Bounding Peer-to-Peer Upload Traffic in Client Networks*

Chun-Ying Huang
National Taiwan University
huangant@fractal.ee.ntu.edu.tw

Chin-Laung Lei
National Taiwan University
lei@cc.ee.ntu.edu.tw

Abstract

Peer-to-peer technique has now become one of the major techniques to exchange digital content between peers of the same interest. However, as the amount of peer-to-peer traffic increases, a network administrator would like to control the network resources consumed by peer-to-peer applications. Due to the use of random ports and protocol encryption, it is hard to identify and apply proper control policies to peer-to-peer traffic. How do we properly bound the peer-to-peer traffic and prevent it from consuming all the available network resources?

In this paper, we propose an algorithm that tries to approximately bound the network resources consumed by peer-to-peer traffic without examining packet payloads. Our methodology especially focuses on upload traffic for that the upload bandwidth for an ISP are usually more precious than download bandwidth. The method is constructed in two stages. First, we observe several traffic characteristics of peer-to-peer applications and traditional client-server based Internet services. We also observe the generic traffic properties in a client network. Then, based on the symmetry of network traffic in both temporal and spatial domains, we propose to use a bitmap filter to bound the network resources consumed by peer-to-peer applications. The proposed algorithm takes only constant storage and computation time. The evaluation also shows that with a small amount of memory, the peer-to-peer traffic can be properly bounded close to a predefined amount.

1 Introduction

The behavior of traditional Internet applications is simple. That is, a client sends a request to an Internet server and then receives replies from the server. Network resource management for these applications is also simple. To manage the network traffic of a specific network service, a network administrator can easily apply traffic control policies to traffic that communicates using corresponding server ports. However, as the emerging of peer-to-peer technologies, modern popular services like file sharing and video streaming now leverage peer-to-peer technologies to increase the availability and the performance of the services. Therefore, it also brings new challenges to network resource management. The major reason is that the peer-to-peer traffic is more difficult to identify. First, peer-to-peer applications tend to communicate using random ports and thus it is hard to define port-based control policies for such network traffic. Second, as any one can develop their own protocols, even if network administrators are able to identify network traffic by analyzing packet payloads, it is impossible to know all peer-to-peer protocols beforehand. Besides, the use of "protocol encryption" (PE), "message stream encryption" (MSE), and "protocol header encryption" (PHE) also complicates the problem. Since the PE, MSE, and PHE encrypts the parts of peer-to-peer protocol messages in payloads, it also increases the difficulties to identify peer-to-peer traffic.

Recent studies have shown that the peer-to-peer traffic has gradually dominated the Internet traffic. While ISPs are usually charged based on the traffic they send upstream to their providers, they would like to keep traffic generated by their customers within the boundaries of their own administrative domains. However, this conflicts with the core spirit of peer-to-peer applications, which encourages clients to *share* what they possess to the public. The more the clients share, the more the uplink bandwidth are consumed for the share. From the view point of network administrators, the precious uplink bandwidth *should be used for client requests, not for the shares*. To reserve the uplink bandwidth for the right purposes, peer-to-peer upload traffic should be properly controlled in a client network. As we already knew that peer-to-peer traffic is hard to identify, how do we control these unknown uplink traffic in a client networks?

An effective method to achieve this goal is adopting a

*This work is supported in part by the National Science Council under the Grants NSC 95-3114-P-001-001-Y02 and NSC 95-2218-E-002-038, and by the Taiwan Information Security Center (TWISC), National Science Council under the Grants No. NSC 95-2218-E-001-001 and NSC 95-2218-E-011-015.

positive listing strategy. That is, the client network allows only outbound requests initiated by clients in the network. At the same time, to keep peer-to-peer applications working, a limited amount of the uplink bandwidth could still be allowed for those applications. While peer-to-peer upload traffic are mostly triggered by inbound requests, by limiting the inbound requests, the upload traffic can be constrained to a given bounds. To do this, a stateful packet inspection (SPI) filter can be installed at the entry points of a client network to maintain the per-flow state of each outbound connection. The SPI filter tracks the states of network flows that pass it. It allows all outbound requests and the corresponding inbound responses. However, on receipt of inbound requests, the SPI filter decides to accept or reject the request according to the uplink bandwidth throughput. Applying such a mechanism in an ISP-like scale network may incur a high computational cost as the required storage space and computation complexity depends linearly on the number of concurrent active connections, which may be in the order of tens of thousands or even millions.

In this paper, we try to solve the above problem with an efficient and effective method. An *bitmap filter* algorithm is proposed to maintain outbound connection states and permit inbound connections according to monitored bandwidth throughput. The effectiveness of the bitmap filter is similar to that of an SPI filter, but it requires only constant storage space and computational resources.

The remainder of this paper is organized as follows. In Section 2, we review some previous works that are related to our solution. In Section 3, we observe several client network traffic characteristics that are useful to construct our solution. In Section 4, we discuss the usage model and the detailed design of the proposed solution. In Section 5, we then evaluate the effectiveness and the performance of the solution. Finally, in Section 6, we present our conclusions.

