Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computer Networks 50 (2006) 859–876
Computer Networks
www.elsevier.com/locate/comnet

Consistency maintenance in dynamic peer-to-peer overlay networks ☆

Xiaotao Liu [a,*], Jiang Lan [b], Prashant Shenoy [a], Krithi Ramaritham [c]

[a] Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA
[b] Verizon Corp, GTE Labs, Verizon Data Services, 40 Sylvan Road, Waltham, MA 02451, USA
[c] Department of Computer Science and Engineering, India Institute of Technology, Bombay, Powai, Mumbai 400 076, India

Available online 30 August 2005

Abstract

In this paper, we present techniques to maintain temporal consistency of replicated objects in data-centric peer-to-peer overlay applications. We consider both structured and unstructured overlay networks, represented by Chord and Gnutella, respectively, and present techniques for maintaining consistency of replicated data objects in the presence of dynamic joins and leaves. We present extensions to the Chord and Gnutella protocol to incorporate our consistency techniques and implement our extensions to Gnutella into a Gtk-Gnutella prototype. An experimental evaluation of our techniques shows that: (i) a push-based approach achieves near-perfect fidelity in a stable overlay network, (ii) a hybrid approach based on push and pull achieves high fidelity in highly dynamic overlay networks and (iii) the run-time overheads of our techniques are small, making them a practical choice for overlay networks.

Keywords: P2P; Consistency; Overlay network

1. Introduction

The design of distributed applications using overlay networks has received increased research attention in recent years. Overlay networks are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality; such networks offer the promise of harnessing the vast numbers of Internet hosts to implement large distributed applications. Many data-centric applications such as global-scale storage [1], peer-to-peer remote backup [2], decentralized file sharing [3,4], distribution of large datasets, peer-to-peer caching, distributed hash tables [5], distributed object location [6,7], and multicast distribution systems [8,9] have been designed using overlay networks.

☆ This research was supported by NSF grants CCR-9984030, CCR-0098060, CCR-0219520 and EIA-0080119.
* Corresponding author.
E-mail address: xiaotaol@cs.umass.edu (X. Liu).

1389-1286/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2005.07.010

A peer-to-peer overlay network consists of a collection of nodes that are interconnected via logical links; each logical link spans multiple physical links in the underlying physical network. Each node (also referred to as a *peer*) in the overlay can communicate with its neighbors via a logical link. Communication with other nodes involves traversing multiple logical links, and nodes in the overlay typically participate in the routing of messages. Typically, overlay networks are dynamic, with nodes that may join or leave at any time. The network must reconfigure itself to handle these topology changes.

Depending on the actual topology, peer-to-peer overlay networks can be categorized into two types: structured and unstructured. Structured overlay networks, including Chord [10], CAN [11], Pastry [12], Tapestry [13], and Kademlia [14], assign each peer a uniform random *peerID* from a large identified space, and assign each application-specific object a unique identifier called *key*, selected from the same id space. Each key is dynamically mapped by the overlay to a unique live peer, called the key's *root*. Structured overlay networks also employ a key-based routing algorithm. The routing algorithm supports deterministic routing of messages to a live peer with responsibility for the destination key. Typically, each peer maintains a routing table consisting of the peerIDs and IP addresses of the peers to which the local peer maintains overlay links; messages targeted at some key f are forwarded across overlay links to peers whose peerIDs are progressively closer to the key in the identifier space. These systems guarantee that messages are delivered in the absence of failures.

In contrast, unstructured peer-to-peer overlay networks do not employ key-based routing and instead use a centralized index or flooding to locate the destination peer. Unstructured overlays such as Napster [15], Gnutella [3], and Freenet [4] are primarily used by file sharing applications. These systems can be centralized or distributed: Napster uses a centralized model, while Gnutella and Freenet use distributed models. Unlike structured networks, unstructured network such as Gnutella or Freenet do not guarantee that files can be located in the absence of failures. The centralized model suffers from two limitations. First, the indexing server can become a bottleneck and a central point of failure. Second, the indexing server can return stale information if a file is deleted at a peer (since the index information is only refreshed periodically). Decentralized systems attempt to overcome these drawbacks. In decentralized systems, queries propagate through the network via flooding; the reach of a query message is limited by a time-to-live (TTL) value, which is decremented at each hop. The responses are transmitted via the reverse path.

