Performance Evaluation of netfilter

A Study on the Performance Loss When Using netfilter as a Firewall

Raik Niemann, Udo Pfingst and Richard Göbel

ABSTRACT
Since GNU/Linux became a popular operating system on computer network routers, its packet routing mechanisms attracted more interest. This does not only concern “big” Linux servers acting as a router but more and more small and medium network access devices, such as DSL or cable access devices.

Although there are a lot of documents dealing with high performance routing with GNU/Linux, only a few offer experimental results to prove the given advices. This study evaluates the throughput performance of Linux' routing subsystem netfilter under various conditions like different data transport protocols in combination with different IP address families and transmission strategies. Those conditions were evaluated with two different types of netfilter rules for a high number in the rule tables. In addition to this, our experiments allowed us to evaluate two prominent client connection handling techniques (threads and the epoll() facility).

The evaluation of the 1,260 different combinations of our test parameters shows a nearly linear but small throughput loss with the number of rules which is independant from the transport protocol and framesize. However, this evaluation identifies another issue concerning the throughput loss when it comes to the address family, i.e. IPv4 and IPv6.

Categories and Subject Descriptors
C.2.6 [Internetworking]: Routers; C.4 [Performance of systems]: Measurement techniques

Keywords
Linux, netfilter, performance

1. INTRODUCTION
One of the benefits of the Linux kernel is the availability for nearly every technical architecture. The combination with the GNU operating system (often referred as GNU/Linux or simply Linux) makes it a good choice for router in computer networks because its memory footprint is quite small based on the modularity of the kernel modules.

In addition to this, the Linux kernel has out-of-the-box routing capabilities as well as advanced packet filter and transformation mechanisms which can be found in the netfilter framework inside of the Linux kernel. Quality of service based classification and prioritization are available, too.

Along with other key features such as the big variety of server software, GNU/Linux is now one of the most preferred operating systems especially for small routing devices, for example DSL or cable access devices in end-user environments. Another famous example for GNU/Linux is the usage in wireless access routers known as DD-WRT.

Those devices as well as “big” GNU/Linux routers, for example PCs or servers, share the same disadvantage: the routing and filtering is based on software whose execution time is influenced by many factors, for example CPU, main memory and hardware drivers. In the worst case, the technical components of the router are not performant enough to process the data packets and they are delayed or discarded.

The main objective of this study is to evaluate the impact of netfilter rules on the throughput rate per client in a distributed client-server application, i.e. a performance test. Although we are aware of the fact, that netfilter features a variety of filter rules, we focus for our experiments only on the most interesting rules for router operators: rules for both permitting clients to pass the router (ACL) and measuring their traffic volume (known as IP accounting) as well as rules for regulating the available network bandwidth among those clients (known as QoS).

In section 2 we describe the reasons why we did not use the test apparatus for this kind of performance test that is suggested in RFC 3511. Additionally, we specify our test apparatus and the extended set of possible influence parameters.

Since we used our own test apparatus, we were able to clarify another aspect in client-server applications: the handling of client connections in the server component. We evaluated two widely used kinds in terms of the throughput rate. The first is to handle each client
connection in its own thread (threading) and the other is the `epoll()` facility offered by the Linux kernel that proclaims to be more performant and easier to implement. Our results along with other observations are discussed in section 3.

2. TEST APPARATUS

The test apparatus follows the guidelines described in RFC 3511. Basically it is a client-server architecture where a central gateway filters and transforms the data transmissions between the clients and the server.

Contrary to RFC3511, we did not use the suggested HTTP benchmark because we were fundamentally interested in the evaluation of a bigger number of influence parameters than only HTTP transactions per second. All test parameters that we were interested in are listed in Table 1. We developed a distributed application instead that has the same semantics like other popular command line benchmark tools like `iperf` or `netperf`, but incorporates a third component gateway in the client-server concept.

