RZ 3482 (# 99522) 09/29/2003
Computer Science 60 pages

Research Report

Enhancements and Prototype Implementation of the ForCES Netlink2 Protocol

Guillaume Goutaudier

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

This document is the professional thesis I submitted to the Eurecom Institute and the École Doctorale STIC in partial fulfillment of the requirements for the degree of telecom Engineer. It describes the Netlink2 protocol I implemented during the 6 months I spent at the IBM Zurich Research Laboratory. This work was based on the background knowledge I obtained at the Ecole Nationale Supérieure des Télécommunications de Bretagne, the Eurecom Institute, and the Université de Nice Sophia Antipolis.

It was supervised by **Robert Haas** (IBM supervisor) and **Ernst Biersack** (Eurecom Professor). My manager at IBM was **Patrick Droz** (IBM Network Processors manager).

I thank all members of the IBM Network Processors team for their useful advice, as well as the authors of the Netlink2 ForCES IETF draft, Jamal Hadi Salim (Znyx Networks) and Steven Blake (Ericsson IP Infrastructure).

– Abstract –

The emergence of off-the-shelf network processor devices has created the need for standard mechanisms to allow these components to be combined into functional wholes. ForCES aims to define a framework and associated mechanisms for standardizing the exchange of information between the logically separate functionality of the control plane and the forwarding plane.

In the Unix world, Linux Netlink already addresses some ForCES requirements: Netlink2 intends to extend Netlink so that it is more compliant with the ForCES requirements. After describing ForCES and Netlink, this paper explains some Netlink2 design choices. A description of our GPL Netlink2 Linux implementation is also given. This description is followed by performance measurements. Before concluding, we point out some open issues which should be addressed in future.

Contents

Chapter 1

ForCES

This chapter introduces ForCES. It provides the main background to further understand the context in which Netlink2 was designed and the topics Netlink2 addresses. First, we describe the ForCES IETF working group, then a description of the ForCES framework and requirements is given. A complete description may be found in [1] and [2].

1.1 ForCES working group

ForCES stands for Forwarding and Control Element Separation. An IETF working group in the Routing Area, its chairs are Patrick Droz and David Putzolu. This working group was created with the following observation in mind:

> The emergence of off-the-shelf network processor devices that implement the fast path or forwarding plane in network devices such as routers, along with the appearance of a new generation of third party signaling, routing, and other router control plane software, has created the need for standard mechanisms to allow these components to be combined into functional wholes. ForCES aims to define a framework and associated mechanisms for standardizing the exchange of information between the logically separate functionality of the control plane, including entities such as routing protocols, admission control, and signaling, and the forwarding plane, where per-packet activities such as packet forwarding, queuing, and header editing occur. By defining a set of standard mechanisms for control and forwarding separation, ForCES will enable rapid innovation in both the control and forwarding planes. A standard separation mechanism allows the control and forwarding planes to innovate in parallel while maintaining interoperability.

The ForCES working group was created in 2002, and the drafts it has submitted provide standard definitions as well as a framework and requirements for future ForCES protocols.

1.2 ForCES framework

A complete description of the ForCES framework can be found in [1]. Here we give just a quick introduction to provide a basic understanding of the terms used:

> An IP network element (NE) appears to external entities as a monolithic piece of network equipment, e.g., a router, NAT, firewall, or load balancer. Internally, however, an IP NE (such as a router) is composed of numerous logically separated entities that cooperate to provide a given functionality (such as routing). Two types of NE components exist: control elements (CE) in the control plane and forwarding elements (FE) in forwarding plane (or data plane). FEs are typically ASIC, network-processor, or general-purpose processor-based devices that handle data path operations for each packet. CEs are typically based on general-purpose processors that provide control functionality such as routing and signaling protocols.
>
> ForCES aims to define a framework and associated protocol(s) to standardize information exchange between the control and forwarding planes. Having standard mechanisms allows CEs and FEs to become physically separated standard components. This physical separation accrues several benefits to the ForCES architecture. Separate components would allow component vendors to specialize in one component without having to become experts in all components. Standard protocol also allows the CEs and FEs from different component vendors to interoperate with each other and hence it becomes possible for system vendors to integrate the CEs and FEs from different component suppliers. This interoperability translates into many more design choices and greater flexibility for the system vendors. Overall, ForCES will enable rapid innovation in both the control and forwarding planes while maintaining interoperability. Scalability is also easily provided by this architecture in that additional forwarding or control capacity can be added to existing NEs without the need for forklift upgrades.

Figure 1.1 shows the logical components of the ForCES architecture and their relationships. The exact CE and FE definition is the following:

Addressable Entity (AE) An entity that is directly addressable given some interconnect technology. For example, on IP networks, it is a device with which we can communicate using an IP address; on a switch fabric, it is a device with which we can communicate using a switch fabric port number.

Physical Forwarding Element (PFE) An AE that includes hardware used to provide per-packet processing and handling. This hardware may consist of (but is not limited to) network processors, ASICs (application-specific integrated circuits) or general processors installed on line cards, daughter boards, mezzanine cards, or in stand-alone boxes.

Physical Control Element (PCE) An AE that includes hardware used to provide control functionality. This hardware typically includes a general-purpose processor.

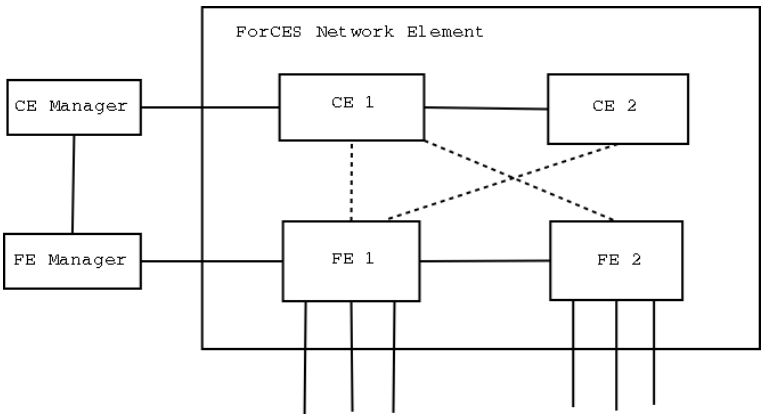


Figure 1.1: ForCES architecture.

Forwarding Element (FE) A logical entity that implements the ForCES protocol. FEs use the underlying hardware to provide per-packet processing and handling as directed by a CE via the ForCES protocol. FEs may be a single blade (or PFE), a partition of a PFE or multiple PFEs.

Control Element (CE) A logical entity that implements the ForCES protocol and uses it to instruct one or more FEs how to process packets. CEs handle functionality such as the execution of control and signaling protocols. CEs may consist of PCE partitions or whole PCEs.

The framework allows multiple instances of CE and FE inside one NE. Each FE contains one or more physical media interfaces for receiving and transmitting packets from/to the external world. The aggregation of these FE interfaces becomes the NE's external interfaces. In addition to the external interfaces, there must also exist some kind of interconnect within the NE so that the CE and FE can communicate with each other, and one FE can forward packets to another FE. Figure 1.1 also shows two entities outside of the ForCES NE: the CE Manager and the FE Manager. These two entities provide configuration to the corresponding CE or FE. Their task includes distributing identifiers to CE or FE as well as deciding which CE should manage which FE. The ForCES protocol is only defined for communication between CE and FE (dashed links on Figure 1.1). The interface between two ForCES NEs is identical to the interface between two conventional routers and these two NEs exchange the protocol packets through the external interfaces. ForCES NEs connect to existing routers transparently.

1.3 ForCES requirements

A complete description of the ForCES requirements can be found in [2]. Here we present most of them to elucidate how they are addressed by Netlink2.

First a clear distinction should be made between what should be covered by the ForCES protocol and what is beyond its scope. It is assumed that the ForCES protocol is not started from scratch but after a pre-association phase,

during which the CE and FE Managers determine which CEs and FEs should be part of the same network element. As stated above, the CE and FE Managers are also responsible of distributing identifiers to CE or FE as well as deciding which CE should manage which FE. As a consequence, all these features do not have to be supported by the ForCES protocol.

Now let us list the most important ForCES requirements:

1. CEs and FEs must be able to connect by a variety of interconnect technologies. Examples of interconnect technologies used in current architectures include Ethernet, bus backplanes, and ATM (cell) fabrics.

2. FEs must support a minimal set of capabilities necessary for establishing network connectivity (e.g., interface discovery, port up/down functions).

3. A NE must support the appearance of a single functional device.

4. The architecture must provide a way to prevent unauthorized ForCES protocol elements from joining an NE.

5. A FE must be able to asynchronously inform the CE of a failure or increase/decrease in available resources or capabilities on the FE.

6. The architecture must support mechanisms for CE redundancy or CE failover.

7. FEs must be able to redirect control packets (such as RIP, OSPF messages).

8. In a ForCES NE, the FEs must be able to provide their topology information (topology by which the FEs in the NE are connected) to the CE(s).

9. The ForCES NE architecture must be capable of supporting (i.e., must scale to) at least hundreds of FEs and tens of thousands of ports.

10. The ForCES architecture must allow FEs AND CEs to join and leave NEs dynamically.

11. The ForCES NE architecture must support multiple CEs and FEs.

12. The CE must understand how the FE processes packets. Therefore, an FE model be created that can express the logical packet processing capabilities of an FE. The FE model must define both a capability model and a state model, which expresses the current configuration of the device.

13. The protocol must provide a means for the CEs to control all the FE capabilities that are discovered through the FE model.

14. ForCES architecture must select a means of authentication for CEs and FEs.

15. The ForCES protocol must provide a means to express the protocol message priorities.

16. Mission-critical payloads must be delivered in a robust reliable fashion, but ForCES must not be restricted to strict reliability.

17. The ForCES protocol must be able to group an ordered set of commands to an FE. Each such group of commands should be sent to the FE in as few messages as possible. Furthermore, the protocol must support the ability to specify whether a command group must have all-or-nothing semantics.

18. The ForCES protocol must provide a means for the CE to query statistics (monitor performance) from the FE. The ForCES protocol must provide mechanisms for controlling FE capabilities that can be used to protect against denial-of-service Attacks.

We will see below how Netlink2 fulfills these requirements.

Chapter 2

Netlink

This chapter presents background information about Netlink. We describe how user applications should use Netlink sockets, how these sockets are implemented in the Linux kernel, and the messages format.

2.1 Netlink overview

The concept of IP control and forwarding separation was first introduced in the early 1980s by the BSD 4.4 routing sockets. The focus at that time was to provide a simple IP(v4) forwarding service and allow the control plane, either via a command line configuration tool or a dynamic route daemon, to control forwarding tables for that IPv4 forwarding service. The IP world has evolved considerably since then. Linux Netlink takes routing sockets one step further by breaking the narrow focus on IPv4 forwarding.

More precisely, Netlink is used to transfer information between kernel modules and user space processes. It provides kernel/user space bidirectional communication links. It consists of a standard sockets-based interface for user processes and an internal kernel API for kernel modules. When used to configure IPv4 forwarding, Netlink interfaces with the following Linux networking layers:

Linux supports numerous advanced networking features, including firewalls, QoS support in the form of queues, classes and filters, traffic conditioning, etc. Since the Linux 2.1 kernel, Netlink has been providing the IP service abstraction for a few additional services other than classical RFC 1812 IPv4 forwarding. In practice, different netlink families can be defined on top of Netlink. Thus, Netlink provides a unified way to configure these services. Netlink is now fully included in the Linux kernel. The Netlink code was written by Alan Cox and Alexey Kuznetsov.

2.2 User applications view

Netlink sockets are created using the standard BSD socket interface:

```
// e.g. socket_type = SOCK_ROW;
//      netlink_family = NETLINK_ROUTE;
netlink_socket = socket(AF_NETLINK, socket_type, netlink_family);
```

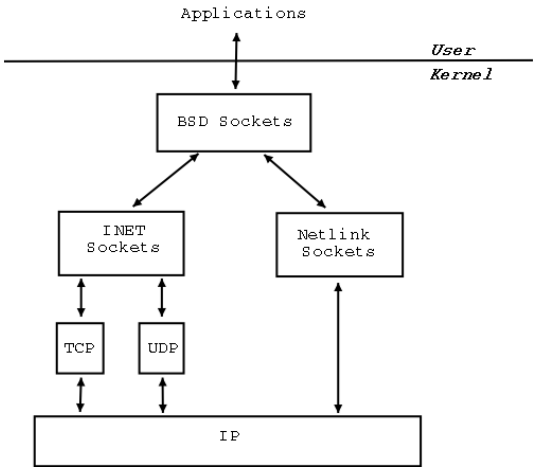


Figure 2.1: Netlink sockets.

netlink_family selects the kernel module or netlink group to communicate with. The currently assigned netlink families are:

NETLINK_ROUTE Receives routing updates and may be used to modify the IPv4 routing table.[1]

NETLINK_FIREWALL Receives packets sent by the IPv4 firewall code.

NETLINK_ARPD Manages the ARP table in user space.

NETLINK_ROUTE6 Receives and sends IPv6 routing table updates.

NETLINK_TAPBASE...NETLINK_TAPBASE+15 are the instances of the ethertap device. Ethertap is a pseudo network tunnel device that allows an Ethernet driver to be simulated from user space.

NETLINK_SKIP is reserved for ENskip.

NETLINK_USERSOCK is reserved for future user space protocols.

As with IP sockets, the Netlink socket then has to be bound to a given `sockaddr` address. In the case of Netlink, the address should be of type:

```
struct sockaddr_nl
{
    sa_family_t nl_family;      /* AF_NETLINK   */
    unsigned short  nl_pad;     /* zero     */
    __u32       nl_pid;         /* process pid  */
    __u32       nl_groups;      /* multicast groups mask */
};
```

`nl_pid` is used to represent the process id; `nl_groups` is used to represent the

[1]NETLINK_ROUTE refers to RT netlink sockets. Most of the time, it is included in the Linux kernel and prints the `Initializing RT netlink socket` message while booting.

multicast groups the process belongs to. Once the Netlink socket has been created and bound, one may send a Netlink message issuing the simple `send` (or `sendmsg`) call:

```
send(netlink_socket,buffer,buffer_length,0);
```

Buffer has to contain a Netlink message in the format described later in this chapter.

2.3 Kernel view

Netlink sockets are registered during kernel initialization:

```
struct net_proto_family netlink_family_ops = {
    PF_NETLINK,
    netlink_create
};
static int __init netlink_proto_init(void)
{
...
sock_register(&netlink_family_ops);
...
}
```

The `sock_register()` function makes the association between the `AF_NETLINK` socket family and `netlink_create()` Netlink socket creation function, so that this function is called when a user wants to create a Netlink socket. But how is the communication with kernel modules actually performed? A key structure in the Netlink architecture is the `nl_table` (see Figure 2.2).

