# HPTP: Relieving the Tension between ISPs and P2P

Guobin Shen[1], Ye Wang[1,2], Yongqiang Xiong[1], Ben Y. Zhao[3], Zhi-Li Zhang[4]
[1] Microsoft Research Asia, Beijing, P.R.China
[2] Electronic Engineering Department, Tsinghua Univ., P.R.China
[3] Computer Science Department, U.C. Santa Barbara
[4] Computer Science Department, Univ. of Minnesota at Twin Cities

## ABSTRACT

Measurement-based studies indicate that there is a severe tension between P2P applications and ISPs. In this paper, we propose a novel HTTP-based Peer-to-Peer (HPTP) framework to relieve this tension. The key idea is to exploit the *widely deployed web cache proxies* of ISPs to trick them to cache P2P traffic. This is achieved via a process we refer to as "HTTPifying": we segment (if necessary) large P2P files or streams into smaller chunks, encapsulate and transport them using the HTTP protocol so that they are cache-able. We outline the design of several key tools of the proposed HPTP framework – HTTPifying, cache detection and usability test tools, and describe a cache-aware tree construction (CATC) protocol for delivering P2P streaming traffic as an example to showcase the HPTP framework. Simulation results demonstrate that HPTP can lead to significant performance improvement. We argue that the HPTP framework will benefit both ISPs and end users (P2P as well as normal web users) by significantly reducing *network overload* caused by repetitive P2P traffic.

## 1. INTRODUCTION

While peer-to-peer (P2P) applications eliminate the problems of "flash crowd" and server overload that afflict servers in the traditional client-server systems, their increasing popularity, in particular, emergence of P2P streaming applications such as P2P IPTV, has created another problem – namely, traffic surges and network congestion at Internet Services Providers (ISPs). Numerical ISPs have reported that P2P traffic accounts for a major portion of the Internet, surpassing any other application category such as web, and and is bound to increase further. The network congestion caused by P2P traffic not only affects users of P2P applications, but also those of other applications such as web. Furthermore, it has also been reported that more than 92% P2P traffic traverse transit/peering links among ISPs [1], thereby affecting the bottom line of (customer) ISPs. As an additional side-effect of this problem, the overwhelming bandwidth consumption of peer-to-peer systems – despite the inherently scalable design – may prevent them from scaling further, at least within University-like environments, as the measurement study [2] concludes.

The excessive traffic overload (and perhaps more importantly, the resultant financial burden) incurred by P2P applications on ISP networks has prompted many of them to resort to blocking or rate-limiting P2P traffic. Such "reactionary" measures, on the other hand, often irk users who may take their business elsewhere. A more "constructive" approach is to attempt to deploy cache proxies to cache P2P traffic, similar to web caching. In fact, several P2P caching schemes have been proposed [2, 4–7], and a few startups have also appeared. Unfortunately, there are a few obstacles in deploying P2P caches: firstly, P2P caching systems are likely to be very complicated. Unlike web traffic standardized in using HTTP transport through few dedicated ports like 80, there is no a standard P2P protocol and every P2P protocol uses its own port. Therefore, P2P caching systems are forced to take an ad hoc approach by enumerating and handling every P2P protocol. So far such an approach appears feasible, since there are only a few popular P2P systems that contribute most of the traffic at the moment. Yet another drawback of this ad hoc approach is the requirement of regular update of the P2P cache engines to handle newly emerged popular P2P protocols. Secondly, extra, possibly huge, investment is required for the equipment and facility purchase and also the administrative cost.

