InHome: A Local Area Peer-to-peer Caching System

Mihir Kedia, Raluca Ada Popa, Irene Zhang

November 25, 2008

Abstract

Demand for wide-area bandwidth is explosively increasing as the use of bandwidth-hungry applications like Bittorrent rises and websites compete to offer a richer browsing experience. ISPs are finding it increasingly difficult to keep up with demand, making wide-area bandwidth an expensive commodity. On the other hand, the capacity of local-area bandwidth has been growing rapidly; capacity that, for the most part, remains unused. This paper presents InHome, a system that trades cheap local-area bandwidth for expensive wide-area bandwidth. InHome is motivated by the observation that many people inside an organization access common data, allowing one to trade wide-area bandwidth for local area bandwidth by fetching commonly accessed data from local peers.

1 Introduction

Wide-area bandwidth is becoming an increasingly scarce and expensive resource. Pages are populated more and more with rich media, and bandwidth-hungry applications like Youtube are growing at an explosive rate. As a result, ISPs are being forced to install increasingly expensive infrastructure, passing the cost onto business and academic customers.

Local-area bandwidth, on the other hand, is becoming increasingly plentiful and cheap. Most computers nowadays have 100-Mbit or Gigabit Ethernet cards, despite the fact that network transfer rates rarely approach such levels. Moreover, laying local cable is cheap; most offices and campuses tend to have strong interconnectivity. Since the majority of network traffic is to external servers, local area bandwidth tends to remain mostly unused.

Given these conditions, it would be nice if we could find a way to trade wide-area bandwidth for local-area bandwidth. The key to doing this is the observation that most internet traffic is redundant – 25 to 40 percent of all browser requests are for web objects already stored in the cache of another local client. If web browsers were smart enough to retrieve the data from a local computer rather than the origin server, global bandwidth usage would be significantly decreased (not to mention the cost savings for organizations paying hefty ISP bills.)

InHome is a peer-to-peer system that operates like a distributed cache. Internet applications with appropriately-designed plugins can query InHome for data that might be on local computers, falling back to the origin server if the request times out or cannot be fulfilled.

The remainder of the paper discusses the design of the InHome system and offers a new search algorithm for quickly locating cached data. Section 2 provides a more detailed description of the problem. Section 3 gives an overview of the design of the InHome system. In Section 4, we elaborate the searching algorithms that allow InHome to guarantee low latency. Section 5 contains evaluations of the performance and resource costs of InHome's search algorithms. In Section 6, we take a more detailed look at how InHome could be used as a distributed browser cache. Section 7 covers related work in the literature. Finally, Section 8 concludes.

2 Problem Description

The main goal of InHome is to reduce external bandwidth usage by satisfying web requests from the internal network of an organization.

The system setting consists of many computers or clients (ranging from 1000 to $10,000$ or even $100,000$) inside an organization. These clients are connected with each other by fast network links. They are also connected to outside entities by slower or more expensive links. Thus, getting data from a client within the same organization can be orders of magnitude higher than getting the same data from an external, remote server. Each client performs web requests for objects such as web pages, music, video, or general files. Some of these objects have already been requested by other clients inside the network. InHome would like to fetch these objects from the local clients rather than from the external origin server.

An ideal solution should have the following properties:

1. The clients should not see a significant increase in latency.
2. The clients should not store data they are not interested in.
3. The system should not require new hardware and maintenance.
4. The system should be customizable for organization sizes.

The first property of InHome is the most important; if InHome significantly increases the latency of fetching web content, clients may not be willing to use the system. As we will discuss in Section 4, clients can tradeoff latency for bandwidth reduction. If they are willing to wait more, there is a higher chance that the desired data is retrived from the InHome local network which results in higher reductions in external bandwidth usage.

The second property states that InHome should not require clients to store data they are not interested in. The reason is that the content from another client may be compromising (such as porn files) or insecure (such as viruses). Furthermore, unnecessary transfer of objects should be avoided so the local area network is not overwhelmed by InHome traffic.

