Delay Reduction for Liner-Search Based Packet Filters

Liang Zhao, Yuji Inoue and Hideo Yamamoto
Faculty of Engineering, Utsunomiya University,
Yoto 7-1-2, Utsunomiya-shi, 321-8585, Japan
Email: zhao@is.utsunomiya-u.ac.jp

Abstract: In general, a packet filter handles incoming packets according to a set of rules. This paper shows that the order of rules is critical to the performance of filters that use the linear search algorithm to handle packets. To reduce the filtering delay, a rule-order optimization problem is formulated. A simple algorithm to solve this problem is also proposed.

1. Introduction

A packet filter is a combination of hardware and software that delivers packets according to a set of rules, where each rule consists of a pattern and an action such that the action is taken for every packet that matches the pattern. Usually a pattern specifies the source/destination address, protocol, port number etc, whereas an action may be accept, reject, address translation (NAT) or others. See Figure 1 for an illustration.

Various packet filters, including routers and firewalls, play together a central role in computer networks. On the other hand, they also introduce extra transmission delays, since all packets have to be processed before reaching destinations. Therefore it is important to develop efficient algorithms to minimize the filtering time (the delay).

For that purpose, many researchers consider the problem of finding a (best)-matched rule for a given packet, called the packet classification problem. They try to solve it by using sophisticated data structures and algorithms corresponding to computational geometry ([1], [3], [4], [7], [9]). However, the state of the art is that, most filters adopt not such an approach but a linear search algorithm ([2], [5], [7], [8]), which simply compares the packet with the rule list, one rule by another, until it is delivered due to a matched rule. Obviously linear search is simple. It requires less storage and computation power, and depends less on the packet specifications.

This paper considers the performance problem of linear-search based filters and propose a solution to reduce the filtering delay. Our study is motivated by the next observation.

Let \( \mathcal{F} \) be a linear-search based filter with \( n \) ordered rules \( r_1, r_2, \ldots, r_n \). For simplicity, suppose that it takes \( s \) time to compare a packet with one rule. Let \( P \) be a packet that matches only \( r_n \). Then it takes \( n \times s \) time for \( \mathcal{F} \) to process \( P \), since all rules have to be compared. Clearly, if there are a lot of such packets, we can reduce the filter delay of \( \mathcal{F} \) by moving \( r_n \) to the front of rule list. (Some simulation tests for proving this observation are shown in Section 4.)

Actually this has been observed for years: some guide books for filter designers suggest to arrange rules appropriately to obtain the maximal performance. Unfortunately, as far as we know, there is no precise way publicly available before, though it is known that an advanced filter can use cache to improve the performance. In this paper, we first give a simple problem formulation in Section 2, then provide a solution in Section 3. In Section 4, we show some simulation results. Finally we conclude in Section 5.

2. A Simple Problem Formulation

Given a filter \( \mathcal{F} \) with a set \( \{r_1, r_2, \ldots, r_n\} \) of rules, we want to minimize the expected (i.e., average) filtering time of \( \mathcal{F} \).

For this, we first consider the comparing time. In general, it may vary by packets and rules. For simplicity, we assume that it depends only on the rule (estimations are considered in Section 3), i.e., letting the comparing time for rule \( r_i \) be \( t_i \).

Assumption 1: \( t_i \) is a constant for rule \( r_i \), \( 1 \leq i \leq n \).

Let \( \pi \) denote an order (i.e., a permutation) of \( \{1, 2, \ldots, n\} \), and let \( \pi[i] \) denote the \( i \)th number (e.g., if \( \pi = 2, 1, 3, \ldots \), then \( \pi[1] = 2, \pi[2] = 1, \pi[3] = 3, \ldots \)). Let \( p_i \) be the probability that rule \( r_i \) is matched, i.e., \( p_i \) is the ratio of packets matching \( r_i \) (it can be estimated from statistics).

Due to the nature of linear search, we can assume that a packet reaches a rule must have been compared with all prior rules. Speaking precisely, given an order \( \pi \), a packet reaches a rule \( r_{\pi[i]} \) must have been compared with all rules \( r_{\pi[j]}, j = 1, \ldots, i-1 \). Thus the processing time for a packet reaches \( r_{\pi[i]} \) is \( \sum_{j=1}^{i} t_{\pi[j]} \). Hence the average filtering time with respect to \( \pi \), denoted by \( AFT(\pi) \), can be written as

\[
\text{AFT}(\pi) = \sum_{i=1}^{n} \left( \sum_{j=1}^{i} t_{\pi[j]} \right) p_{\pi[i]}. \tag{1}
\]

Therefore we can formulate the problem as the following.