2 Related Works

A great deal of research effort has been devoted to peer-to-peer networks. In [1], the authors investigate several characteristics of peer-to-peer traffic, which includes the bottleneck bandwidths, latencies, the degree of peer cooperations, etc. In [2], the authors analyze the peer-to-peer traffic by measuring flow level information and show that the high volume and good stability properties of peer-to-peer traffic makes it a good candidate for being managed in an ISP network. Authors of [3] and [4] also show that the amount of peer-to-peer traffic keeps growing and now it has now become one of the major Internet applications.

In contrast to our solution, authors of [5] purpose to save the download bandwidth by caching those shared data. The cache system works only when it can identify and understand the peer-to-peer protocols. To identify peer-to-peer traffic, besides counting on well-known ports, Sen et al. [6] developed a signature-based methodology to identify peer-to-peer traffic. However, the use of "protocol encryption" (PE) makes it difficult to detect peer-to-peer traffic using payload identification. In [4], Karagiannis et al. try to identify peer-to-peer traffic without examining the payloads. The proposed PTP algorithm performs well on identification of unknown peer-to-peer traffic. Nevertheless, the algorithm use a table to records flow states, which may be not suitable to operate in a real-time and large-scale environment.

To limit the peer-to-peer upload traffic, we believe that an SPI-based filter is a possible solution for client networks. However, since SPI-based filters have to keep all per-flow states in detail, adopting it incurs high cost for an ISP. Take a popular SPI implementation in the Linux open-source operating system as an example. The required storage space grows linearly according to the number of kept flows. Besides, the data structures used to maintain these states are basically link-lists with an indexed hash table. It is obvious that both the storage and computation complexities are $O(n)$, which is not affordable for a larger ISP containing several client networks.

3 The Client Network Traffic Characteristics

3.1 Network Setup

Our packet traces are collected in a subnet of our campus network. Most of hosts in the subnetwork are clients. The trace collection environment is illustrated in Figure 1. A traffic monitor is used to receive and analyze both inbound and outbound traffic of the subnetwork. The traffic monitor is a Fedora Core 5 Linux equipped with dual-processor Intel Xeon 3.2G and a Broadcom BCM95721 gigabit network interface. To save the storage space for packet traces, the traces are collected in three different stages. First, we collect *full packet traces* (including both packet headers and full payloads) using the well-known `tcpdump` [7] program. The full packet traces are then used to verify the correctness of our customized traffic analyzer. The verified analyzer is finally used to extract useful information from packet payloads on-line and simultaneously collect *header packet traces*, which contains only layer 2 to layer 4 packet headers, for future use. The design of the customized traffic analyzer is introduced later in Section 3.2.

3.2 The Traffic Analyzer

One purpose of the traffic analyzer is to identify *network applications* from current network connections. A network connection is identified by a five tuple socket pair, which includes the layer 4 protocol (TCP or UDP), the

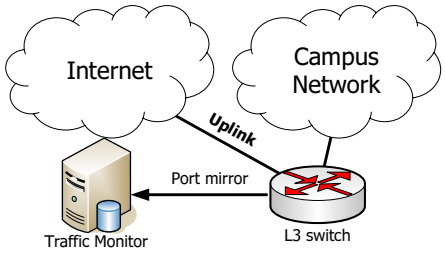

Figure 1. The network setup for packet trace collection. The traffic sent to the campus network is inbound traffic while traffic in the other direction is outbound traffic.

source address, the source port, the destination address, and the destination port. An example of a socket pair s is $\{\text{TCP}, A, x, B, y\}$. Since packets of the same connection are transmitted in different directions between two end hosts, the inverse of a socket pair, $\overline{s} = \{\text{TCP}, B, y, A, x\}$, also identifies the same connection. In our traffic analyzer, it first classifies packets into connections and then try to identify the application of each connection.

Two methods are used to identify the applications. As we know that many modern applications, for example, peer-to-peer applications, do not use fixed ports to communicate, we first try to identify application by matching the packet payloads against several predefined patterns. To do so, the analyzer must have the ability to examine the payloads either by reading the full packet traces collected by the `tcpdump` program or by accessing packets directly through the network interface. The analyzer focuses only on TCP and UDP traffic for that these two are the major data transmission protocols used over Internet. Packets with incorrect checksum values are not considered for examination.

The payload of each UDP packet is always examined. However, to guarantee the completeness of payloads in a TCP connection, we only examine TCP connections with an explicitly TCP-SYN packet, which indicates the beginning of a TCP connection. Unlike the examination for UDP data packets, the pattern matching algorithm does not match for a single TCP data packet. Instead, it matches a concatenated TCP data stream against the patterns. For a TCP connection, we have to concatenate payloads of several very first data packets[1] to form a short TCP stream. The algorithm then matches the concatenated data stream against all the patterns. The patterns used for pattern matching are written in the form of regular expressions. Most of these patterns are adopted from the L7-filter project [8]. Examples of some of these patterns are listed in Table 1. If it is failed to identify an application by pattern matching, the analyzer then tries to identify by matching the port numbers of the connection against well-known port numbers.

To focus more on file exchanging applications, we use two alternative strategies to identify peer-to-peer and FTP applications, respectively. For the ease of explanations, a network connection c is denoted by $c = \{A : x \rightarrow B : y\}$, where A is a client that connects to a service provider B on port y using port x. In the first strategy, if c is identified as one of the peer-to-peer applications, all future connections to $B : y$ are also identified as the same application. In the second strategy, since we know that the FTP command and the FTP data are transmitted in separated connections, if c is identified as an FTP application, all payloads of the identified connection are examined to identify the corresponding FTP data connections specified in a FTP command connection.