The third property is aimed at increasing adoption of the system. New hardware requires additional expenditure, overhead of maintenance, and time overhead of installing the needed hardware. A typical solution to reduce external bandwidth traditionally used by organizations is to install a centralized proxy for caching web data. Clients in the organization first attempt to get their desired web content from the proxy before falling back to the origin server. The problem with this approach is the overhead of maintenance, the possibility of centralized failure, and poor scalability. In contrast, InHome should be a distributed peer-to-peer system that does not require additional hardware.

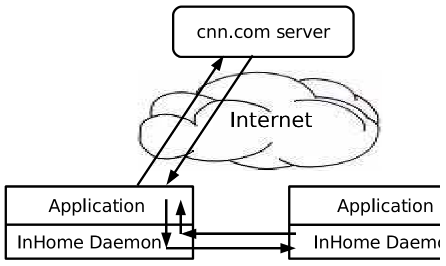

Figure 1: InHome system overview. The application running on top of InHome wants to fetch "cnn.com". It first requests the data via the InHome system. The request goes through the InHome daemon to other InHome daemons in the network. These return the data or a message saying they do not have the data. Alternatively, the application can directly query the "cnn.com" server.

The last property states that the system should be flexible enough to be adapted to different settings. Organization sizes vary from very small, such as start-up companies with about $100 - 1000$ computers, to very large, with $10,000 - 100,000$ computers such as a university campus or large organization (e.g. Microsoft).

3 System Design

InHome is implemented as a peer-to-peer network that operates like a distributed cache.

3.1 System Architecture

Figure 1 illustrates the overview of the system. Web applications or plugins for existing applications run on top of the InHome system. The data is stored in the form of objects identified by an ID-s. The content of the objects consists of raw bytes. Thus, InHome is application independent because we do not make any assumption about the content of the data; it can support web pages, video, music, or other files.

Clients run a background InHome daemon whose tasks are:

- Maintain the metadata and routing information in the system synchronized with the other InHome daemons. This information is used to locate objects in the system.

- Perform web object lookups as requested by the application which runs on top of it.

- Service content to other InHome daemons that request it.

3.2 Interface

InHome exports a simple and general interface which is summarized in Table 1.

Upon startup, the InHome client needs to join the InHome network. The *join* function takes the client's network information (such as IP address) and the network information of another peer that is known to be in the network. The easiest way to find another peer is for the organization to set up a centralized membership tracker. This solution is not ideal because it violates the third property presented in Section 2. All other solutions are vastly more complicated, so the simplicity of the tracker may outweigh any other considerations. The membership tracker will not be heavily loaded so any machine will do, even a well known client that does not leave the network can do the job.

The *get* and *put* functions are responsible for placing and reading data from the system. The *name* identifies the objects and it can be a URL, a file name, a video name, etc. The data is stored as a sequence of bytes and it is not interpreted by InHome clients. Using a general interface, InHome can support a range of applications from web pages to videos or other general files.

If the application is interested in security, the name can be self-certifying. That is, if the application that runs on top of InHome wants to verify that the content it receives via InHome from another client is correct, it can use self-certifying names such as a hash of the content. The check for correctness is then done at the application level. If the data is found to be incorrect, it is left to the application to react to this. For example, it can choose to fall back to the origin server for that data.

get is a blocking function and it does not return until it finds the data, confirms the data is not in the cache, or times out. The *get* function uses consistent hashing to locate the data, which we describe in Section 4.

4 Search Algorithms

This section discusses the search algorithms our system uses to quickly locate cached data. In order to adapt the system to different membership sizes, we explored two different search algorithms. Both algorithms use consistent hashing, a commonly used algorithm for locating data in distributed systems [6]. For our project, we looked at consistent hashing with full membership and a variant of consistent hashing with partial membership based on Chord.

4.1 Consistent Hashing with Full Membership

Consistent hashing is a distributed algorithm that maps keys to data by assigning responsibility for some of the mappings to each node [5]. Each node and key in the system is hashed to a common ID space. Nodes are arranged in a ring ordered by ID. Each node is responsible for all of the keys that fall between its ID and the ID of its predecessor.

For the InHome cache, the node that is responsible for a key stores the list of nodes that have the cached data associated with that key. To insert a key, the node notifies the responsible node. The responsible node will add the node to the list of nodes that have the data for that key. To look up a key, the node just finds the successor of the key. Since the nodes have global knowledge, finding the node responsible for a key takes constant time.