In this paper, we propose a novel HTTP-based Peer-to-Peer (HPTP) framework to relieve this tension. The key idea is to exploit the *widely deployed web cache proxies* of ISPs to trick them to cache P2P traffic. This is achieved via a process we refer to as "HTTPifying": we segment (if necessary) large P2P files or streams into smaller chunks, encapsulate and transport them using the HTTP protocol to ensure them cache-able by properly specifying the cache-control related directives of the HTTP request/response header. The key difference between HPTP and other P2P caching proposals lies in that we utilize the *existing* web cache infrastructure deployed by ISPs

The efficacy of HPTP depends on how successfully we can trick the web cache proxies to cache the HTTPified P2P traffic. To increase the hit rate, a cache-aware P2P overlay construction protocol is highly desired. However, unlike normal P2P applications where peers' addresses are known, most caching proxies (especially those deployed by ISPs which are *transparent caches*) are unknown. This necessitates a tool to detect the caches in the first place. In this paper, we present a light weight cache detection tool called $\mathcal{H}$Ping and, as a first case study to cache-aware overlay construction, a cache-aware tree construction protocol that can be applied to a practical streaming scenario. We perform some experiments and simulations to demonstrate the effectiveness of the $\mathcal{H}$Ping tool for cache detection and usability test, and the significant performance improvement to P2P users and traffic reduction on the backbone and transit/peering links among

ISPs due to HTTPification.

The remainder of this paper is organized as following. In Section 2 we provides a brief introduction to web caching and discuss related work. In Section 3 we present the proposed HPTP framework and the key tools. Experimental results are reported in Section 4. The paper is concluded in Section 5.

## 2. BACKGROUND AND RELATED WORK

### 2.1 Brief Introduction to Caching Proxies

A caching proxy (or cache for short) usually intercepts the TCP connection of a web request and *splits* it into two separate TCP connections, one to the client and the other to the server. The logic behind this design is to *always* perform cache checking first before attempting to make a connection to the server. The latter connection will be established only if a cache miss happens. It is such design that leads to shorter response latency and reduces the traffic to the server.

Upon receiving a request, the cache engine must quickly determine if it still stores the response. This requires the response to be uniquely indexed with hints from its request and lookup to be performed efficiently. The unique indexing is achieved by indexing the response using its URL which is intrinsically unique and efficient lookup is achieved through hashing.

The network host address in a URL can be expressed using hostnames or IPs, and more interestingly, in an HTTP session, up to three network host addresses may be specified, therefore, we want to understand if the hostname and IP are interchangeable and which network host address are used in the cache's indexing scheme. We experimented with three popular caching proxies, a Cisco Cache Engine (Model: 505), Microsoft ISAS and Squid. We found that 1) hostnames and IPs are considered different in indexing a response; 2) the response is indexed with preference `Hostname_get`, `Hostname_host`, `Hostname_con`. Our test message is:

```
telnet Hostname_con 80
GET Hostname_get/helloworld.html http/1.1
HOST Hostname_host
```

Many different factors can affect the cache-ability of a particular response, and these factors interact in a complicated manner. In general, for a response to be cache-able, one needs to ensure the size of the object is suitable and certain cache-control directives are properly set in both the request and the response.

Finally, because caching proxies are shared among many users, they are, therefore, essential services for ISPs and many organizations (e.g., corporations and universities). As a result, they are typically deployed at some strategic points such as near the organization's network gateways or near ISPs' Point of Presence (POP) in different locations.

### 2.2 Related Works

P2P traffic of a small ISP was found to be highly repetitive, showing great potential for caching [4]. In [2], initial analysis revealed that the outbound hit rate could reach approximately 85%, and the inbound hit rate reaches to 35% even though the cache has not fully warmed. Significant locality in the Kazaa workload was further identified in [5], which implies a 63% cache hit rate under extremely conservation trace-driven estimation. P2P systems exhibit good stability and persistence at the prefix and AS aggregation levels and suggest inserting local indexing/caching nodes or applying traffic engineering may be a promising way to manage the P2P workload in an ISP's network [3]. Besides the data messages, query messages in Gnutella networks are found to exhibit temporal locality and therefore cache-able [7].

The aggregate popularity distribution of objects is found to deviate from Zipf curves [5] and further modeled by a Mandelbrot-Zipf distribution [6]. A novel caching algorithm based on object segmentation and partial caching is proposed to cache each object by a portion that is proportional to its popularity. Trace-based simulations show that a relatively small cache size would lead to up to 35% byte hit rate [6].

A few startups like CacheLogic [1], Sandvine (`www.sandvine.com`), P-Cube (`www.p-cube.com`, acquired by Cisco in August 2004) etc., have developed hardware or software P2P caching systems. They all use routing policy enforcement to look up ISP's own network before looking further afield and adopt the divide and conquer method to handle different P2P protocols. It is well anticipated that regular updates are needed as P2P protocols evolve, so the annual support service is necessary.

The key difference between HPTP and all above mentioned P2P caching proposals is that we utilize the *already deployed* caching proxies and there is no extra adoption cost for ISPs.

Finally, Content Delivery Networks (CDN) has been a mature business since they are compelling to content providers because the responsibility for hosting content is offloaded to the CDN infrastructure. CDN is related to our work in the sense that they also use caches, but unlike HPTP that uses already deployed caching proxies, they deploy and use reverse proxy caches.

## 3. HPTP FRAMEWORK

In this section, we present several key components of the HPTP framework and a specific cache-aware tree construction protocol.

### 3.1 HTTPifying Tool

Since caching proxies typically caches only HTTP traffic, an indispensable component of HPTP framework is the HTTPifying tool, which segments (if necessary) the files or streams into small-sized chunks, use HTTP protocol for the transport of the resultant chunks and ensure such HTTP wrapped chunks are cache-able by specifying the correct cache-control related directives of the HTTP response header.

The reason we want to segment the original file is three folds: 1) to make it cache-able since most web caches imposes constraints on the size of cache-able objects; 2) to allow partial caching and fine cache replacement, which has proved to be crucial with certain cache replacement schemes [6]; 3) to exploit the potential of soliciting content from multiple senders as in BitTorrent.

Obviously, HTTPifying incurs some overhead. The over-

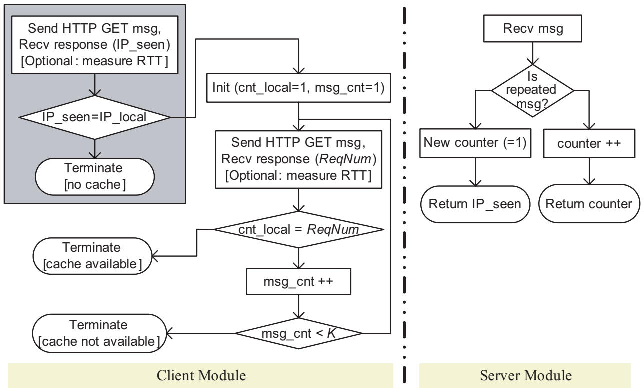


**Figure 1: The overall process of $\mathcal{H}$Ping. In the client module, the shadowed box is for cache detection, and the rest for usability detection.**

head equals to the size of HTTP wrapper divided by the segment size. If the segment size is set to 256KB, then the overhead is less than 1%.

## 3.2 $\mathcal{H}$Ping Tool

Optimal cache placement problem has attracted in depth studies and we expect it is worth more study in a P2P setting. However, in the HPTP context, because the caches are already deployed, we need to discover where such caches are deployed. Moreover, besides telling the existence of caches, we also want to learn the usability (i.e., how likely the cache will cache our HPTP traffic) of discovered caches. We have developed a light weight cache detection and cache usability test tool, called $\mathcal{H}$Ping, that can fulfill these requirements.

### 3.2.1 Caching Proxy Detection

The $\mathcal{H}$Ping performs cache detection based on the fact that a caching proxy *splits* a web request into two separate TCP connections, one to the client and the other to the server. This fact implies that the source IP the server sees from the request will be different from the original source IP (the IP of the requesting client) if there exists a cache in between. Therefore, we can tell the existence of a cache by comparing the original source IP against the source IP seen by the server.