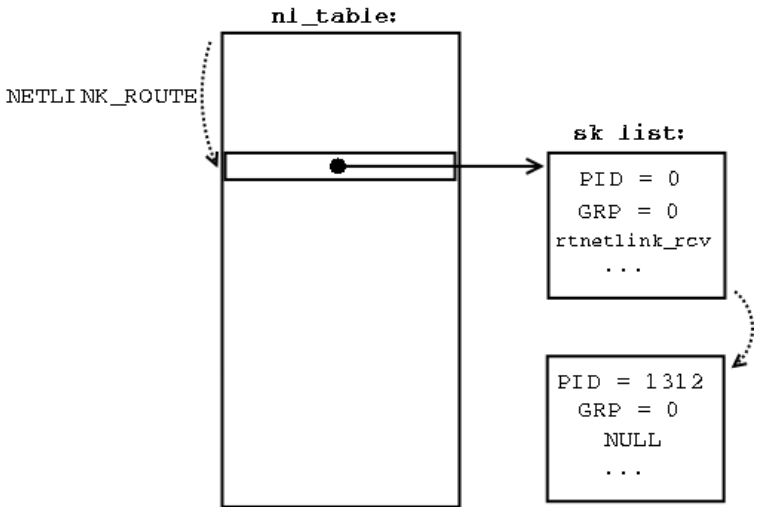


Figure 2.2: Netlink message passing table.

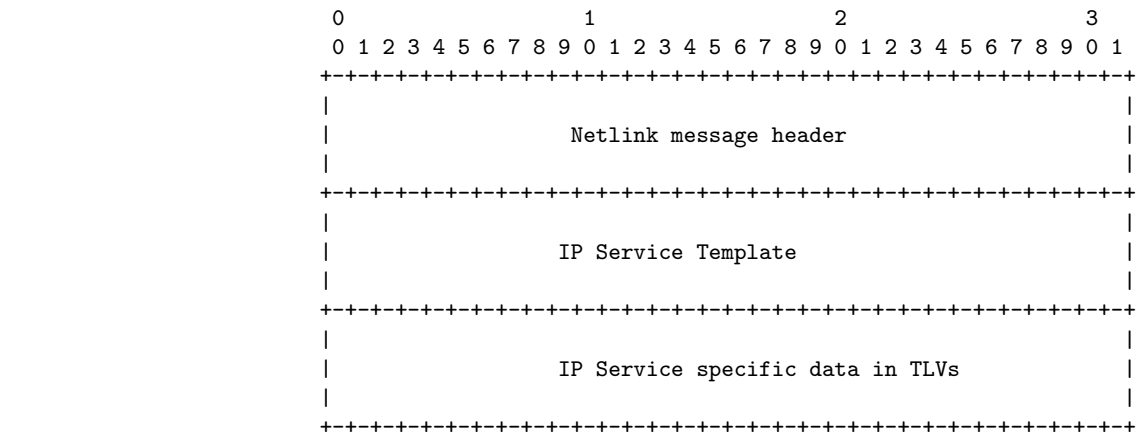
This table is indexed by the Netlink protocol, e.g. `NETLINK_ROUTE`. Each entry is a list of processes that communicate using this protocol. The list is made

up of `sk` structures containing data such as the PID, GROUPS, and handling function. These data are created either by the `netlink_create()` function in the case of a call issued by a user process, or by the `netlink_kernel_create()` function in the case of a call issued by a kernel module. In the first case, the handling function is set to `NULL`, so that a listening process has to make a read-like call on the socket. In the second case, the handling function is called directly. Each time a process issues a send-like call on a Netlink socket, Netlink delivers the message to the proper receiver that performs a lookup on this table.

For a deeper understanding, one may look directly into the kernel code, which is in the `net/netlink` and `net/core` directories of the kernel tree.

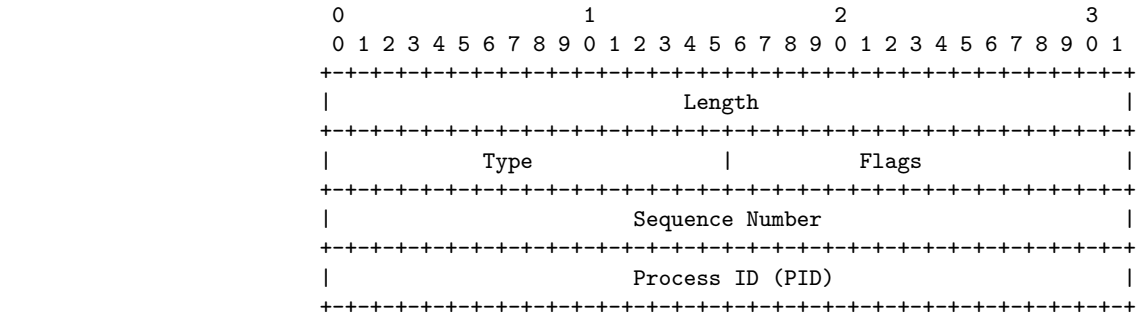
2.4 Messages format

A complete description of Netlink IP Service templates can be found in [3]. There are three levels to a Netlink message: The general Netlink message header, the IP service-specific template, and the IP service-specific data.



The Netlink message header is generic for all services, whereas the IP Service Template header is specific to a service. Each IP Service then carries parameterization data. These parameterizations are in TLV (Type-Length-Value) format and are unique to the service.

The Netlink message header is in the following format:



Length The length of the message in bytes, including the header.

Type Describes the message content. The three special values `NLMSG_NOOP`, `NLMSG_ERROR`, and `NLMSG_DONE` correspond to a message which has to be ignored, an error, or the end of a multipart transaction. All others are used to identify the IP Service Template that follows the Netlink message header.

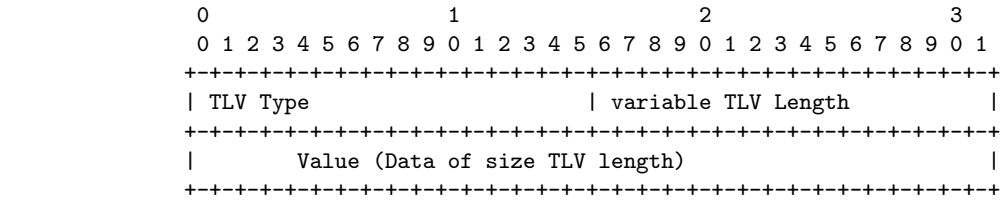
Flags Additional information on how the message should be processed. Examples are the `NLM_F_REQUEST` flag, which must be set on all request messages, the `NLM_F_ACK` flag, which requests an acknowledgment on success, or the `NLM_F_ECHO` flag which ask the kernel to echo the request.

Sequence number The sequence number of the message.

Process ID (PID) The PID of the process sending the message. PID is set to zero on messages coming from the kernel.

Different IP service templates exist for each Netlink protocol. We will not enter into the details here.

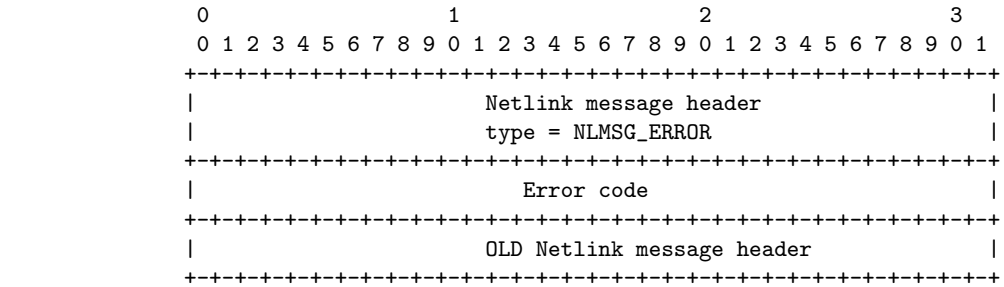
If these IP service templates accept optional arguments, they are given in the generic TLV format:



The TLV type field indicates the type of data encapsulated within the TLV. The TLV length field indicates the length of this TLV including the TLV type, TLV length, and the TLV data. If the total size of the message is known, TLV processing is easy: one need only look at the successive lengths, calculate each time the beginning of the next TLV, and stop when the sum of the lengths equals the length of the message.

2.5 Acknowledgments

Netlink has built-in acknowledgment messages (ACKs). This message is actually used to denote both an ACK and a NACK. The format of the ACK message is the following:



An error code of zero indicates that the message is an ACK response. An ACK

response message contains the original Netlink message header, which can be used for comparison.

A non-zero error code message is equivalent to a Negative ACK (NACK). In such a situation, the Netlink data that was sent down to the kernel is returned appended to the original Netlink message header.

Netlink uses a cumulative ACK policy, i.e. an ACK with a given sequence number acknowledges all messages with a lower sequence number. It is important to note that cumulative ACK only applies to "positive" ACK, i.e. ACKs with error code set to 0. A NACK in the sequence would turn off the cumulative ACK method. To illustrate this, let us take the example of the ACK/NACK sequence to be sent:

```
ACK   ACK   ACK   NACK   ACK   ACK
(1)   (2)   (3)   (4)    (5)   (6)
```

One may think of sending the following sequence:

```
 -     -    ACK   NACK    -    ACK
(1)   (2)   (3)   (4)    (5)   (6)
```

But what if the NACK is lost? The receiver will think everything was right from (1) to (6)! Actually when an irregular event occurs, the receiver sends a NACK (step 4) and stops processing any message until that message has been taken care of. This helps the sender because it does not have to rely on a timeout to retransmit, and leading to shorter delays.

2.6 Two-phase commit

Netlink provides all the semantics to make two-phase commit operations, but they are not used. A two-phase commit transaction should be made using the `NLM_F_MULTI` and `NLM_F_ATOMIC` flags as well as the `NLMSG_DONE` message type. The `NLM_F_MULTI` flag indicates that the Netlink message is part of a longer message sequence. The `NLM_F_ATOMIC` flag indicates that the operations from the multipart message have to be done using an "all-or-nothing" policy. An empty message of type `NLMSG_DONE` should be sent at the end of the two-phase commit transaction.

Let us take the example of two routes that have to be updated using the two-phase commit functionality and that require acknowledgments. The sequence of messages would be:

1. Send 1st Netlink message to add a route:

   ```
   type = RTM_NEWROUTE;
   flags = NLM_F_REQUEST | NLM_F_CREATE | NLM_F_ACK | NLM_F_MULTI | NLM_F_ATOMIC;
   ```

2. Await ACK.

3. Send 2nd Netlink message to add a route:

   ```
   type = RTM_NEWROUTE;
   flags = NLM_F_REQUEST | NLM_F_CREATE | NLM_F_ACK | NLM_F_MULTI | NLM_F_ATOMIC;
   ```

4. Await ACK.
5. Send empty Netlink message to close the transaction:

```
type = NLMSG_DONE;
flags = NLM_F_ACK;
```

The received ACKs merely indicate that the command has been received but they are committed only after the last `NLMSG_DONE` message.

The two-phase commit feature should be implemented in Netlink2.

Chapter 3

Netlink2

In the previous chapters, we have presented ForCES and Netlink. ForCES and Netlink have been conceived with different objectives in mind: ForCES was designed to unify network element design, whereas Netlink was linked to Linux kernel architecture. In this chapter, we show which ForCES requirements Netlink already addresses and the features Netlink lacks. Then we present Netlink2 and how it addresses these missing features.

3.1 Why use Netlink?

Looking at how Netlink currently works, one may identify a CE to a user application and an FE to the Linux kernel. As is, Netlink already addresses some ForCES requirements (see ForCES chapter). Let us examine them one by one.

- *Requirement 2 – FEs must support a minimal set of capabilities necessary for establishing network connectivity (e.g., interface discovery, port up/down functions).*
  Having identified an FE to the Linux kernel, this requirement is obviously fulfilled (Linux is well known for its high network connectivity features).

- *Requirement 3 – A NE must support the appearance of a single functional device.*
  If the NE only contains a single CE (user application) and a single FE (kernel), an NE would be seen by external entities as a single Linux box. In the case of multiple FEs, the control Linux allows on the data flow would allow it to emulate a single functional device.

- *Requirement 4 – The architecture must provide a way to prevent unauthorized ForCES protocol elements from joining an NE.*
  Netlink provides no means of authentication because it is to be run on a single box and allows only logged users to create Netlink sockets. Moreover, it is not because a user succeeded in creating a Netlink socket that all his requests will be accepted by the kernel: most of the commands would actually require root privileges.[1] This requirement should be fulfilled in Netlink2 while allowing remote applications to send Netlink messages.

[1] Note that to use the group multicasting Netlink feature, root rights are also required.

- *Requirement 5 – A FE must be able to asynchronously inform the CE of a failure or increase/decrease in available resources or capabilities on the FE.*
  Kernel modules are able to broadcast a failure message on a given Netlink channel.

- *Requirement 7 – FEs must be able to redirect control packets (such as RIP, OSPF messages).*
  Again, if we identify an FE to the Linux kernel, such a redirection would be done easily: the CEs simply have to listen on the port reserved for OSPF messages (which is actually 89), just as an OSPF daemon would do.

- *Requirement 9 – The ForCES NE architecture must be capable of supporting (i.e., must scale to) at least hundreds of FEs and tens of thousands of ports.*
  FEs are represented in Netlink by a 32 bits PID and a 32 bits GROUPS identifiers. This potentially allows thousands of FEs. Note that the semantics of these identifiers is different in Netlink2 (see the Addressing section).

- *Requirement 10 – The ForCES architecture must allow FEs AND CEs to join and leave NEs dynamically.* The way Netlink currently works, CEs (user applications) would join the NE creating netlink sockets or listing on routing dedicated port, and FEs (kernel modules) would join the NE calling the `netlink_kernel_create()` function. Likewise, to leave the NE, the Netlink socket should be closed or the `netlink_release()` function should be called.

- *Requirement 11 – The ForCES NE architecture must support multiple CEs and FEs.* See requirement 9.

- *Requirement 12 – The CE must understand how the FE processes packets. Therefore, an FE model be created that can express the logical packet processing capabilities of an FE. The FE model must define both a capability model and a state model, which expresses the current configuration of the device.*
  This is the major advantage of building a ForCES protocol on Netlink: Netlink already defines many IP service templates (see previous chapter) which enables a CE to interface with numerous services. These services define a complete FE model, including features such as IPv4/IPv6 forwarding, classification, QoS, packet redirection, and IPsec. Each individual Linux-based running router contributes to validating the wideness and consistency of this model.

- *Requirement 13 – The protocol must provide a means for the CEs to control all the FE capabilities discovered through the FE model.*
  Again, FE capabilities are well defined and each type of service can be controlled by means of a dedicated IP Service Template.

- *Requirement 14 – ForCES architecture must select a means of authentication for CEs and FEs.*
  See requirement 4.

- *Requirement 18 – The ForCES protocol must provide a means for the CE to be able to query statistics (monitor performance) from the FE. The ForCES protocol must provide mechanisms for controlling FE capabilities that can be used to protect against Denial of Service Attacks.* The IP Service Templates already define messages for querying statistics. Moreover, any given module is allowed to broadcast statistics on a given Netlink channel.