To maintain full membership information, a node notifies all of the other nodes in the system when it joins and leaves. If a node fails, the next lookup for a key that the failed node is responsible for will time out and the node that is performing the lookup will notify the other nodes of the failed node.

Although consistant hashing with full membership guarantees fast lookup, the disadvantage is that the overhead of maintaining full membership information grows linearly with the number of nodes in the system. Thus, as the number of nodes in the system grows, the overhead of maintaining membership may become unacceptably high.

The trade-off between search performance and bandwidth usage must be carefully considered. The latency of searches in our system impacts

Table 1: System Interface

Function	Description
`join(info, peer)`	Joins the local client the network information in $info$ by contacting a *peer* that is already in the network.
`put(name, data)`	Creates a binding between a *name* of an object and the data in the object.
`data = get(name, timeout)`	Returns the data object corresponding to *name* or null if not found.
`remove(name)`	Removes the binding of *name*.

the number of cache hits because if the lookup takes too long, it will time out and the application will fall back to the origin server. The hit rate determines how much bandwidth is saved in the end, so using a higher latency search protocol will hurt the performance of the system. But as the membership size grows, the overhead used by consistent hashing to maintain full membership information may overwhelm the local area network.

For situations where the membership size is guaranteed to remain small, consistent hashing with full membership will be the best choice. But for a system that scales with a large number of nodes, we explore consistent hashing with partial membership information using data-oriented Chord.

4.2 Data-oriented Chord with Partial Membership

The goal of partial-knowledge consistent hashing is to reduce the number of other nodes a node must know about to perform lookups. At a minimum, each node must know about at least one other node. If each node just maintained a correct successor pointer, a lookup could traverse the entire ring until it finds the successor for the key.

Data-oriented Chord is based on the Chord lookup protocol [7]. Chord is a variation of partial-knowledge consistent hashing where each node knows about its successor and $\log n$ other nodes. The other $\log n$ nodes, called fingers, are used to optimize lookups. Fingers are placed at powers of two; the i-th finger is a node that is at least 2^i away in the ID space. With $\log n$ fingers, Chord can perform a lookup in $O(\log n)$ time because the distance to the successor of the key can be at least halved by each hop.

Data-oriented Chord is a variation on Chort where clients store data by inserting *virtual nodes* for each new mapping. Each client (hereafter referred to as physical nodes) stores one virtual node for each piece of data it contains, and all virtual nodes form a giant Chord ring. We'll talk about how we construct the virtual node IDs in a bit.

To join the system, physical nodes only need to contact a well-known node. A physical node does not join the Chord ring until the new node has a piece of data to insert. Until the new node inserts its first piece of data, the node can use the well-known node to run lookups. Since our system is used for caching, it is expected that the node re-inserts every piece of data it receives, so the new node should not need to use the well-known node for lookups for very long[1].

Lookups function similarily to Chord except that each physical node considers the fingers of *all* of its virtual nodes. This optimization reduces the runtime of the algorithm from $\log m$ where m is the number of *virtual* nodes down to $\log n$ where n is the number of *physical* nodes in the system. If the successor ID matches the key, the object exists in the system. If the successor is not the key, the object is not in the cache.

To insert a piece of data, the physical node has the virtual node representing that piece of data "join" the Chord ring using the Chord join protocol. The physical node will find the virtual node's successor using lookup and fetch an initial finger table for the virtual node from the successor. Figure 2 shows a Chord ring with virtual nodes and successor pointers.

Since each virtual node represents a piece of data, a problem arises when two physical nodes have one piece of data. This problem occurs fre-

[1]Unless the node is malicious, in which case, the well known node should cut off the malicious node after some number of lookups

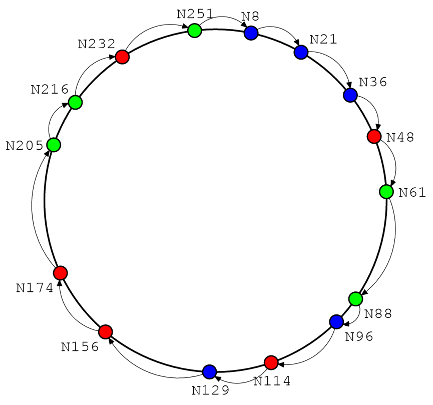

Figure 2: **Data-oriented Chord ring.** Each node in the Chord ring represents a virtual node and the virtual nodes of the same color reside on the same physical node. Even with only successor pointers, each physical node already has quite a bit of information about the other nodes. For example, the red physical node knows the blue physical node had an object associated with key 129 and the green physical node has objects associated with keys 61 and 251.