**Problem (Rule Order).** Given \( n \) rules \( r_1, r_2, \ldots, r_n \) with matching probabilities \( p_1, p_2, \ldots, p_n \) and comparing times \( t_1, t_2, \ldots, t_n \), find an order \( \pi \) of rules that minimizes (1).

In general, we do not know how to solve the above problem efficiently. For this purpose, we let first consider with the next...
Assumption (a solution that does not require the assumption will be shown in the next section).

Assumption 2: Rules can be arbitrarily reordered without changing the function of $\mathcal{F}$ (thus $p_i$ is invariant to the order). By Assumption 2, we can solve the problem exactly.

Theorem 1: An order $\pi$ is optimal if and only if

$$\frac{p_{\pi[i]}}{t_{\pi[i]}} \geq \frac{p_{\pi[i+1]}}{t_{\pi[i+1]}}, \quad i = 1, 2, \ldots, n - 1. \quad (2)$$

Proof. (only if) If not, there exists $i$ such that $\frac{p_{\pi[i]}}{t_{\pi[i]}} < \frac{p_{\pi[i+1]}}{t_{\pi[i+1]}}$.

By exchanging $r_{\pi[i]}$ and $r_{\pi[i+1]}$, we can obtain a new order $\pi'$ with average filtering time

$$AFT(\pi') = AFT(\pi) + t_{\pi[i+1]}p_{\pi[i]} - t_{\pi[i]}p_{\pi[i+1]} < AFT(\pi).$$

This contradicts the optimality of $\pi$. Thus (2) must hold.

(if) By the above proof, we see that any optimal order must satisfy (2). On the other hand, it is not difficult to see that all orders satisfies (2) have the same average filter time. This proves the theorem.

Remark. Theorem 1 shows that a descending order of $\frac{p_1}{t_1}, \frac{p_2}{t_2}, \ldots, \frac{p_n}{t_n}$ is an optimal solution. This implies that the problem can be solved in $O(n \ln n)$ time and $O(n)$ space by one sorting. It also shows that a cache algorithm may not perform well if it considers only $p_i$. Instead, a more clever algorithm should use $\frac{p_i}{t_i}$ as the cache index.

In practice, of course, one cannot expect this theorem to work unconditionally. Let us discuss in the next section.

3. The Solution

Let us consider a solution for the (general) problem. We will work with the Linux packet filter software — `iptables` ([5]), but the idea should also work for other filters.

We consider that a solution must solve the next problems. Firstly, it must estimate $p_i$ and $t_i$. Secondly, it must verify the Assumptions, or to consider a solution if they are not satisfied. The last problem, called default rule, will be explained later. Let us see them in the following. A figure showing our algorithm flow will be given at the end of this section.

3.1 Parameter estimations

Estimating matching probabilities $p_i$ is not difficult, since logging packets is a standard feature of filters. We can observe the packets for a period $T$ (e.g., one week). Let $N$ be the number of processed packets, and let $N_i$ be the number of packets that match rule $r_i$. Then $p_i$ can be estimated by

$$p_i = \frac{N_i}{N}.$$ 

On the other hand, we do not know how to estimate the comparing times $t_i$ exactly. For simplicity, we can roughly assume $t_i = 1$ for all rules $r_i$. For more accuracy, we can estimate it by the number of specifications in the pattern, since it is the maximum number of comparisons needed to handle a packet. For example, for the next pattern

(protocol == TCP) and (port == 80),

we can estimate the comparing time as 2. Similarly, we can estimate the comparing time for pattern (port == 80) as 1. Summarizing the above, we can roughly estimate $t_i$ as

$$t_i = \text{number of specifications in the pattern of } r_i. \quad (3)$$

3.2 On the assumptions

To apply Theorem 1, we have assumed two assumptions in Section 2. Assumption 1 can be satisfied due to our estimation (see above). On the other hand, however, Assumption 2 may not be satisfied. For instance, a filter designer may write the next rule set to accept only the HTTP (port 80) request among all TCP connections.

$$r_1 : \quad \text{(protocol == TCP) and (port == 80)} \quad \rightarrow \text{accept},$$

$$r_2 : \quad \text{(protocol == TCP) and (port != 80)} \quad \rightarrow \text{reject}.$$

Obviously the order of these two rules cannot be exchanged, otherwise the filter’s function is destroyed.

