Network Interface Grouping in the Linux Kernel

Vlad Dogaru, Octavian Purdilă, Nicolae Țăpuș
Automatic Control and Computers Faculty
Politehnica University of Bucharest
Emails: {vlad.dogaru,tavi,nicolae.tapus}@cs.pub.ro

***Abstract*—The Linux kernel is a prime field for implementing experiments, many of which are related to networking. In this work we tackle a part of the network scalability issues of the kernel. More specifically, we devise a means of efficiently manipulating large numbers of network interfaces. Currently, the only approach to handling multiple interfaces is through repeated userspace calls, which add significant overhead. We propose the concept of network device groups, which are completely transparent to the majority of the networking subsystem. These serve for simple manipulation of large number of interfaces through a single userspace command, which greatly improves responsiveness. Changes are proposed both in kernel and userspace, using the iproute2 software package as support. Improvements in speed are visible, with changing parameters of thousands of interfaces taking less than a second, as opposed to over 10 seconds using a conventional approach. Implementation is simple, unintrusive and, most importantly, user-defined. This leaves room for future improvements which use the group infrastructure, some of which have already been proposed by third parties.**

***Index Terms*—Linux, kernel, network, device, scalability, grouping, iproute2**

I. Introduction

Computer networking poses interesting problems, both in the design of new protocols and techniques and in improving the scalability of existing concepts. It is particularly interesting to see what happens when the quantity of networking equipment, rather than the number of clients, grows. More specifically, we study the performance of the Linux kernel when dealing with thousands or tens of thousands of network interfaces. Because these cannot be physically fitted into a single machine, virtual interfaces are used.

Linux already offers a kernel module that emulates network interfaces. Before we can measure performance of these interfaces, a means of efficiently manipulating them is needed. The simple way to do this would be to repeatedly use an administrative tool for every interface. However, when dealing with a large number of interfaces, this proves inefficient. The creation of a new process for each of the interfaces offsets the useful work which is done. Moreover, even if the entire work could be done in a single process, communicating to the kernel that each interface is to be modified becomes a bottleneck. What is needed is an kernel API for specifying that a number of interfaces need to be modified in an identical manner. This includes activating or deactivating the interfaces, setting their MTU and other link-level parameters.

We propose a solution that introduces the concept of network interface group. A group is nothing but a simple integer tag; no relationship is implied between the devices in the same group. Their parameters can be modified either individually or collectively, thus no flexibility is lost. Because there is no intrinsic relationship between members of a group, group membership and policy is entirely defined by the administrator, not forced by the kernel. In many ways, the concept of an interface group is similar to that of a packet mark. It is used exclusively from userspace and its meaning is flexible.

By introducing network interface grouping, both of the factors slowing down interface manipulation are addressed. The administrator can use a single command to modify the parameters of a large number of interfaces; thus, scheduler overhead is eliminated. Further, userspace can address all the members of a group using a single request to the kernel, thus minimizing message transfers and system calls.

The solution described has been fully implemented using Linux and the iproute2 software package as support. The changes to existing code are unintrusive, with respect to both performance and code complexity. The code has undergone several rounds of review from the community and is, at the time of this writing, pending inclusion in the upstream kernel. This enables continued development in the area of network interface grouping and collective configuration. It also provides guaranteed maintenance from the community.

Performance tests feature a virtual machine setup. This enables harmless recovery from kernel errors and contained testing. Moreover, it speeds up the testing process because rebooting is low-cost. KVM was chosen for the task because, except for the live boot media, it demands no additional files or daemons. Additionally, KVM incurs little performance penalty on the host system (provided the host hardware supports the Intel VT-x or AMD-V extensions) in comparison to other virtual machines or emulators. Userspace utilities are provided by the busybox suite; this enables a full array of Linux utilities, with minimal dependencies.

Tests have been made featuring the `dummy` module of the Linux kernel. This module enables the user to add as many interfaces as they like by providing a parameter when inserting the module into the kernel. Because these interfaces have no physical meaning and are not critical to virtual machine operation, they are easily the target of tests involving batch modifications. We have tested two setups: one in which the user repeatedly calls the iproute2 `ip` command for each interface (using a shell script); the other involves using our proposed solution and changing the parameters of an entire group of interfaces using a single command. Test results are consistently orders of magnitude in favor of the latter technique, particularly for larger numbers of interfaces.