quently because when a node fetches a piece of data from another node in the system, the node should insert its own copy of the data into the ring. Furthermore, we want to load balance all requests for a piece of data amongst all nodes that contain that data.

We solve this problem as follows. To insert a piece of data, the node first tries to insert a virtual node with an ID where the lower order bits are all set to 1. If this insert fails (i.e. another node has already inserted the data, the node picks a random number for the lower bits of the ID and re-inserts until it succeeds. Any time a node requests the data, it chooses a random number for the lower order bits (Chord lookup will return the first node with a greater ID number than the request.)

Since the first node to insert the data selected the all-1 vector for its lower order bits, any data request is guaranteed to return the correct data[2] All requests choose a random set of lower-order bits, so on average all nodes containing the data will satisfy an equal number of requests.

To leave the system, the node can just exit without contacting any other nodes. If the node has a lot of virtual nodes, the node might end up contacting every other physical node in the system, defeating the purpose of minimizing the bandwidth overhead of membership maintenance.

Each node runs stabilize according to the Chord protocol to keep successor pointers and finger tables up to date. To reduce the amount of bandwidth spent running the stabilize protocol, each physical node cycles through its virtual nodes and only runs stabilize for one virtual node at a time. Thus, the amount outgoing bandwidth used for stabilize at each physical node be the same as in Chord, but the incoming bandwidth used at each node will be proportional to how much data the node has with respect to the other nodes in the system.

5 Evaluation

This section presents performance evaluations for the two search algorithms used in our system.

5.1 Consistent Hashing Comparison

In order to compare consistent hashing with full membership against data-oriented Chord with partial membership, we implemented a simulation of each algorithm. Each simulation can show nodes joining, inserting keys, searching for keys and failing or leaving. The simulations record a number of performance measurements: the number of messages sent and received by each node, the amount of routing data stored by each node and the number of hops needed for each search.

For the lookup performance and bandwidth measurements shown in Figures 3 and 4, the simulation created n nodes and picked random nodes to insert $n \cdot m$ objects, where m is the average number of objects per node. Then, each simulation had different randomly chosen nodes run a total of $n \cdot s$ key lookups, where s is the average number of searches each node runs. The simulations used varying numbers of nodes with $m = 10$ and $s = 10$. No nodes leave or fail throughout the experiment.

Figure 3 shows the average number of hops per search for different membership sizes. As expected, the consistent hashing algorithm has lower latency lookups than data-oriented Chord

[2]In the corner case where that node unexpectedly drops and no other node has tried to insert the data, our lookup protocol queries the predecessor pointer of the node to find a node containing the data.

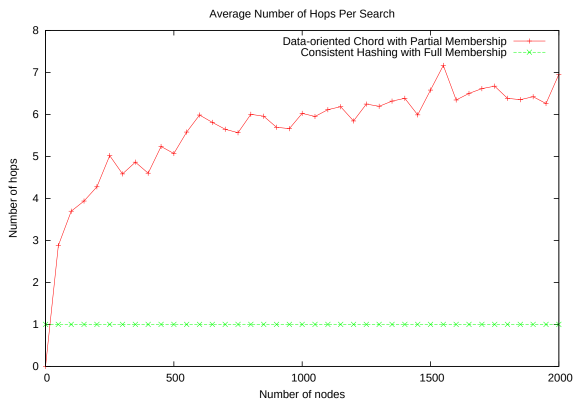

Figure 3: A comparison of the average number of hops a search takes as membership size grows for consistent hashing with full membership and data-oriented Chord.