$\mathcal{H}$Ping contains two modules: a client module and a server module (i.e., the daemon). The overall process of $\mathcal{H}$Ping is illustrated in Figure 1 and is elaborated below. Let *Peer A* ($P_A$) and *Peer B* ($P_B$) denote the pinging peer and the peer being pinged, respectively.

$P_A$ first sends an HTTP GET request message (referred to as $\mathcal{H}$Ping message hereafter) to $P_B$. If it is the first time $P_B$ receives the request, it creates a counter (initialized to one) for the new unique request and responds with a cache-friendly HTTP response with the content being the requestor's IP address it saw; otherwise, $P_B$ increments the counter associated with that request and responds only the counter. $P_A$ compare the IP address returned from $P_B$ with its own IP address. If they are the same, then we can conclude that there is no cache between the two peers; otherwise, there exists a cache and of which the IP address is also known.

Note that $\mathcal{H}$Ping may lead to possible false positive conclusion for the NAT/NAPT users. That is, there is actually no cache in between, but the $\mathcal{H}$Ping would conclude that there exists one because the IP address seen by the server is actually the client's NAT'ed (external) IP and differs from the client's own (internal) IP. Fortunately, the false positive conclusion does not hurt much (except possible waste of few $\mathcal{H}$Ping requests) because the "non-existing", falsely claimed cache is doomed not to pass the usability. Also, for most of the organizational networks, caching proxies are usually deployed on the gateway which implies that most of the seemingly false positive conclusions are actually correct. One limitation of $\mathcal{H}$Ping is that it can only tell the one closest to $P_B$ even if there may exist multiple caches in the path from $P_A$ to $P_B$. Nonetheless, we can progressively refine the locations of caches by recursively applying the cache detection logic, as done in the cache-aware tree construction.

### 3.2.2 Cache Usability Test

$\mathcal{H}$Ping performs cache usability test using *chained $\mathcal{H}$Ping messages*. The message chain is formed by $K$ subsequent same $\mathcal{H}$Ping messages. Still using $P_A$ and $P_B$ as examples. $P_A$ issues up to $K$ same $\mathcal{H}$Ping messages, one by one, immediately after the response to a previous request is received and processed. As above said, during the cache detection phase, $P_B$ has already associated a counter to each unique request, and the counter will be incremented (i.e., $ReqNum$++) for repeated request and included in $P_B$'s cache-friendly response. $P_A$ checks the response and test if the $ReqNum$ has increased. If the $ReqNum$ does not change, we can conclude that there indeed exists a cache between $P_A$ and $P_B$ (i.e., not a false positive case) and the cache is immediately usable, and the procedure terminates. If all $K$ $\mathcal{H}$Ping messages are sent but no conclusion can be drawn, then we simply conclude that the cache in between is not immediate usable such as running out of capacity or a false positive case caused by NAT/NAPT.

In $\mathcal{H}$Ping, $K$ is a system parameter related to the available caching capacity and also the cache replacement policy. We have not obtained a good estimation method for it. Instead, we follow the intuition by setting an initial large $K$ and dynamically reduce it by looking at the incremental steps of the returned $ReqNum$. Its rationale lies in that fact that the incremental speed of $ReqNum$ gives the hint of how many other peers are performing the probing concurrently, i.e., $ReqNum$ is an indicator of popularity. Moreover, $\mathcal{H}$Ping does not differentiate the requests from different peers, therefore, all peers are actually performing the cache detection and usability test collectively. This will lead to an accuracy estimation if the user base is large. Otherwise, if each peer issues a unique $\mathcal{H}$Ping message, it would be hard to tell the likelihood of the usability of the cache.