II. Related Work

It is interesting to note that, while much research has gone into Linux network scalability, there is little interest in *configuration-time* optimization. Most progress is made towards *runtime* performance, be it packet handling [4] or routing performance [1].

III. Architecture

Network interface grouping targets the Linux operating system and, as such, must adhere to the kernel API. Because the group infrastructure proposed is simple, no additional data structures or complex algorithms were needed. Changes are entirely in existing source code files, so no modification was brought to the build system.

Although Linux supports loadable modules, changes were deemed sufficiently unintrusive not to warrant the modularity of the interface grouping routines. The only core data structure that was modified is `struct net_device`, which is by no means a performance-critical element. Thus, cache performance was not a problem and the necessary modifications could be made without resorting to memory profiling.

Kernel-side, the contributions are located in the `net_device` structure and in the API that the kernel provides to manipulate it. Historically, there have been two choices for modifying network parameters. The first interface is based on the `ioctl` system call. This involves creating a special descriptor and using successive `ioctl` calls to modify kernel parameters. It is used by the `ifconfig` utility, but both the API and the userspace components are currently deprecated.

The alternative is Netlink [3], a socket mechanism for interprocess communication. Netlink can serve as a means of communication between userspace and the kernel, as well as between two processes. Unlike other sockets, however, Netlink only works for a single host. Netlink provides multiple socket families, corresponding to different operating system parameters ranging from the neighboring system (ARP) to firewalling (Netfilter) or IPSec. The one of most interest to us is `NETLINK_ROUTE`, which is used to query and change routing and link information. This socket family is used by routing protocol implementations such as Quagga, as well as the `iproute2`, which we chose as support for our userspace modifications. Figure 1 shows how Netlink is used by `iproute2`.

Meant as a modern network configuration tool, the `iproute2` utility is developed in close relation to the Linux kernel. The development teams are closely related and patches are sent to the same mailing list for both the kernel networking subsystem and `iproute2`. As such, it was the natural choice for implementing userspace support for network interface groups. `iproute2` is fairly modular in architecture, with components for link-level, IP addressing, routing, tunneling and IPSec. Network group logic was added to the lowest-level component, informally named `iplink`.

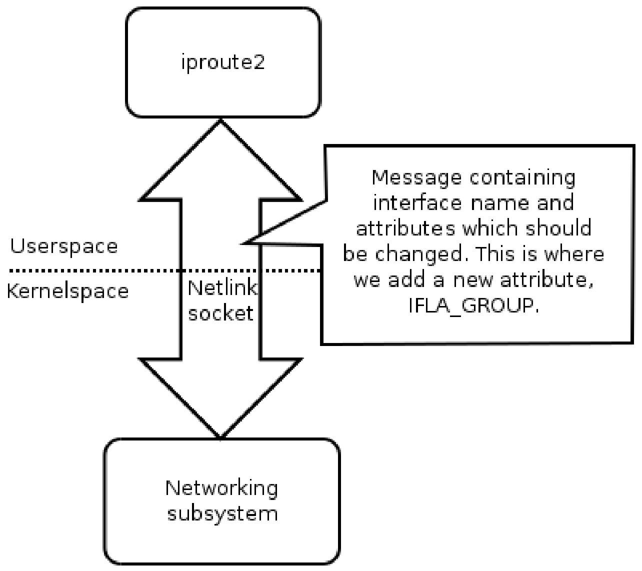
iproute2
Message containing interface name and attributes which should be changed. This is where we add a new attribute, IFLA_GROUP.
Userspace
Kernelspace
Netlink socket
Networking subsystem

Figure 1. `iproute2`, a userspace tool, uses a Netlink socket to communicate with the kernel. We add new attributes to the message so that Netlink will be group-aware.