To detect the existence of such a precedence requirement is not so difficult: we can scan the rule set for $n - 1$ rounds. In round $i$, compare the pattern of rule $r_i$ with rules $r_j, j = i + 1, \ldots, n$. (More efficient algorithm may be available.) On the other hand, unfortunately, we are not aware of any easy method to eliminate the precedence requirements for a given rule set. For this, we propose two algorithms.

Algorithm 1 (Rewrite). Rewrite rules so that a packet can match at most one rule.

For instance, in the previous example, we can rewrite $r_2$ to

$$r_2' : \quad \text{(protocol == TCP) and (port != 80)} \quad \rightarrow \text{reject}.$$ 

In this way, we see that Assumption 2 is satisfied, which implies that reordering does not destroy the filter’s function.

Unfortunately, this method is not easy to achieve (itself is a study). Even worse, modifying a rule may change the comparing time (e.g., in the previous example, it changes from 1 to 2 according to estimation (3)). Thus it is almost impossible to modify a large set of rules without significant loss of performance. This drove us to consider the next solution.

To see it, first notice that, given an optimal order $\pi$,

$$\frac{p_{\pi[i]}}{t_{\pi[i]}} \geq \frac{p_{\pi[i+1]}}{t_{\pi[i+1]}}, \quad \text{if } r_{\pi[i]} \text{ and } r_{\pi[i+1]} \text{ are exchangeable},$$

where “exchangeable” means that the exchange of two rules has no damage on the filter’s function. Based on this observation, we consider the next algorithm:

Algorithm 2 (Reorder). For $i = 1, 2, \ldots, n - 1$, exchange $\pi[i]$ and $\pi[i+1]$ if they are exchangeable and $\frac{p_{\pi[i]}}{t_{\pi[i]}} < \frac{p_{\pi[i+1]}}{t_{\pi[i+1]}}$.

Repeat the above until there is no exchange happened.

This algorithm does not require Assumption 2 (of course, it works correctly if Assumption 2 is satisfied). Therefore, it is a correct algorithm for the Rule-Order problem, i.e., the function of the finally obtained rule set works equivalently to the input rule set.

On the other hand, however, we note that the optimality may not be obtained. There are examples showing this fact. Further details are omitted due to the limit of space.
3.3 The default rule problem

A special precedence requirement is due to the default rule. Default rule, also called the policy, is such a rule that is matched by all packets that match no other rules. Therefore a default rule must stay at the end of the rule list (see Figure 1).

We observe that, since there can exist a large number of packets that only match the default rule, the performance of the filter may be further improved by extracting patterns (i.e., creating new rules) from the policy. Let us see an example.

\[
\begin{align*}
    r_1 : & \quad \text{(protocol == TCP) and (port == 80)} \quad \rightarrow \text{accept}, \\
    r_2 : & \quad \text{(protocol == TCP)} \quad \rightarrow \text{reject}, \\
    r_3 : & \quad \text{default} \quad \rightarrow \text{drop}.
\end{align*}
\]

Let \( p_1 = 0.1, p_2 = 0.2, p_3 = 0.7 \). According to (1) and (3), the average filtering time of this rule set is

\[
2 \times 0.1 + (2 + 1) \times 0.2 + (2 + 1 + 0) \times 0.7 = 2.9.
\]

Notice that we cannot reorder the three rules. However, let us suppose that there are 60\% (probability = 0.6) routing packets (UDP, port 520) that match \( r_3 \). Then, if we create a new rule

\[
r_4 : \quad \text{(protocol == UDP)} \quad \rightarrow \text{drop},
\]

and order rules by \( r_4, r_1, r_2, r_3 \), the filter still works correctly, but this time, the average filtering time is reduced to

\[
1 \times 0.6 + (1 + 2) \times 0.1 + (1 + 2 + 1) \times 0.2 + (1 + 2 + 1 + 0) \times 0.1 = 2.1.
\]

Thus extracting new rules from the default rule can improve the performance. Our detailed solution goes follows.

