IEICE TRANS. COMMUN., VOL.E91–B, NO.12 DECEMBER 2008

INVITED PAPER *Special Section on Peer to Peer Networking Technology*

A Remedy for Network Operators against Increasing P2P Traffic: Enabling Packet Cache for P2P Applications

Akihiro NAKAO[†,††a)], **Kengo SASAKI**[†b)], ***and*** **Shu YAMAMOTO**[††c)], ***Members***

SUMMARY We observe that P2P traffic has peculiar characteristics as opposed to the other type of traffic such as web browsing and file transfer. Since they exploit *swarm effect* — a multitude of end points downloading the same content piece by piece nearly at the same time, thus, increasing the effectiveness of caching — the same pieces of data end up traversing the network over and over again within mostly a short time window. In the light of this observation, we propose a *network layer packet-level caching* for reducing the volume of emerging P2P traffic, transparently to the P2P applications — without affecting operations of the P2P applications at all — rather than banning it, restricting it, or modifying P2P systems themselves. Unlike the other caching techniques, we aim to provide as generic a caching mechanism as possible at network layer — without knowing much detail of P2P application protocols — to extend applicability to arbitrary P2P protocols. Our preliminary evaluation shows that our approach is expected to reduce a significant amount of P2P traffic transparently to P2P applications.
key words: *P2P, overlay networks, packet cache, P4P, future Internet*

1. Introduction

Recent Internet tomography has revealed that peer-to-peer (P2P) traffic has become dominant in volume over HTTP traffic, which used to represent the majority of the entire traffic exchanged on the Internet [1], [2]. The prevalence of P2P traffic indicates that users on the Internet have found advantages over the traditional communication in innovative data transmission where end points *directly* exchange data making use of *traffic dispersion* — dividing data into small pieces and transmitting them over multiple (disjoint) paths constructed by indirection (relaying at intermediaries) at participating peers — and also *caching* along such paths rather than traditional, single-path, server-client type of data exchange. P2P file sharing applications such as BitTorrent [3] exploit this direct data exchange, traffic dispersion, and caching to achieve efficient large-file transfer. It is also worth noting that the other work such as [4], [5] has reported to have achieved even better performance than BitTorrent in sharing large files through similar caching and traffic dispersion observed in P2P applications. Since recent statistics shows that a majority of P2P traffic carries large contents such as video, it makes sense for users to utilize the direct, dispersed, and cached data transmission.

Manuscript received September 7, 2008.
Manuscript revised September 17, 2008.
[†]The authors are with the University of Tokyo, Tokyo, 113-0033 Japan.
[††]The authors are with NICT, Koganei-shi, 184-8795 Japan.
a) E-mail: nakao@iii.u-tokyo.ac.jp
b) E-mail: qq086407@iii.u-tokyo.ac.jp
c) E-mail: shu@nict.go.jp
DOI: 10.1093/ietcom/e91–b.12.3810

However, dispersion and caching are made possible by indirection at peers. That is, ISPs that have customers running P2P file-sharing applications may be carrying traffic for those who belong to the other ISPs and therein running the same applications, since the applications may automatically reroute pieces of data being indirected at their customers. Accordingly, ISPs are not necessarily serving their own customers alone any more, but are subject to extra cost incurred by customers of the other ISPs [6]–[8].

Several ISPs regard this extra cost of carrying traffic for the other ISPs' customers as a significant problem and it has come to our attention whether ISPs should take actions specifically on P2P traffic [9], [10]. The actions recently taken by ISPs to mitigate P2P traffic range from banning P2P traffic completely or restricting it [11] to control peering of P2P applications by a new protocol looking at the underlying Internet topology, e.g., restricting peers to only connect to the others inside the same ISP or those topologically nearby such as in P4P [12]–[14]. However, recent articles have reported that "network neutrality" problem has been raised and discussed — there exists a general consensus that deeply examining packet contents and restricting packet forwarding according to the contents is largely considered problematic in public services and that ISPs should refrain from banning a particular traffic [15], [16].

On the other hand, the solution such as P4P approach appears to be appealing [12]–[14]. P4P claims to be able to confine P2P traffic effectively to a network domain, achieving 45% improvement in completion time of P2P transmissions, while improving the link utilization of P2P traffic at backbone links of major ISPs by 50% to 70% [14]. The test conducted in P4P signifies a turning point in the history of P2P technology and it turns out the problem is not P2P technology itself but is how we deploy it. However, such solution is still under active research and yet not known to be a viable solution to the problem. For instance, even though P4P offers a method to restrict selection of peers within a certain vicinity in the Internet, such enforcement may not be realistic in practice, since we may not be able to restrict any communication patterns under "network neutrality" as long as the communication is achievable through overlay networks, let alone P2P. In addition, P4P approach may deprive P2P applications of the advantage of traffic dispersion.

In this paper, we take a different approach to tackle this problem than the above existing proposals. We observe that P2P traffic has peculiar characteristics as opposed to the other type of traffic such as web browsing and file transfer.

Since they exploit *swarm effect* — a multitude of end points downloading the same content piece by piece nearly at the same time, thus, increasing the effectiveness of caching — the same pieces of data end up traversing the network over and over again within mostly a short time window. Although this characteristics causes apparently inefficient redundant data transmission, it means, on the other hand, that if we cache the packet-level data at network layer, we should be able to greatly benefit from high hit-ratio of the cache. In the light of this observation, we propose a *network layer packet-level caching* for reducing the volume of emerging P2P traffic by compressing it transparently to the P2P applications — without affecting operations of the P2P applications at all — rather than banning or restricting it. Although there have been a few proposals to cache P2P traffic [17]–[19], the criticism is that these caches must be designed to speak specific protocols, thus limiting their generality and applicability to closed protocols. Our approach is different in that we aim to provide as generic a caching mechanism as possible at network layer — without knowing much detail of P2P application protocols — to extend applicability to arbitrary P2P protocols.

2. Design

This section describes design decisions we have made to enable packet-base caching for P2P applications.

Figure 1 depicts the design of our proposed scheme for reducing the volume of P2P traffic. There exists a cloud of target transit networks between ISPs where end systems exchange P2P application packets and where our system is supposed to reduce the P2P traffic. Since a P2P application often exploits *swarm effect* where a multitude of peers download the same content nearly at the same time or within a short time window to accelerate the data transmission, we expect that these P2P application packets get largely duplicated and the packets carrying the same data repeatedly traverse the cloud of transit networks in a fairly short period of time.

Therefore, we place a set of intelligent routers so that we can compress the traffic at an *ingress* router to the cloud and decompress it at an *egress* router on a packet basis so that we can significantly reduce the traffic traversing the cloud of the transit networks. If we can drastically reduce the *size* of the packets by compression, especially to much less than the maximum transmission unit (MTU), we may also be able to decrease the *number* of the packets by consolidating multiple small packets.

There are various scenarios possible as to where this scheme can apply. For instance, two ISPs peering each other could cooperatively install routers with this cache capability to reduce the P2P traffic between them. An ISP might pay its neighboring ISPs for installing such routers on behalf of itself. For another example, an ISP could install these routers within its domain to reduce its local traffic.

As Fig. 1 shows, there could be multiple ingress and egress routers that can perform compression and/or decompression. However, for the sake of simplicity, we focus on a pair of ingress and egress routers for the rest of this paper.

In order to achieve the goal of reducing P2P traffic, we have made the following design decisions.

Compression and Caching

Although naïve compression on a packet basis reduces the size of the packet, we can gain more benefit if we use caching as well as compression at the same time. At the egress of the transit networks, we cache P2P traffic on a packet basis with a hash of the packet payload as a key and the packet payload as a value. At the ingress to the transit networks, we calculate a hash value of the payload of each packet, and send only a hash value if we already have a cached entry for its corresponding packet content at the egress. Through this method, we can implicitly compress the packet effectively without any explicit compression, since a hash value is usually small compared with a data given. The only caveat is that we need a coordination between the ingress and the egress routers as shown in Fig. 1.

Transparency to P2P Applications

We decide to achieve two kinds of transparency to P2P applications. First, our compression and caching scheme needs to be transparent to end-to-end communications, since introduction of intermediate agents may affect the original P2P applications' behaviors. In addition, such intermediaries may invite a security concern such as unnecessarily creating a target for various attacks on end-systems such as DDoS attack. Second, our system needs to be independent of P2P application protocols. Our designed scheme must be applicable to a wide range of P2P applications. Application-layer proxy-cache [19] often supports only a limited number of P2P applications, since it must deal with application protocols to interact with peers. Our approach requires a minimal assumption that a P2P application packet usually has a specific leading byte-sequence for a piece of data carried in a packet. We call such byte-sequence Prefix. As far as we detect Prefix, we can maximize the effectiveness of compression and caching by aligning caching units with Prefix.

Deep Packet Inspection and Signature Detection

We must be able to *deep-inspect* packets traversing the routers to distinguish compressible P2P packets from the others. Just as in the other P2P traffic analyzer, our system can perform deep packet inspection (DPI) to identify and locate Prefix. For instance, BitTorrent uses a specific Prefix right before data pieces in a packet payload. After locating a Prefix, we apply generic hashing and caching described in the previous paragraphs above, expecting that the subsequent bytes right after the Prefix should appear repeatedly in a multitude of flows, thus, give us an opportunity to perform effective caching on a packet basis. In other words,

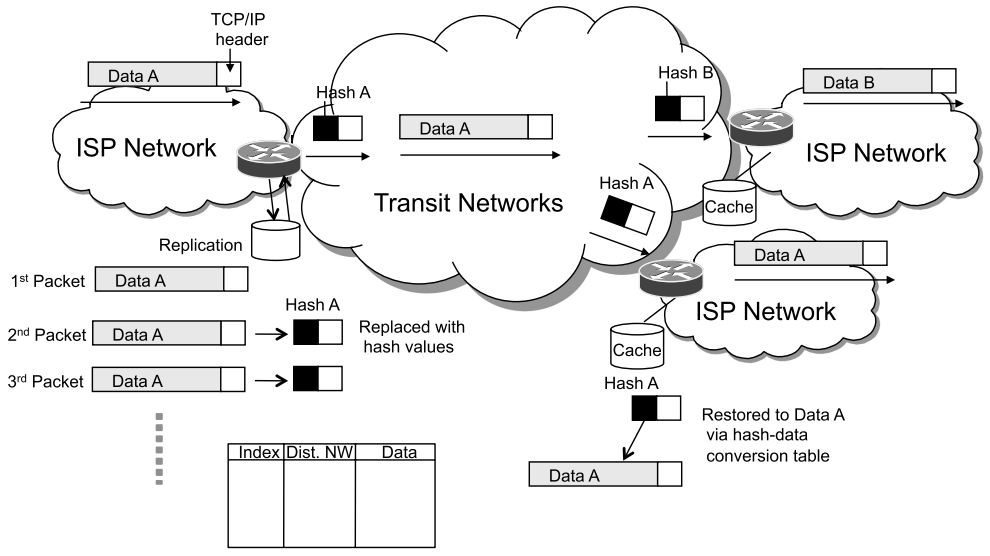

Fig. 1 Design overview.

we should be able to apply the same technique to arbitrary P2P applications as long as they use specific leading byte-sequence. We assume most P2P applications have such Prefix's to denote data (piece). Note that although, in this paper, we mainly discuss our method with this assumption for Prefix, as long as we can detect the beginning of each data piece, our proposal is applicable as discussed in Sect. 3. As an aside, we could actually propose as a standard that P2P applications should be equipped with a specific Prefix in a packet to facilitate caching, but this is left for our future research.

3. Architecture

This section describes the detailed architecture of our proposed prototype system that enables packet-base caching for P2P applications.

3.1 Overview

Our prototype system consists of a set of ingress and egress routers at the edge of the target transit networks and cache storage attached to routers as shown in Fig. 2. As briefly described in Sect. 2 and Fig. 1, a packet gets cached at the egress router. Everytime a packet gets cached, a cache entry maps a hash value of the payload as a key to the payload itself as a value. Also, the egress router reports on this cache entry (send the hash value) to the ingress router so that everytime it sees a packet containing the content that has the same hash value traversing, it replaces the payload content with its hash value, thus, compresses the packet to smaller size. For this reason, the ingress router also has a cache. When the egress router receives this compressed packet, it will locate the hash value in the packet, retrieve the original content, and replace the hash with the content. As Fig. 2 shows, the roles of ingress and egress routers are interchangeable according to the direction of packet flows, thus, they are depicted symmetrically except that packets get compressed at the ingress router and reconstructed at the egress router in practice.

3.2 Packet Layout

This section describes the layout of a compressed packet in detail. The layout of the original (uncompressed) packet and that of the compressed packet are shown in Fig. 3 (in case for the packet with Prefix) and Fig. 4 (in case for the packet without Prefix), respectively. In this example, we divide the payload of a packet into 128-byte long chunks and compress each chunk into a 16 byte (128-bit) MD5 hash value for the simplicity's sake, but obviously these values could be changed to arbitrary ones to achieve various flavors of compressions and hashing schemes.

When an ingress router discovers the packet with a specific Prefix in its payload through DPI, it records the flow (5-tuple) and the pointer to the Prefix in TCP sequence number space as *soft-state*. Since we know the byte-offset of the Prefix from the beginning of the payload whose TCP sequence number is recorded in the TCP header, we can easily figure what TCP sequence number the Prefix corresponds to by adding the byte-offset to the TCP sequence number in the TCP header. Starting from this Prefix, the router divides the payload into 128-byte chunks and attempts to compress them into their hash values if the ingress router knows that the egress router already has inserted the cache entries for the hash values. Since the ingress router records the flow and the Prefix pointer in the TCP sequence number space, it can continue this 128-byte aligned compression across multiple packets by comparing TCP sequence numbers and the Prefix pointer. When it observes a new Prefix for a particular flow, it just updates the Prefix pointer to the new value.

As Figs. 3 and 4 show, the compressed packet has the

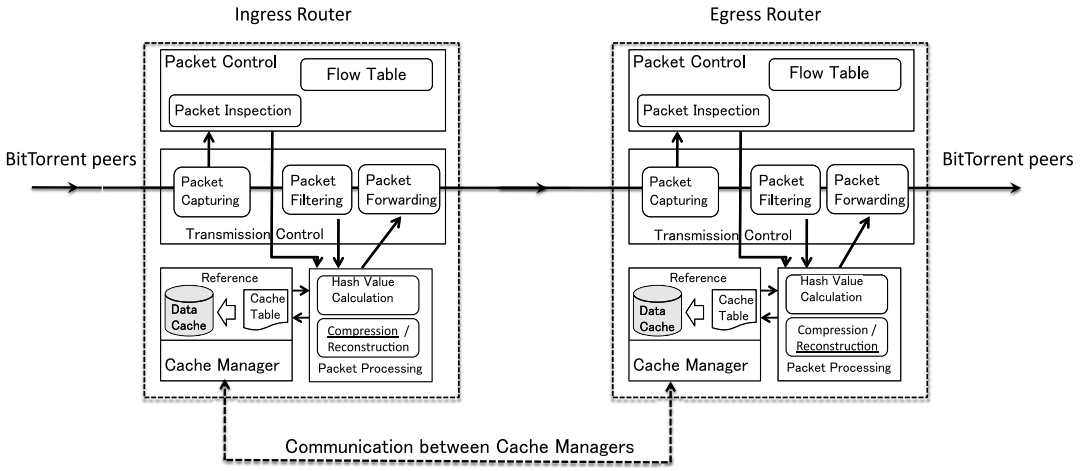

Fig. 2 Architecture overview.

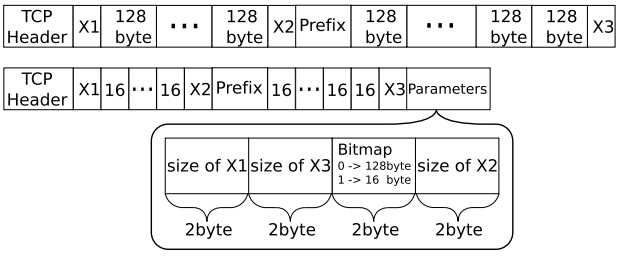

Fig. 3 Packet layout with Prefix (Leading byte-sequence ahead of data piece).

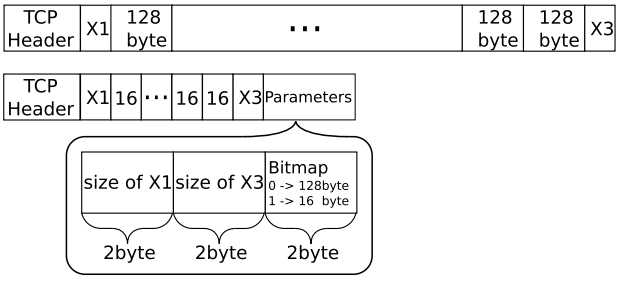

Fig. 4 Packet layout without Prefix (Leading byte-sequence ahead of data piece).

field called Parameters at the end that contains Bitmap flag (2 bytes) indicating which 128-byte chunks have been compressed, i.e., replaced with their 16-byte hash values[†].

For example, a bit 1 means compressed while a bit 0 means uncompressed. The Parameters field also contains X_1, X_2, and X_3 (2 bytes each) that denote the offset bytes at the beginning, right before the Prefix, and at the end of the packet (before Parameters), respectively. These offset bytes are to enable caching of the content in 128-byte chunks that are precisely aligned with 128-byte boundaries, since usually a packet has a variable length due to TCP congestion control and flow control so the beginning of the payload and the end of it may not necessarily be aligned with the 128-byte boundaries. If the beginning of the payload is not aligned with the 128-byte boundaries from Prefix, the first $X_1(< 128)$ bytes are not compressed. Similarly, if the end of the payload is not aligned, the last $X_3(< 128)$ bytes are not compressed. If the payload contains the Prefix, the same offset strategy applies to the last $X_2(< 128)$ bytes before the Prefix are preserved [††]. Note that X_2 will not appear in the packet without Prefix in its payload. Note that letting $X_i(t)$ $(i = 1, 2, 3)$ be the offset in the t-th packet, we have the following equation to quickly figure out $X_1(t+1)$ from $X_3(t)$.

$$X_1(t+1) + X_3(t) = 128 \tag{1}$$

Thanks to the adjustment through these offset bytes $X_i(t)$ $(i = 1, 2, 3)$, all the 128-byte chunks are divided precisely aligned with the 128-byte boundaries starting from the last observed Prefix. This means a content is always divided into the same set of 128-byte chunks, no matter how a content is divided into packets, or no matter which flow these packets are conveyed through. Because of the alignment mechanism achieved through these offset bytes $X_i(t)$ $(i = 1, 2, 3)$, we can effectively cache the content piece by piece across multiple flows as long as they convey the same content.

3.3 Router Composition in Click Model

Both ingress and egress routers are supposed to perform compression and decompression of the packets. Although these processes are not tied to a particular implementation, in this section, we attempt to describe our proposed architecture by showing the compression and decompression process by using Click [20] element diagrams.

[†]In our prototype, the bitmap has only two bytes that are enough to cover MTU of 2048 bytes (= 16×128 bytes). However, we are not tied to these specific numbers of bytes allocated in Parameters, data piece size, compression schemes, etc.

[††]Note that the current prototype only deals with up to one Prefix in each packet for the sake of simplicity, but we can easily extend it to support multiple ones.

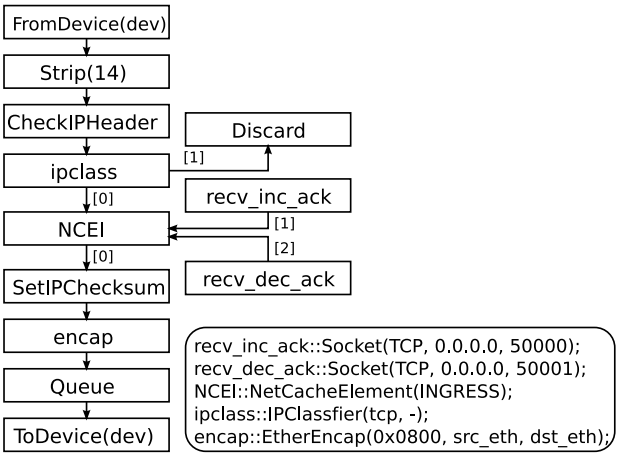

Fig. 5 Structure of ingress router in the Click model.

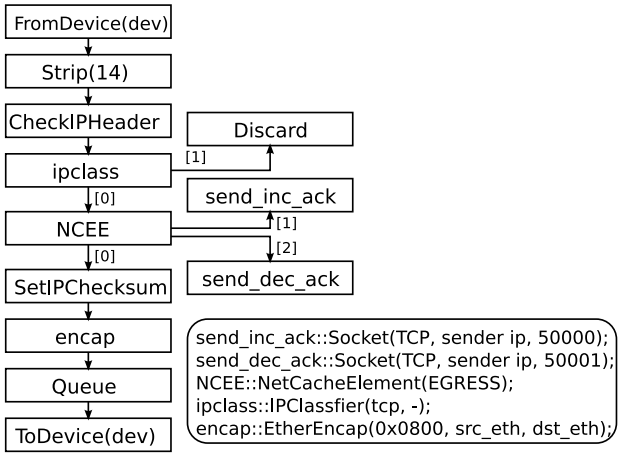

Fig. 6 Structure of egress router in the Click model.

Click [20] is an open-source *extensible* router implementation developed to facilitate experiments with a new routing protocol and with innovative packet processing at routers. In the Click model, a router is composed of multiple modules called *elements* chained together to achieve packet processing and routing. Each element implements a simple function such as stripping off an Ethernet header from a packet given, packet queuing, packet scheduling, etc. Many useful elements are already built-in and thus just combining them and configuring them with parameters allow us to build a complex piece of router software. If we would like to implement a new element, it is fairly easy to do so, and after adding a new element, we can chain it together with the existing elements.

For example, for our purpose, we add a new Click element called NetCacheElement that compresses and decompresses packets if they are inserted into the packet processing path. In describing our proposed architecture, we explain how this NetCacheElement works to compress and decompress P2P packets as in the Click model.

3.3.1 INGRESS

The structure of an ingress router in the Click model that compresses P2P packets is depicted in Fig. 5. This section describes how a packet gets processed along the chain of the elements including our new element NetCacheElement.

First, after a packet is received through the FromDevice(dev) element, where dev is a network device that the packets get captured from, its Ethernet header gets stripped by the Strip(14) element. After the packet has its integrity checked as an IP packet by the CheckIPHeader() element, the packet gets compressed through the NetCacheElement(INGRESS) element. Here, the parameter INGRESS passed to the NetCacheElement denotes that this element works as compressor at an ingress router. In our current prototype architecture, this compression is simply done by calculating MD5 hash values of 128-byte chunks in the payload of the packet. After this compression is performed, the compressed packet gets through SetIPChecksum() to have its checksum corrected and through encap() to attach its Ethernet header back, and finally gets forwarded to ToDevice(dev) to be pushed on the wire.

NetCacheElement(INGRESS) will not compress all the received packets. As Fig. 5 shows, only the 128-byte blocks that have the hash values reported from the recv_inc_ack element will be converted into the hash values. Note that NetCacheElement(INGRESS) has two incoming ports, in addition to one outgoing port [0], where a port represents an entrance or an exit for packets in general. Ports [1] and [2] are used for receiving signals from an egress router in this scenario. This recv_inc_ack element receives a signal (a set of hash values) from the egress router and feeds the signal to NetCacheElement(INGRESS) through the port [1] so that it may only convert 128-byte chunks of data that correspond to the hash values received from the egress router.

On the other hand, the recv_dec_ack element receives a signal (a set of hash values) from the egress router and feeds the signal to NetCacheElement(INGRESS) through the port [2] so that it may *stop* converting 128-byte chunks of data that correspond to the hash values received from the egress router. This element and the signaling is necessary since when the egress router decides to evict some of the cache entries, it needs to notify of the ingress router to stop compressing packets so that it may not fail to reconstruct the original content from the hash values.

3.3.2 EGRESS

The structure of an egress router is shown in Fig. 6. Just as in the case with the ingress router, after a packet is received through the FromDevice(dev) element with some device dev, its Ethernet header gets stripped by the Strip(14) element. After the packet has its integrity checked as an IP packet by the CheckIPHeader() element, the packet gets decompressed through the NetCacheElement(EGRESS) element. After this decompression is done, the reconstructed

(original) packet goes to SetIPChecksum() element to have its checksum corrected and through encap() to attach its Ethernet header back, and finally gets forwarded to ToDevice(dev).

However, NetCacheElement(EGRESS) may receive uncompressed packets or partially compressed packets where only a few 128 byte blocks in their payloads are compressed. In this case, NetCacheElement(EGRESS) generates a 16-bit hash value for each 128-byte uncompressed block and stores a mapping between the hash value and the uncompressed block. Then it sends the hash value to send_inc_ack element. Now send_inc_ack packages multiple such hash values into a signal and send it to the ingress router so its recv_inc_ack may receive the signal and start compressing the blocks corresponding to the hash values. When the cache storage of the egress router needs to evict some cache entries, their hash values get sent to send_dec_ack. Then send_dec_ack packages these hash values into a signal and send it to the ingress router so its recv_dec_ack may receive the signal and stop compressing the blocks corresponding to the hash values.

3.4 Compression and Caching Process

As described in Sect. 3.3, both ingress and egress routers are constructed using NetCacheElement element in the Click model. This element takes one argument to denote if it works for the ingress router (NetCacheElement(INGRESS)) or for the egress one (NetCacheElement(EGRESS)). This section explains the detail of this element and how it operates.

3.4.1 NetCacheElement(INGRESS)

Figure 7 shows the flowchart of internal operations of NetCacheElement(INGRESS), the building block of the ingress router as depicted in Fig. 5.

NetCacheElement(INGRESS) (NCEI in short hereafter) has three input ports and one output port. It captures P2P traffic from the port [0] and performs compression and caching of the traffic, while it also communicates with its counter-part egress router using the ports [1] and [2]. Note that Deep Packet Inspection (DPI) and signature detection for specific P2P traffic before the P2P traffic gets captured into the port [0] are omitted here, since this stage is considered rather straightforward and often discussed elsewhere.

When a P2P flow gets captured at the port [0], NCEI first checks if the flow has been observed before or not. If NCEI detects a new flow, it registers a soft-state flow entry that retains a pointer to Prefix in its TCP sequence number space. If NCEI observes the existing flow, it updates the pointer to the last observed Prefix. The flow entry gets evicted if either a preset timer expires or NCEI sees TCP FIN and RST.

When NCEI receives a packet with Prefix, it calculates X_1, X_2, and X_3; otherwise X_1 and X_3. After removing these offset bytes $Xi(i = 1, 2, 3)$ and Prefix, the packet

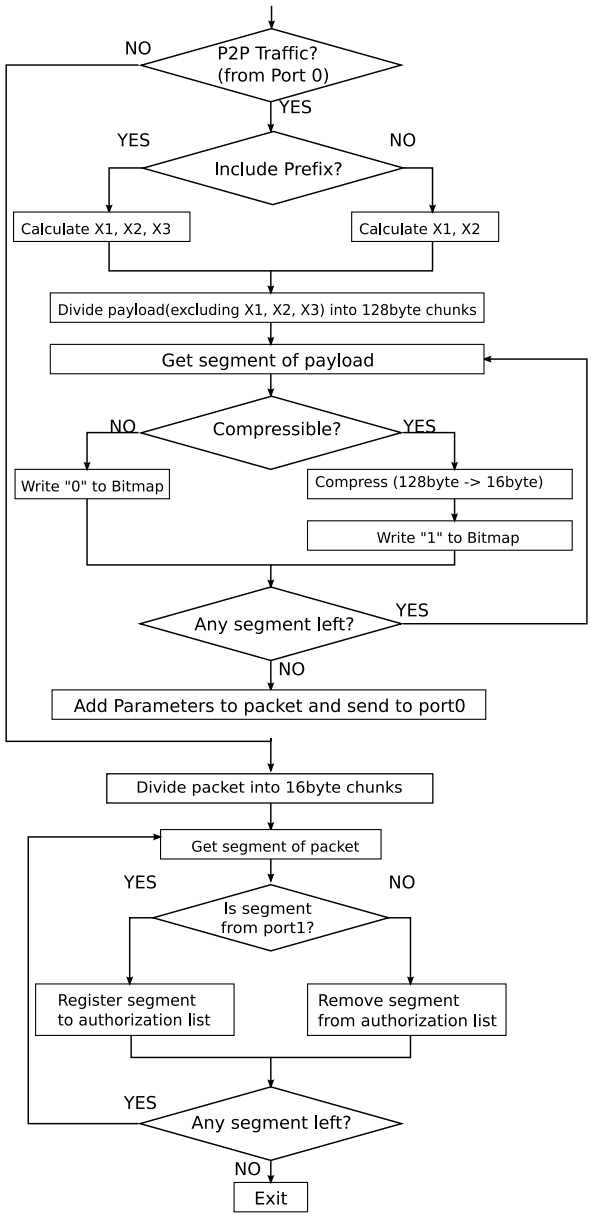

Fig. 7 Flow of operations at ingress router.

payload gets divided into 128-byte blocks. Then NCEI judges whether each 128-byte block can be compressed, i.e., replaced with its 16-byte hash value or not, according to 16-byte hash values notified by the egress router through recv_inc_ack and recv_dec_ack elements. If the 16-byte hash value of a 128-byte block matches one of the hash values received through recv_inc_ack element, NCEI replaces the 128-byte block with its 16-byte hash value; if it matches one of the hash values through recv_dec_ack element, NCEI will not compress the block. After iterating this process through the packet payload, depending on which 128-byte blocks have been compressed, NCEI composes a bitmap, Bitmap where 1 is set for a compressed block and 0 for an uncompressed one and attaches it to Parameters bytes at the end of the packet.

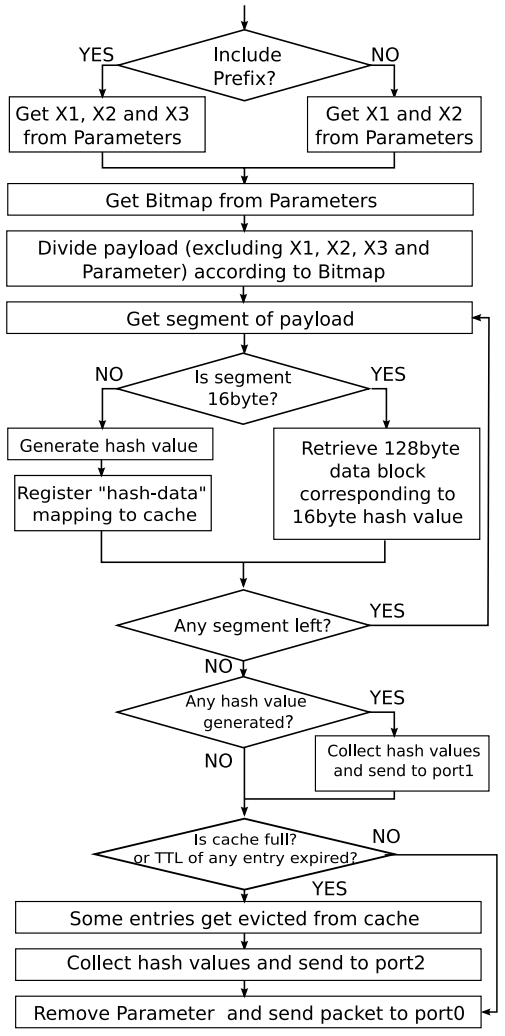

Fig. 8 Flow of operations at egress router.

Communication with the counter-part egress router is rather simple. NCEI just receives a series of 16-byte hash values from the port [1] or the port [2]. The port [1] connects to recv_inc_ack and the port [2] to recv_decc_ack as shown in Fig. 5. Depending on from which ports NCEI receives a series of 16-byte hash values, it registers the hash values as compressible ones (if from the port [1]) and unregisters them as incompressible ones (if from the port [2]).

3.4.2 NetCacheElement(EGRESS)

Figure 8 shows the flowchart of internal operation of NetCacheElement(EGRESS), the building block of the ingress router as depicted in Fig. 6.

NetCacheElement(EGRESS) (NCEE in short hereafter) has one input port [0] and three output ports [0], [1] and [2]. When NCEE receives a packet with Prefix from the port [0], it calculates X_1, X_2, and X_3 just in the same way as in NCEI; otherwise X_1 and X_3. After removing these offset bytes $X_i (i = 1, 2, 3)$, Prefix and Parameters, the packet payload gets divided into 128-byte and 16-byte blocks according to the bitmap, Bitmap. Then NCEE generates a 16-byte hash value for each 128-byte block and registers the mapping between the 16-byte hash value and the 128-byte data in its cache. For each 16-byte block in the packet payload, NCEE looks up in the cache to retrieve the corresponding 128-byte data that has been registered previously.

In the course of the process above, if any new hash values are generated, they are collected and sent to the port [1] so that they may be conveyed to the ingress router through send_inc_ack element. If the cache gets filled and NCEE needs to evict some of the cache entries, NCEE collects the hash values evicted from the cache and send to the port [2] so that these hash values may be notified to the ingress router through send_dec_ack element.

4. Evaluation

This section presents preliminary evaluation of our proposed scheme. We conduct two kinds of evaluations here. One is the evaluation of a Click implementation of our architecture on Emulab [21] and the other is the application of our algorithm to the real packet trace.

4.1 Preliminary Evaluation on Emulab

We have implemented NetCacheElement in Click and run ingress and egress routers on Emulab. Emulab is a network testbed where we can reserve a set of PC servers and run arbitrary operating systems (OSes) connected in whatever configuration we link via VLAN. On Emulab, we use four PC servers of identical hardware specification and connect them in serial via 100Mbps Ethernet. The configuration of these four PC servers is BitTorrentPeer-NetCacheIngress-NetCacheEgress-BitTorrentPeer where BitTorrentPeer sends P2P packets to the other one over NetCacheIngress/-Egress. The packets are transmitted from a BitTorrentPeer, compressed at NetCacheIngress, then decompressed at NetCacheEgress, and finally reach the other BitTorrentPeer. This process is completely transparent to BitTorrentPeers. We use MD5 for the hash function as described before.

Figure 9 compares the traffic observed before NetCacheIngress and that observed between NetCacheIngress and NetCacheEgress when we transfer The same file[†] up to *three times* between BitTorrentPeers. At the first time, since NetCacheEgress has an empty cache, nothing is compressed so the traffic amount stays the same. However, at the second time and the third, since NetCacheEgress has populated its cache at the first time, packets are accordingly compressed and the traffic is significantly reduced to 29% at the second time and to 26% at the third time. The number of the cache entries after the third transmission is 1226 in this experiment.

The compression gets improved at the third time compared to the second time, since the cache gets gradually

[†]The file contains 180 KB of zipped web content including html, jpg, gif, and various scripts.

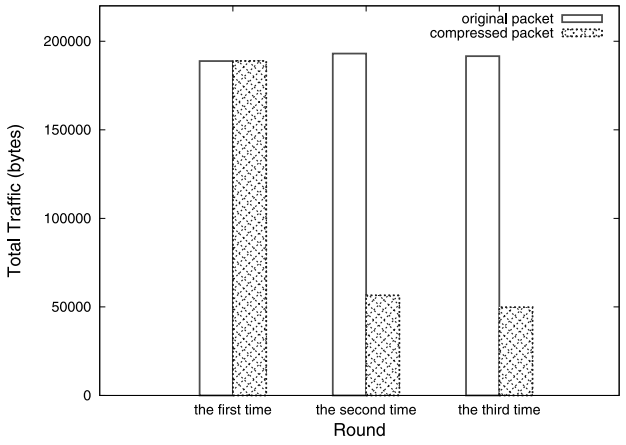

Fig. 9 Comparison between the traffic observed before NetCacheIngress and that observed between NetCacheIngress and NetCacheEgress.

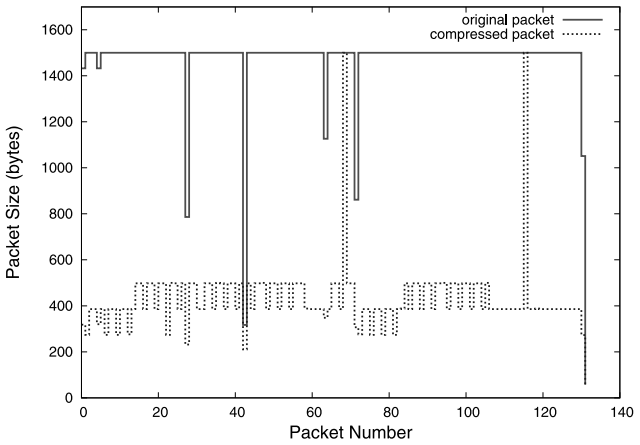

Fig. 10 Comparison between the traffic observed before NetCacheIngress and that observed between NetCacheIngress and NetCacheEgress (the second round).

warmed up (populated). The reason is, as described at the end of this section in detail, with a small probability, Prefix may span across a packet boundary, and 128-byte boundaries may be calculated based on a wrong (the previously observed) Prefix, thus, wrongly aligned data chunks may be compressed into wrong hash values, so the cache gets contaminated with wrong hash values at first. However, since packet boundaries are random, the same Prefix may not span across a packet boundary at later times on different flows, so the cache gradually gets populated with the correct hash values when the same contents are repeatedly sent at different times over different flows[†]. Note that for each round of experiments, the total traffic generated is not the same, since BitTorrentPeers exchange control messages of which amount varies according to the situation.

Figure 10 compares the packet size observed before NetCacheIngress and that observed between NetCacheIngress and NetCacheEgress *at the second time*. The horizontal axis shows the number of packets forwarded and the vertical axis represents the packet size. The packet size before NetCacheIngress is shown as "original packet" and that between NetCacheIngress and NetCacheEgress is shown as "compressed packet." Most packets are suppressed to 20–30% in bytes compared to the original size. We also observe that some packets are not compressed at all even in the second time.

Figure 11 shows cumulative distribution of compression ratio of packet size at the second time. More than 95% of the packets are suppressed below 35% in bytes compared to the original size.

Figure 10 shows that two packets are not compressed at all. There are two possible reasons why this occurs. One is that Prefix spans across a packet boundary with a marginal probability. In this case, with the current scheme, packets may get divided using a wrong pointer to Prefix (the previously observed Prefix) at egress and the hash values observed at ingress at the second time would not match any of these hash values generated at the egress if the Prefix in question observed at ingress does not span across packet boundary at the second time.

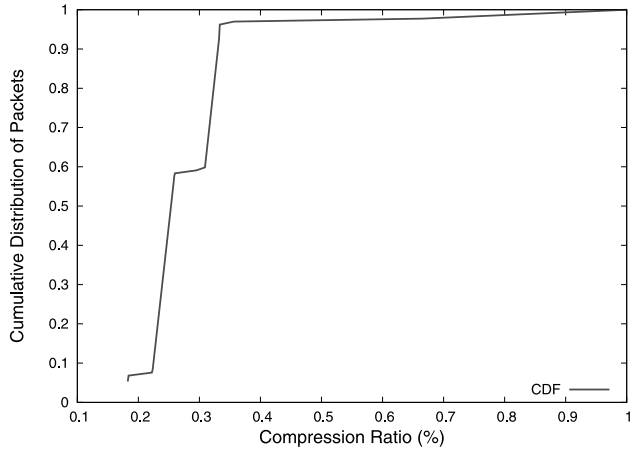

Fig. 11 Cumulative distribution of the traffic compression ratio (the second round).

The other possible reason is that notification of cacheable hash values from the egress to the ingress arrive later than compression attempts occur at the ingress. As Fig. 9 shows, the reason why the third time compresses better than the second time is exactly due to this reason.

In this preliminary study, hash collision has not been considered yet. However, we can easily incorporate the hash collision checking by adding a finger print of each packet and verify it after reconstructing the packet to be the same finger print, etc. The other considerations such as retransmission triggered by hash collision and cache eviction strategy are the topics for our future work.

[†]This artifact can be easily resolved by taking care of Prefix spanning across a packet boundary, but, the system becomes more complex than the current one. In this paper, it is left for future work.

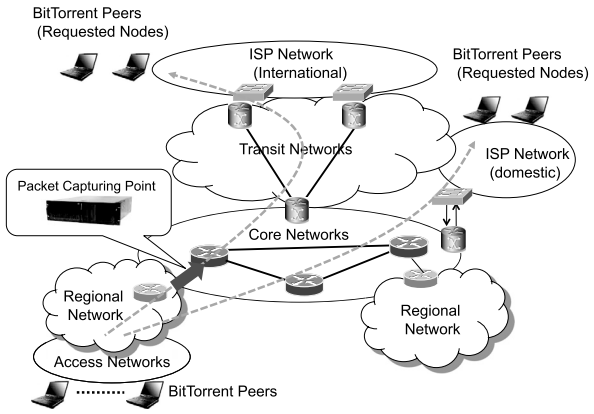

Fig. 12 Configuration of packet capturing at a border gateway router.

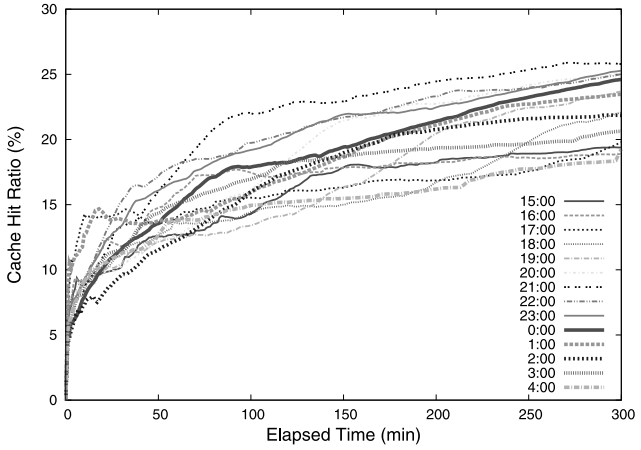

Fig. 13 Cumulative cache hit ratio during each 5-hour (300-minute) period starting from every hour between 15:00 and 4:00 on the following day.

4.2 Preliminary Evaluation Using Packet Trace

As shown in Fig. 12, we have captured packets at a transit router of an ISP for the duration of 18 hours starting from 15:00 on the 18th of June in 2008 till 9:00 on the following day. Note that P2P traffic such as BitTorrent usually dominates during these hours every day.

During this 18-hour period, we have captured about 1.7 TB traffic data and among them we have observed about 25% of BitTorrent traffic in volume on average. We divide this 18-hour observation period into 5-hour time slots and run our caching algorithm ignoring notification delay between ingress and egress. That is, we divide each packet payload into 128-byte blocks starting from Prefix pointers and generate MD5 hash values. Then we simply account for duplicate hash values for the subsequent packets. As long as subsequent duplicate hash values are observed no matter how much later, we regard this event as a cache hit; otherwise a cache miss.

Figure 13 shows cumulative cache hit ratio during each 5-hour time slot starting from every hour between 15:00 and 4:00 on the following day. Figure 14 averages all the 14

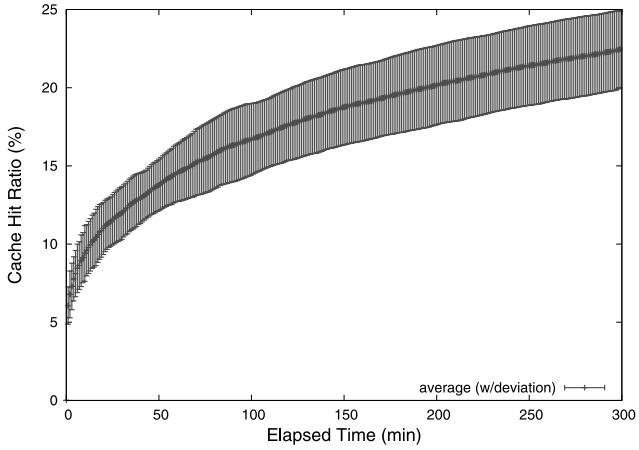

Fig. 14 Cumulative cache hit ratio averaged at every minute in the elapsed time for each 5-hour (300-minute) period.

curves shown in Fig. 13 at every minute in the elapsed time for 5 hours. Error bars shown with the average curve in Fig. 14 represent the standard deviation.

As we can see from these graphs, the averaged cumulative cache hit ratio slowly increases over time and reaches 22.5±2.5% after the elapsed time of 5 hours. Although not shown here for the sake of brevity, after the elapsed time of 9.5 hours, the cumulative cache hit ratio reaches close to 30%. The result of our preliminary study indicates that our algorithm is expected to reduce the BitTorrent traffic by about 20–30% depending on how much cache space we could afford.

5. Related Work

There have been a number of studies on application layer cache for eliminating redundant traffic across the network [19], [22]–[24]. Web proxy caches deployed within ISP localizes Web response traffic and effectively eliminates redundant data transmissions among ISP networks. For example, pCache [19], an application layer proxy cache for P2P traffic, reduces redundant P2P traffic.

The application layer approach, however, requires caching semantics to interact with clients using a specific protocol. Application layer caching approach eliminates application objects, but it cannot eliminate redundant data in packet or byte granularity. Another disadvantage is that collocation of application cache in ISP is often regulated with privacy, copyright, etc. It could also be a target of DDoS attacks, then it would not work any more.

The packet-level caching schemes [25], [26] are universal schemes for eliminating redundant traffic that can work on a protocol independent architecture. It is a generic scheme that can be applied to any redundant packet context. Thus it can eliminate more redundant traffic in packet or byte granularity rather than in application-layer object granularity. The packet level cache can effectively eliminate high level of repetition in P2P packets.

The packet cache is implemented using a shared cache

architecture, where each packet gets its payload cached with its hash value in a device collocated at a border gateway router in a network domain. The cache device can detect a fraction of payload data identical to the previously observed one. In the shared cache architecture, only the pointers are passed in place of duplicate bytes between the two cache devices placed at different network domains, while end users behind caches exchange the original packets. This concept has been studied to eliminate redundancy in any kind of data packets.

The prior work [25], [26] leverages a Rabin fingerprint algorithm for finding repetitive content. The fingerprint algorithm allows faster computation than MD5 hash calculation, but the footprints consume a large amount of memory. To reduce the memory size, it is required to sample a subset of fingerprints per packet or to interleave fingerprint generation [26]. The fingerprint algorithm, which is effective for packets repeated in short time interval, will not be applicable to P2P traffic. The P2P traffic has a long tail distribution in the repetition interval. The cached data has to stay for a long time to improve the cache hit ratio according to our preliminary observation of P2P traffic. In our approach, instead of inspecting all the packets, we enforce redundancy elimination specifically for P2P traffic. Our novel packet cache architecture that detects Prefix indicating the beginning of a data piece and enables caching piece by piece allows us to eliminate a large volume of redundant traffic generated by P2P overlays. In addition, while the existing work on packet caches has not discussed the data synchronization mechanism between far-end pairs of caches, our packet caching architecture aims to reduce traffic between ingress and egress routers located possibly at a distance thus addresses the importance of the synchronization between the far-end caches.

6. Future Work

We plan to extend our work in several directions. First, we would like to evaluate the effectiveness of our proposal by extending our prototype implementation and investigating more traffic data than we have examined. Second, we are currently devising more space-saving caching scheme to make our system more effective and efficient. Third, another extension of our system is to support a multiple sets of ingress and egress routers. Especially, we would like to investigate more on the communication between multiple caches installed on those routers and report its performance evaluation. Fourth, we intend to apply this scheme to the other P2P application than BitTorrent to evaluate the effectiveness, with or without specific Prefix bytes. There are a couple of interesting P2P systems that employ end-to-end encryption. Although the report [2] shows that such encrypted traffic is unpopular (less than 20%), we plan to devise a way to deal with such applications as well. Finally, we may actually be able to set a standard that P2P applications should install a specific Prefix in a packet to facilitate caching demonstrated in our proposed system.

7. Conclusions

The traffic characteristics of the Internet has greatly changed from a web-centric pattern to P2P-dominant one. ISPs and network operators are facing a dilemma that they would reduce P2P bandwidth that serves not their own customers but those of others, while they would like to keep their pipes full.

In order to resolve the dilemma of ISPs and network operators, we propose a *network layer packet-level caching* for reducing the volume of emerging P2P traffic, transparently to the P2P applications — without affecting operations of the P2P applications at all — rather than banning it, restricting it or modifying P2P systems themselves. Since P2P traffic has peculiar characteristics *swarm effect* — a multitude of end points downloading the same content at the same time, caching the packet-level data at the network layer could enable a great hit-ratio of the cache. Although there have been different caching methods proposed, our approach is different in that we aim to provide as generic a caching mechanism as possible at the network layer to extend applicability to arbitrary P2P protocols.

We have done two kinds of preliminary evaluations. First, we have built a prototype of our proposed scheme in Click on Emulab and show that with our prototype system, more than 95% of the packets are suppressed below 35% in bytes compared to the original size. Second, we have taken packet trace at a transit router of an ISP for the duration of 18 hours and exercise our algorithm on the packet trace. The averaged cumulative cache hit ratio of our system have reached 22.5±2.5% after the elapsed time of 300 minutes, and close to 30% after the elapsed time of 9.5 hours. These results show that our algorithm is expected to reduce the P2P traffic with a specific Prefix significantly by about 20–30% depending on the cache space.

Acknowledgment

The authors would like to thank Tomohiko Ogishi, Yuichiro Hei, Kiyohide Nakauchi and Megumi Shibuya for insightful discussions for this research.

References

[1] A. Parker, "The true picture of peer-to-peer filesharing," July 2004. http://www.cachelogic.com
[2] "iPoque: Internet study 2007," Aug. 2007. http://www.ipoque.com
[3] "BitTorrent." http://www.bittorrent.com/
[4] K. Park and V.S. Pai, "Deploying large file transfer on an HTTP content distribution network," Proc. First Workshop on Real, Large Distributed Systems (WORLDS 2004), San Francisco, CA, Dec. 2004.
[5] K. Park and V.S. Pai, "Scale and performance in the CoBlitz large-file distribution service," Proc. Third Symposium on Networked Systems Design and Implementation (NSDI 2006), San Jose, CA, May 2006.
[6] S. Seetharaman and M. Ammar, "Characterizing and mitigating inter-domain policy violations in overlay routes," ICNP'06: Proc. 2006 IEEE International Conference on Network Protocols, pp.259–268, IEEE Computer Society, Washington, DC, USA, 2006.

[7] S. Guha, N. Daswani, and R. Jain, "An experimental study of the skype peer-to-peer VoIP system," Proc. IPTPS 2006, 2006.
[8] H.X.Y.R. Yang, "A measurement-based study of the skype peer-to-peer VoIP system," Proc. IPTPS 2007, 2007.
[9] T. Karagiannis, P. Rodriguez, and K. Papagiannaki, "Should Internet service providers fear peer-assisted content distribution?," Proc. Internet Measurement Conference (IMC'05), Oct. 2005.
[10] K.P. Gummadi, R.J. Dunn, S. Saroiu, S.D. Gribble, H.M. Levy, and J. Zahorjan, "Measurement, modeling, and analysis of a peer-to-peer file-sharing workload," Proc. 19th ACM Symposium on Operating Systems Principles (SOSP-19), Oct. 2003.
[11] "Cisco. network-based application recognition (NBAR)." http://www.cisco.com/en/US/docs/ios/qos/configuration/guide/clsfy_traffic_nbar.html
[12] H. Xie, A. Krishnamurthy, Y.R. Yang, and A. Silberschatz, "Explicit communications for cooperative control between P2P and network providers." http://www.dcia.info/activities/#P4P
[13] H. Xie, Y.R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz, "P4P: Provider portal for applications," Proc. ACM SIGCOMM'08 Conference, pp.351–362, Aug. 2008.
[14] H. Xie, A. Krishnamurthy, A. Silberschatz, and Y.R. Yang, "P4P: Eplicit communications for cooperative control between P2P and network providers," May 2007. P4PWG Whitepaper.
[15] "The Washington post news," Sept. 2008. http://www.washingtonpost.com/wp-dyn/content/article/2008/09/04/AR2008090402280.html
[16] "CBC news," May 2008. http://www.cbc.ca/technology/story/2008/05/12/tech-bell.html
[17] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak, "Cache replacement policies revisited: The case of P2P traffic," Proc. GP2P, 2004.
[18] G. Shen, Y. Wang, Y. Xiong, B.Y. Zhao, and Z.L. Zhang, "HPTP: Relieving the tension between ISPs and P2P," Proc. IPTPS 2007, 2007.
[19] M. Hefeeda, C.H. Hsu, and K. Mokhtarianz, "pCache: A proxy cache for peer-to-peer traffic," Proc. ACM SIGCOMM'08 Conference, p.539, Aug. 2008.
[20] R. Morris, E. Kohler, J. Jannotti, and M.F. Kaashoek, "The Click modular router," ACM Trans. Comput. Syst. (TOCS), vol.18, no.3, pp.263–297, Aug. 2000.
[21] "Emulab." http://www.emulab.net/
[22] "Velocix(formerly CacheLogic)." http://www.velocix.com/
[23] "PeerApp." http://www.peerapp.com/
[24] "Oversi." http://www.oversi.com/
[25] N.T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," Proc. ACM SIGCOMM'00 Conference, Sept. 2000.
[26] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: The implications of universal redundant traffic elimination," Proc. ACM SIGCOMM'08 Conference, pp.219–230, Aug. 2008.

Akihiro Nakao received B.S. (1991) in Physics, M.E. (1994) in Information Engineering from the University of Tokyo. He was at IBM Yamato Laboratory/at Tokyo Research Laboratory/at IBM Texas Austin from 1994 till 2005. He received M.S. (2001) and Ph.D. (2005) in Computer Science from Princeton University. He has been teaching as an Associate Professor in Applied Computer Science, at Interfaculty Initiative in Information Studies, Graduate School of Interdisciplinary Information Studies, the University of Tokyo since 2005. (He has also been an expert visiting scholar/a project leader at National Institute of Information and Communications Technology (NICT) since 2007.)

Kengo Sasaki received B.S. in Faculty of Electro-Communications, from the University of Electro-Communications. He is a first year Master student and belongs to Applied Computer Science Course, Graduate School of Interdisciplinary Information Studies, the University of Tokyo.

Shu Yamamoto received B.S., M.E., and Ph.D. degrees in electronics engineering from the University of Tokyo, Japan, in 1977, 1979, and 1989, respectively. He joined KDD (currently KDDI) in 1979. He has been engaged in the research on optical communication and mobile networks. Since 2007, he is working on new generation networking in NICT. He received the best paper and the achievement awards from IEICE in 1995 and 1997, respectively.