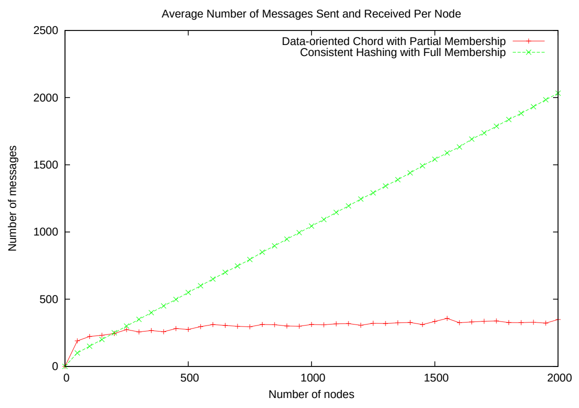

Figure 4: A comparison of the average number of messages sent per node as the membership size grows for consistent hashing with full membership and data-oriented Chord.

because each node has full membership information. When each node has global knowledge, the node performing the lookup can go directly to the responsible node in one hop. With partial membership information like in data-oriented Chord, lookups have to be routed to different nodes that have more information in order to finally find the responsible node.

Figure 4 shows the bandwidth consumed by consistent hashing with full membership and data-oriented Chord for different membership sizes. The advantage of data-oriented Chord can be seen in the bandwidth usage graph; data-oriented Chord uses significantly less bandwidth as the number of nodes increases than regular consistent hashing. Consistent hashing with full membership has a linear increase in traffic because each node that joins has to notify every other node in the system. Failing or leaving nodes are not included in the measurements.

Another advantage of data-oriented Chord as compared to consistent hashing with full membership is that data-oriented Chord has fate sharing. Lookups only fail when they are for data stored on a failed node because there is no additional indirection. In contrast, in the full membership consistent hashing scenario, upon a node failure, the files that become unreachable are not only the ones that it stores locally, but also the files for which it keeps the metadata. This means that even if a client storing a particular file is live, that file may still not be reachable because the consistent hashing client who stored the binding for the file failed.

Figure 5 shows the hit rate for the files that

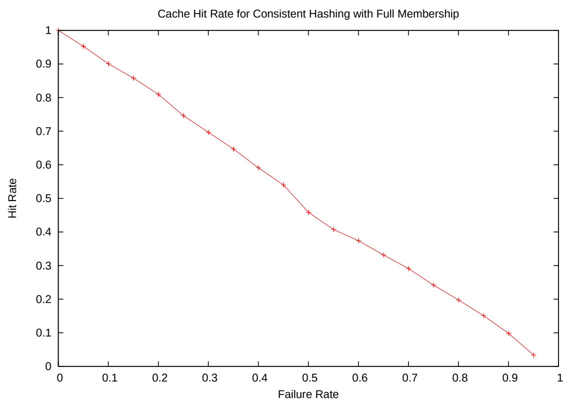

Figure 5: Hit rate depending on failure rate.

were inserted in the system as the failure rate of the clients increases. The data was obtained by running simulations of our consistent hashing with full membership protocol on 1,000 clients and 10,000 files with random id-s. For each failure rate, we lookup all the files we inserted. We consider a lookup for a file to be successful if the metadata node storing the binding is alive. We measure the hit rate as the number of files with successful lookups divided by the total number of files in the system. We can see that the hit rate decreases linearly in the failure rate. This is obviously an undesired behavior. One can mitigate the problem by using replication of the bindings in the system, although the problem would not fully disappear.

5.2 Data-oriented Chord Evaluation

Using the simulation of data-oriented Chord, we measured the lookup performance and band-