Another purpose of the analyzer is to measure and log some fundamental properties of network connections for further traffic analyses. These properties include the direction (inbound or outbound) of a network connection, the number of packets and bytes transmitted in each direction, the lifetime of a connection, and the out-in packet delays. To keep the original traffic patterns and save the storage spaces, payloads of all processed packets are stripped and then stored using the same format as the `tcpdump` program.

3.3 Traffic Characteristics

Based on the information collected by the traffic analyzer, we make several observations on these traffic. The observations are done on a 7.5-hour TCP and UDP packet trace, which was collected in the environment introduced in Section 3.1. In the 7.5-hour packet trace, there were 6739733 collected connections. Among all the connections, 29.8% were TCP connections and 70.1% were UDP connections. Although there are more UDP connections, 99.5% bandwidth are contributed by TCP traffic. The average bandwidth throughput of this trace was 146.7 Mbps, where 10.2% were download traffic and 89.8% were upload traffic. The first observation is the distributions of each observed applications. Among the observed applications, 5% are HTTP/HTTP-PROXY traffic, 55% are peer-to-peer traffic (including bittorrent, edonkey, and gnutella), 5% are other traditional internet services, and most of traffic (35%) are still unidentified. A brief summary of the protocol distribution can be found in Table 2.

The second observation focuses on the port number distributions of network connections. We classify all the port numbers into four different classes, namely "ALL", "P2P", "Non-P2P", and "UNKNOWN". For each TCP connec-

[1] In our program, we concatenate at most four TCP data packets. This is because most of the patterns used to check the connection type are short and thus it is not necessary to store and check the full TCP data stream.

Table 1. Patterns and ports used to identify network applications.

Application	Regular Expressions	Ports
bittorrent	\x13bittorrent protocol|d1:ad2:id20:|\x08'7P\)[RP]|^azver\x01$|^get /scrape?info_hash=	N/A
edonkey	^[\xc5\xd4\xe3-\xe5].?.?.?.?([\x01\x02\x05\x14\x15\x16\x18\x19\x1a\x1b\x1c\x20\x21\x32\x33\x34\x35\x36\x38\x40\x41\x42\x43\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58[\x60\x81\x82\x90\x91\x93\x96\x97\x98\x99\x9a\x9b\x9c\x9e\xa0\xa1\xa2\xa3\xa4]|\x59.....?[~\]|\x96....$)	TCP/UDP: 4662
fasttrack	^get (/.download/[~\]*|/.supernode[~\]|/.status[~\]|/.network[~\]*|/.files|/.hash=[0-9a-f]*/[~\]*) http/1.1|user-agent: kazaa|x-kazaa(-username|-network|-ip|-supernodeip|-xferid|-xferuid|tag)|^give [0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]?[0-9]?[0-9]?	N/A
gnutella	^(gnd[\x01\x02]?.?.?\x01|gnutella connect/[012]\.[0-9]\x0d\x0a|get /uri-res/n2r\?urn:sha1:|get /.*user-agent: (gtk-gnutella|bearshare|mactella|gnucleus|gnotella|limewire|imesh)|get /.*content-type: application/x-gnutella-packets|giv [0-9]*:[0-9a-f]*/|queue [0-9a-f]*[1-9][0-9]?[0-9]?\.[1-9][0-9]?[0-9]?\.[1-9][0-9]?[0-9]?\.[1-9][0-9]?[0-9]?:[1-9][0-9]?[0-9]?[0-9]?[0-9]?|gnutella.*content-type: application/x-gnutella|.....?lime)	N/A
http/http-proxy	http/(0\.9|1\.0|1\.1) [1-5][0-9][0-9] [\x09-\x0d ~\]*(connection:|content-type:|content-length:|date:)|post [\x09-\x0d ~\]* http/[01]\.[019]	TCP: 80, 3128, 8080
ftp	^220[\x09-\x0d ~\]*ftp	TCP: 21

Table 2. Summary of Protocol Distributions in the Trace Data

Protocol	Connections	Utilizations
HTTP	2.17%	5%
bittorrent	47.90%	18%
gnutella	7.56%	16%
edonkey	22.00%	21%
UNKNOWN	17.55%	35%
Others	2.82%	5%

tion, we only count the port number that is used by the service provider, i.e. the destination port of the corresponding TCP-SYN packet. This is because the source ports of TCP connections are usually randomly generated. We also gather the same statistics for UDP connections. However, since there is no explicitly signals to determine the direction of an UDP connection, for UDP connections, both source ports and destination ports are counted. Figure 2 and Figure 3 show the cumulative distributions of TCP and UDP port numbers, respectively. In Figure 2, we can find that most of the "Non-P2P" connections use several well-know ports. We also found that besides these well-known peer-to-peer ports, a great deal of random ports between port 10000 and port 40000 are also used for peer-to-peer communications. Although there are many unidentified connections in our trace, we found that the port distributions of these "UNKNOWN" connections are close to "P2P" applications. As the development of proprietary peer-to-peer protocols and the use of "protocol encryption" technologies, we believe that many of those unidentified connections have a high probability to also be peer-to-peer traffic.