## 3.3 Cache-aware Tree Construction (CATC)

In a naive case, we can simply let the source HTTPify the P2P data and ask all peers to request data from the source directly, using HTTP transport. We refer to this scheme as *naive HPTP*. In some sense, naive HPTP is similar to HTTP tunneling except that we deliberately make the traffic cache-able through HTTPifying. However, this is a passive and best-effort leverage of caches. The extent to which the caches are utilized depends on the (geographical) distribution of peers and the caches. Nevertheless, it is still beneficial because the caches are usually strategically deployed. Another drawback of this naive scheme is that the source may risk heavy burden and becoming performance bottleneck since there is no guarantee on the cache hit.

To avoid such situation, we build cache-aware delivery tree with explicit control on the selection of caching proxies. This is achieved via the cache-aware tree construction (CATC) protocol described below. Once the tree is built, each peer only requests data from its parent, instead of the source as in the naive HPTP case.

#### 3.3.1 The CATC Protocol

We regard all peers and the source as in a large cluster at the beginning with the source being the cluster head.

1. All peers in the same cluster perform cache detection and usability test against the cluster head, and record (in stack order) the head information locally.

2. All peers report their results and own IP addresses to a (new) DHT node and remove their records from the previous one. Peers are further clustered (naturally) according to their detected caches. Those failed to discover new usable caches remain at their previous cluster and form an orphan set.

3. The DHT nodes appoint the peer whose IP address[1] is the closest to the source as the new cluster head (through IP matching) and inform all peers in the same cluster.

4. Above steps are recursively applied until there is no new usable caches can be found any further.

5. Finally, the tree is constructed recursively in a reverse order, starting from the finest clusters: peers in the same cluster form a subtree by directly connecting to the cluster head. This step is repeated until all the peers are recruited into the tree. In case of a large orphan set, we may build a tree out of it using normal P2P tree building logic, but use HPTP transport strategy.

Note that we have chosen to use a DHT to organize the collected cache information. Alternatively, we can use a server for this purpose. However, DHT naturally helps to cluster the peers since peers reporting to the same DHT nodes are covered by the same caching proxy. This avoids an explicit clustering process as would be the case if a server were used. Also, using DHT is a more robust and scalable way if we want to collect the cache information for a longer term.

#### 3.3.2 Handling Peer Dynamics

Peer dynamics handling usually represents a big obstacle in any P2P system design. Unlike other P2P systems, peer dynamics handling in the HPTP framework becomes much easier because of the recruited caches are indeed "giant peers": powerful, reliable, dedicated and strategically deployed. Their existences help to hide away the peer dynamics, besides boosting the delivery performance, as detailed below:

*Peer leave or failure*: the system keeps *silent* as much as possible to peer leave or failure. If leaf nodes left the system, there is no impact at all. If some intermediate nodes of the resulting tree (i.e., those who have been $\mathcal{H}$Ping'ed) left the system, there is no change to children peers at all (because the content may have been cached already and cache can help to response) unless the children peers receive a "connection refused" error message (indicating the content is not cached). In this case, the children peers will react by simply popping up another peer from their local stacks that have been built during the tree construction process.

*Peer joining*: newly joining peers always follow the CATC procedure to reach the finest cluster. When no new useful cache can be found, it adds itself to the orphan set at the corresponding level and directly connects to the last successfully $\mathcal{H}$Ping'ed peer. One interesting artifact is that even if an intermediate node has actually left the system when a later peer joins, it is still possible for that peer to reach a finer subtree of that intermediate node, as long as its response to $\mathcal{H}$Ping is still cached. Peers in orphan set may periodically perform peer joining procedure in case there are caches warmed up after their usability test.

We want to emphasize here that the robustness of the cache-aware tree to the peer dynamics is a direct result of the design logic of caching proxies: always perform cache checking first before attempt to make connections. This property of caching proxy also makes the maintenance of the cache-aware tree very simple. Unlike other tree maintenance protocol, we do not need heartbeat message to test the liveness of the peers. Similarly, there is no need to perform periodical optimization for the cache-aware tree. Instead, only peers experiencing low performances may perform opportunistic optimization by rejoining the tree.

## 4. EXPERIMENTS

We have performed some preliminary cache detection and usability test experiments with peers from many universities and also performed some simulation study on the cache-aware delivery tree.