IV. Implementation

To have a properly functioning, albeit minimal, device group infrastructure, the following aspects need to be addresses:

1) A mechanism for grouping devices together.
2) A means of exporting group information to userspace, as well as a way to filter certain groups. The user can then view devices from a single group.
3) The possibility to change the group a device belongs to. We generally avoid referring to this action as *moving*, because no copying is required, as we shall see.
4) Finally, a way of specifying that a change affects an entire group and not just a single device.

The `net_device` structure has been modified to include group information in the form of an integer field. Thus, each network device belongs to a single group, there are no overlappings between groups. By default, all devices are created in group 0. The network device structure is not critical to performance (for instance, it is not used during routing), so modifying it does not raise cache efficiency problems. Nor is this structure part of intricate lists and other data, so it is fairly easy to understand the purpose of all its fields.

Group information is exported from kernel to userspace via route Netlink. The message format used in communication consists of a header followed by a variable number of attributes. Every time information is sent to the user, typically as a response to a `RTM_GETLINK` message, the group number is also packed in the message. Because of the flexible nature of Netlink messages, adding this information was trivial, in the form of a new attribute type. In turn, userspace unpacks the message information and interprets the attributes. The `list` operation in `iproute2` has been modified to accept a supplementary argument, the `devgroup` keyword, which filters devices that belong to other groups. The user can now choose information they wish to see by filtering the relevant group.

The same Netlink attribute, called `IFLA_GROUP`, is used to set the group of a device. The sole difference is the direction of the message – user to kernel – and the type of message sent, typically `RTM_NEWLINK`. This type of message also contains the interface to operate on, specified either by the internal index number or the device name. Kernel-side modifications need concern with locking mechanisms, since the entire Netlink system is already protected by a lock. By ensuring that, kernel programmers have enabled future contributions that are less prone to bugs. Changing the group of a device means simply unpacking the message and assigning the requested value to the group field of the network device structure. No costly copying is involved.

Finally, the target of the whole infrastructure is being able to batch device parameter modifications. The aim is to be able, using a single Netlink message, to modify many network interfaces. Initially, another attribute was added, `IFLA_-FILTERGROUP`. If this was specified in a user-originated message, the modifications described by the message were to be made on all the interfaces in a group, not just on a single one. However, the community expressed concern over adding two attributes for a relatively simple task, and a later iteration of the patch set uses a small hack and a single attribute. More precisely, if:

1) a userspace request has a negative interface identifier; normally, interface identifiers in Linux start at 1 and are all positive;
2) **and** it has no interface name specified;
3) **and** it contains a group attribute

then the modifications are to be made on the entire group, not just a single network device. By using the same attribute both for setting parameters and filtering devices, we eliminate the possibility of changing the group of an entire group of devices in a single request. But this was deemed an unnecessary corner case by the community in contrast to the API bloat it would introduce.

V. Experimental Setup

The initial tests we ran for the patch set have run into a tricky problem. We could not efficiently test our changes using many interfaces because of speed issues. More precisely, creating even only 1024 interfaces took around a minute. Because of the repetitive nature of testing our patch, we took the decision to test the setup without including `sysfs` in the kernel build. The `sysfs` filesystem is a modern way of configuring kernel parameters through a file-like interface, but, in our case, it was an important obstacle to performance. Without `sysfs`, new interface creation dropped from 55 seconds to 7 seconds, for 1024 interfaces. This is an important improvement for testing, not a real optimization.

Soon after, another performance obstacle had to be addressed, again not directly. VMWare, which is the de facto virtual machine in our environment, had a high overhead, not in terms of memory or processor consumption, but with respect to (re)booting and parameter modification. We migrated to the Kernel-Based Virtual Machine [2], which provides simple command-line manipulation, faster startup times, and generally behaves more like a simple program; for instance, when closing the KVM window, the virtual machine is stopped without further user queries. This might corrupt the machines disks, but we were not concerned about that.

Another aspect of the testing setup is the lack of a hard disk for the virtual machine. Because the kernel is the main part being tested and modified, storing it on a hard drive and copying it through the network to the virtual machine made little sense. Instead, we booted the machine from a live medium, generated by an in-house script. The script copies a few essential programs into an ISO image and bundles them with a kernel and an initial ramdisk. Aside from the iproute2 suite, scripts to create necessary entries in `/dev` and a minimal `passwd` file are provided. Manual creation of device files is necessary because the setup does not include Udev.