Since an overlay network may be dynamic with frequent joins and leaves, replication is commonly used by data-centric applications to ensure high availability and also to improve performance. With data replication comes the problem of maintaining consistency of various replicas—if a data object is modified by the application, all replicas must be updated to ensure data consistency and correctness. For example, in global-scale data storage application [1], when a data block is modified due to a write, all the replicas of modified data blocks must be updated to ensure consistency. In a peer-to-peer file sharing application used in a collaborative setting, any update to a shared file will require replicas to be either updated or invalidated. In a distributed hash table [5], when the items of hash table are changed, all the replicas of the updated items must be updated to maintain consistency. In distributed object location systems [6,7], once the object mappings are changed, all replicas of the updated object mapping locations must also be updated to reflect the new mapping. In a recent work, we built a repository [16] of multi-gigabyte traces using BitTorrent [17]; our system enables traces files to be updated with new versions and uses consistency mechanisms to eliminate stale replicas of these files from the system.

Data consistency techniques have been widely studied in distributed systems, most recently in the context of Web proxy caches [18–22]. However, most of these techniques are not directly applicable to peer-to-peer overlay networks. Such networks can be highly dynamic—nodes may dynamically join and leave the network at any time; studies have shown the mean session duration of a nodes is only a few hours for file sharing applications [23]. Since nodes containing replicated content may not be part of the network

when an object is modified, maintaining consistency is more challenging in these environments. In contrast, much of the work on Web proxy caching has assumed that proxies are mostly available and failures are rare. Bayou is an example of a distributed system that assumes weakly connected replicas and implements a weak form of consistency called eventual consistency to deal with disconnections and reconnections [24,25]. However, many overlay network applications may have stronger consistency requirements than a eventual consistency mechanism can provide. Due to these differences, novel data consistency techniques specifically designed to handle the needs of applications built using overlays are required.

1.1. Research contributions

In this paper, motivated by the need to support data coherency for replicated objects, we propose three consistency maintenance techniques for data-centric peer-to-peer overlay applications. Our first two techniques are based on the notions of push and pull, respectively, and have complementary trade-offs. A *push*-based approach can provide near-perfect fidelity but has high communication overheads and is suitable for overlay networks where nodes are mostly static. In contrast, a *pull*-based approach has lower communication overheads and is better suited for dynamic networks but provides weaker guarantees than push. Based on these observations, we propose a hybrid approach that combines the best features of push and pull and attempts to provide good fidelity in highly dynamic networks at a reasonable cost. We propose enhancements to a structured overlay network protocol such as Chord [10], and to an unstructured overlay network protocol such as Gnutella [3] to incorporate all three techniques. We also implement our hybrid technique into Gtk-Gnutella—a public source implementation of the Gnutella protocol.

We evaluate our techniques using a combination of simulations and prototype implementation. Our results show that while push is more suitable for stable overlay networks, our hybrid approach can provide good fidelity even in highly dynamic environments. Our measurements from the prototype implementation on a laboratory testbed and the PlanetLab testbed for a Gnutella file sharing application indicate that this fidelity can be provided at a reasonable run-time cost.

The remainder of this paper is structured as follows. We present our consistency maintenance techniques in Section 2. Enhancements to the Chord and Gnutella protocol and details of our prototype implementation are presented in Section 3. Section 4 presents our experimental results. Section 5 presents related work, and finally, Section 6 presents our conclusions.