### 4.1 Cache Detection and Usability Test

Ideally we would have liked to test the existence of caching proxies between PlanetLab nodes. Unfortunately, we cannot run our daemon on the PlanetLab nodes, due to the fact that the Port 80 is reserved for administration purpose. As a result, we used a node from Tsinghua university (China) as the $\mathcal{H}$Ping target, and performed cache detection tests from various PlanetLab nodes using $\mathcal{H}$Ping. Rather disappointingly, *no* caches are found. This somehow confirmed the message in [8] that reads: "Although interception caches can also be located on backbone networks, it is not very common."

The second set of experiments is that we asked some (now 19) friends from many universities (spreading China, USA,

[1]For peers behind NAT/NAPT, external IPs are required.

Canada, and Hongkong) to $\mathcal{H}$Ping the same target at Tsinghua. Table 1 shows the cache detection and usability test results. We see that 11 out of 19 nodes have detected a usable caching proxy on their way to Tsinghua University.

| Nodes from | Success | False Positive | Failure |
|---|---|---|---|
| Univ's in China | 5 | 2 | 2 |
| Univ's in USA | 1 | 0 | 1 |
| Univ's in Canada | 1 | 0 | 1 |
| Univ's in HongKong | 1 | 0 | 0 |
| MSR Asia | 3 | 1 | 1 |

**Table 1: Cache detection and usability test results.**

Although preliminary, the positive results indicate that there a large amount of web caches already deployed in the Internet. We feel the basic idea of HPTP, i.e., to turn these hidden cache into "giant peers", is very promising.

## 4.2 Performance of CATC

To evaluate the performance of HPTP and compare it with normal P2P, we used the GT-ITM [9] to generate an Internet-like topology with 100 routers, four of which are purely transit routers while the rest are stub ones providing access services. 1000 peers are randomly connected to different stubs and their access bandwidths are randomly distributed between 1Mbps to 10Mbps. In the text below, $K$ denotes the number of usable caches and $H$ denotes the actual byte hit rate of those caches.

We simulate and compare three schemes, namely normal P2P, naive HPTP and the CATC HPTP, using three metrics: average latency between a peer and its parent (could be another peer, a cache or the source), average bandwidth for peers, backbone traffic. The normal P2P delivery tree (i.e., an application layer multicast tree) is built using the shortest path routing with latency as path metric. Naive HPTP is achieved by directly HTTPifying the resulting P2P tree. Note that we do not constrain the number of children a peer can supply, but all the children will compete for the egress link bandwidth of that peer.

In the first experiment, we set $K = 30$ and let $H$ be fixed to 67% (according to the results in [4]) or varying with the request popularity. More specifically, we set $H_i = \min(c \cdot \log W_i, 80\%)$ with $W_i$ be the number of peers sharing the same cache and $c$ a normalization constant. To be more realistic, the hit rate is capped at 80%. The results (averaged over 10 runs) are shown in Figure 2 to Figure 4 for the four metrics.

Figure 2-(a) and Figure 2-(b) show the comparison for the average parent-to-child latency perceived by peers for the three schemes under a high fixed cache hit rate and a varying hit rate, respectively. It is evident that in both cases naive HPTP outperforms the normal P2P while the CATC HPTP performs the best with a large margin. Compare the two sub-figures, we see immediately the advantages of CATC over naive HPTP under a more realistic cache hit rate model. This is obviously due to the explicit effort in CATC to better leverage caches. Similarly, we can see the performance boost of HPTP schemes with regard to the average bandwidth metric