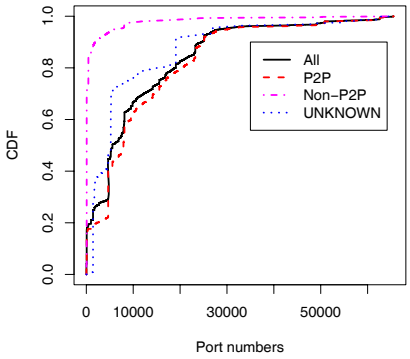

Figure 2. The port number CDF plot of TCP connections. Only ports that used to accept TCP connections are counted.

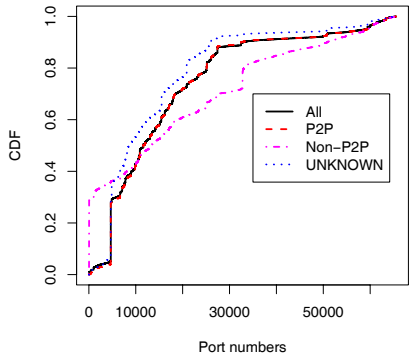

Figure 3. The port number CDF plot of UDP connections. Both source ports and destination ports of UDP connections are counted.

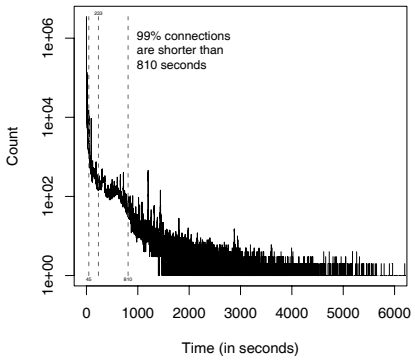

Figure 4. Statistics for connection lifetime. The average connection lifetime is 45.84 seconds.

Figure 3 shows the results of UDP connections. While we count both source ports and destination ports for UDP connections, the result also reflects that the port numbers are almost uniformly distributed. However, we can still identify several frequently used ports, like DNS (port 53) and the edonkey ports (port 4661, 4662, 4672, etc).

We have mentioned that 89.8% of the throughput was contributed by outbound traffic. Among all the outbound traffic, it should be noticed that 80% are sent along with inbound connections while the other 20% are actively sent out by inner clients. In general, the design of data transmission protocols can be classified into two categories. Data can be delivered either within the same connection to the request or using a different connections. This statistics show that most applications prefer the former design.

We also examine the lifetime of connections from the packet trace. The lifetime of TCP connections are counted from the first TCP-SYN packet to the appearance of a valid TCP-FIN or TCP-RST packet. The connection lifetime varies widely from a minimum of several milliseconds to a maximum of six hours, as shown in Figure 4 (data exceeding the 6000^{th} second are removed, since there are no more peaks). However, the lifetime of most connections is short. The statistics show that 90% of connections are under 45 seconds, 95% are under 4 minutes, and less than one percent last for more then 810 seconds.

Although the lifetime for each connection varies greatly, an interesting phenomenon is that *the out-in packet delay is always short*. Before introducing out-in packet delay, we define two types of packet. An *outbound packet* is a packet sent from a client network, while *inbound packet* is a packet received by a client network. A packet always contains a socket pair σ of $\{protocol, source\text{-}address, source\text{-}port, destination\text{-}address, destination\text{-}port\}$. Thus, for an outbound packet with a socket pair of $\sigma_{out} = \{protocol, saddr, sport, daddr, dport\}$, the socket pair of its corresponding inbound packet should be in an inverse form, that is $\sigma_{in} = \{protocol, daddr, dport, saddr, sport\}$. Note that for an outbound packet and its corresponding inbound packet, $\overline{\sigma_{in}}$, which is the inverse of the socket pair σ_{in}, and σ_{out} should be the same. Based on these definitions, the out-in packet delay is then obtained as follows:

1. On receipt of an outbound packet with a socket pair $\sigma_{out} = \{protocol, saddr, sport, daddr, dport\}$ on an edge router at time t, the router checks if the socket pair has been recorded previously. If the socket pair is new, it is associated with a timestamp of time t and stored in the edge router's memory. Otherwise, the timestamp of the existed socket pair is updated with the time t.

2. On receipt of an inbound packet with a socket pair $\sigma_{in} = \{protocol, daddr, dport, saddr, sport\}$ at time t, the edge router checks if the inverse socket pair $\overline{\sigma_{in}}$ has been recorded before. If it already exists, the timestamp associated with the inverse socket pair $\overline{\sigma_{in}}$ is read as t_0 and the out-in packet delay is computed as $t - t_0$.

3. To avoid the problem of port-reuse, which affects the accuracy of computing the out-in packet delay, an expiry timer T_e deletes existing socket pairs when $t - t_0 > T_e$.

The out-in packet delay may be caused by network propagation delay, processing delay, queueing delay, or mechanisms

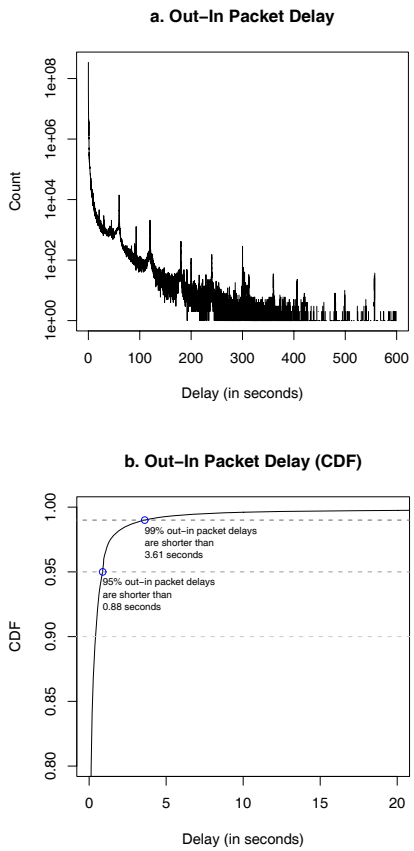

Figure 5. The measured out-in packet delay in the trace data. Part-(a) shows the raw data with observed port-reuse effects on peaks. Part-(b) shows the CDF of the out-in packet delays.

like delayed-ACK. However, they should not be too long. The statistics of out-in packet delay are shown in Figure 5-b. Since we use a large expiry timer, $T_e = 600$ seconds, to handle expired socket pairs, in Figure 5-b, the effect of port-reuse can be observed roughly at the peaks. Although the port-reuse timer varies in different implementations, we find that most of them are in multiples of 60 seconds. The statistics also show that most out-in packet delays are very short. In Figure 5-c, 99% of out-in packet delays are under 2.8 seconds. The result also implies that the most Internet traffic is bi-directional and has high locality in the temporal domain.

4 The Bitmap Filter

By definition, a client network should have only client hosts, such as a business enterprise customer, a group of

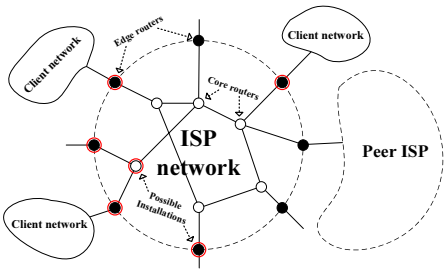

Figure 6. An ISP network with bitmap filters.

DSL users, a wireless network, or buildings in a campus. Usually client hosts only initiate requests and seldom receive requests from the Internet. However, as the peer-to-peer softwares become more and more popular, client hosts now also wait for inbound connections from other peers and thus generate huge volumes of upload traffic. The bitmap filter is a lightweight and efficient algorithm that can be used to bounding upload traffic from client networks. In this section, we first illustrate the usage model of our solution and then introduce the detailed design of the algorithm.

4.1 The Usage Model

Bitmap filters should be installed in an ISP network. As shown in Figure 6, an ISP usually has edge routers (black nodes) and core routers (white nodes). The bitmap filter can be installed on an edge router directly connected to a client network or on a core router, which is an aggregate of two or more client networks. In Figure 6, the nodes with an outlined circle are possible locations to install the bitmap filter. Actually, the bitmap filter can be installed at any location through which traffic from client networks must pass.

4.2 Construct the Bitmap Filter

The design of the bitmap filter leverages certain client network traffic characteristics to improve the filter performance. Based on the observations that 1) the client network traffic is bi-directional, 2) most out-in packet delays are short, and 3) most of the outbound traffic are triggered by inbound requests, a naïve solution is proposed to limit the upload traffic. The solution basically keeps only the outbound requests initiated by inner clients. When the upload bandwidth throughput is low, all the inbound packets, either responses to previous outbound requests or inbound requests to the client network, are permitted. However, if the upload bandwidth throughput is high, only the inbound packets that are responses to previous outbound requests are permitted. The solution works as follows: Suppose that a timer with an initial value of T is associated with the socket pair $\sigma_{out} = \{protocol, source\text{-}address, source\text{-}$

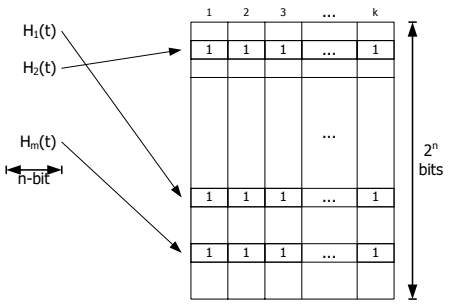

Figure 7. An example of a $\{k \times n\}$-bitmap, the core architecture for a bitmap filter.

$port$, $destination\text{-}address$, $destination\text{-}port\}$ of each outbound packet that is new to an edge router. If the socket pair σ_{out} is not new to the router, the value of the associated timer is simply reset to T. The timer reduces every time unit Δt. When the timer expires (reaches zero), the associated socket pair is deleted. For each inbound packet, the router extracts the socket pair σ_{in} and checks if its inverse $\overline{\sigma_{in}}$ exists. If it exists, the packet is bypassed; otherwise, it is dropped under certain probability P_d. The dropping probability P_d in our algorithm varies according to the uplink bandwidth throughput. It can be lower when the throughput is low and higher when the uplink is fully utilized.

The problem of the above solution is that the complexities for both storage and computations make it infeasible to deploy in a large scale network. Thus, a bitmap filter, which is a composite of k bloom filters [9] of equal size $N = 2^n$ bits, denoted as a $\{k \times N\}$-bitmap filter, is used instead. An example of a bitmap is illustrated in Figure 7. Each column in the bitmap represents a bit-vector of a bloom filter. For the convenient of explanation, in the algorithm, the bit-vector of the i^{th} bloom filter is written as $bit\text{-}vector[i]$.

At the initialization phase, all the bits on the $\{k \times n\}$-bitmap are set to zero and an index of the current bit vector idx is set to the first bit-vector. All the bloom filters in the bitmap share the same m hash functions, each of which should only output an n-bit value. An output that exceeds n-bit should be truncated. The bitmap filter comprises two algorithms, the $b.rotate$ algorithm, which clears expired bits from the bitmap, and the $b.filter$ algorithm, which marks and looks up bits in the bitmap. The algorithms are detailed in Algorithm 1 and Algorithm 2, respectively. The $b.rotate$ algorithm is quite simple. The algorithm runs every time unit Δt. When it is activated, the index of the current bit vector idx is set to the next bit vector and all bits in the previous bit vector are set to zero. For example, assume there are k bit vectors in a bitmap filter indexed from $\{1, \cdots, k\}$. If the current index is set to the 1^{st} bit vector, then the last bit vector will be the k^{th} bit-vector; however, if the current index is set to the $j^{th}(1 < j \leq k)$ bit vector, then the last bit vector will be the $(j-1)^{th}$ bit-vector.

Algorithm 1 The Timer Handler - $b.rotate()$

Require: An initialized $\{k \times n\}$-bitmap and an index to current bit vector idx.

1: $last = idx$
2: $idx = (idx + 1) \pmod k$
3: set all bits in $bit\text{-}vector[last]$ to zero
4: **return** idx

The bitmap is marked and looked up using the $b.filter$ algorithm, as shown in Algorithm 2. When a packet is received by an edge router, the $b.filter$ algorithm is applied to determine whether the packet should be bypassed or dropped. For an outbound packet, the $b.filter$ iteratively applies all the m hash functions on the socket pair σ_{out} and marks the corresponding bits in all bit vectors to a value of 1. Outbound packets are always bypassed. On the other hand, when an inbound packet is received, the $b.filter$ iteratively applies all the hash functions on the socket pair $\overline{\sigma_{in}}$ and checks if the corresponding bit in the current bit vector indicated by the index idx is marked or not. If a bit is not marked, then the packet will be dropped under a probability of P_d. The value of P_d can be dynamically adjusted according to the upload bandwidth throughput. An example of generating P_d is a similar form to the random early detection (RED) algorithm [10]. Given two threshold L, H $(L < H)$, and an indicator of upload bandwidth throughput b, the P_d is computed by Equation 1.

$$P_d = \begin{cases} 0 & \text{, if } b \leq L \\ \frac{b-L}{H-L} & \text{, if } L < b < H \\ 1 & \text{, if } b \geq H \end{cases} \quad (1)$$

Note that the bitmap filter is not necessary to use all fields in the socket pair σ to compute the hash value. Instead, for an outbound packet, the hash functions can be applied only to the parts of $\{protocol$, $source\text{-}address$, $source\text{-}port$, $destination\text{-}address\}$. In contrast, for an inbound packet, only $\{protocol$, $destination\text{-}address$, $destination\text{-}port$, $source\text{-}address\}$ are used to compute the hash value. The reason not to use all fields is to support the "hole-punching" [11] technique, which is usually used for a client host to create bypass rules on the network address translation (NAT) or firewall device for future inbound connections. The support to "hole-punching" can be enabled or disabled depending on the network administrator's choice.

In summary, the "mark" action is always performed on all bit vectors, the "look up" and the "clean up" actions are only performed for the current bit vector and the last bit vector, respectively. The combination of these operations achieves the same purpose as the naïve solution described

Algorithm 2 The Filtering Function - $b.filter()$

Require: An initialized $\{k \times n\}$-bitmap, an index of current bit vector idx, a conditional dropping probability P_d, and a packet pkt to be inspected.

```
if pkt is an outbound packet then
  for h ∈ hash-function list do
    j = h(σ_out)
    mark the j^th bit in all bit vectors as 1
  end for
else if pkt is an inbound packet then
  for h ∈ hash-function list do
    j = h(σ_in)
    if the j^th bit in bit-vector[idx] is 0 then
      p = a randomly generated number in [0, 1]
      if p < P_d then
        return DROP
      end if
    end if
  end for
end if
return PASS
```

at the beginning of this sub-section, which effectively limits the upload traffic sent from a client network.

4.3 Choose Proper Parameters

As stated in Section 4.2, several parameters for the bitmap filter must be decided. They are the k - the number of bit vectors in a bitmap, the N - the size of a bit vector, the Δt - the time unit to clean up a bit vector, and the m - the number of hash functions used in the bitmap filter. The k and N parameters decide how much storage space is required for the bitmap filter; and the k and Δt parameters decide the countdown time of the timer T_e mentioned in Section 3.3. Thus, given a moderate expiry timer T_e and a proper time unit Δt, the value k can be decided by $\lfloor \frac{T_e}{\Delta t} \rfloor$.

Recall the result in Section 3.3. T_e should not be too long, since the port-reuse effect may incur more false positives[2], which decrease the precision of the bandwidth limiter. In other words, a packet that should be dropped may be accepted by the limiter. However, to prevent overkilling connections with longer delays, T_e should not be too short either. A value below 60 seconds, such as 20 or 30 seconds, would be acceptable. On the other hand, the time unit Δt need not to be too short. Although a shorter Δt improves the timer's granularity, a Δt that is too short may raise the frequency of running bitmap clean-ups too much and thus reduce the overall performance of the system. A value of 4 or 5 seconds would be appropriate.

[2]The definition of false positives is defined in Section 5

The n is a flexible parameter. A network administrator can decide the value according to the number of concurrently active connections and the memory space that they are willing to devote to the system. Note that a small n will raise the possibility of false positives and reduce the effectiveness of traffic limiter. To avoid the problem, more hash functions (i.e., m) may be used to reduce false positives. When deploying such a system, administrators should consider a trade-off between storage space and computation power to decide the value of n and m. We further evaluate the effects of different sets of parameters in the next section.

5 Evaluations

In this section, we evaluate several aspects of the proposed solution by analyses and simulations.

5.1 False Positives and False Negatives

As our solution adopts an approximate algorithm to maintain outbound connection states, it may incur false positives and false negatives. The definition of a false positives is similar to that used in the original bloom filter paper [9]. That is, an inbound packet that should be dropped is accepted by the filter. In contrast, a false negative is an inbound packet that should be accepted is dropped. Since the bitmap filter works in flavor of a positive listing, only inbound packets with an out-in packet delay longer than the expiry timer T_e are filtered out. Thus, the number of false negatives is very low. As the result in Section 3.3 shows, false negatives should be lower than 1% when T_e is greater than 3.61 seconds.

However, we should focus more on false positives. Assume m hash functions are applied to a single inbound packet and the utilization of the current bit vector is $U = \frac{b}{N}$, where b is the number of marked bits in a bit vector. The probability p that a random inbound socket pair σ will penetrate the bitmap filter is

$$p = U^m = \left(\frac{b}{N}\right)^m. \tag{2}$$

The number of marked bits on the bit vector should be proportional to the number of active connections c inside a time unit of T_e. If we assume that the results of the hash functions seldom collide when the utilization of the bit vector is low, Equation 2 can be rewritten as

$$p \simeq \left(\frac{c \cdot m}{N}\right)^m. \tag{3}$$

Given a bit vector size N and the expected max number of active connections c, then to minimize the desired penetration probability p, we differentiate Equation 3 and get

$$p' = \left(\frac{c}{N} \cdot m\right)^m \left(1 + \ln\left(\frac{c}{N} \cdot m\right)\right). \tag{4}$$

Thus, m that minimizes the penetration probability p can be obtained by solving $1 + \ln(\frac{c}{N} \cdot m) = 0$, which is

$$m = \frac{e^{-1} \cdot N}{c}, \tag{5}$$

where e is the base for the natural logarithm. By replacing m in Equation 3 with $\frac{e^{-1} \cdot N}{c}$ when m minimizes the penetration probability p, the ratio of the expected max number of active connections c should satisfy

$$\frac{c}{N} \leq -\frac{1}{e \ln p}. \tag{6}$$

For example, if we adopt a bitmap filter of size $N = 2^{20}$ (about 1-million bits) with $k = 4$, and $\Delta t = 5$ seconds, and set the desired penetration probability to be roughly 10%, 5%, and 1%, the number of active connections inside a time unit $T_e = 20$ seconds should be less than 167K, 125K, and 83K, respectively. Compared with our trace data, which has only average 15K active connections inside a time unit of 20 seconds, these upper bounds are much higher than the actual traffic. The number of used hash functions m in the setup can be 3, and the memory space required by the bitmap filter is only $(k \times N)/8 = 512K$ bytes.

5.2 Performance

The bitmap filter is efficient because almost all operations can be performed in constant time. The processing time for an outbound packet is $O(m \times t_h) + O(m \cdot k \times t_m)$, where m is the number of used hash functions, t_h is the time taken to execute a hash function, k is the number of bit vectors to be marked, and t_m is the processing time to mark a bit. Processing inbound packets is simpler than for outbound packets. The required processing time is $O(m \times t_h) + O(m \times t_c)$ where t_c is the processing time need to check whether a bit on a bit vector is marked or not. Inbound packet processing is also a constant time operation. When an inbound packet is considered to be dropped, the bitmap filter drops the packet according to the dropping probability P_d. Computing the P_d requires only the knowledge of current bandwidth throughput, which is an essential component in off-the-shelf network devices.

The most time consuming operation may be the $b.rotate$ algorithm, which executes every Δt seconds. The algorithm first advances the current index idx to set to the next bit vector, and then resets all bits in the last bit vector to zero. Thus, the operation is proportional to the size of a bit vector, which is $O(n)$. However, since the memory space of a bit vector is fixed and continuous, implementing such an algorithm in software is simple and efficient. As all the components used in the algorithm already have corresponding hardware implementations, it is also easy to accelerate the algorithm by using hardware coprocessors.

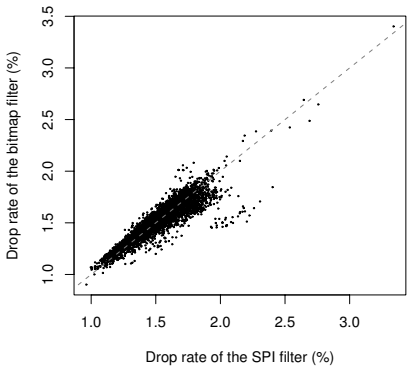

Figure 8. Comparison of the packet drop rates of the SPI and the bitmap filters. The gray-dashed line has a slope of 1.0**.**

5.3 Simulation with the Packet Trace

We also perform several simulations to verify the effectiveness of the bitmap filter. A bitmap filter and an SPI-based filter are both implemented. The input to both filters is the packet trace used in Section 3.3. First, we compare the packet drop rate of the two filters. The SPI filter is set to delete idle connections after 240 seconds, which is the default TIME_WAIT timeout used in the Microsoft windows operating system The bitmap filter is configured as follows: $N = 2^{20}$, $k = 4$, $T_e = 20$, $\Delta t = 5$, and drop all inbound packets without states. This constructs a 512K-byte bitmap filter that handles the out-in packet latency shorter than 20 seconds. As Figure 8 shows, the filters have similar packet drop rates, and the gray-dashed line has a slope of 1.0. The SPI filter has an average drop rate of 1.56% compared to 1.51% for the bitmap filter. This is because that the SPI filter knows the exact time of closed connections and can therefore drop packets precisely than the bitmap filter.

The second simulation is to show the effectiveness of the bitmap filter on the same packet trace data. The bitmap filter now monitors the bandwidth throughput of upstream traffic and blocks incoming connections when the uplink bandwidth throughput is high. The dropping probability P_d is generated by Equation 1 with a upper bound bandwidth limit H of 100Mbps and a lower bound bandwidth limit L of 50Mbps. To simulate a blocked connection, when an inbound packet is decided to be dropped by the bitmap filter, the socket pair σ of that packet is stored and all the future packets that match any stored σ or $\overline{\sigma}$ are all dropped without checking the bitmap. The configuration of the bitmap filter tries to control peer-to-peer upload traffic below an upper bound of 100Mbps. Figure 9-a and Figure 9-b show the

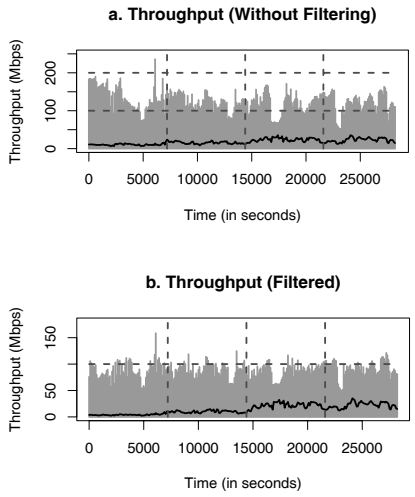

Figure 9. The performance of the bitmap filter to limit upload traffic.

original and the filtered bandwidth throughput, respectively. In the two figures, the black line indicates the downlink throughput and the gray part indicates the uplink throughput. It should be noted that both parts of the downlink and uplink traffic are limited. This is because some download peer-to-peer traffic are transfered in different inbound connections. Since the simulation is done with replayed packet trace, as the simulation is unable to block the outbound connections that may triggered by previously blocked inbound requests, the effect of the traffic filtering is limited. We believe that the filter can perform better in a real network environment. The result of simulation also shows that the 512K bytes $\{4\times2^{20}\}$-bitmap filter with 3 hash functions can properly limit uplink traffic for the small- or medium-scale client network.

6 Conclusions

The core spirit of peer-to-peer applications is to share with the public. Thus, a client host that running peer-to-peer applications always generates a considerable amount of upload traffic, which should be limited in a client network. However, with randomly selected port numbers and the use of protocol encryption, peer-to-peer traffic is hard to identify and manage. As the upload traffic are usually triggered by inbound request, in this paper, we propose a bitmap filter to bound the peer-to-peer upload traffic by controlling inbound requests. The proposed algorithm requires only constant storage and computation power. Analyses and simulations show that with a small amount of resources, an ISP can efficiently prevent the peer-to-peer traffic from affecting the normal operations of traditional Internet services.

References

[1] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, Jan. 2002.

[2] S. Sen and J. Wang, "Analyzing peer-to-peer traffic across large networks," *IEEE Transactions on Networking*, vol. 12, no. 2, pp. 219–232, Apr. 2004.

[3] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos, "Is P2P dying or just hiding?" in *Proceedings of IEEE Global Telecommunications Conference*, vol. 3. IEEE, Nov. 2004, pp. 1532–1538.

[4] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy, "Transport layer identification of P2P traffic," in *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM Press, 2004, pp. 121–134.

[5] N. Leibowitz, A. Bergman, R. Ben-Shaul, and A. Shavit, "Are file swapping networks cacheable? characterizing P2P traffic," in *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW)*, Aug. 2002, pp. 121–134.

[6] S. Sen, O. Spatscheck, and D. Wang, "Accurate, scalable in-network identification of P2P traffic using application signatures," in *WWW '04: Proceedings of the 13th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2004, pp. 512–521.

[7] V. Jacobson, C. Leres, and S. McCanne, *TCPDUMP public repository*. [Online]. Available: http://www.tcpdump.org/

[8] J. Levandoski, E. Sommer, and M. Strait, *Application Layer Packet Classifier for Linux*. [Online]. Available: http://l7-filter.sourceforge.net/

[9] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communication of ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[10] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[11] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," in *USENIX Annual Technical Conference*, Apr. 2005.