3.2 Missing features

- *Requirement 1 – CEs and FEs must be able to connect by a variety of interconnect technologies. Examples of interconnect technologies used in current architectures include Ethernet, bus backplanes, and ATM (cell) fabrics.*
  Netlink is designed for communication between user-space applications and kernel modules. Netlink messages are not carried by IP packets. But Netlink is built on BSD sockets. In Netlink2, **an effort has been made to convert Netlink into an independent Layer 4/5 protocol, which may rely on various underlying technologies**. This is an important issue because some of the following requirements should now be taken into consideration.

- *Requirement 4 – The architecture must provide a way to prevent unauthorized ForCES protocol elements from joining an NE.*
  This issue should be taken into consideration to enable Netlink to run in a distributed environment. Any security mechanism can be used in the underlying protocol. In the case of IP, one may think of taking advantage of all IPsec security features.

- *Requirement 6 – The architecture must support mechanisms for CE redundancy or CE failover.*
  The new addressing semantics introduced in Netlink2 enables CE redundancy or CE failover (see Redundancy section).

- *Requirement 8 – In a ForCES NE, the FEs must be able to provide their topology information (topology by which the FEs in the NE are connected) to the CE(s).*
  This requirement has not yet been addressed by Netlink2, but is work-in-progress. However, we will see in this chapter that the high flexibility of Netlink2 would allow such queries easily.

- *Requirement 10 – The ForCES architecture must allow FEs AND CEs to join and leave NEs dynamically.*
  In Netlink2, new messages have been introduced to allow FEs and CEs to join and leave NEs dynamically (see next section).

- *Requirement 15 – The ForCES protocol must provide a means to express the protocol message priorities.*
  In Netlink2, new messages have been introduced to allow FEs and CEs to set message priorities (see next section).

- *Requirement 16 – Mission-critical payloads must be delivered in a robust reliable fashion, but ForCES must not be restricted to strict reliability.*
  Netlink is to be run between user-space and the Linux kernel, i.e. in a reliable channel. Netlink then provides no reliability mechanism. However, to enable Netlink to run in a distributed environment, such a mechanism should be provided. We saw that Netlink2 should be run on top of a transport protocol. Reliability should be provided by this transport protocol layer. In the current Netlink2 state, no specific message has been created to select the reliability level, which could be chosen statically or derived from the message priority.

- *Requirement 17 – The ForCES protocol must be able to group an ordered set of commands to an FE. Each such group of commands should be sent to the FE in as few messages as possible. Furthermore, the protocol must support the ability to specify if a command group must have all-or-nothing semantics.*
  See the section on batching.

3.3 Netlink2 overview

A detailed description can be found in [4]. Here we give the guidelines followed while designing Netlink2, how Netlink2 fits into the network stack and the Netlink2 message format.

3.3.1 Aims

Netlink2 intends to extend Netlink for greater compliance with ForCES requirements. To achieve this, Netlink2 leaves the Linux kernel internals and becomes an independent protocol that can be used in contexts other than Linux boxes. More precisely, Netlink2 provides a new addressing scheme as well as Netlink header extension. Until now we have only talked about CEs and FEs. These entities could be broken into smaller entities: control plane components and forwarding engine components.

Control Plane Components (CPCs) Control plane components encompass signaling protocols ranging from dynamic routing protocols, such as OSPF, to tag distribution protocols, such as CR-LDP. Classical management protocols and activities also fall in this category. These include SNMP, COPS, and proprietary CLI/GUI configuration mechanisms. The purpose of the control plane is to provide an execution environment for the above-mentioned activities with the ultimate goal of configuring and managing the second NE component: the FE. The result of the configuration defines the way packets traversing the FE are treated.

Forwarding Engine Components (FECs) The FE is the entity of the NE that incoming packets (from the network into the NE) first encounter. The FE's service-specific component massages the packet to provide it with a treatment to achieve an IP service, as defined by the Control Plane Components for that IP service. Different services will utilize different FECs. Service modules may be chained to achieve a more complex service.

If built to provide a specific service, the FE service component will adhere to a forwarding model.

Note that CPCs and FECs are also called control element components (CECs) and logical forwarding blocks (LFBs). The aim of Netlink2 is to enable these entities (which may be part of a distributed system) to communicate with the means Netlink already provides in the Linux kernel.

3.3.2 Netlink2 in the network stack

While designing Netlink2, it was difficult to use the IP addressing model "as is": ForCES elements from a given service group could be on different hosts (i.e. have different IP addresses), and ForCES elements on a given host do not necessarily belong to the same service group. We then introduced a new addressing scheme to provide a new addressing abstraction layer over the IP layer. As Netlink2 provides an addressing scheme, it may be regarded as a Layer 3/4 protocol, even if it already relies on a Layer 3 protocol. The following comparison with TCP could be drawn:
Netlink2 takes full advantage of the features offered by the TCP/IP stack, but

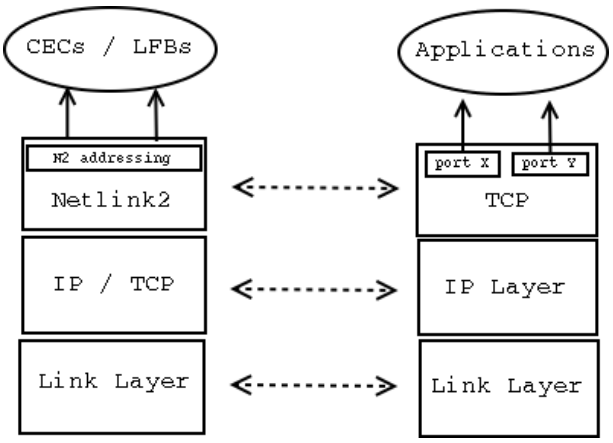


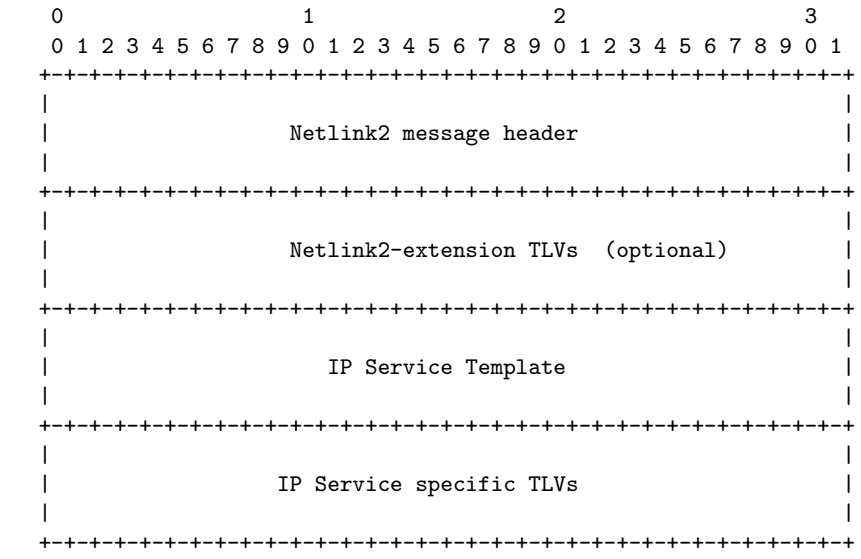
Figure 3.1: Netlink2 in the Network stack.

addresses CECs and LFBs rather than applications bound to a given TCP port. Also note that Netlink2 does not necessarily have to be run on top of TCP or IP.

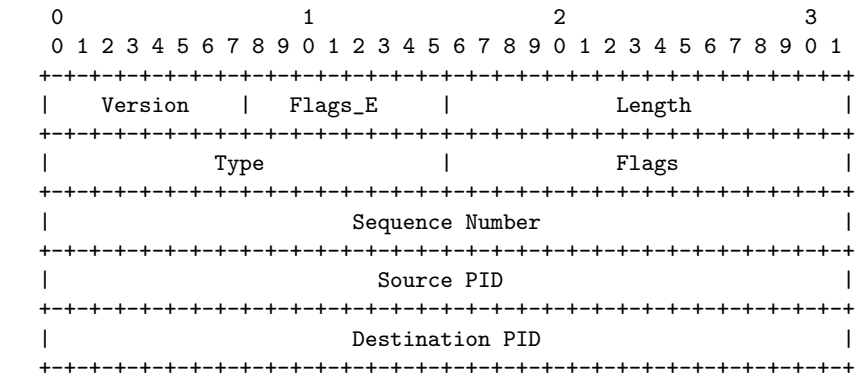
3.3.3 Message format

The exact message format has been subject to modifications. Here we give the initial Netlink2 message format we implemented. A discussion of possible changes is given in Chapter 6.

As with Netlink, the Netlink2 message format is divided into sublayers, with the addition of optional Netlink2 extended TLVs:



The Netlink2 message header is shown below:



A quick comparison with the Netlink message format shows that the `Length` field has been reduced to 16 bits. These 16 bits are replaced in Netlink2 by the `Version` and `Flags_E`. The new `Destination PID` field has also been introduced. The meaning of the fields is the following:

Version The version field is split into major:minor (4:4 bits) subfields. The value for Netlink2 is 0x20.

Flags_E This field contains flags specific to Netlink2. It provides support for new features. The already defined values are:

NLM_F_SYN Set on the first message. Interpreted as a boot message.

NLM_F_FIN Set on the last message. Interpreted as a departure message.

NLM_F_PRIO Message priority: 1 for high and 0 for low. Additional QoS level set in QOS TLV.

NLM_F_ASTR Set the ACK strategy: 1 for partial ACKs and 0 for full ACKs.

NLM_F_ETLV Extended TLVs on. This flag indicates whether optional TLVs are used or not.

The NLM_F_SYN and NLM_F_FIN flags address requirement 10. The NLM_F_PRIO addresses requirement 16. The NLM_F_ASTR helps avoid the ACK implosion problem when many FEs are involved in a Netlink2 transaction. An example showing how this flag should be used in given in Chapter 5.

Length The length of the Netlink2 message in bytes including the header.

Type This field describes the message content. The semantic is exactly the same as in Netlink.

Flags This field provides more information about the way the message should be treated. The semantic is exactly the same as in Netlink.

Source PID Netlink2 address of the sender.

Destination PID Netlink2 address of the destination of the message. The Source PID and Destination PID are explained in detail in the next section.

3.4 Addressing: from Netlink to Netlink2

In this section we give some background on addressing in distributed systems, explain Netlink addressing internals, and then we present the Netlink2 addressing. An example is given at the end of the section for clarity.

3.4.1 Related work

Before presenting the addressing format we chose for Netlink2, let us see how global addressing is managed in two distributed systems: CORBA and TIPC.

In the latest version of CORBA, the concept of *interoperable object reference (IOR)* has been introduced for object request broker (ORB) interoperability. When a client wants to call a method from a remote object, it makes requests using a simple object reference (e.g. a string) advertised by a naming service. Then the ORB embeds object key information in the IOR as well as the IP+Port of the remote object host. Given this information, the ORB is able to forward the call to the correct server, and this server is able to retrieve the correct object from the object key. This procedure is summarized in Figure 3.2. More information about CORBA addressing can be found in [5].

TIPC is a high-speed, message-oriented communication service designed for cluster environments. It provides location transparency using a logical addressing and maintaining a hot address translation table. The logical address has the format:

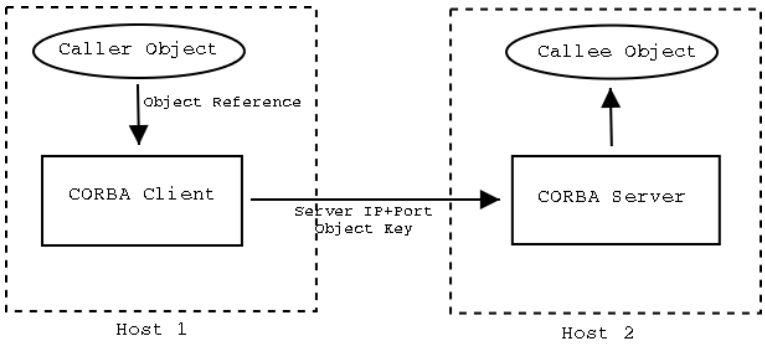


Figure 3.2: CORBA IOR.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Type                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Instance                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The *type* identifies a certain service type, and the *instance* is used as a qualifier for accessing a certain instance of the requested service. These logical addresses are converted internally by TIPC into volatile addresses, which have the following format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Random number                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Zone     | Subnetwork    |        Processor              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

These fields indicate the processor localization. More information about TIPC addressing can be found in [6].

3.4.2 Netlink addressing

Looking at the Netlink header, it seems that no information about the message destination is given. A Netlink message is actually delivered using the following pieces of information:

- While creating a Netlink socket, a Netlink *protocol* is specified in the call to socket: `socket(AF_NETLINK,SOCK_RAW,NETLINK_ROUTE)`. Here, for example, the protocol is `NETLINK_ROUTE`.
- As for Internet addresses which are represented by `sockaddr_in` structures, `sockaddr_nl` structures are used to represent Netlink addresses. One may send a Netlink message by calling the well-known `sendmsg()` function, and passing to it a `sockaddr_nl` structure as an argument.

The `sockaddr_nl` structure contains the *destination pid* as well as the *destination groups* to which the Netlink message has to be delivered. The *destination groups* is a 32-bit mask of the groups to be reached. For instance, if the *destination groups* has a value of `0x0009`, the message will be received by groups `0x0001` and `0x0008`. The *destination pid* is the Unix PID of the receiver.

It is important to note that the previous data is not mapped into Netlink header fields, but is used directly by the kernel to dispatch the message. The message is delivered using the *protocol, destination groups*, and *destination pid* following these three steps:

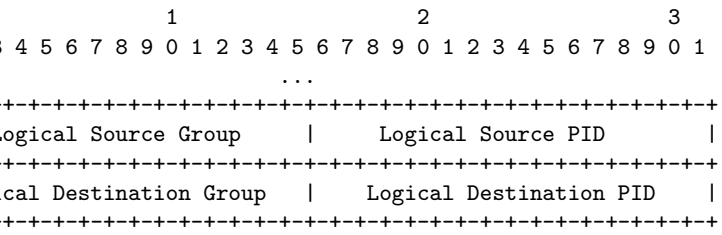
1. The set of receivers having registered the Netlink *protocol* is selected.
2. If the *destination groups* is not null, then the message is delivered to all receivers in the selected set belonging to one of the groups indicated by the *destination groups*.
3. If the *destination groups* is null, then the message is delivered to the process having the Unix PID *destination pid*.

3.4.3 Netlink2 addressing

While designing Netlink2, it was impossible to retain Netlink addressing. A one-to-one mapping between Netlink and Netlink2 identifiers was also impossible for two reasons:

1. Netlink addressing relies on information that is not present in the Netlink header.
2. Extra information is required to make Netlink2 distributed.