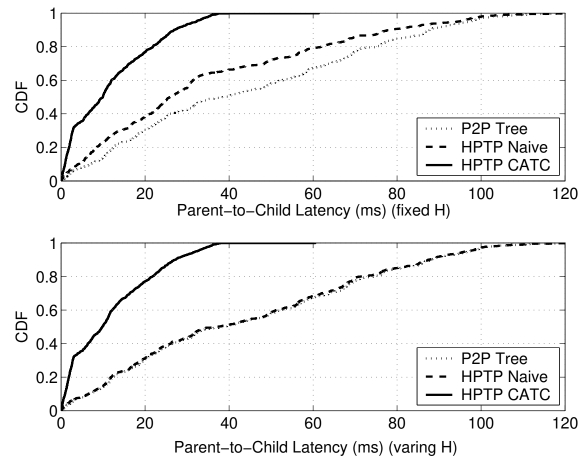


**Figure 2: CDF of average parent-to-child latency: (a) fixed cache hit rate, (b) varying cache hit rate.**

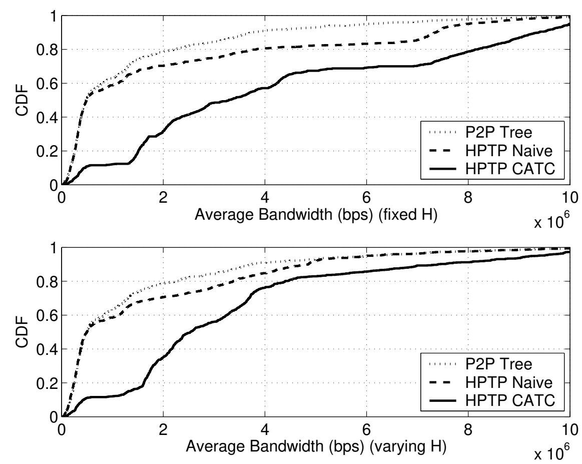


**Figure 3: CDF of average bandwidth of all peers: (a) fixed cache hit rate, (b) varying cache hit rate.**

in Figure 3, under different cache hit rate models.

We stated before that HPTP can lead to significant reduction in the transit traffic as well as the Internet backbone traffic because of the caches' help. This is clearly confirmed in Figure 4. The reduction of backbone traffic also implies that HPTP achieves, implicitly but automatically, locality awareness and reaches a finer level than an application level locality-aware protocol can achieve. Note that the way how normal P2P tree is built in our experiment actually represents excellent locality awareness because closer peers are assigned shorter latency.

We also conduct another set of experiments to measure the path length (in hops) from a peer to the source. The results reveal that there are indeed some peers travel more hops (i.e., detouring) to reach the source in order to seek for caches. But when transferring, on average, they enjoy a shorter route due to cache hitting. We omit the figure for sake of space.

Finally, as aforementioned, the performance of HPTP depends on both the hit rate and deployment of those web caches. So we perform another two sets of experiments to study how the number of caches ($K$) and their hit rate ($H$) influence the HPTP performance. To obtain more insight on

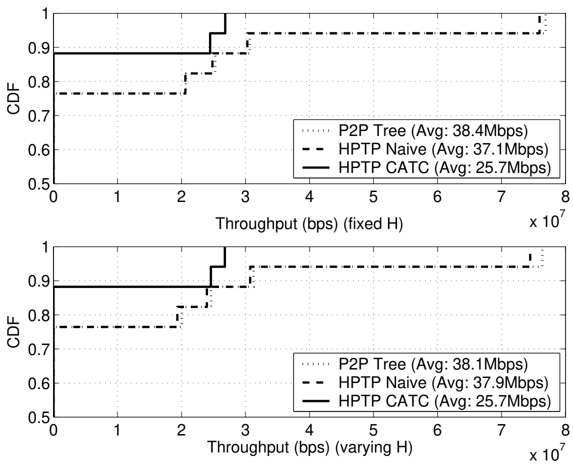


**Figure 4: CDF of traffic on backbone links: (a) fixed cache hit rate, (b) varying cache hit rate.**

their respective impact, we fix one of them while adjusting the other. Due to the space limit, we are not able to include the figures there. The results reveals that: 1) the performance of HPTP goes better almost linearly with the growth of the caches' byte hit rate; 2) the number detected-usable web caches also has almost linear impact on the HPTP performance; 3) the placement of caches is very important where the number of caches is small.