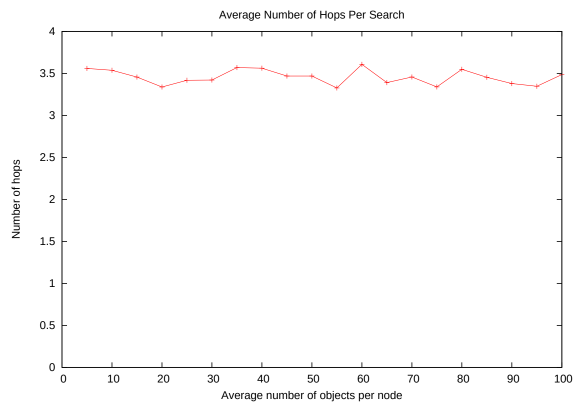

Figure 6: Lookup latency

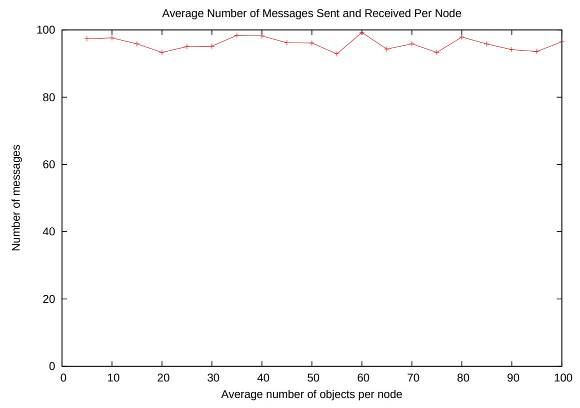

Figure 7: Bandwidth

width usage of the protocol as the number of objects increases. The simulation used 100 physical nodes with varying amounts of data. The simulation inserted $100 \cdot n$ objects in total on random physical nodes, where n is the average number of objects per node. The simulation then performed 100 searches for random keys.

Figure 6 shows the lookup performance of data-oriented Chord as the number of objects in the system increases. As expected, the latency stays about constant because search latency depends on the number of physical nodes, which is constant in this test.

Figure 7 shows the bandwidth usage of data-oriented Chord. The bandwidth measurements include 10 cycles of stabilize and 100 random key searches. A stabilize cycle runs the Chord stabilize algorithm on a virtual node on each physical node. The bandwidth measurements do not include the amount of bandwidth consumed inserting the objects.

6 Case Study

We studied a distributed browser cache as a concrete example of an application using InHome. Rather than always going to the origin server to retrieve web content, we first see whether any local peers have already retrived and stored it in their cache.

This application is implemented as a Mozilla Firefox plugin. The plugin intercepts HTTP requests and dispatches a query to InHome for each. If a response is not received within 200 milliseconds (or if the page is not available), the plugin retrieves the data from the origin server. Pages are identified via their URL, so two identical web objects in different locations would be treated as separate objects. 200 millseconds was chosen based on empirical measurements of local area latency balanced with a desire to avoid worsening the browser experience.

In addition, every time a new object is added to the cache, the plugin inserts the new object into the InHome system, allowing the computer to serve the new object to other peers. The plugin also removes pages with expired TTLs, ensuring that clients always fetch the latest version of dynamic pages.

6.1 Security and Caching Policy

Security is an exceptionally important and challenging problem for a web cache. Critical business is executed over the Internet, and retrieving pages from local clients opens up many new avenues of fraud and phishing attacks. Guaranteeing security is exceptionally difficult, however, because content is not static or self-certifying.

If we were allowed to make radical changes to the status quo, we could have servers cryptographically sign their content. At the moment, though, this is impossible. Instead, we make sure that no compromising pages are cached by excluding pages with SSL encryption or pages with password fields.

Our final caching policy can be written as follows:

- If the URL is SSL-encrypted (e.g. HTTPS protocol), fall back to the origin server (for obvious reasons.)
- Attempt to retrieve data from InHome. Timeout after 200 milliseconds.

- Fall back to the origin server if the query was unsucessful.

- If the data contains a password field, fall back to the origin server (this prevents malicious peers from attempting to steal passwords.)

6.2 Bandwidth Savings

The efficacy of this system is dependant on how similar user browsing habits are. The internet is vast – if different users usually visit different sites, this scheme is useless. Fortunately, this is not the case. We analyzed multiple sets of users traces and constructed a statistical model to analyze multiuser web browsing behavior.

6.2.1 University of California, Berkeley

The first trace we studied was a four-hour trace from November, 1996 from U. C. Berkeley [2]. A cross-section of student, staff, and faculty agreed to install internet tracking software on their home computers, and the results were anonymized and made public. Since these traces were from the client computers, it is possible to figure out which requests were served by the local browser cache and exclude them from the measurements.

Analyzing the trace, 24.3% of web requests are for web objects previously requested by a different client. Factoring web object sizes in, applying InHome could save 27.6% of total bandwidth.