Basic tools are provided by Busybox [5]. The Busybox suite, dubbed *a swiss army knife for Linux*, is a collection of utilities designed to replace GNU coreutils, with a focus on small binary size and minimalism. Busybox consists of a single executable which encompasses the functionality of many utilities, from filesystem to process and text manipulation. It even contains a small version of `vi`. When launched, the Busybox executable looks at either its first parameter or the name it has at launch; this dictates its behaviour. So, launching `busybox ls` would trigger similar functionality to the `ls` command, while launching the Busybox executable using a link (symbolic or hard) named `vi` will have the effect of a `vi` clone. Busybox can be statically linked and it often is, but we chose not to do so. The live medium which we use to boot already contains the needed libraries. Finally, it should be noted that Busybox modularity is exemplary; it has a configuration menu in the spirit of the Linux kernel, from which the user can choose the modules which should be included. Configuration detail is surprisingly high – there are, for instance, options for more advanced `vi` features.

The final live image for the virtual machine is generated with `genisoimage`. The image is made bootable by including `isolinux`, a tool used for booting Linux from ISO images. The final image – containing a minimal kernel, an initial ramdisk, iproute2, Busybox, isolinux and the necessary libraries – amounts to little over 13 megabytes. Booting the virtual machine takes under 5 seconds, which is a boon to a generally error-prone testing cycle.

VI. Scenarios and Results

The sole method of testing the improvements is measuring the time needed to configure interfaces with and without using network device groups. We used an emerging technology called `perf` to profile the operations executed when bringing interfaces up, down, and when changing the Maximum Transfer Unit (MTU) of a device group. `perf` is a tool for manipulating the hardware performance counters of the processors, similar in fashion to `oprofile`. The advantage of `perf` is that development is synchronized with the kernel, so communication between userspace and privileged mode is always up to date. Furthermore, user interface is much friendlier, with less complicated syntax and intuitive defaults.

Profiling has yielded expected results, but a bit disproportionate. It is clear that, by using device groups, modifying device parameters has a higher throughput, with proportionately less time spent in userspace and more in kernel mode. This is especially true because, when using device groups, we only create a single process; previously, a process was created for each interface. Even if the device manipulation was a straightforward action, it is important to note that creating a process, loading a new program into the address space, scheduling the process to run, and finally waiting for proper termination and cleanup are all very costly operations.

Indeed, operating system literature recognizes that performance critical applications should not create a large number of processes. This stems from the significant cost of a context switch. On systems which support virtual memory, which are virtually all systems nowadays, a specialized cache memory is employed in order to speed up virtual-to-physical address translation. The Translation Lookaside Buffer, or TLB, is an associative memory with pairs of virtual and physical addresses. When a context switch is triggered, the address spaces need to be changed, which in turn causes the flushing of all TLB entries.

Even factoring out userspace time spent, the total time spent in kernel mode is also significantly lower when using interface groups. Previously, a Netlink request was constructed for each interface which needed to be modified. This was an iterative process, constructing a message, then sending it, waiting for a reply, all this being done for each interface. Sending the request and receiving the response are translated into system calls, which are another recognized source of latency in the context of operating systems. Interface groups solve this problem by necessitating a single request for any number of interfaces, as long as they are in the same group. System call overhead is thus greatly reduced.

What is, in our view, remarkable about the proposed solution is that it actually scales better the more interfaces are involved. Without device grouping, the number of processes created and system calls is a multiple of the number of interface manipulated. Conversely, when employing interface grouping, a single process is created and a single Netlink request is made, regardless of the interface count. This leads to 50 times better performance with 1024 interfaces, but 65 times better with 2048 interfaces. We deem this an encouraging start into scalable network configuration.

The experiment results are shown in Table I. We timed the operation of changing the MTU of a number of interfaces, both traditionally (columns labeled *No group*) and using the new group interface (columns labeled *Group*). Because `sysfs` introduces a significant overhead, the experiments have been run on two different kernel configurations: one includes `sysfs` and the other does not.