2. Consistency techniques for data-centric overlay applications

In this section, we present techniques for maintaining consistency of replicated objects in a data-centric overlay application. Our techniques are designed to work in the context of both structured and unstructured peer-to-peer overlay networks. Each data object in our system is assumed to have a unique owner and a unique identifier (key). Typically, the owner of the object is the node where the object originated (i.e., the node where the object was created or first shared). Observe that, for structured overlays, the owner node may be different from the root node. In general, we assume that the owner is the node that stores the master copy of the object, while the root node stores the object location information (i.e., the ID of the owner node). In the event the object is replicated, we assume that the root node maintains a list of all nodes that hold replicas of the object. For unstructured overlays such as Gnutella, only the owner node is assumed to be known for each data object. The list of replicas is not known and must be discovered dynamically if needed.

Each object is also associated with a version number; the version number is incremented by the owner upon each update. Modifications to an object can only be made by its owner. While this assumption may seem overly restrictive, it is not—any user (node) may modify an object, but upon doing so, it is required to transmit these modifications to the owner to "commit" the changes. This ensures that the owner always has

the most up-to-date version of the object at all times. Observe that additional mechanisms such as distributed locking are required to prevent multiple peers from simultaneously updating an object and introducing write–write conflicts; these techniques can be implemented separately and are beyond the scope of this paper.

Next, we present three techniques for maintaining consistency of replicated data objects.

2.1. Push: owner-initiated consistency

Our first approach is based on the notion of push and puts the onus of consistency maintenance on the owner node. In this approach, the owner sends invalidation messages to all replicas upon each update to the object (alternatively, the new version of the object may be sent). Upon receiving an invalidation message, the node deletes the local replica if its version number is smaller than that specified in the invalidation message. If updates are sent instead of invalidates, the node simply replaces the old version of the object with the new version. While the choice to send update or invalidation is indeed dependent on the size of the update, and there exists cost–benefit trade-off between sending invalidations and pushing updates: (i) sending invalidations incurs smaller network overhead than pushing updates since only a small invalidation message compared to the update itself traverses the overlay network; (ii) sending invalidations may overwhelm the owner at a later time since all replicas may request update simultaneously or in a relatively short time interval; while pushing updates does not encounter this problem; (iii) pushing updates has smaller latency compared to sending invalidations as replicas are updated immediately once receiving updates while they have to download updates from the owner after receiving invalidations.

Due to their different routing mechanisms, Chord and Gnutella handle the invalidation messages in different ways:

1. In Chord, the owner first sends the invalidate message to the root node. The invalidate message propagates from the owner to the root via Chord's overlay routing mechanism. Since the root maintains a list of all replicas, the root then forwards the invalidate message to each node on the list. Again, messages propagate via Chord's overlay routing mechanism.
2. In Gnutella, the owner broadcasts the invalidate message to all nodes. The broadcast message propagates through the network via flooding, much like query or ping messages—the owner forwards the message to its neighbors, who then propagate the message to their neighbors and so on until the TTL limit is reached.

The main advantage of such a push-style approach is its simplicity. Further, the approach provides good consistency guarantees, so long as all nodes holding a replica are reachable. A limitation of push for unstructured networks is the high control message overhead due to flooding. Although a push-based approach is suitable for static overlays, the following limitations arise in dynamic networks:

1. Not all the peers in the network may receive the invalidation messages. There are two scenarios when this can happen: one is if the network is partitioned; the other, applicable only for unstructured networks, is if a peer is beyond the reach of the specified TTL limit.
2. Nodes in the overlay can join and leave the network dynamically. After a node leaves the network, it would not receive any further *invalidation* messages. Upon a subsequent rejoin, the node will contain a stale replica.

Based on the above observations, we conclude push alone is not sufficient for maintaining consistency in large overlay networks. Next, we present a pull-based approach for maintaining consistency.

2.2. Pull: peer-initiated consistency

Unlike a push approach where the owner is responsible for consistency maintenance, a pull approach puts the burden of consistency maintenance on individual nodes holding replicas. Implementing a pull-based consistency technique in an overlay network is no different from imple-

menting it in a client–server system such as the Web. In this approach, a node polls the owner to determine if a replica is stale. A node can employ different policies to determine when and how frequently to poll the owner to check for consistency. Regardless of the policy, the following information must be stored with each replica for consistency maintenance:

1. *Version number:* the version number indicates the version of the object currently stored at the peer node. The last modification time of the object also be used to determine this information instead of explicit version numbers.
2. *Owner ID:* This allows a peer to locate the owner of an object. In Chord, the peerId of the owner is stored, whereas in Gnutella, the IP address of the owner is stored.
3. *Consistency status:* The consistency status of an object can take one of three values: (i) *valid*, indicating the file is consistent with the version at the owner, (ii) *stale*, indicating that the file is older than the version at the owner, and (iii) *possibly stale*, indicating that the object could possibly be stale but the peer is unable to determine the actual status since the owner is unavailable (i.e., has left the overlay network).
4. *Time-to-refresh (TTR) value:* The TTR denotes the next time instant the node must poll the owner, and thus, determines the polling frequency.

Observe that a pull-based approach is more resilient to dynamic joins and leaves. Upon rejoining the network, a node can poll the owners of all locally cached objects to check if these objects were updated in the interim, and thereby ensure consistency of replicated objects.

Rather than determining the poll frequency statically, we employ an adaptive poll approach to dynamically vary the polling frequency based on the update rate of the object.

A node can observe the frequency of updates to an object and poll more often when the object is being modified frequently and less frequently when it is not. The update rate can be easily determined since the response to each poll contains the last modification time and the latest version number of the object. A history of modification times can be maintained and used to determine the update rate to the object. Rather than using a history of modification times, a simpler approach is to vary the poll frequency based only on the result of the most recent poll: the TTR is increased by an additive amount if the object was not modified since the last poll and reduced by a multiplicative factor if the object was modified. This notion has been explored in the context of Web cache consistency [20,26] and we use a similar idea here. In essence, an additive increase multiplicative decrease (AIMD) algorithm (see Eq. (1)) is used to probe for the update rate. A key advantage of the technique is that it can adapt to changing update rates by recomputing the TTR value after each poll.

$$\mathrm{TTR} = \begin{cases} \mathrm{TTR}_{\mathrm{old}} + C & \text{if object did not change between two polls} \\ \frac{\mathrm{TTR}_{\mathrm{old}}}{D} & \text{otherwise} \end{cases} \tag{1}$$

where the $\mathrm{TTR}_{\mathrm{old}}$ is previous TTR value, $C > 0$ is an additive constant and $D > 1$ is the multiplicative decrease constant.

After the above computation, the TTR is bound by a maximum and minimum value to prevent the TTR from becoming very large or very small, both of which can be problematic. Thus,

$$\mathrm{TTR} = \max(\mathrm{TTR}_{\min}, \min(\mathrm{TTR}_{\max}, \mathrm{TTR})) \tag{2}$$

in which $\mathrm{TTR}_{\min}$ and $\mathrm{TTR}_{\max}$ is the minimum TTR value and the maximum TTR value, respectively. In general, the values of $\mathrm{TTR}_{\min}$ and $\mathrm{TTR}_{\max}$ are dependent on the coherency tolerance of users—while there are not optimal values for these parameters.

This TTR value is used to determine the time of the next poll. Such adaptive TTR techniques have the following advantages:

1. They provide tunable parameters C and D which allow a peer to control its behavior. The constants determine how quickly the TTR is increased or decreased after each poll.
2. Only the most recent TTR and the last modification time (i.e., version number) needs to be stored with each file. No other history information is necessary, resulting in a very small per-file state space overhead.

3. The techniques can handle dynamic joins and leaves. Upon rejoining the network, the node simply resets the TTRs of all cached files to TTR_{min}. This enables the node to poll each owner quickly to determine the consistency information.

2.3. Hybrid push and adaptive pull technique

A push-based technique can provide good consistency guarantees for peers that are on-line and reachable from the owner. Pull, on the other hand, is better suited for dynamic networks but provides weaker guarantees. Push can be combined with the adaptive pull approach in a hybrid technique that combines the best features of the two approaches.

The push part of the hybrid approach works exactly as the invalidation-based push technique. In addition, the hybrid technique requires peers to occasionally poll the owner to check if the object was updated. Ideally, only those peers who are unable to receive invalidation messages should poll the owners of objects. An invalidation may not reach a peer either because it is beyond the reach of the specified TTL, or because the peer has temporarily left the network or because the overlay network is partitioned. In either case, a poll at a subsequent time allows the peer to refresh an object with the updated version. In general, it is difficult to achieve the ideal scenario where only peers who miss an invalidation message poll the owner, but we can modify the adaptive pull technique to make the polling less aggressive. Less aggressive polling reduces wasted polls from peers who are within the reach of the owner.

Since an invalidation message is an indicator of an update and the reachability from the owner, the TTR value must be updated upon receiving a push-based invalidation message to reduce wasted polls from peers who are within the reach of the owner. We use a simple approach to modify the TTR value upon receiving invalidation message as

$$\mathrm{TTR} = 2 \times \frac{\text{Time difference between two versions}}{\mathrm{Version}_{new} - \mathrm{Version}_{old}} \tag{3}$$

where $\mathrm{Version}_{new}$ is the version number in the invalidation message and $\mathrm{Version}_{old}$ is the previous version number. This TTR is used for future polls if the object is subsequently refreshed by the user.

Furthermore, in addition to adapting the TTR to the update rate and the receiving of invalidation messages, we can take into account the stability of the overlay network when computing the TTR. In general, a peer should poll more frequently when the network sees frequent joins and leaves, since frequent changes to the overlay topology increases the probability of missing an invalidate message. Similarly, the peer should poll less frequently when the network is stable.

In Chord, the number of updates to the routing table can be used as an indicator of network dynamics—a peer should poll more frequently when it sees frequent changes to its routing table.

In Gnutella, we use the number of active neighbors of a peer as an indicator of the network dynamics. Suppose that a peer has N_{conn} active connections to its neighbors and let $N_{avgconn}$ denote the average connectivity of a peer in the network (Gnutella uses $N_{avgconn}$ as a pre-defined parameter to ensure good connectivity—upon joining, a peer attempts to create logical links to these many other peers). In such a scenario, the TTR is chosen more aggressively when the number of neighbors drops below average and is made larger when a peer is well-connected and has more neighbors than the average peer. Therefore, the TTR value from Eq. (1) is further tuned as

$$\mathrm{TTR} = \mathrm{TTR} + \frac{N_{conn}}{N_{avgconn}} \times \alpha \tag{4}$$

where α is a constant. The TTR is decreased if the peer has a small number of neighbors and increased otherwise. As before, this TTR value is constrained by the maximum and minimum allowable TTR values TTR_{max} and TTR_{min}.

2.4. Analysis of our techniques

In this section, we analyze the control message overhead and scalability of our techniques, and discuss issues related to dynamic IP addresses.

2.4.1. Control message overhead

Both the push and pull techniques introduce some amount of control message overhead.

Regardless of the overlay networks used (Chord and Gnutella), the total control message overhead introduced by the pull technique is proportional to the number of replicas and the poll frequency. The total control message overhead introduced by the push technique is proportional to the update rate and the message overhead introduced by each invalidated message. Due to the different routing mechanisms used in Chord and Gnutella, the control message overheads introduced by an invalid message in these two overlay networks are different.

As shown in [10], routing a message toward the destination key in Chord requires that $O(\log N)$ messages be exchanged with high probability where N is the number of servers in the Chord network. Therefore, the control message overhead introduced by one push-based invalidation in Chord is given by $O(R \log N)$, where R is the number of replicas of a file. While this $O(R \log N)$ message overhead is comparable to $O(\log N)$ for unpopular (also referred as cold) objects that are cached only at a few peers, it can be much larger than $O(\log N)$ for popular (also referred as hot) objects that are likely to be cached by a large number of peers. For very popular objects, the number of duplicate intermediate messages introduced by unicasting invalidation message to each replica can be very large. In this case, it is more efficient to multicast invalidate messages in Chord by using the object key as the multicast group address [27]. Each replica will then join the group with the same group ID as the key of their cached object. This multicast-based invalidation mechanisms for popular objects can totally eliminate the duplicate intermediate messages introduced by unicasting message to each replica. An owner can distinguish between hot and cold object based on the frequency of query messages—hot objects will be queried more frequently than cold objects.

In Gnutella, a peer simply drops the messages which it has seen before. Thus a message can only traverse the logical link between any two adjacent peers once, and then the control message overhead introduced by one push-based invalidation in Gnutella is simply the number of all logical links within the reach of TTL limit from the owner (assuming only one logical link between any two adjacent peers). The number of all logical links within the reach of TTL is more than the number of peers within the reach of TTL. Given a fixed overlay topology, the control message overhead introduced by one push-based invalidation is only affected by the specified TTL. As shown in [28], 95% of all peers are less than seven hops away (more than 50% are less than five hops away). Therefore, given the message time-to-live ($TTL = 7$) preponderantly used and the flooding-based routing algorithm employed, the control message overhead introduced by on push-based invalidation is in the order of $\Omega(N)$ where N is the number of servers in the Gnutella network. From the perspective of efficiency, a different set of concerns arise for hot and cold objects. Since invalidations are broadcast in Gnutella using flooding, the resulting overhead is wasteful for cold objects as only a few peers need to receive this invalidation message. One possible approach to reduce this control message overhead is to use push infrequently (e.g., push every kth update) and depend on adaptive pull for consistency maintenance of cold objects. The weaker consistency guarantee provided by pull may suffice in this scenario, since cold objects are not of interest to a majority of the peers in the system. Alternatively, multicast can be used to address this issue. While the use of IP multicast can simplify the design of the overlay network, it has some limitations: (i) replicas of different objects may be in the same IP multicast group, since the possible object's key space is much larger than the possible IP multicast address space, and hence, the wasted overhead of invalidations for cold objects cannot be totally eliminated; (ii) native multicast requires support from the Internet Service Provider (ISP) and is frequently not always available. A second alternative is to use application-level multicast. Similar to the multicast mechanisms proposed for Chord, application-level multicast uses the object key as the multicast group address, and each replica joins the group with the same group ID as the key of their cached object. Such an application-level multicast mechanisms can eliminate the high overhead for cold objects and the need for ISP support, while making the overall system design more complex (since

the mechanism needs to be supported within the overlay network).

Since the control message overhead introduced by a push-based invalidate is determined by the routing algorithms employed in Chord and Gnutella, we can only control the total message overhead introduced by push and pull techniques by controlling the push and pull frequency, respectively.

2.4.2. Scalability

From the perspective of scalability, the overhead of pull becomes an issue for hot objects. If an object is hot, a large number of peers will cache the object and begin to poll the owner for updates to this object. For very popular objects, the overhead of polls can be excessive for the owner. An orthogonal issue is the stability of the owner peer. If the owner of a popular objects leaves the network frequently, then peers will no longer be able to poll the owner, nor will the owner push invalidations during such periods. Scalability and availability can both be improved by sharing owner responsibilities among a small set of peers. Rather than assigning a single owner for a file, each object can be collectively owned by a small group of peers. This allows the various owners to share the burden of polls—each peer can randomly choose a peer from the group of owners for purposes of polling. Further, it is more likely that at least one owner peer will be available at any given instant, which in turn improves availability and allows for better consistency guarantees. Note that owners of an object will need to ensure that an update received by any one owner is transmitted to all other owners as soon as possible to ensure that owners themselves store consistent version of the object. We note here that the notion of super-nodes used by P2P system such as Kazaa [29] can be used in this context when choosing owners of an object—since a super-node has good bandwidth connectivity and has long-lived sessions, having at least one super-node as an owner of an object will improve availability and scalability.

2.4.3. Handling nodes with dynamic IP addresses

Many Internet nodes use dynamic IP addresses via the DHCP protocol; this is especially prevalent for nodes that use cable modem and DSL broadband technologies. Our techniques associate a unique owner with each object. The IP address of an owner can potentially change every time they rejoin a network, making it harder to identify owners by their IP addresses. This can be problematic, since the IP address of the owner is necessary to poll for changes in the adaptive pull and the hybrid approaches. One solution to this problem is to use the Dynamic DNS service [30]. In this service, a hostname is associated with each node and the DNS mappings associated with the hostname are transparently changed every time the IP address of the node changes (this is often done via an update script that tracks changes to the host IP address and updates the DNS servers automatically on each change). Addressing the node via its hostname ensures that the node can be accessed despite changes to its IP address. Dynamic DNS is a increasingly popular service for home broadband users, and more than two dozen organizations offer free or commercial dynamic DNS services to DHCP users. This simple solution can be used for consistency maintenance as well. All owners can be tracked by their hostnames (e.g., peer5.dyndns.org) rather than their IP addresses, which makes the effects of DHCP transparent to our techniques.

While a dynamic DNS service is essential for consistency maintenance in Gnutella networks, nodes in Chord can use either a dynamic DNS hostname or use the owner's peerID for addressing the node—so long as the peerID remains unchanged across joins and leaves, this form of addressing is also resilient to changes in the node's IP address.

3. Prototype implementation

In this section, we first describe how to implement our hybrid push–pull algorithm into the Chord protocol with Chord's library. We then present details on implementing our hybrid push–pull algorithm into Gtk-Gnutella [31], an open-source implementation of the Gnutella protocol.

3.1. Implementation in Chord

In [27], Dabek defines a *key-based routing API* (*KBR*), which represents basic capabilities that are common to all structured overlays. They also demonstrate the ways to build distributed hash tables, group anycast and multicast, and decentralized object location and routing upon the basic KBR. The KBR API supports a *route* method to forward a message to a peer. Our implementation is based on this method.

The push-based invalidation message announces the availability of a new version. First, the owner of the object sends an invalidate message to to the root node using *route*. The root maintains a list of all replica peers, and upon receiving this message, sends invalidates to all replica peers using *route*. Upon receiving invalidates, a peer marks the cached object as stale.

The adaptive pull can be implemented by HTTP/1.1 through the if-modified-since (IMS) HTTP messages.

3.2. Implementation in Gnutella

To implement our hybrid push–pull algorithm into Gtk-Gnutella version 0.17 [31], we extend the Gnutella protocol to incorporate push-based invalidations and use HTTP/1.1 to implement adaptive pull. We add a new message type for push-based invalidations. Each invalidation message contains a 16 byte object identifier (an MD-5 [32] hash of the object name), the object name, its last modification time, the owner's IP address, port number, and the TTL for the invalidate message. Upon receiving an invalidate, a peer invalidates the object if present in its local cache, decrements the TTL, and forwards the message to its neighbors.

Gtk-Gnutella uses HTTP to download objects from a peer. Since support for HTTP is already built into the system, we can use this functionality to implement adaptive pull. Specifically, a peer uses if-modified-since (IMS) HTTP messages to poll for updates to objects. Each peer computes TTR values as discussed in the previous section. The TTR value is recomputed after each poll and in response to changes in a peer's neighbors.

Out prototype sets the TTR value of an object to TTR_{min} when it changes from *Stale* or *Possible Stale* to *Valid* or if the object is newly downloaded. In order to be backward compatible with current Gnutella protocol, we always set the status of the objects downloaded from peers that do not support cache consistency to *valid* and set the TTR value to -1. We do not poll cached files with negative TTRs.

4. Experimental evaluation

In this section, we demonstrate the efficacy of our techniques using simulations and experiments with our prototype implementation. We use simulations to explore the parameter space along various dimensions and use our GTK-Gnutella prototype to measure implementation overheads (an aspect that simulations do not reveal). In what follows, we first present our experimental methodology and then our experimental results.

4.1. Experimental methodology

4.1.1. Simulation environment

We have extended the Chord simulator [33] to include our push–pull cache consistency techniques. We also designed an event-based Gnutella simulator to evaluate our cache consistency techniques for unstructured networks. The topology of the overlay and various system parameters are initialized using observed statistics. We borrow heavily from recent measurements studies [28,34,35] to initialize parameters such as link bandwidths, network diameter, node connectivity, session times, object popularities, etc.

In these two simulators, each peer is responsible for answering queries and propagating query messages to successors (Chord) or neighbors (Gnutella), and for servicing object download requests. Each peer can also initiate query requests; inter-arrival times of queries are exponentially distributed and a certain fraction of the query responses is assumed to result in object downloads. Our simulators also incorporate an update process that generates updates to objects stored at owners. An update causes the last

modification time and the version number of the object to be updated at the owner, and initiates a push-based invalidation message being sent to the root of the object in Chord or being broadcasted in Gnutella. The default values of various parameters used in our simulations are listed in Table 1.

The workload for generating queries, object downloads and updates to objects is generated synthetically. While measurements of query rates and object downloads are available from recent studies, realistic distributions of object update rates are not available since current file sharing applications only share static objects. Consequently, we use update distributions of Web pages [18,19] as a reasonable indicator of updates in overlay environments. Specifically, we assume four types of objects: highly mutable, very mutable, mutable, and immutable. Each category has a different mean update rate. The percentage of the objects in each category and the mean update rate in each category is (0.5%, 15 s), (2.5%, 7.5 min), (7%, 30 min), and (90%, 1 day). Note that the mean lifetime of an immutable object is longer than our simulation duration of 10 h.

4.1.2. Metrics for performance analysis

We use two different metrics to evaluate our techniques.

- *Fidelity:* Fidelity is the degree to which a technique can provide consistency guarantees. We use a metric called *False Valid Ratio* (*FVR*) to determine the fraction of query responses or downloaded objects that return stale requests (i.e., are falsely reported as valid by their peers). The query false valid ratio (QFVR) is the fraction of query responses that list stale objects. A query that returns some stale objects is not necessarily bad, since the user can pick one of the matches for an actual download (e.g., a match with the largest version number or the most recent modification time). The download false valid ratio (DFVR) is the fraction of the downloaded objects that are stale. The false valid ratio for queries and downloads should be as close to zero as possible. It is possible to reduce the false valid ratios by having a peer return the version number/last modification time of a object in response to query messages. The information can be used by a peer to download the most recent version of an object (note that this version can still be stale if the owner of an object has a newer version and is not within reach of the query message).
- *Control message overhead:* The control message overhead is the number of control messages that are exchanged to maintain consistency of replicas. In the push approach, this is evaluated by the number of invalidates per Update. In the pull approach, the control message overhead is defined to be the number of poll messages per Update. We note that, while flooding of invalidations in Gnutella is not necessarily efficient, flooding is currently the mechanism of choice to propagate queries and ping messages in Gnutella. Thus, the overhead of pushing invalidates is not significantly larger than other overlay network functions. It is also possible to reduce this overhead by piggybacking invalidates on query or ping messages; we do not consider such optimizations in this paper.

4.1.3. