the device. The socket used is opened by calling “create\_socket” – this is the only state the user must manage (to avoid the overhead of reopening the socket for each communication). Internal to the compilation unit, a global variable (seq) is kept to create an ordering of packets sent to the kernel.

The implementations consist of crafting a special netlink packet, then using the internal “send\_and\_ack” routine to send the packet to the kernel and wait for a response. The netlink packets consist of a header (struct nlmsghdr). Note in the implementation that all calls are aligned according to the macros NLMSG\_LENGTH/NLMSG\_ALIGN or RTA\_LENGTH/RTA\_ALIGN (the \_LENGTH variant includes the header size; \_ALIGN doesn’t). All bytes are kept in host-order.

## Low-Level Netfilter Code

**Files in question**: src/condor\_utils/network\_manipulation.h, src/condor\_utils/network\_netfilter.c

Netfilter controls the host’s firewall. This code provides the ability to read information from a firewall – specifically, we want to read the number of packets that have gone through each rule for accounting purposes.

The code interacts with netfilter by creating a raw socket, then reading out the firewall configuration via a syscall, getsockopt (compare this to the message-based approach used by netlink). The firewall configuration is returned as an in-memory data structure that we must then parse.

The exposed function is perform\_accounting; this takes a chain name and a callback function, and invokes the callback for each rule in the chain. The callback is invoked with the contents of the rule’s comment section, the number of bytes matched by the rule, and the user-data pointer (passing void\* instead of doing type-safe object-oriented code is done because the Linux headers are not compatible with C++, and we must use straight C).

The parsing code is currently simple – we first iterate through the binary blob, searching for the correctly named chain. Then, we have a separate function that iterates through each rule in the chain, looking for the comment and matched bytes information.

If we decide to augment the existing implementation with a built-in NAT setup, we will have to greatly extend this code to add rules and create chains.

## Integration with Condor Accounting

(TODO: Write section)

## Test Code

Files in question: src/condor\_starter.V6.1/network\_netlink\_main.c, src/condor\_starter.V6.1/ns\_exec.c

This is standalone test code for namespaces and the netlink/netfilter manipulations.

* **ns\_exec.c** – A CLI from IBM; it’s floating around the internet in several places, but isn’t packaged as part of a library. It allows you to fork/clone a process where you can choose the namespace flags manually. It’s a great help for testing out namespace ideas by command-line – you can fork a bash process in its own namespace and do various testing.
* **network\_netlink\_main.c** – A test of all the low-level netlink/netfilter functions in the (unfortunately, includes a lot of duplicate code). This will create the namespaces, configure the routing, and perform accounting. If this works, it will verify most of the networking functionality.

## Code Projects

* Upgrade to IPv6. Some of the internal structures are IPv4-specific. (Zhe)
  + The netlink code is fairly good about this (only have to change a function or two).
  + This is particularly nasty with netfilter. A lot of the code is going to be duplicated between IPv4 and IPv6. Iptables takes the approach of doing typedefs/defines, and compiling the same code twice – one with the appropriate defines for IPv4, and one with the defines for IPv6.
    - Aside: The netfilter interfaces are quite horrid.
* Pluggable manager. The NetworkNamespaceManager is a fairly generic interface. It would be nice to push the implementation off to a separate, loadable library. (Brian)
* Turn network\_netlink\_main.c into stand-alone program. Would be nice to be able to use it to launch a bash shell in the configured network environment. (Zhe or Todd?)
  + Great for sysadmin debugging.
* Contrib-ify. Move all code to condor\_contrib. Integrate with build system so it can be turned on/off. (Brian)
* Better default NAT setup. The existing NAT configuration code is a bit ad-hoc. Make it so it works out-of-the-box at *any* site.
  + Turn it to C program instead of script-based? Maybe still retain callouts to iptables so we don’t have to replicate that code?
  + Would like to be able to iterate through the available interfaces and pick an appropriate job-internal address by default.
  + Add locking so we can pick an unused address in a race-free manner.
* Bridging/VLAN isolation. Come up with a strategy to automatically bridge jobs, instead of hiding them behind a NAT. This perhaps could make VLAN-tagging a bit more plausible.
* C-based creation and deletion of the firewall chains. (Brian)
* Startd-based cleanup of chains and devices. The startd should be able to check for leaked chains and devices.

## Future Design Thoughts

(These are just Brian’s personal thoughts, not a matter of fact)

Right now, there’s only one policy (the one defined by the sysadmin via the scripts they implement). The out-of-the-box policy is to create a NAT and do various networking statistics.

What we really want to do is to:

* Have a way for the job to advertise the policy it would like.
* Allow the startd and job to negotiate various policies (normal Rank and Requirements dance).
* The startd puts together the final policy, passes it to the starter (make the network policy part of the machine’s ClassAd?).
* The network plugin in the starter will get the machine’s ClassAd, with the network description, and execute a lark tool to configure the network appropriately.

So, we break things into three executables and one library:

1. lark\_network\_policy: Takes in a job and machine classad, writes out a policy for the network configuration.
2. lark\_network\_configure: Takes in a machine classad and configures the machine’s network appropriately.
3. lark\_network\_cleanup: Cleans up the network configuration (or all Condor-related network configs)
4. lark\_network (library): Manages the Condor internals