## 5. CONCLUSION AND FUTURE WORKS

In this paper we have proposed an HTTP-based Peer-to-Peer (HPTP) framework to leverage the already deployed ISPs' caching proxies to relieve the tension between P2P and ISPs. We presented the basic concept of HPTP framework, designed necessary tools like the HTTPifying tool and the cache detection and usability test tool, $\mathcal{H}$Ping. We also performed case study by building a cache-aware tree to demonstrate the potential gains of HPTP. Experimental and simulation results confirmed that the tools are effective and HPTP can indeed lead to significant performance improvement for peers and traffic reduction on transit links and the Internet backbone.

We believe that the proposed HPTP framework will benefit both ISPs and end users (P2P as well as web users), as HPTP proactively recruits many "giant-peers" - powerful, stable, dedicated and strategically deployed caching proxies - to help deliver P2P traffic. HPTP can produce substantial reduction of the transit/peering traffic across ISPs and the traffic on the Internet backbone, at no extra adoption cost as compared with P2P caching solutions or other alternatives. It is therefore more appealing to ISPs. While P2P users will benefit immediately and directly from HPTP, normal web users will indirectly benefit from HPTP as well because the reduced P2P traffic on the backbone will make downloading of uncacheable dynamics web contents faster, given the fact that more and more web pages are using uncacheable dynamic content heavily. In addition, as the hit rate increases only logarithmically with the cache size [4], caching proxies can be more efficiently utilized by caching P2P traffic, while its negative impact on the web cache can be mitigated by using intelligent P2P caching schemes such as the one developed in [6].

As a final remark, we would like to point out that the current cache index schemes prevent us from most efficient utilization of caching proxies because the same content from different peers are indexed differently which would cause huge waste of cache storage in a P2P setting. We are currently investigating some practical work-arounds. Moreover, as the Internet evolves towards a data-oriented architecture where files can be referred to with location-independent flat identifiers [10] or *Uniform Resource Names* (URNs, RFC 2141), the efficacy of HPTP would be maximized.

## Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0546216. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## REFERENCES

[1] CacheLogic, "http://www.cachelogic.com."

[2] S. Saroiu, et al, "An analysis of internet content delivery systems," in *Proc. of OSDI'02*, 2002.

[3] S. Sen and J. Wang, "Analyzing peer-to-peer traffic across large networks," *IEEE/ACM Trans. on Networking*, vol. 12, no. 4, pp. 219–232, 2004.

[4] N. Leibowitz, et al, "Are file swapping networks cacheable? characterizing p2p traffic." in *Proc. of WCW'02*, Boulder, Colorado, Aug. 2002.

[5] K. P. Gummadi, et al, "Measurement, modeling, and analysis of a peer-to-peer file-sharing workload," in *Proc. of SOSP'03*, Oct. 2003.

[6] O. Saleh and M. Hefeeda, "Modeling and caching of peer-to-peer traffic," in *Proc. of ICNP'06*, Santa Barbara, CA, Nov. 2006.

[7] S. Patro and Y. C. Hu, "Transparent query caching in peer-to-peer overlay networks," in *Proc. of IPDPS'03*, Washington DC, 2003.

[8] D. Wessels, *Web Caching*. O'Reilly & Associates, Inc., 2001.

[9] K. Calvert, M. Doar, and E. W. Zegura, "Modeling internet topology," *IEEE Comm. Magazine*, 1997.

[10] M. Caesar, et al, "Rofl: Routing on flat labels," in *Proc. of the ACM SIGCOMM*, Pisa, Italy, 2006.

[11] V. S. Pai, et al, "The dark side of the web: an open proxy's view," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 1, pp. 57–62, 2004.

[12] A. Wolman, et al, "On the scale and performance of cooperative web proxy caching," in *Proc. of SOSP'99*, 1999, pp. 16–31.