In Netlink2, we introduce the concept of *logical address* to embed the previous data. Two *logical addresses* are present in the Netlink2 header: the *logical source address* and the *logical destination address*. The *logical source address* is the *logical address* of the sender. This field replaces the previous *Process ID* Netlink header field. The *logical destination address* is the *logical address* of the receiver. Both the *logical source address* and the *logical destination address* have the same structure and are composed of two subfields: the *logical group* subfield and *logical PID* subfield. The Netlink2 header then contains the following fields:



The semantics of this fields is the following:

- *Logical Destination Group*. This 16-bit subfield is to be compared with the *destination groups* field used in Netlink, but it no longer represents a

bit mask of groups to reach. Instead, it now has a more general meaning, and represents a *"set of receivers"*. The meaning of a *"set of receivers"* depends on the NE topology and could represent:

- each entity belonging to a protocol, e.g. `NETLINK ROUTE`.
- a CE or FE in the NE.
- any logical set of Netlink2 receivers. To enable a hierarchical addressing, a group can be a set of groups.

Groups may be built to map the existing Netlink protocols (e.g. `NETLINK_ROUTE`, `NETLINK_FIREWALL`), but no special values are currently reserved. The `0xFFFF` special value is reserved for broadcast delivering.

Thus, the *Logical Destination Group* field could have different meanings. This design choice is driven by the aim to keep the addressing simple as well as to provide high flexibility to an NE designer.

- The *Logical Destination PID*. This 16-bit subfield is to be compared with the old Netlink *destination pid*. It uniquely identifies a Netlink2 receiver. The `0xFFFF` special value is reserved for broadcast delivering. The `0x0000` special value is reserved for targeting a group itself instead of a PID inside a group. This value is useful to address CEs/FEs directly. Typically, this special value will be used to address an FE having a given *Logical Group* and containing a single component. On most *nix systems, process identifiers do not have values greater than 32767, so a direct mapping between the *Logical Destination PID* and the Unix process identifier is possible. However, while running Netlink2 between different hosts, one should keep these values globally unique. The `0xDFFF` special value is reserved for CE broadcast delivering. The `0xEFFF` special value is reserved for FE broadcast delivering.

The *Logical Source Group* and *Logical Source PID* are dual to the *Logical Destination Group* and *Logical Destination PID*: they have the same meaning but they refer to the sender.

To have a consistent addressing, it is recommended that each host in the NE SHOULD be assigned a *Logical Group* so that any Netlink2 end point has a unique *Logical Source Address*, namely the *Logical Group* of its host plus its *Logical PID*. Also note that it is up to the implementer to reserve subsets of the *Logical Group* address space to a particular type of group.

The fields presented here only aim to represent a logical addresses. The actual group location addresses, which could be Unicast or Multicast IP addresses, are part of the Netlink2 implementation internals. For this purpose, one may think of maintaining a "hot translation table", either centralized or distributed, keeping track of the *Logical Address* / Real Address translation.

3.4.4 Example

In this example, we take an NE having the following topology:

Logical pids: the OSPF daemon has logical pid `0x0001`, the BGP4 daemon has logical pid `0x0002`, the Scheduler daemon has logical pid `0x0003`. The

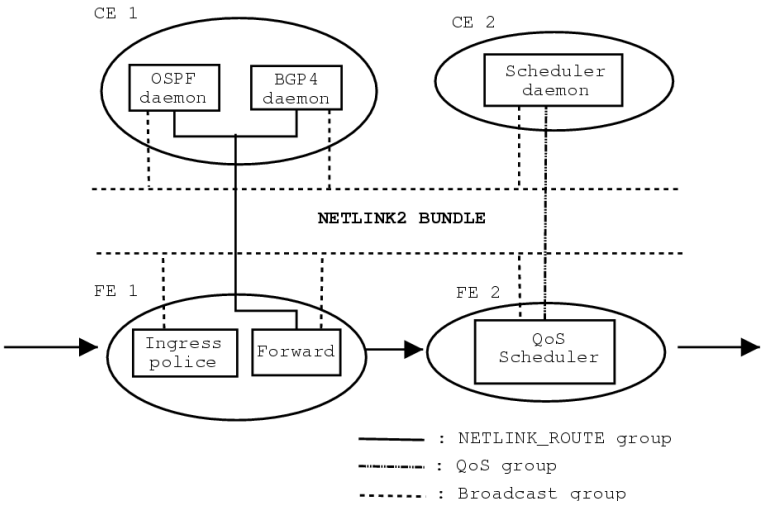


Figure 3.3: Example of an NE.

Ingress police component has logical pid `0x0004`, the Forwarder component has logical pid `0x0005`, the QoS scheduler has logical pid `0x0006`.

Logical groups: CE1, CE2, FE1, FE2 are represented by logical groups `0x0001`, `0x0002`, `0x0003`, `0x0004`. The `NETLINK_ROUTE` group has logical group `0x0005`. The `QoS` group has logical group `0x0006`. The `broadcast` group has logical group `0xFFFF`. Here we clearly see that logical groups could have different meanings: for example, the logical group `0x0001` represents the CE1 host, whereas the logical group `0x0005` represents all end points interested in routing information.

The previous information defines the addresses:

Element	**Address(es)**	**Attached group**
OSPF daemon	0x00010001	CE1
	0x00050001	NETLINK_ROUTE
BGP4 daemon	0x00010002	CE1
	0x00050002	NETLINK_ROUTE
Scheduler daemon	0x00020003	CE2
	0x00060003	QoS
Ingress police	0x00030004	FE1
Forwarder	0x00030005	FE1
	0x00050005	NETLINK_ROUTE
QoS Scheduler	0x00040006	FE2
	0x00060006	QoS
CE1	0x00010000	CE1
CE2	0x00020000	CE2
FE1	0x00030000	FE1
FE2	0x00040000	FE2
All	0xFFFFFFFF	All

Imagine the OSPF daemon sends a route update to the forwarder. It will use as the *logical source address* the concatenation of its host *logical group* (i.e. `0x0001`) and its *logical PID* (i.e. `0x0001`) so the *logical source address* will be `0x00010001`. Symmetrically, the *logical destination address* of the message will be the concatenation of the forwarder *logical group* and the forwarder *logical PID* (i.e. `0x0005`). If the OSPF daemon knows that the forwarder belongs to FE1 having *logical group* `0x0003`, it could send the message to the *logical PID* `0x00030005`. Instead, it could send the message to the `NETLINK_ROUTE` group, i.e. to the *logical PID* `0x00050005`.

Another example is the QoS scheduler having buffer overflow: it might send a message to `0x0006FFFF`, so that each member of the QoS group will receive it.

Open questions:

- Logical addresses should be unique. An open issue is to know whether this means that each PID should be globally unique, or whether two CEs or FEs may have the same PID has long as they do not belong to the same group. With the latter option, an FE dynamically joining a group containing an FE with its PID should be assigned a new one. For reasons of simplicity, the first option is preferable.
- At current Netlink2 state, all addresses are assigned statically, and the exact joining procedure has not yet been well defined. The question is to know whether the address request should be handled in the pre-association phase, in the reply to the `NLM_F_SYN` (which should be sent when an FE joins the NE, see next section), or in a separate message.

3.5 SYN message

At NE startup, or when an FE joins an NE, a SYN message with the ACK flag set should be sent. The exact procedure has not yet been defined, but we may follow the guidelines below:

- The SYN message should be sent to all CEs, i.e. to the `0xFFFFDFFF` address, so that all CEs know that this new FE is joining the NE. Only its "master" CE should reply to the SYN message. The "master" CEs (i.e. which CE should control which FE) are chosen during the pre-association phase.
- The SYN message could also be used as an address request message. The SYN message might contain as source logical address the host logical group of the FE and the PID it would like to use. The CE should ACK this message or return a NACK. In the latter case, the error code inside the NACK may represent either a rejection of the FE by the CE or an acceptance with a free PID the FE could use.

3.6 Redundancy

The Netlink2 addressing features can be used to provide high availability at either the CE or the FE level. Here we give two examples of how this could be done.

3.6.1 FE High Availability

Figure 3.4 shows a scenario with one CE and two FEs: an active FE and a backup FE. The CE could send all its messages with the following source and

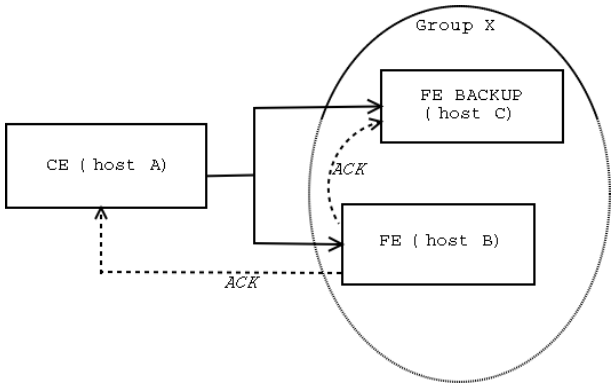


Figure 3.4: FE high availability.

destination addresses:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             0xFFFF            |             0xFFFF            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             0x0001            |             0xFFFF            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The group `0x0001` contains the two FEs. At the IP layer, it could be a multicast address on which these two FEs listen. After receiving the command, the FE will reply to address `0xFFFFFFFF`, i.e. to everyone, so that the backup FE will also receive the ACK. Obviously, the backup FE should be configured not to send an ACK back to the CE and should interpret the ACK it receives from the FE as a commitment. In this scenario, the CE does not have to be aware of the backup CE's existence. The backup FE could detect an FE failure (e.g. no longer receiving ACKs from it) and choose to replace it transparently. In such a case, the recovery process has to be fixed by the NE designer.[2]

3.6.2 CE High Availability

Figure 3.5 shows a scenario with one FE and two CEs: an active CE and a backup CE. The CE could send all its messages with the following source and destination addresses:

[2]In this example, one could have used the 0xFFFFEFFF special value to target all FEs.

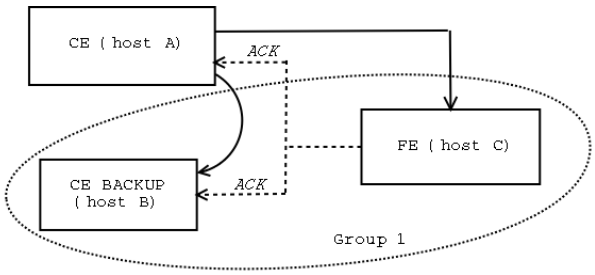


Figure 3.5: CE high availability.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             0xFFFF            |             0xDFFF            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|             0x0001            |             0xFFFF            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The group `0x0001` contains the FE and the backup CE. At the IP layer, it could be a multicast address on which the FE and backup CE listen. After receiving the command, the FE will reply to address `0xFFFFDFFF`, i.e. to all CEs, so that the backup CE will also receive the ACK. Obviously, the backup CE should be configured to interpret the previous messages so that it does not interpret them in the standard way but rather use them to be aware of the CE and FE states. The backup CE could detect a CE failure (e.g. the CE not responding to heartbeat messages), and choose to replace it transparently. In such a case, the FE does not have to be informed of the CE changes. The exact recovery process has to be fixed by the NE designer.

3.7 Capability query

This section only contains ideas which have not been yet been included in the Netlink2 draft.

While initiating a connection between a CE and an FE, the ForCES framework requires a way for the FE to advertise its capabilities:

> The FE needs to inform the CE of its own capability and its topology in relation to other FEs. The capability of the FE is represented by the FE model.
>
> ...
>
> The model would allow an FE to describe what kind of packet processing functions it contains, in what order the processing happens, what kinds of configurable parameters it allows, what statistics it collects and what events it might throw, etc.

Linux Netlink has no capability query features, since the capabilities are implicitly advertised by kernel modules, which register a particular Netlink protocol.

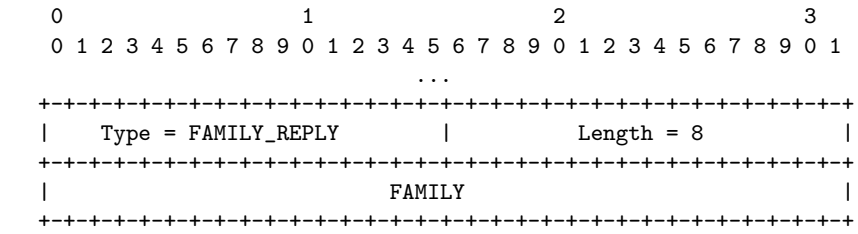
To enable capability query in Netlink2, we may introduce three new netlink types:

- `CAP_GETFAMILY`: this message type is issued by a CE to get a CE family (e.g. NETLINK_FIREWALL).

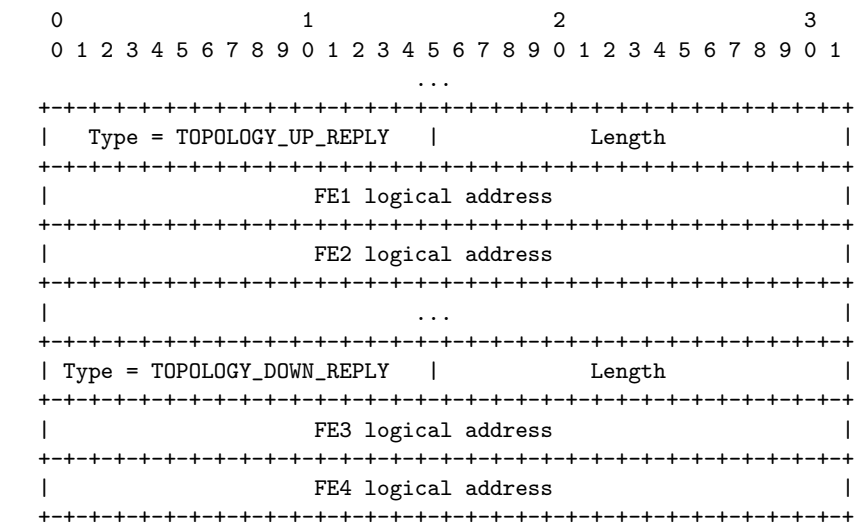
- `CAP_GETTOPOLOGY`: this message type is issued by a CE to ask which FEs the FE is connected to.

- `CAP_REPLY`: this message is issued by an FE in response to one of the previous messages. The data contained by this message is in TLV format. It could either contain a 32-bit integer indicating the FE protocol (in the case of a `CAP_GETFAMILY` query) or a list of FEs to which the FE is connected (in the case of a `CAP_GETTOPOLOGY` query).

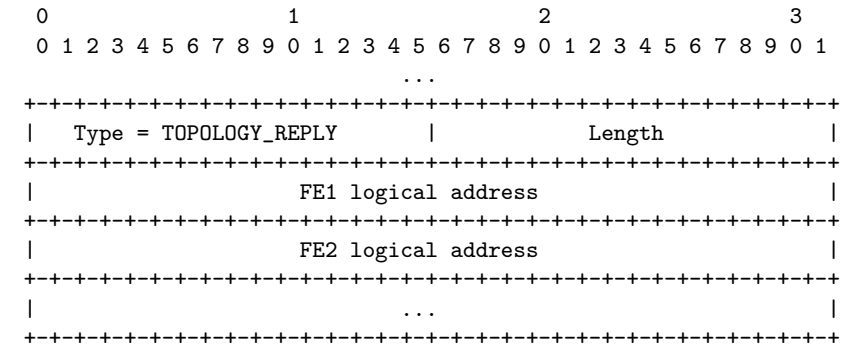
The first reply has the following format:



The second reply contains two parts: the list of upward FE(s) and the list of downward FE(s), according to the data path.