![Figure 1: netfilter performance testing architecture](image)

As shown in figure 1, our application consists of three independant command line tools. The communication between those three components can be divided into two parts: a) the control connection between the client and the gateway as well as the server component and b) the data connections between the client threads and the server thread(s). The control connection is used for sending the test parameters to the components and (once they configured themselves according to the parameters) to signal the start and the end of the specific experiment.

Every experiment follows the same steps:

1. The test parameters are given as command line parameters when the client component starts. The parameters are validated and transferred to the gateway component. The gateway component configures the netfilter subsystem according to the submitted test parameters by inserting appropriate filter rules as specified in test parameter $F$.

2. When the gateway components signals its readiness for the configuration, the client component submits the test configuration to the server component. The server component then awaits any client connections by opening a server socket. When this socket is successfully opened and bound, the server component signals its readiness back to the client component.

3. The client component initializes and executes $n$ client threads. They subsequently connect to the server component. According to the parameter $T$ the server component handles each of the client connections in a) its own thread or b) in a single thread using the `epoll()` facility.

4. Once all client threads are connected, they begin to send and to receive data packets according to the test parameters $P,A$ and $f$. Every of the $n$ client-server-connections has its own independant sending/receiving cycle as depicted in figure 2: the client sends a specific amount of data (measurement point 1), the server thread receives the data (measurement point 2) and echos it back to the client thread (measurement point 3). The client thread finally receives the data (measurement point 4).

5. When the test duration $t$ is reached, the client threads get a signal to end the current sending/receiving cycle, to disconnect from the server component and finally to end. The client component instructs the server and then the gateway component to restore the system state that existed before the experiment.

For each of the four measurement points as shown in figure 2 several values were recorded. For measurement point 1 and 3 the number of successful sent/unsent data

---

**Table 1: Parameters of a test case**

| Parameter | Description and tested values |
|-----------|-------------------------------|
| $n$       | Number of client threads (5,10,20,40,80,160,320) |
| $t$       | Duration of the experiment (100s) |
| $f$       | Frame size (either fixed [64,128,256,512,1024] or ranged between 64 and 1024) |
| $P$       | Transport protocol for the transmission (either TCP, UDP or SCTP) |
| $A$       | Address family (either IPv4 or IPv6) |
| $T$       | Server component uses threads for handling the client connections (only valid for stream oriented protocols) |
| $F$       | netfilter rule generation per client thread: 0 for plain forwarding, 2 for up- and download and 4 for additional QoS marks |

---

(Link removed according to double-blind review process).
frames and the frame sizes were saved. The term “un-
sent” in this context means that a data frame could not
be send successfully within a specific timeout (500 ms).

For measurement point 2 and 4 the number of re-
ceived data frames as well as the frame size and the
result of the validation were saved. Please note that a
read timeout was possible, but not used. For the valida-
tion process every data frame sent by a client was filled
with a data record that contains the following informa-
tion: a) the number of the client in the range from 1
to \( n \) b) the chosen frame size according to test param-
eter \( f \) and consecutive sequence number starting with
1 and raised with every send/receive cycle. This allows
to validate if a received data frame belongs to the asso-
ciated sender and the data was successfully transmitted
by comparing the received amount of data with test pa-
rameter \( f \). In addition to this, it allows to detect “gaps”
in the sending/receiving cycle.

All those recorded values formed the basis for our
evaluation.

2.1 Test series

We composed three test series based on the test para-
meter \( F \) (refer to table 1):

1. Plain forwarding: this test series only makes use
   of netfilter’s forwarding capabilities. This means
   that the gateway component is instructed to for-
   ward all data transmission between the client and
   the server component without any limitations, i.e.
   no netfilter rules were inserted.