	Without sysfs		With sysfs	
Interfaces	Group	No group	Group	No group
128	0.01	0.49	0.01	0.53
256	0.02	1.11	0.03	1.15
512	0.05	2.57	0.06	2.59
1024	0.17	7.02	0.20	7.51
2048	0.32	21.74	0.36	23.05

Table I
TIMING OF CHANGING INTERFACE MAXIMUM TRANSFER UNIT WITH AND WITHOUT INTERFACE GROUPING. ALL TIMES ARE IN SECONDS.

VII. CONCLUSION AND FURTHER WORK

The infrastructure implemented so far has proved to be scalable and generally accepted by the Linux kernel community. The patch set has already been through a feedback iteration, and is currently pending a second one. Valuable lessons in both design and communication with the kernel ecosystem have been learned in the process. Particularly, we have seen a surprising amount of suggestions for further improvement.

Although the patch, as it stands, is simple and generic, there have been suggestions to transform it into a hierarchical approach. The user would be able to define a tree-like structure, where a group can contain devices and other groups. Modifying a group would then have the semantic of modifying its devices and all the groups contained in it. A slight disadvantage of this would be little applicability, which is generally a sign of over-engineering. With respect to code, arborescent groups would require a separate group data structure, as opposed to a simple filed in the network device structure.

Another direction towards which the infrastructure can be developed concerns the handling of related devices, such as PPPoE (Point-to-Point Protocol over Ethernet) virtual devices and their supporting Ethernet device. It has been argued that group membership should be inherited, thus creating large groups without the need for explicit group changing. For instance, a PPP (Point-to-Point Protocol) server would add its relevant interface in group 42, then create hudreds or thousands of PPPoE interfaces having it as layer 2 support. These interfaces are implicitly created in group 42 and easily modifiable using a single command.

Because group membership is internally represented by a simple data type, an integer, there is no reason it should not be exported to userspace as such. The `sysfs` virtual filesystem is the current manner for the Linux kernel to expose information. A relatively simple addition would be the presence of the group tag in the corresponding `sysfs` directory of each device. An authorized user can then change the group of a device by using either `iproute2` or `sysfs`.

One particular critique of the grouping infrastructure was that it is not particularly user friendly, specifically that numbers have little meaning to users. Handling strings in kernel space is usually frowned upon when fixed-width tags can do better, so another solution is needed. Luckily, `iproute2` already has a convention in place. In the `/etc/iproute2` directory, the user can store associations between kernel numbers and userspace significant names. We intend to add support for such an association. This remains within the initial self-imposed restriction of the implementation being simple in the kernel, but flexible in userspace.

Finally, the real scope of this endeavour is to improve the scalability of configurating network parameters. So far, we have treated part of the second level of the stack, but much work needs to be done for a fully functional interface. The most difficult problem we face is assigning relevant addresses to interfaces. That cannot be achieved with the current implementation, as identical MAC addresses would be next to useless for any setup. What needs to be implemented is incremental assigning of layer 2 and 3 addresses. One cannot efficiently do that in userspace, because it would bring about the problem of system call saturation discussed earlier. So the issue remains open, despite it being one of the key points of the original endeavour.

All in all, interface grouping is a simple, flexible interface that attempts to open the way to scalable network device configuration in the Linux kernel. Community feedback to it was positive, which means we can expect to see it in a future kernel release. Performance improvements are significant, even in the simple case of interface activating and deactivating. Improvement ideas abund, both in extending the existing structure and in improving the essential idea.

ACKNOWLEDGMENT

The authors would like to thank Jamal Hadi Salim for his initial implementation suggestion and constant feedback. Also, Răzvan Deaconescu and Laura Gheorghe provided valuable feedback for this article.

REFERENCES

[1] O. Hagsand, R. Olsson, and B. Gördén. Towards 10gb/s open-source routing. *Linux Kongress*, 2008.
[2] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: The linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.
[3] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549 (Informational), July 2003.
[4] J. H. Salim. When NAPI comes to town. Technical report, 2005.
[5] N. Wells. Busybox: a swiss army knife for Linux. *Linux Journal*, October 2000.