Note that it is assumed that a data path has already been defined. If this is not the case, the FE may reply with another format that does not distinguish between upward and downward FEs:



Open question: One may wonder whether the capability query message should be interpreted at the FE or the LFB level. At the FE level, it could have no meaning if the NE is composed of a single LFB (e.g. a Linux box) containing different LFBs. At the LFB level, the Netlink family may not be sufficient to describe the LFB capabilities.

3.8 Loss detection

One of the ForCES requirements is CE redundancy and CE failover:

> The ForCES protocol MUST support mechanisms for CE redundancy or CE failover. This includes the ability for CEs and FEs to determine when there is a loss of association between them.

To provide such a mechanism, as mentioned in [3], one could create a heartbeat protocol between the FE and CE by using the ECHO flags and the `NLMSG_NOOP` message.

3.9 Batching

Batching enables grouping series of operations. The main objective is to improve performances. Here we show what is done in the Network File System Version 4 (NFSv4) protocol, and specify some possible ways batching could be performed in Netlink2. A complete description of the NFSv4 protocol may be found in [12].

3.9.1 Related work

The newly designed NFSv4 protocol has some common properties with Netlink. Although old NFS versions were defined only in terms of remote procedure calls (RPCs), NFSv4 introduces a new COMPOUND procedure:

> The COMPOUND procedure provides the opportunity for better performance within high latency networks. The client can avoid cumulative latency of multiple RPCs by combining multiple dependent operations into a single COMPOUND procedure. A compound operation may provide for protocol simplification by allowing the client to combine basic procedures into a single request that is customized for the client's environment.

While receiving a COMPOUND request, the server processes it by evaluating each of the operations within the COMPOUND procedure in order. The results of each operation are then returned to the client in a single message. As in Netlink, results are represented with an error code (0 for an ACK). If an operation results in a non-zero status code, the evaluation of the compound sequence will halt and the reply will be returned. Again, Netlink behaves in the same way.

3.9.2 Netlink2 batching

Batching could be performed in many different ways. Let us take the example of a CE wanting to update 10,000 routes to a remote FE. To perform this we have tested the three following methods:

1. No batching:

```
send Netlink2 message #1 in a single packet, wait for ACK #1
send Netlink2 message #2, wait for ACK #2,
...
send Netlink2 message #10,000, wait for ACK #10,000
```

2. Do the same as method 1, but send multiple messages per packet. If we send 10 messages per packet, we would:

```
send first 10 messages (packet #1), wait for ACKs #1 to #10
...
send last 10 messages (packet #1,000), wait for ACKs #9,991 to
#10,000
```

3. Have multiple IP service templates in a single Netlink2 message. If we have 10 IP services templates per Netlink2 message, we would:

```
send Netlink2 message #1 (packet #1), wait for ACK #1,
...
send Netlink2 message #1,000 (packet #1,000), wait for ACK #1,000
```

4. Do the same as method 2, but use a cumulative ACK policy. If we send 10 messages per packet, we would:

```
send first 10 messages (packet #1), wait for ACKs #1 of message #10,
...
send last 10 messages (packet #1,000), wait for ACK #1,000 of
message #10,000
```

The first two methods correspond to Netlink normal behavior. Methods 3 and 4 could be done by introducing new flags to turn on batch mode or to set the cumulative ACK policy. The performance of these methods is studied in Chapter 5.

Chapter 4

Linux implementation

The previous chapter presented the Netlink2 protocol. Here we present our Linux implementation, explaining the design choice we made.

4.1 Architecture

We had the choice between two opposite architectures:

- Handle new Netlink2 functionalities inside the kernel.
- Handle new Netlink2 functionalities inside a daemon.

Figure 4.1 and Figure 4.2 illustrate those two approaches.

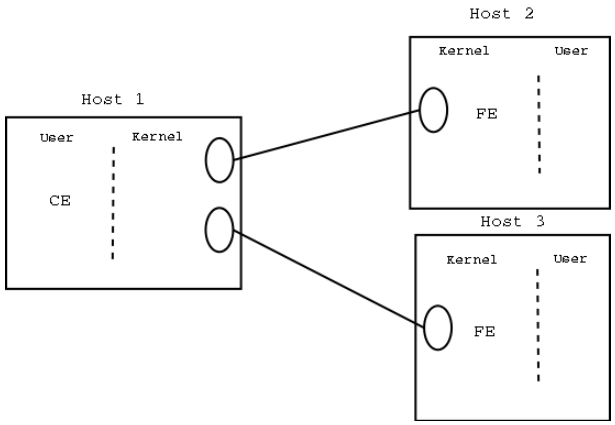


Figure 4.1: Netlink2: Architecture 1, kernel modification.

The fist solution (Figure 4.1) consisted of making the kernel support new Netlink2 features. As mentioned above, one of those features was support for a distributed Netlink2 broadcast wire. It implied that the kernel would have to deal with connection management with remote CEs/FEs. Moreover, this architecture was deeply linked to Linux, and interoperability with proprietary

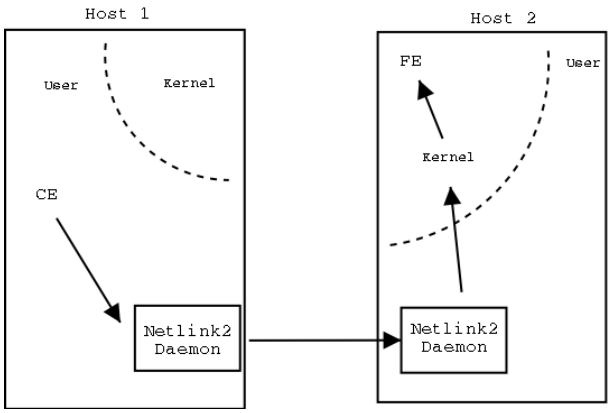


Figure 4.2: Netlink2: Architecture 2, daemon.

CEs/FEs would be difficult. Indeed, interoperability would be simple if we succeed in extracting the Netlink protocol from Linux internals. This implied limiting the number of "Linux-dependent" system calls and generalizing Netlink so that it no longer relies on a particular type of socket (see chapter 2). Even if this solution were feasible, it rapidly turned out not to be the most simple nor the most appropriate. The main reason is that it would break Netlink's backward compatibility. Another reason is that it would lead solving a Linux system call on a remote host, which would cause significant security problems.

The second solution was disconnected from the kernel: a daemon is in charge of interfacing with CEs (which most of the time will be user applications) and with FEs (which could be local or remote kernel services). Figure 4.2 shows the path taken by a message issued by a CE in a given host, which has to be transmitted to an FE in a remote host. The protocol between the CE and the daemon is Netlink2. After receiving the Netlink2 message, the daemon forwards it to the correct host/port so that the remote daemon can process it. The remote daemon has to ensure backward compatibility with Netlink to be able to send appropriate message to the remote kernel. With this solution, interoperability is possible: the daemon can send/receive Netlink2 messages to/from different architectures. Moving Netlink to a distributed environment is also easier since location information is handled by the daemons.

For all the previous reasons, we based our implementation on the second solution. In the rest of this chapter, we will explain it in detail.

4.2 Netlink2 daemon

4.2.1 Interfaces

Figure 4.3 shows the Netlink2 daemon interfaces:

Local application interface This interface enables local applications to connect to the Netlink2 daemon. These applications typically represent CEs. The protocol used through this interface is Netlink2.

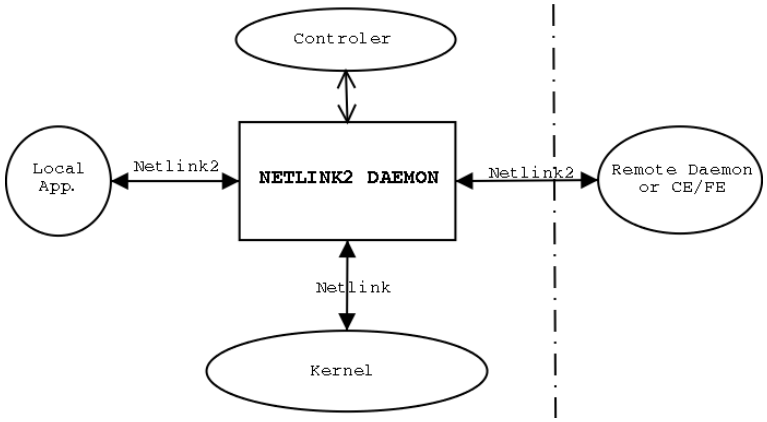


Figure 4.3: Netlink2 daemon: interfaces

Kernel interface This interface ensures Netlink backward compatibility. Typically, the Netlink2 daemon connects to kernel services using the Netlink protocol and emulates FEs. The Netlink2 daemon also acts as a gateway between these FEs and CEs.

Outside world interface This interface allows the sending/receiving of Netlink2 messages with the outside world. The Netlink2 daemon dispatches incoming Netlink2 messages to the appropriate CEs/FEs, and sends outgoing messages to remote hosts.

Configuration interface This interface does not provide any ForCES functionality, and is only used for "hot" configuration of the Netlink2 daemon.

4.2.2 Big picture

The Netlink2 daemon is implemented in `netlink2_daemon.c`. Figure 4.4 shows its internal structure.

The blocks in the figure are explained as follows:

Input Dispatcher The input dispatcher takes care of the incoming Netlink2 messages. It uses the information stored in the LLT to decide whether the message should be forwarded to a local CE, to the kernel gateway, or discarded. The input dispatcher is also in charge of interpreting some of the Netlink2 header flags of the message such as the `NLM_F_BATCH` flag we introduced for batching.

Output Dispatcher The output dispatcher takes care of the outgoing Netlink2 messages. It uses the information stored in the GLT to decide whether the message should be forwarded to a local CE, to the kernel gateway, to a remote CE/FE, or discarded.

Both the input and output dispatcher are part of the daemon core and are implemented in `netlink_daemon.c`.

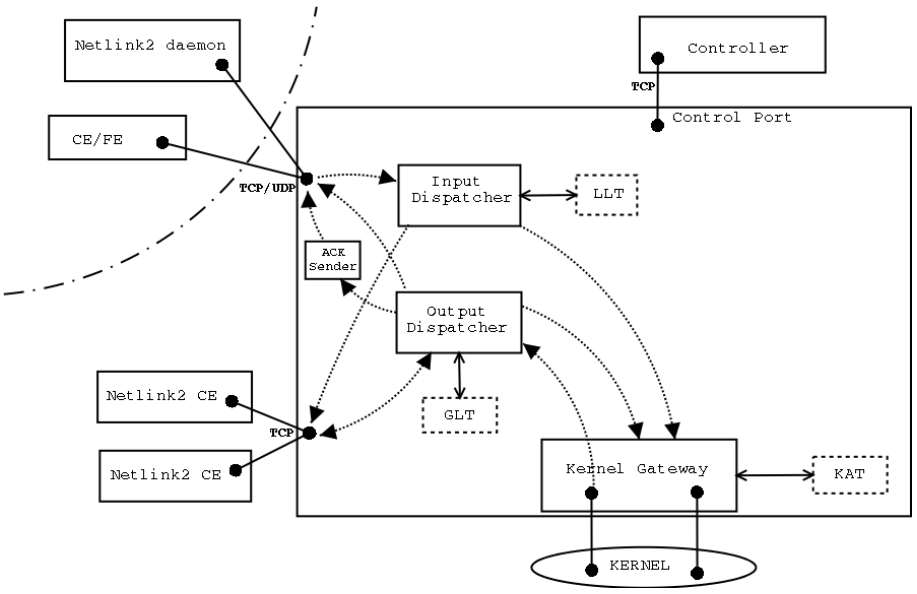


Figure 4.4: Netlink2 daemon: overall picture.

LLT The LLT block represents the local location table. This table stores the logical addresses of the CEs, which are locally connected to the daemon, and the logical addresses of the FEs that the kernel gateway emulates. In the case of CEs, the identifier of the socket by which they are connected is also stored. The LLT is implemented in `llt.c`.

GLT The GLT block represents the global location table. This table stores the logical addresses of the CEs/FEs in the NE and their corresponding IP address. The GLT is implemented in `glt.c`.

Kernel Gateway / KAT The kernel gateway is in charge of converting the Netlink2 messages it receives into Netlink messages that are understandable by the kernel. If a message requires an acknowledgment, the kernel gateway saves the message header and associated sequence number. These data are used to generate an acknowledgment in Netlink2 format after receiving the kernel reply. Figure 4.5 illustrates this process. After receiving the Netlink2 message, the kernel gateway generates a Netlink message and sends it to the Kernel. It stores the sequence number of the Netlink message and the Netlink2 header in the kernel acknowledgment table (KAT). Then, having received the kernel ACK, it generates a Netlink2 ACK with the help of the previously stored data. The KAT is implemented in `kat.c`. The kernel gateway is implemented in `kernel_gw.c`.

Controller A controller can connect on a special port to modify the LLT and GLT while the daemon is running. A controller is implemented in `netlink2_controller.c`.

Local CEs Local CEs can connect to the Netlink2 daemon. They can then

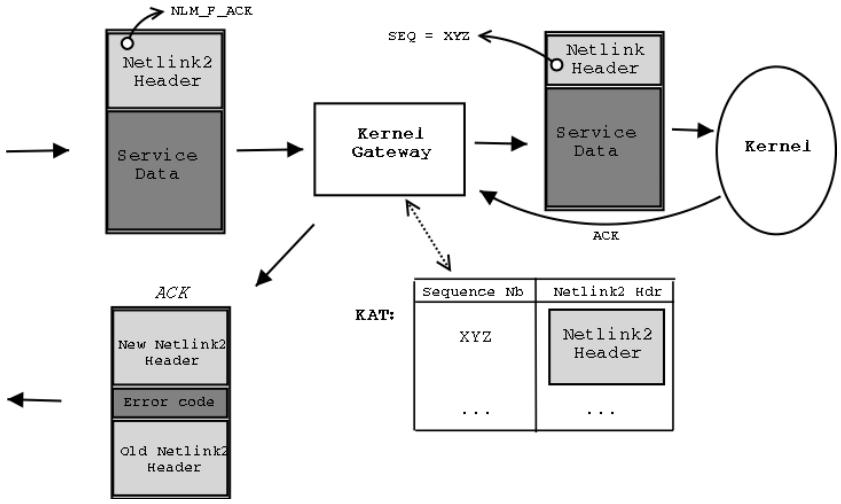


Figure 4.5: Netlink2 daemon: KAT table.

send a Netlink2 message to the daemon, which will either forward the message locally or to the appropriate IP address.

Remote daemon The standard way of transmitting a Netlink2 message between two hosts is to use a Netlink2 daemon at each host, which forwards messages to its connected CEs/FEs.

Remote CE It is also possible for a remote CE to connect directly with the daemon. In this case, it has to be aware of the IP addresses of the CEs/FEs in the NE (in a sense, it has to have its own GLT).

A Netlink2 Client implemented in `netlink2_client.c` is capable of emulating a local or remote CE.