**Algorithm 3 (Rule Extraction)**

1. For each class of packets that match the default rule \( r_n \), create a new rule. Let the new rules be \( r_{n+1}, \ldots, r_{n+m} \).
2. For each new rule \( r_{n+k} \), determine its matching probability \( p_{n+k} \) and comparing time \( t_{n+k} \). Put the new rules right before \( r_n \) in the descending order with respect to \( \frac{p_{n+k}}{t_{n+k}} \).
3. Apply the Reorder Algorithm to the new rule set.
4. Remove new rules between \( \{ r_i \mid 1 \leq i \leq n - 1 \} \) and \( r_n \).

**Remark.** The performance of this algorithm depends on the efficiency of packet classifications. Unfortunately, it is known that, in general, finding good classifications from given data is difficult. In practice, we consider that the classification by protocol, by port number, or by both can work fine. Further details are omitted due to space limit.

**Solution Flow.** Let us summarize the above. The rough flow chart of our solution can be written as Figure 2 (see also the Reorder Algorithm and the Rule Extraction Algorithm).

We note that minor enhancements can be inserted into the implementation, e.g., if \( \frac{p_{n}}{t_{n}} \) is the minimum among all \( \frac{p_{i}}{t_{i}} \), then there is no need to extract new rules.

4. Simulation Results

Let us show some preliminary simulation results. They are performed using the next equipments.

**OBS filter:** OpenBlockS 266 (see [6] for details).

CPU: IBM PowerPC 405GPr 266MHz
RAM: 64MB
LAN Card/Chip: Dacicom DM9102/DM9102A
OS: SSD Linux 0.2 (kernel 2.4.20)
Filter software: iptables v1.2.7a

**PC 1 filter:** Notebook PC (Panasonic Let’s note CF-S2218)

CPU: Intel Mobile Pentium MMX 266MHz
RAM: 160MB
LAN Card/Chip: 3Com 3C575A Fast EtherLink XL
OS: Redhat Linux 8.0 (kernel 2.4.18)
Filter software: iptables v1.2.6a

**PC 2 filter:** Notebook PC (Panasonic Let’s note CF-T1)

CPU: Intel Mobile Pentium III - M 933MHz
RAM: 256MB
LAN Card/Chip: 3Com 3C575A Fast EtherLink XL
OS: Fedora Core 1 (Linux kernel 2.4.20)
Filter software: iptables v1.2.8

Tests are done as follows.

For each of the three filters, we do the performance tests 21 rounds. In each round \( i \), we create a rule set consisting of 201 rules that are different only by the port number. A packet is allowed to pass through if it matches the 10\(i^{th}\) rule (the granted rule). In other words, we only grant connections to the 10\(i^{th}\) port. To test the performance, we use the well-known tcpdump program. In each round, we have tested for 5 times and take the average as the result.

First, let us see Figure 3. It shows the relation between the (maximum TCP) throughput and the number of the granted rule. It is clear that, for filters OBS and PC 1, the throughput decreases almost linearly due to the increase of rule number.

Next, let us see Figure 4. Inversely to Figure 3, it shows that, for OBS and PC 1, the filtering time increases almost linearly as the number of the granted rule increases.

Filter PC 2 shows an almost-constant high performance in the previous test. This can be understood because it owns a much more powerful CPU. To test its performance in detail,
we increase the number of rules to 700 with sampling interval 35. The results are shown in Figure 5, from which we can see that basically the filtering time increases linearly except for numbers below 200. (We note that, as a rough estimation, a common firewall filter usually has a rule set of size as small as several tens to as large as several thousands.)

Summarizing the above, we conclude that basically the performance of a linear-search based filter varies by the order of rules linearly, which matches our observation. Therefore our algorithm can improve the performance for such filters. On the other hand, we note that our model does not consider the computational overhead for filters, which is interested in a more detailed study.

5. Conclusion

This paper considered linear-search based packet filters. We have shown that the order of rules is critical to the performance. To reduce the filtering time, we formulated a rule-order optimization problem, and have solved it under certain assumptions. In practice, the assumptions may not be satisfied. For that purpose, we proposed a simple solution, which works correctly but may not obtain the optimality. (It can find an optimal solution if the assumptions are satisfied.) Now we are working for an implementation of the proposed solution.

If there exist precedence requirements, it is much more difficult to find an optimal solution. In addition, extracting good new rules from the default rule is another difficult problem. In summary, we see that two problems (precedence requirement and rule extraction) require a thorough study in the future.

References