2. Simple up- and download rules: this test series is
   like the first one but the gateway component in-
   serts a upload and a download netfilter rule per
   client thread. The rules simply checks the IP ad-
   dresses and the protocol to test. In total \( 2 \cdot n \)
   rules are active for a specific experiment. At the end
   netfilter is instructed to discard any other data
   packet that does not conform with the inserted
   rules. This is done by setting the policy of the
   specific rule table to drop anything that was not
   matched by any existing rule.

3. Simple up- and download rules as well as QoS
   marks: this test series does the same as the sec-
   ond one but additionally inserts netfilter rules per
   client thread that are responsible to tag in- and
   outgoing network data packets with a QoS mark.\footnote{The rule template for this is “iptables -t mangle -A
PREROUTING -s <client> -d <server> -p <protocol> -j MARK --set-mark <QoSmark>” for the upload and vice
versa for the download direction.}

Those marks can be used within the iproute2 util-
ity collection to manipulate the QoS subsystem of the
Linux kernel. In total \( 4 \cdot n \) netfilter rules
are inserted for a specific experiment. Please note that
the QoS subsystems of all three test machines
were not modified and used the default (pfifo_fast,
a simple packet first-in-first-out queue with almost
no overhead).

The results of the first test series served us as a baseline
for the other two. During the experiments the hardware
metrics were recorded, e.g. CPU and main memory usu-
age.

2.2 Test machines characteristics

The machine for the client component has two AMD
Opteron 870 CPUs with 4 cores each and 2 GHz
frequency. The machines for the gateway and server com-
ponent have two AMD Opteron 890 CPUs with 4 cores
each and 2.8 GHz frequency. Each of the three ma-
hines have 32 GByte of main memory (DDR2, ECC
error correction).

All test machines used a recent GNU/Linux distribu-
tion (Ubuntu 14.04 in the 64 bit server edition) as an
operating system with a recent Linux kernel (3.13-03).
All unnecessary services were turned off.

2.3 Network configuration

Each of the three machines used for the tests has a 4-
port network adapter with two Intel 82546EB chipsets.
This allows four physical GBit connections. The gate-
way machine is dual-homed with a physical GBit con-
nection to each the client and server component ma-
hine. Each the client and server component has its
own IP network.

Since the gateway machine is dual-homed, it can con-
nect both networks and uses netfilter to route, to filter
and to transform the data transmissions between the
two networks.

All settings that were available for the tested trans-
port protocols and address families were left to their
defaults. Although they offer the potential to raise the
processing performance, the complexity in conjunction
with our test parameters was too high.
3. DISCUSSION OF THE TEST RESULTS

We executed every test series three times and all shown results use the mean value; the variance was uniformly low. In total we executed 3,780 single experiments.

3.1 General observations

All three test series gave us a first impression of the throughput rate for the tested protocols and address families. The average throughput rate for all 3,780 experiments is depicted in figure 3 and 4. Both show the results for the tested address families and scaled to the potential transmission maximum of 1 GBit per second.

The first figure shows the throughput rate grouped by the tested number of concurrent client threads. This way it is possible to estimate the average throughput for any application where the number of clients is known. Please note that the shown throughput rates already include the decrease resulted by netfilter’s filtering and routing. As visible in figure 3, the average throughput rate is quite stable but decreases with a higher number of concurrent clients. The only exception is SCTP where the throughput rate is surprisingly higher for 320 concurrent clients than for 80 and 160.

The latter figure 4 shows the throughput rate grouped by the tested frame sizes. This figure also include all experiments where netfilter rules were involved. Unsurprisingly the throughput rate increases with a higher frame size. The general case is shown in the last bar group labeled “ranged”. In this case the frame size was randomly chosen\(^4\) in a range between 64 and 1024 before every send/receive cycle in every client thread.

We can confirm the widely known fact that SCTP in terms of throughput is slower than TCP which is slower than UDP. Our results show that SCTP is in average 32.65 percent slower than TCP (minimum/maximum difference: 9.28 and 48.23 percent) for all experiments. TCP however is in average 8.42 percent slower than UDP (minimum/maximum difference: 5.18 and 10.64 percent).