ACK sender The ACK sender implements the "partial ACKs" ACK strategy. In the context of multiple receivers, one may set the NLM_F_ASTR flag to indicate that the receivers should use a partial ACKs strategy. This strategy intends to avoid the ACK implosion problem: the ACK are broadcast, and the receivers delay randomly the date at which they broadcast the ACK. When a receiver sees the ACK it should later send on the broadcast wire, it simply cancels it. The ACK sender is implemented in `ack_sender.c`.

4.2.3 Running threads

To design the Netlink2 daemon, we were inspired by from the Apache HTTP daemon. Apache 1.3 and previous versions were "process-based". Apache 2.0 introduced a Hybrid policy,[1] meaning that it mixes forking and thread spawning. Figure 4.6 illustrates this. However, this new feature is optional and the default configuration is to handle requests in a non-threaded, pre-forking manner. A

[1]Here we refers only to the Linux version of Apache!

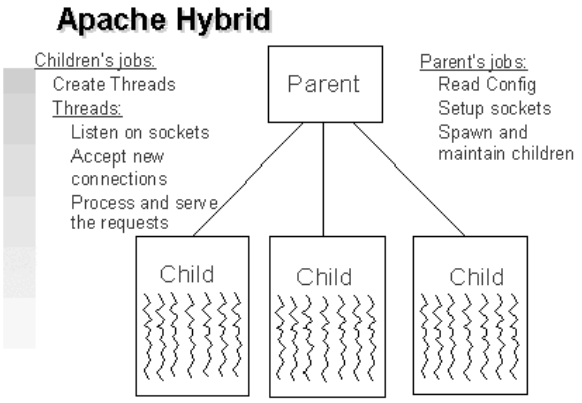


Figure 4.6: Apache 2.0 HTTP daemon: threads.

more complete description of what is done in Apache can be found in [13]. While implementing the Netlink2 daemon, we chose to use threads because we wanted share data. The deferent Netlink2 daemon threads and their interaction are shown in Figure 4.7. The daemon has multiple threads. A "Local TCP

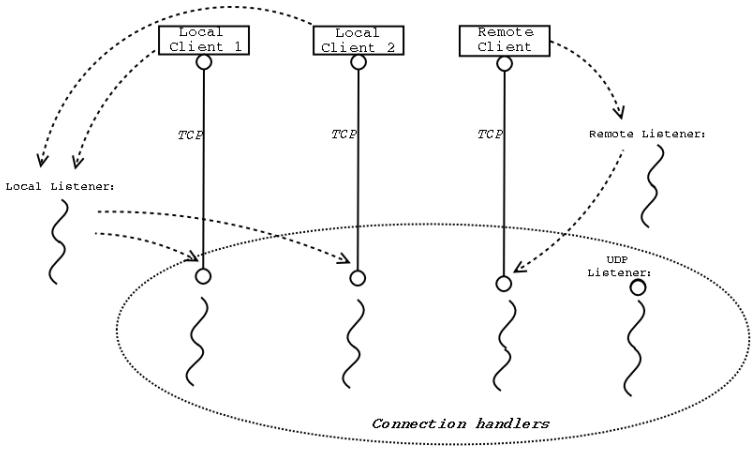


Figure 4.7: Netlink2 daemon: threads

Listener" thread listens for local CE TCP connections. After a connection is accepted, a new thread is spawned to handle the connection. A "Remote TCP Listener" thread listens for remote CE TCP connections. After a connection is accepted, a new thread is spawned to handle the connection. Note that the daemon has no means of knowing whether a connecting CE is local or remote, so the "Local TCP Listener" and "Remote TCP Listener" run on different dedicated ports.

The figure also shows a "UDP listener". Messages coming from the associated port are treated in FIFO order, regardless of their source address. A

thread listening to control messages (i.e. a message modifying the local or global location tables) also listens on a dedicated port but is not represented in the figure.

4.3 Terminology

ForCES only defines the CE and FE logical entities. Looking at Figure 4.4, one may ask which block(s) represent(s) a CE or FE.

CE mapping is the easiest: CEs are represented by local applications, which implement the Netlink2 protocol. If these applications have to be connected to the Netlink2 daemon, it is only for logical address/real address translation.

FE mapping is more difficult: the services offered by the kernel cannot be viewed as FEs, because the protocol used is Netlink (and not Netlink2). The connection point is at daemon level, so the daemon emulates an FE. This FE may contain different components directly addressable by Netlink2 (i.e. they have a Netlink2 logical address). In this case, messages could be addressed to a given FEC but processed at FE level. Note that this is Linux-dependent: one may easily imagine a remote FE that supports Netlink2.

Figure 4.8 gives an example of a mapping between entities we introduced for the Linux implementation and the logical entities defined by the ForCES framework. The figure represents a single NE with multiple CEs/FEs. For reasons of

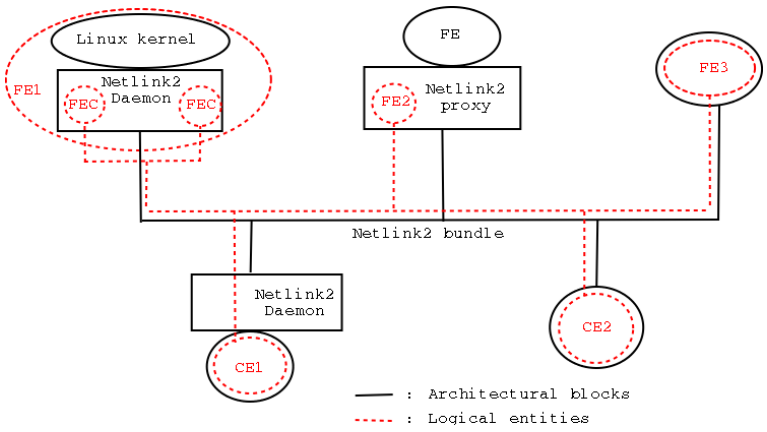


Figure 4.8: Logical entities mapping

simplicity, the role of each CE/FE within the NE is not mentioned. The entities from the Linux implementation view are represented in black. On the FE side, there are three connection points. On the first is a Linux kernel+Netlink2 daemon: the daemon emulates two different FEs. On the second is an FE+Netlink2 proxy: the proxy is necessary because this FE does not handle the Netlink2 protocol. On the third is a single FE, which implements the Netlink2 protocol. On the CE side, there are two connection points. On the first is a CE+ Netlink2 daemon: the daemon is only there to translate logical addresses into real ones. On the second is a single CE, which is able to make this translation itself.

The entities from the logical view are represented in red: they could either map directly with the previous entities or be emulated by a Netlink2 daemon.

Open question: How should the location table synchronization among all daemons be handled?

4.4 Underlying protocol(s)

4.4.1 Requirements

Netlink2 relies on a Layer 4 protocol for transportation. Concerning this underlying protocol, the following guideline is given in [1]:

> ForCES will make use of an existing RFC2914 compliant L4 protocol with adequate reliability, security and congestion control (e.g. TCP, SCTP) for transport purposes.

Even if the Netlink2 header contains fields that could be used to deal with reliability, this task should be left to the underlying protocol. TCP offers all the required features, but may not scale well: imagine an NE composed of a CE controlling thousands of FEs. Having thousands of TCP connections between the CE and each FE would add significant overhead and require expensive processing time at CE. While sending a message to a given logical group, it would be more efficient to use a multicast protocol.

We thus need a *reliable multicast* protocol that should be scalable, have *acceptable overhead*, and be *implemented under Linux*.

4.4.2 State-of-the-art

Reliable multicast has given rise to intensive research during the past few years. Even if many protocols have been proposed, no standard for Unix systems has emerged. One may classify these protocols into four types [7]:

ACK-based protocols ACK-based protocols are an extension of the reliable unicast protocols: each packet sent by the sender (using multicast) is acknowledged by each receiver to ensure reliability. These protocols suffer from the *ACK implosion* problem since the sender must process all acknowledgments for each packet sent.

NACK-based protocols In NACK-based protocols, the receivers send non-acknowledgments (NACK) only when a retransmission is required. By reducing the number of transmissions from the receivers, the ACK implosion problem can be overcome. However, the NACK-based protocols require large buffers and must use additional techniques such as polling to guarantee reliability.

Ring-based protocols In ring-based protocols, a designated site is responsible for acknowledging packets to the source. Receivers send NACKs to the sender when a transmission is required.

Tree-based protocols In tree-based protocols, receivers are grouped into sets where each set contains a leader. The group members send acknowledgments to the group leader, while each group leader summarizes the acknowledgments within its group and sends the summary to the sender.

Also note that it is possible to design protocols making use of several of the previous techniques. A theoretical comparison of these protocol can be found in [8].

The Reliable Multicast Transport Working Group of the IETF has proposed PGM [9] (Pragmatic General Multicast) as a reliable multicast transport protocol. This protocol offers all the required features but it is hard to find a Linux implementation of it. A complete description of existing protocols is beyond the scope of this document. Below are the characteristics of the most promising ones for the Netlink2 appliance: H-RMC [10], PGM [9], LGMP [11], RMTP [14], and NORM [15].

	H-RMC	PGM
Type	NACK-based	Tree-based
Description	A hybrid reliable multicast protocol for the Linux kernel that combines membership state maintenance, NACK-based feedback, updates, probes, and packet retransmissions.	A reliable multicast transport protocol for applications that require ordered or unordered, duplicate-free, multicast data delivery from multiple sources to multiple receivers. PGM guarantees that a receiver in the group either receives all data packets from transmissions and repairs, or is able to detect unrecoverable data packet loss.
Implementation	Kernel network driver	User-space application
API	BSD sockets	Sockets or API
Availability	Kernel 2.2. A port should be made for 2.4	Talarian test drive
Known tests	30 receivers	Thousands of receivers
Comment	Well documented	Good but not free.

	LGMP	RMTP
Type	Tree based	Tree based
Description	A protocol implementation based on the ideas defined by the local group concept (LGC). It supports reliable and semi-reliable transfer of both continuous media and data files. LGMP is based on the principle of subgrouping for local error recovery and for local feedback processing.	A reliable multicast transport protocol for the Internet. RMTP provides sequenced, lossless delivery of a data stream from one sender to a group of receivers.
Implementation	Library	User-space application
API	Library	?
Availability	OK	?
Tests	?	18 receivers on multiple Internet areas
Comment	Receivers get nothing	No implementation found

	NORM
Type	Based on MDP2 (NACK based)
Description	NORM uses a selective, negative acknowledgment (NACK) mechanism for transport reliability and offers additional protocol mechanisms to conduct reliable multicast sessions with limited "a priori" coordination among senders and receivers. A congestion control scheme is specified to allow the NORM protocol to fairly share available network bandwidth with other transport protocols such as the transmission control protocol (TCP).
Implementation	Library
API	Library
Availability	Free
Known tests	?
Comment	No documentation

4.4.3 Retained solution

In the ForCES context, the protocol should be run under low rate error. Moreover, all CEs/FEs in the NE may be connected to the same Ethernet segment or the same bus. Thus, protocols designed to be deployed over the Internet are not well suited. NACK-based protocols such as H-RMC or NORM could be a good choice. However, the protocol performance depends on the quality of the implementation. The choice of a given reliable multicast protocol should be made by the NE designer, taking into account the NE topology and the available connectivity.

Chapter 5

Evaluation

In the previous chapters, Netlink2 and our implementation have been presented. In this chapter, we check some of the Netlink2 features and we present some performance measurements.

5.1 Testbed description

The network on which we conducted our tests was composed of three machines, as described in Figure 5.1. The three machines are PII 250 MHz PCs running

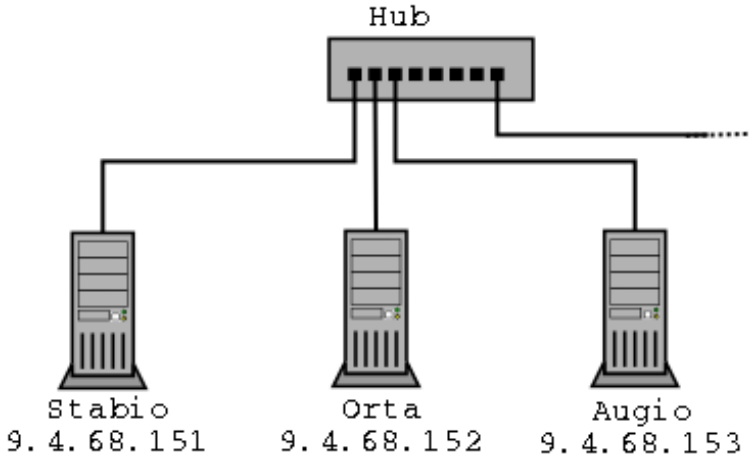


Figure 5.1: testbed description

under Linux (Redhat 7.1 with kernel 2.4.7), connected via a hub and 100 Mb Ethernet cards. As shown in the figure, the IP addresses of stabio, orta, and augio are 9.4.68.151, 9.4.68.152, and 9.4.68.153, respectively. We used IP multicast address 239.128.0.1 port 1503 as necessary.

5.2 Unicast box-oriented groups

The network configuration of this first test is given in Figure 5.2.

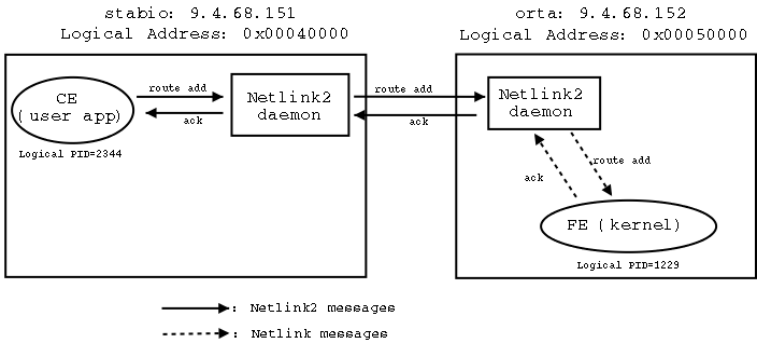


Figure 5.2: Route add using host logical addresses.

In this first test two Netlink2 daemons run at stabio and orta. These daemons communicate using UDP port 1501. The scenario is the following: a CE at stabio wants to add a route on a FE (which is actually the kernel) at orta. The CE also requests an acknowledgment. The CE and the FE are not aware of the network topology. They only know their logical addresses, which are the following: as shown is the picture, stabio has logical address `0x00040000` and orta has logical address `0x00050000`. The CE has logical address `0x0004XXXX` where `XXXX` is its Unix PID, say 2344. The FE has logical address `0x00051229` where 1229 has been assigned statically. The mapping between logical addresses and IP addresses is maintained by global location tables (GLT) in the Netlink2 daemons (cf. Table 5.1).

GLT at stabio

Group	Address
5	9.4.68.152:1501
4	localhost

GLT at orta

Group	Address
4	9.4.68.151:1501
5	localhost

LLT at stabio

Group	logical PID	Socket
4	2344	8

LLT at orta

Group	logical PID	Socket
5	1229	3

Table 5.1: Location tables into Netlink2 daemons.