6.2.2 IRCache

The second set of traces is from a public caching server that provides its data for research purposes called IRCache [1]. Unlike the Berkeley traces or the typical use case for this system, IRCache data is gathered from a set of users scattered throughout an entire city. However, this trace can still serve as a good lower bound for hit rate and bandwidth savings – browsing habits could only become more similar if confined to a smaller geographical area.

Since this trace is from an entire day of requests, its more representative of the cache duration that our solution provides. Although the records don't clearly identify differing clients, we know that requests served by the local browser cache will not appear in the trace output.

Analyzing the trace, 37.6% of web requests are for web objects previously requested by a different client. Factoring web object sizes in, applying InHome could save 41.5% of total bandwidth.

6.2.3 Statistical Model

Prior work has shows that the Zipf Model accurately represents user browsing behavior [3]. Our model uses a set of 1000 independent Zipf distributions to model clients, with each distribution using $N = 50000$. In addition, each client randomizes the order of the top 100 sites to vary user behavior. Since these top 100 sites represent over 50% of all requests, we've determined that this change produces enough variance to accurately measure the behavior of multiple users.

Running our statistical tests, 43.2% of web requests are for web objects previously requested by a different client. Factoring web object sizes in, applying InHome could save 45.7% of total bandwidth.

7 Related Work

So far, there haven't been any distributed cache systems like InHome. Companies interested in saving bandwidth tend to use a centralized caching proxy server – an effective, but costly solution. In addition a centralized caching proxy can only work in situations where there is a central point of entry for data into the system. As more and more organizations use more than one ISP, a centralized solution become more impractical.

The closest prior work to InHome are Content Distribution Networks like Codeen[8]. Although these too are distributed caching systems, CDNs are focused on relieving server pressure – a specific set of content providers replicate their data on these networks to reduce the number of data requests they process. In contrast, InHome is focused on reducing the bandwidth of a specific set of users, storing their most commonly accessed data.

There has been similar work in reducing external Bittorrent traffic. Since Bittorrent forms a significant chunk of cross-ISP bandwidth, systems focusing on reducing external Bittorrent traffic can make a significant impact on global bandwidth usage. Ono[4] is a project from Northwestern that piggybacks off existing CDNs to locate local peers for Bittorrent downloads. Although there are no concrete measurements determining how much bandwidth is saved, Ono transfers draw at least 33% peers from the local network on average. In addition, a study from Stanford on local peer selection has shown that such schemes can reduce the number of redundant copies of data downloaded by an ISP from 50 to 4.

Our system is more general than Ono – with the appropriate plugin, a Bittorrent client could discover whether a copy of the data was located on the local network and download it if possible. Bittorrent was designed to download from a large number of low-throughput connections; for local area networks, its easier to directly download the file from a single peer.

8 Conclusion

We've presented InHome, a peer-to-peer system that can significantly reduce external bandwidth usage by fetching data from local peers when available. Since many web requests are for data previously requested by another peer on the network, InHome eliminates these duplicate requests and ensures optimal use of outgoing network links.

InHome uses a novel scheme to guarantee low latency: Data-Oriented Chord. Here, pieces of data are stored as virtual nodes in the Chord ring, eliminating the layer of metadata present in most DHTs. Using Data-Oriented Chord balances low latency with minimal network overhead, preventing InHome's operation from having a significant impact on typical network activity.

InHome is well-timed for widespread adoption. Current hotly debated topics like Network Neutrality and Bittorrent Filtering are instigated by a lack of bandwidth capacity, and InHome provides an easy-to-deploy technical solution that realizes significant bandwidth savings.

References

[1] Ircache. http://www.ircache.net/.

[2] Uc berkeley home ip web traces - 18 days. http://ita.ee.lbl.gov/html/contrib/UCB.home-IP-HTTP.html.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM*, 1999.

[4] D. R. Choffnes and F. E. Bustamante. Taming the torrent: A practical approach to reducing cross-isp traffic in p2p systems. In *To appear in Proc. of ACM SIGCOMM 2008*, 2008.

[5] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[6] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. Acms: the akamai configuration management system. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 245–258, Berkeley, CA, USA, 2005. USENIX Association.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.

[8] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and security in the codeen content distribution network. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Boston, MA, 2004.