Our test results also showed that the throughput rate for IPv6 is noticeable lower than for IPv4. All tested protocols using IPv4 are in average 9.22 percent faster than with IPv6 (minimum/maximum difference: 4.59 and 13.8 percent).

As mentioned before, we recorded the available hardware usage statistics during all experiments. Compared to the statistics for our router machine in its idle state, the impact of the netfilter routing during the experiments in average is marginal. In fact, this depends on the utilized network adapters and the system drivers. Our network adapters featured a special network processor that massively reduced the CPU load by validating incoming network data packets natively, e.g. calculating checksums and verifying packet headers, which is otherwise done by the operating system.

3.2 Impact of netfilter

As stated in the previous section, the first test series (without any netfilter rules) served us as a baseline for the other two that we executed (with different numbers of netfilter rules).

\(^4\)A Mersenne random number generator was used.
For the second and third test series, we calculated the difference with the first one. The results showed a decrease of 2.25 percent in average for all experiments where netfilter rules were involved. We summarized the average throughput decrease in figure 5 (IPv4) and 6 (IPv6). These figures show the average throughput decrease grouped by the tested client thread numbers and additionally for every tested protocol and number of active netfilter rules per client.

As depicted in figure 5 and 6, the decrease is different for the tested address families: the decrease for IPv6 is lower than for IPv4 (2.71 vs. 1.79 percent in average). By considering the decrease percentages as a function of the number of inserted netfilter rules, we calculated the gradient for each tested protocol and address family. In average the gradients are nearly constant. This can be barely seen on figure 5 and 6 because the x-axis is not linearly scaled. To confirm the nearly constant impact of netfilter on the throughput rate, we reviewed our test results with respect to the number of routed data packets between the client and server thread(s) rather than the throughput rate. The review also proves our main findings:

1. netfilter’s performance in terms of throughput is independant from the used transport protocol, frame size and address family as long as simple netfilter rules are active

2. the throughput loss increases roughly linear with the number of inserted (simple) netfilter rules although this loss is quite insignificant

The second main finding shown above allowed us to express the throughput loss per netfilter rule: one can assume a throughput loss of 0.05 percent for any (simple) IPv4 rule and 0.03 percent for any (simple) IPv6 rule.

### 3.3 Client handling techniques

The last objective of this study was to evaluate the client handling techniques in a client-server application: the server component was instructed to handle the data transmissions of stream-oriented clients either in a separate thread per client or in a single thread using epoll().

To consider the differences in the throughput between those two client handling techniques, we only used the experiment results of the first test series and only for SCTP and TCP as well as for both address families. The average difference is illustrated in figure 7 as a percentage between the threaded and unthreaded technique.

This figure clearly indicates that there is a turning point which technique offers a higher throughput rate for a specific number of client connections to handle in a server process.

In our experiments this turning point was around 40 concurrent client connections. The technical specifications of our test machine executing the server component states the native handling of 32 concurrent threads.
This brought us to examine the system usage statistics that were recorded during the experiments. We noticed a significant increase of the number of context switches for our experiments with more than 40 concurrent threads. A context switch takes place when the operating system saves the current state of a process or thread for a later execution in favor of the execution of another process or thread. This storing/restoring of contexts is quite expensive in terms of computation time and can cause the system to slow down. In contrast the same experiments with 40 or more client connections that were handled via epoll() in a single thread did not show this impact.

In summary we recommend to use the epoll() facility of the Linux kernel in a client-server architecture in general. The reason is the better scalability compared to a client handling with threads for a higher number of client connections. Although the throughput rate is higher when threads are used, the rate difference is not significantly higher compared to the handling with epoll(). In addition to this, an application using epoll() can prevent the operating system from unnecessary context switches that also effects other concurrent applications.