In this example, the IP addresses associated with groups are unicast. But one may easily replace them by multicast addresses (cf. next section). At stabio, the daemon also knows the logical PID of the CE to which it is connected. At orta, the Netlink2 daemon knows the logical PID of the FE (i.e. the kernel) to which it is connected. This information is stored in local location tables (LLT)

(cf. Table 5.1). Some of the values in these tables are only given for illustration purposes and may change from one experiment to another.

Now, let us analyze the data flow from the CE to the FE and backwards: at the beginning the CE sends a Netlink2 message requiring a route addition. The `NLM_F_ACK` flag is set such that an acknowledgment must be returned on success. The logical addresses fields of the Netlink2 header are the following:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Logical Source Group = 4    |  Logical Source PID = 2344    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Logical Dest Group = 5     |   Logical Dest PID = 1229     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The netlink2 daemon at stabio uses the `Logical Dest Group` value to determine the IP destination address of the message, and forwards it to orta. At orta, the netlink2 daemon then uses the `Logical Dest PID` value to forward the message to the kernel. Having received the acknowledgment from the kernel in the netlink format, the netlink2 daemon at orta generates an acknowledgment in the netlink2 format and sends it back to stabio. The logical addresses fields of this message are the following:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Logical Source Group = 5    |  Logical Source PID = 1229    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Logical Dest Group = 4     |   Logical Dest PID = 2344     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Finally, the netlink2 daemon at stabio receives the acknowledgment and forwards it to the CE.

5.3 Multicast service-oriented groups

In this example we take the previous example and add another host: augio with IP address 9.4.68.153 and logical address `0x00060000`. The network configuration of this second test is given in Figure 5.3.

The two FEs are part of the same logical group, e.g. `NETLINK_ROUTE`, which has been assigned logical group identifier 7. The scenario is similar to the previous test, except the CE at stabio now wants to add a route in the routing table of both FEs (at orta and at augio), requiring an acknowledgment. The CE knows only that the two FEs are members of the `NETLINK_ROUTE` group, so it will use the 0xFFFF logical PID special value to have its messages processed by each group member. The location tables are given in Table 5.2.

Now, let us analyze the data flow from the CE to the FE and backwards: at the beginning the CE sends a Netlink2 message requiring a route addition. The `NLM_F_ACK` flag is set so that an acknowledgment must be returned on success. The logical addresses fields of the Netlink2 header are the following:

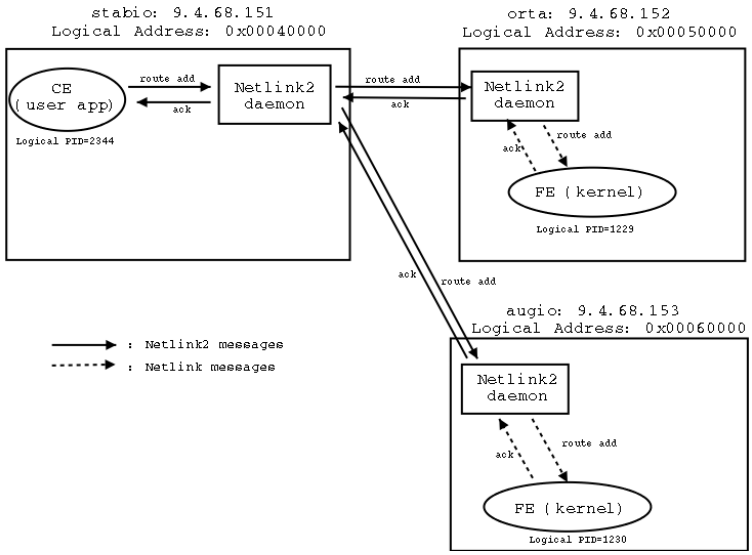


Figure 5.3: Route add using service groups

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Logical Source Group = 4    |  Logical Source PID = 2344    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Logical Dest Group = 7     |   Logical Dest PID = 0xFFFF   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The Netlink2 daemon at stabio uses the `Logical Dest Group` value to determine the IP destination address of the message, and forwards it to the multicast address 239.128.0.1:1503 so that the daemons at orta and augio receive the message. At orta and augio, the Netlink2 daemons check the `Logical Dest PID` value and both process the message because of the `0xFFFF` broadcast value. Having received the acknowledgment from the kernel in the Netlink format, each daemon generates an acknowledgment in the Netlink2 format and sends it back to stabio. The logical addresses fields of this message are the following:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Logical Source Group = 5 or 6  |  Logical Src PID=1229 or 1230 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Logical Dest Group = 4     |    Logical Dest PID = 2344    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Finally, the Netlink2 daemon at stabio receives the acknowledgments and forwards them to the CE.

Note that two acknowledgments are received by the CE. In certain cases, e.g. to avoid the ACK implosion problem, the CE may choose to set the `NLM_F_ASTR` flag so that the acknowledgments are multicasted, where each receiver discards

GLT at stabio

Group	Address
4	localhost
5	9.4.68.152:1501
6	9.4.68.153:1501
7	239.128.0.1:1503

GLT at orta

Group	Address
4	9.4.68.151:1501
5	localhost
6	9.4.68.152:1501
7	239.128.0.1:1503

GLT at augio

Group	Address
4	9.4.68.151:1501
5	9.4.68.152:1501
6	localhost
7	239.128.0.1:1503

LLT at stabio

Group	logical PID	Socket
4	2344	8

LLT at orta

Group	logical PID	Socket
5	1229	3

LLT at augio

Group	logical PID	Socket
6	1230	3

Table 5.2: Location tables into Netlink2 daemons.

its own acknowledgment if it sees an acknowledgment with the same sequence number. To enable this mechanism, the logical destination address should be changed to the following:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Logical Source Group = 5 or 6 | Logical Src PID=1229 or 1230  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Logical Dest Group = 7     |  Logical Dest PID = 0xFFFF    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This situation is illustrated in Figure 5.4: the Netlink2 daemon at orta is the first to multicast an acknowledgment, whereas the Netlink2 daemon at augio silently discards its own acknowledgment.

5.4 Throughput analysis

To analyze the performance of the protocol, we take again the configuration from the first test, and request multiple route addition (with acknowledgment) and route deletion (with acknowledgment), from 10 to 50000 times. Note that before initiating a new transaction, the CE waits for the acknowledgment of the previous one. This test is depicted in Figure 5.5.

We repeated the test three times. For these three simulations, Figure 5.6 represents the execution time as a function of the number of messages. We see that the execution time is linear as a function of the number of messages. The average rate is approximately 1000 route modifications / sec.

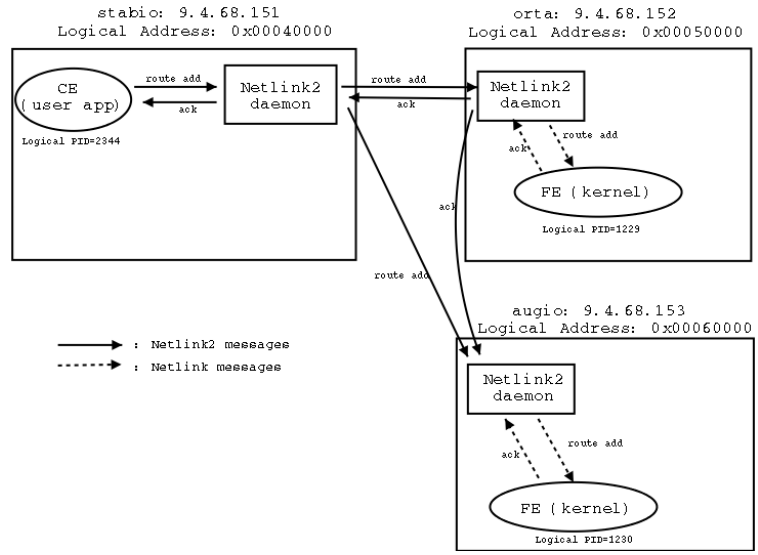


Figure 5.4: Route add using service groups with another ACK strategy.

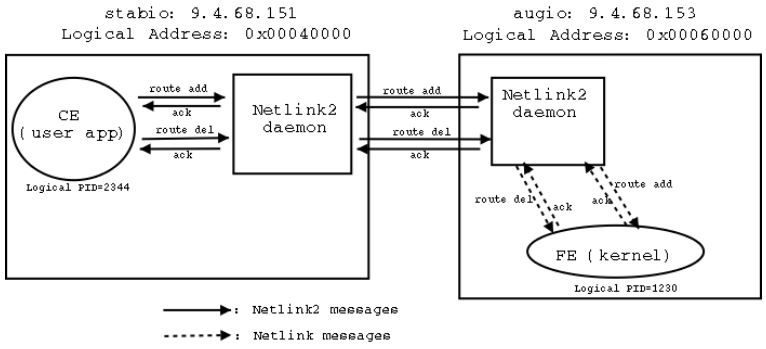


Figure 5.5: Performance analysis: messages sent.

5.5 Performance boost

To improve the previous results we first used a “brute force” approach by changing the testbed to have much more powerful servers (Xeon 2.4 GHz) connected with 1 GB/s Ethernet links, as described in Figure 5.7.

The test scenario is exactly the same as the previous one. The performances are approximatively 5 times better, as shown in Figure 5.8. Looking at the end of the curve, we modified 1,000,000 entries in the routing table (500,000 route addition + 500,000 route suppression) with acknowledgment within 150 seconds, which means more than 6500 route modifications per second. As in the previous test, the CE waits for the acknowledgment of the previous transaction before initiating a new one.

It should be noted that one UDP packet containing one Netlink2 header is required for each single route modification. To reduce this overhead, we intro-

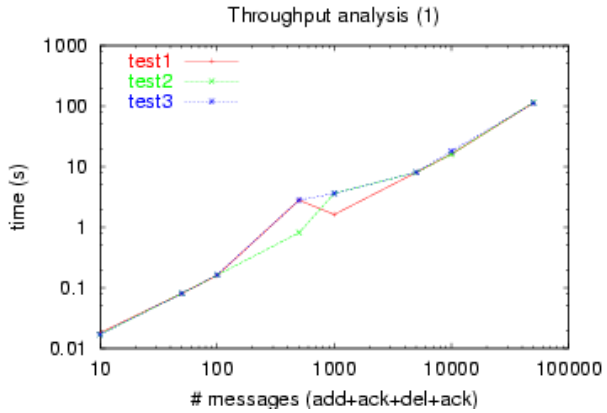


Figure 5.6: Performance analysis (1): execution time.

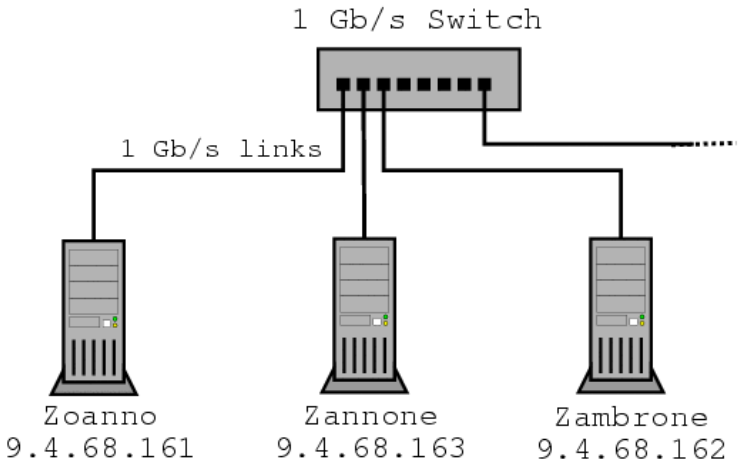


Figure 5.7: Testbed 2.

duced the `NLM_F_BATCH` Netlink2 extended flag. This flag allows the IP services requests to be batched using a single Netlink2 header. If an acknowledgment is required, an "all-or-nothing" policy is applied: this means that an acknowledgment is sent back only if all transactions have been accepted. If not, all of them should be canceled.

From the Netlink2 daemon view, as many Netlink messages should be generated as different IP Services data blocks. If an acknowledgment is required, the daemon should wait until the last Netlink message is acknowledged before sending back a Netlink2 acknowledgment. Otherwise, all the previous modifications should be canceled. This process is illustrated in Figure 5.9.

Note that the type of the message (e.g. `RTM_NEWROUTE`) is normally specified in the `type` field of the Netlink2 header, and not in the IP service part. It implies that when using batching, all the IP service blocks should be of the

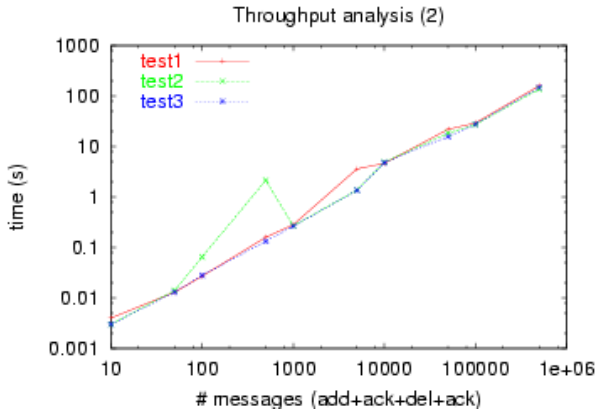


Figure 5.8: Performance analysis (2): execution time.

same type.

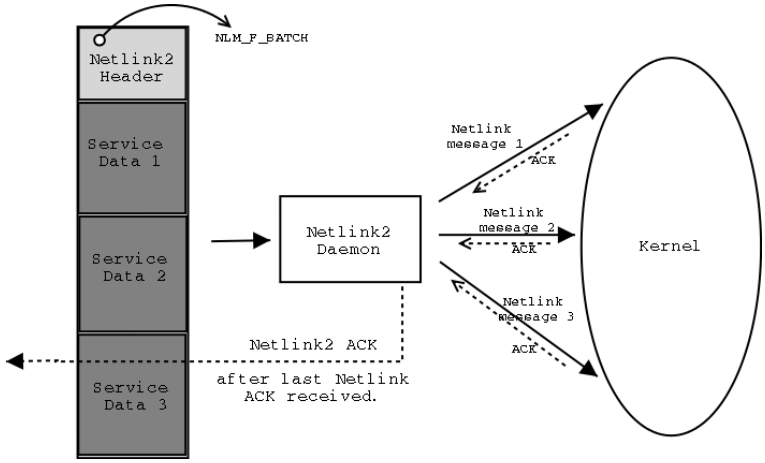


Figure 5.9: Batching inside Netlink2 messages.

We conducted series of tests with different batch values, i.e. with netlink2 messages containing various number of IP service blocks. It turned out that the best performances were with 10 IP service blocks. In this case, the results were 4 times better than in the previous scenario (cf. Figure 5.10), and ended up with 1,000,000 route updates within less than 30 seconds, which means around **35,000 route updates / sec**. The batching method we used here corresponds to the 3d method introduced in Chapter 3. Some tests have also been conducted using method 4, which produced similar results. The 3d method performed slightly better. The performances ratio was equal to the ratio between the message sizes.

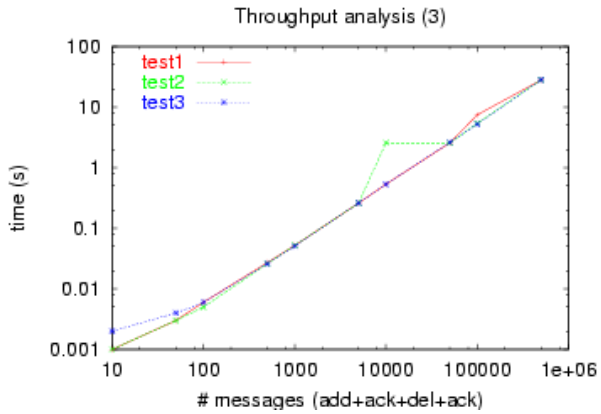


Figure 5.10: Performance analysis (3): execution time.

Chapter 6

Future Work

In this chapter, we present some ideas that should be discussed at length before being integrated into the Netlink2 draft. We also show some possible ways of using Netlink2 in Linux-based network systems.

6.1 Message format

In the previous chapter, we added the `NLM_F_BATCH` extended flag to batch some `RTMNEWROUTE` commands. This flag indicated that the Netlink2 message contained multiple commands **of the same type**. This, however, was merely a "hack" because the receiving daemon already knew that the IP services templates were all 28 bytes long. The problem comes from the fact that the length, type, and flags fields of the Netlink2 headers actually refer to a given IP service template. To allow multiple IP service templates in a given Netlink2 message, one may think of moving these fields from the Netlink2 header to the IP service template.

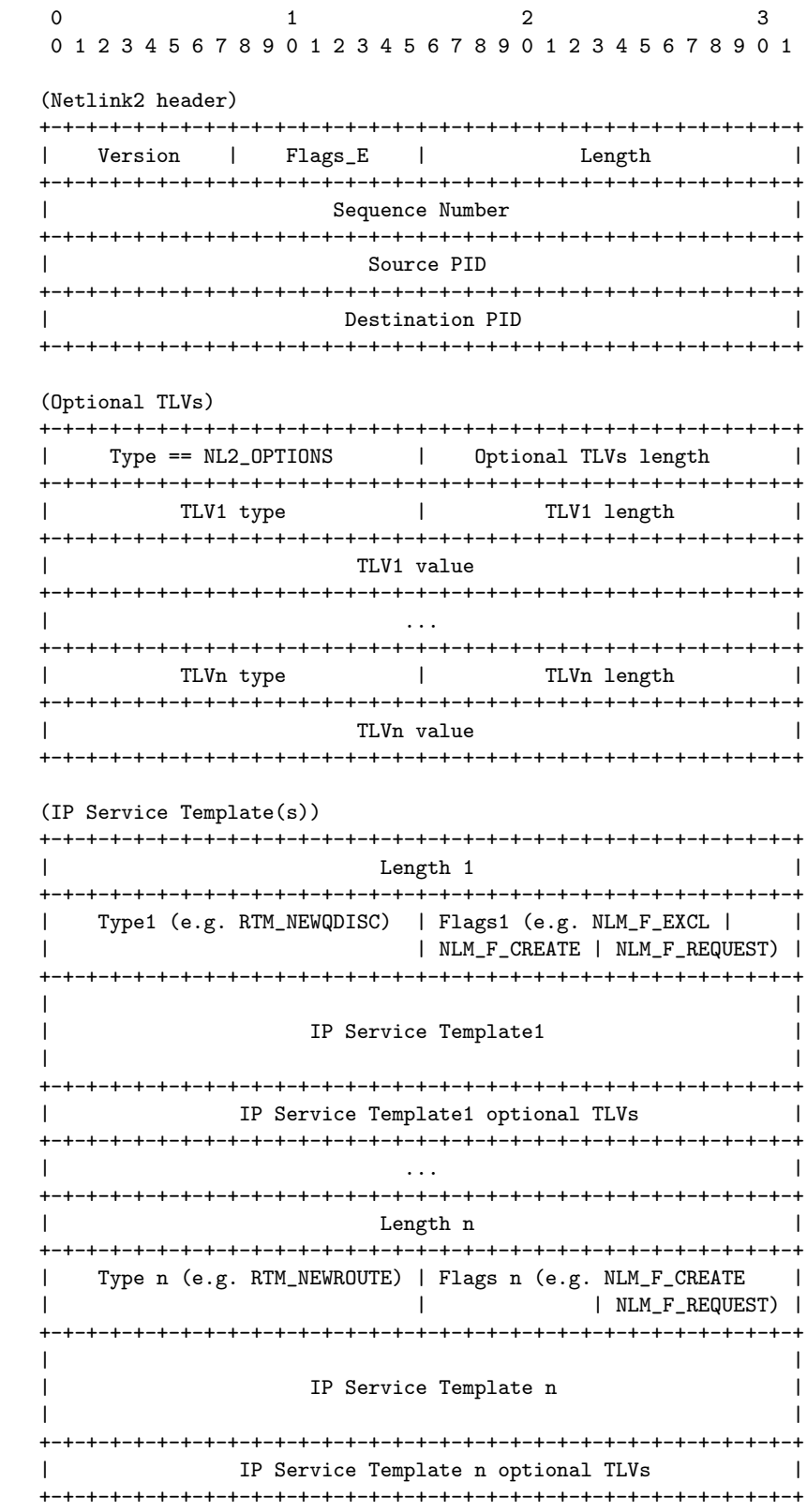
Another issue is the parsing of optional TLVs. If a Netlink2 message contains multiple optional TLVs, a parser would have no means to separate them from the beginning of the IP service template because the length field of the Netlink2 header only indicates the length of the entire message. One may think of encapsulating all the optional TLVs in another one as indicative of the total length of the optional TLVs. The parser would:

1. Read the Netlink2 header, which indicates the total length of the message, and determine whether optional TLVs are in use.

2. If optional TLVs are in use, read the first TLV indicating the length of all optional TLVs, then each TLV up to this length.

3. Read the IP service templates.

After putting together the two previous ideas, the format below could be suggested:


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
(Netlink2 header)
Version
Flags_E
Length
Sequence Number
Source PID
Destination PID
(Optional TLVs)
Type == NL2_OPTIONS
Optional TLVs length
TLV1 type
TLV1 length
TLV1 value
...
TLVn type
TLVn length
TLVn value
(IP Service Template(s))
Length 1
Type1 (e.g. RTM_NEWQDISC)
Flags1 (e.g. NLM_F_EXCL | NLM_F_CREATE | NLM_F_REQUEST)
IP Service Template1
IP Service Template1 optional TLVs
...
Length n
Type n (e.g. RTM_NEWROUTE)
Flags n (e.g. NLM_F_CREATE | NLM_F_REQUEST)
IP Service Template n
IP Service Template n optional TLVs

6.2 Multiple-ACK format

In the previous chapter we used an all-or-nothing batching policy: we kept the ACK format and changed the semantics so that a given ACK acknowledges all the previous messages. One may think of introducing a new multiple-ACK format similar to the one used in NFSv4 [12]:

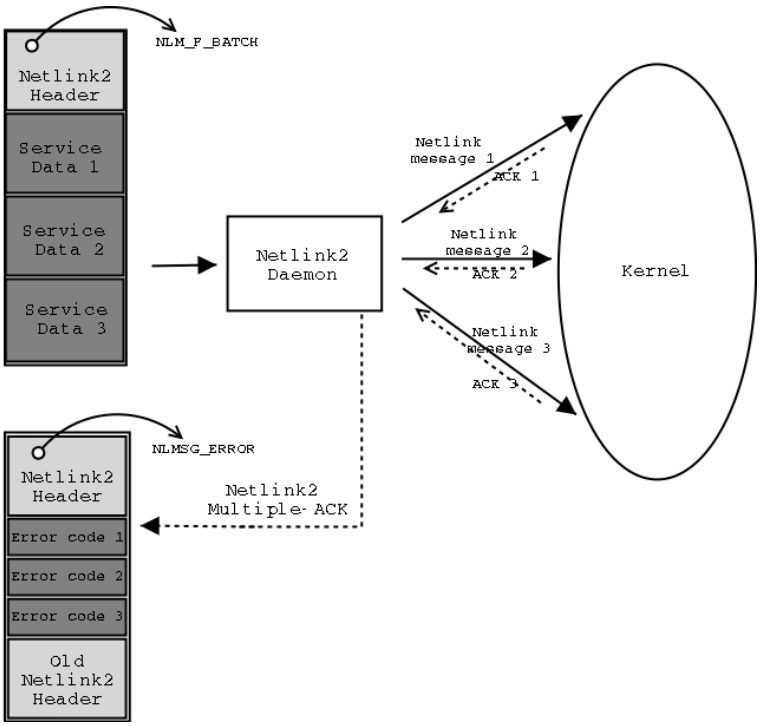


Figure 6.1: Multiple-ACK message format

6.3 Kernel patch

Many applications including iproute2 or tc [16] have already been written for Netlink. It would constitute a lot of work to rewrite them so that they would be Netlink2-compliant. A work-around would consist of making the remote FEs appear in the kernel as local devices. The idea is that Netlink will still be used for communication between the user space processes and the Linux kernel. There would be remote network devices of FEs showing up in the CE. So when an FE joins, its netdevs will be displayed locally. When a local or remote netdev has to be configured, Netlink is still used because that is what most tools use. When it gets to the kernel, if the device being referenced is remote, then an exception handling happens back to user space where the Netlink2 daemon will transmit using Netlink2 to the remote FE. Communication happens (if Linux is on that side) via Netlink to the kernel. If all goes well and an ACK was requested the

response goes back to user space where it gets shipped to the Netlink2 daemon via Netlink2. Upon receiving the Netlink2 message, the Netlink2 daemon sends it to the kernel, which may compute it before sending it to user space.[1] This process is illustrated in Figure 6.2.

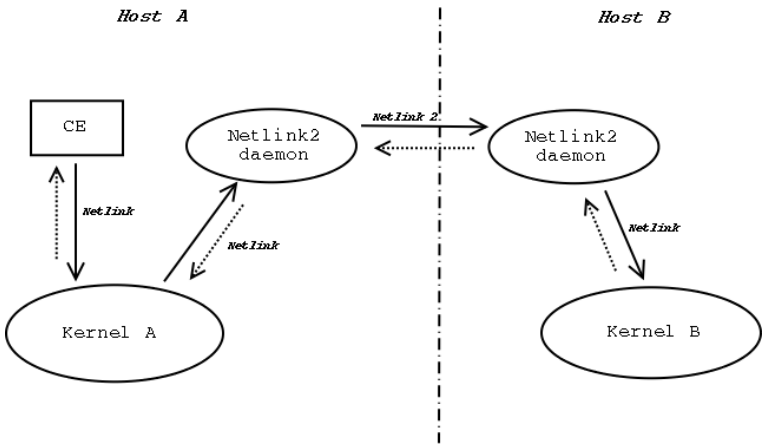


Figure 6.2: Netlink2 kernel patch

6.4 Have Netlink2 kernel-native?

Netlink2 received good feedback at the 2003 Kernel Developers Summit (July 21 and 22 in Ottawa). It would be possible to have it native. It seems to be a lot of work to port existing Netlink applications to Netlink2. But most of these applications use upper-layer libraries to interface with Netlink (e.g. iproute2 uses libnetlink). Moreover, Netlink2 does NOT introduce any changes in the IP service templates, which represent the biggest part of Netlink. For now, the code we release is compatible with the brand new `linux-2.6.0-test4` Linux kernel.

Whatever new features Netlink2 includes, the code modularity of our implementation will enable both an easy kernel integration and future tests (e.g. using the NS-2 network simulator).

[1] This new way of using Netlink2 can be view as an alternate architecture combining the ones we detailed in chapter 4

– Conclusion –

In this paper, we introduced ForCES and Netlink separately. Then we explained what make Netlink a good ForCES protocol candidate, and showed how the Netlink2 protocol extends Netlink to make it more compliant with the ForCES requirements.

Having presented our Linux Netlink2 implementation, we demonstrate the feasibility of some of the new Netlink2 features. Our main contributions to Netlink2 are the new addressing semantics, the acknowledgment strategy, and the study of various batching methods.

We tested the above features with a high-performance objective in mind. The results we obtained showed that Netlink2 may be used in the context of high-speed processing routers.

Netlink2's assets are its high flexibility as well as years of Netlink testing in the open-source community.

Access to the Linux Netlink code enabled us to gain a complete understanding of its internals, and to build Netlink2 on solid bases. IBM has chosen to release the Netlink2 code. This would allow for future improvements and keep Netlink2 ForCES-compliant.

Bibliography

[1] L. Yang, R. Dantu, T. Anderson, *Forwarding and Control Element Separation (ForCES) Framework*, `draft-ietf-forces-framework-04.txt`, December 2002

[2] H. Khosravi, T. Anderson, *Requirements for Separation of IP Control and Forwarding*, `draft-ietf-forces-requirements-09.txt`, May 2003

[3] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, *Netlink as an IP Services Protocol*, `draft-ietf-forces-netlink-04.txt`, December 2002

[4] J. Salim, R. Haas, *Netlink2 as ForCES protocol*, `draft-jhsrha-forces-netlink2-00.txt`, December 2002

[5] OMG Inc., X/Open Co Ltd., *The Common Object Request Broker: Architecture and Specification*, **CORBA V2.2**, February 1998, chap. 11

[6] J. Maloy, *Telecom Inter Process Communication*, **Ericsson**, January 2003

[7] R. Lane, *A Comprehensive Study of Reliable Multicast Protocols over Ethernet-Connected Networks*, **Florida State University**, November 2000

[8] B. Levine, *A Comparison of Known Classes of Reliable Multicast Protocols*, **University of California, Santa Cruz**, June 1996

[9] T. Speakman, J. Crowcroft, J. Gemmell, D. Farinacci, S. Lin, D. Leshchiner, M. Luby, T. Montgomery, L. Rizzo, A. Tweedly, N. Bhaskar, R. Edmonstone, R. Sumanasekera, L. Vicisano, *PGM Reliable Transport Protocol Specification*, **RFC 3208**, December 2001

[10] P. McKinley, R. Rao, R. Wright, *H-RMC: A Hybrid Riliable Multicast Protocol for the Linux Kernel*, **IEEE SC99**, November 1999

[11] M. Hofmann, *Local Group based Multicast Protocol (LGMP)*, `http://hofmann.us/lgmp/lgmp.html`, 1999

[12] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, *Network File System (NFS) version 4 Protocol*, **RFC 3530**, April 2003

[13] Apache HTTP Server Documentation Project, *Multi-Processing Modules (MPMs)*, `http://httpd.apache.org/docs-2.0/mpm.html`

[14] John C. Lin, Sanjoy Paul, *RMTP: A Reliable Multicast Transport Protocol*, **IEEE INFOCOM 96**, March 1996

[15] B. Adamson, C.Bormann, M.Handley, J. Macker, *NACK-Oriented Reliable Multicast Protocol (NORM)*, `draft-ietf-rmt-pi-norm-07.txt`, June 2003

[16] Mark Lamb, *iproute2+tc notes*, `http://snafu.freedom.org/linux2.2/iproute-notes.html`, 1999