4. RELATED WORK

In March 2000, the netfilter routing subsystem was merged into the Linux kernel as the successor of the former subsystem ipchains. The first performance evaluations regarding this new subsystem were made by Hartmeier et al. and Podey et al. ([4], [7]). The comparison between their results concerning the throughput rate with ours for a high number of netfilter rules indicates the same correlations but also illustrates the improvements in the Linux kernel and netfilter subsystem since then.

Further publications dealt with the architecture of netfilter to raise the performance. The netfilter rules are organized in tables that are consulted according to the state of a network data packet. In general, rule evaluation is done sequentially in each table. Lyu et. all as well as Fulp ([6], [3]) classified rules for a later elimination of unnecessary rules. This decreased the overall effort to inspect a data packet within netfilter and lead to a better throughput.

In addition to this, user-defined sub-tables can be created in each of netfilter’s pre-defined tables and can be used as a target for a rule. This allows the segmentation of the rule evaluation. Fulp et al. [3] showed that the rules can be organized as a trie to achieve a faster rule evaluation.

In [2], Acharya et. all collected real-world firewall rule sets of tier-1 internet service providers and the associated usage statistics to form a model for analyzation. This model was later used to improve the rule sets in order to increase the throughput.

Accardi et. all [1] used a special expansion card with a programmable network processor to relocate the network data packet inspection in combination with a netfilter module for this purpose. Their results show a tremendous increase of packet processing in a worst-case scenario, e.g. a Denial-of-service attack. In this attack scenario, the netfilter router faces a massive amount of invalid packets.Accardi et. all demonstrated that their setup of programmable network processor and corresponding netfilter module can prevent the effects of a Denial-of-service attack.

5. CONCLUSION AND FUTURE WORK

In this study we presented the results of our experiments studying the impact of netfilter on the throughput rate. We tested different combinations of transport protocols, address families and frame sizes for an increasing number of netfilter rules. In summary we found out that the throughput loss does not depend on those parameters. The throughput loss is also quite insignificant and rises roughly linear with the number of rules. Our experiments showed an average throughput loss of 0.05 percent for any (simple) IPv4 rule and 0.03 percent for any (simple) IPv6 rule.

In addition to this, we evaluated two prominent client handling strategies for the server component in a client-server application. We proved that up to a certain point a client handling with threads offer a higher but only slight performance gain compared to the counterpart
using the epoll() facility. After this point the thread management is too expensive in terms of computation time and causes the throughput rate per thread to degrade. The epoll() facility in contrast does not show this behaviour.

With the introduction of nftables as the designated successor of the current iptables, a performance gain is expected (although it is based on netfilter, too). As soon as nftables becomes stable, we will redo our experiments using this tool.

6. REFERENCES

[1] K. Accardi, T. Bock, F. Hady, and J. Krueger. Network processor acceleration for a linux* netfilter firewall. In Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems, pages 115–123. ACM, 2005.

[2] S. Acharya, J. Wang, Z. Ge, T. Znati, and A. Greenberg. Simulation study of firewalls to aid improved performance. In Simulation Symposium, 2006. 39th Annual, pages 8–pp. IEEE, 2006.

[3] E. W. Fulp. Optimization of network firewall policies using ordered sets and directed acyclical graphs. In Proc. of IEEE Internet Management Conference, 2005.

[4] D. Hartmeier and A. Systor. Design and performance of the openbsd stateful packet filter (pf). In USENIX Annual Technical Conference, FREENIX Track, pages 171–180, 2002.

[5] B. Hickman, D. Newman, S. Tadjudin, and T. Martin. Benchmarking methodology for firewall performance. http://tools.ietf.org/html/rfc3511.

[6] M. R. Lyu and L. K. Lau. Firewall security: Policies, testing and performance evaluation. In Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International, pages 116–121. IEEE, 2000.

[7] B. Podey, T. Kessler, and H.-D. Melzer. Network packet filter design and performance. In Information Networking, pages 803–814. Springer, 2003.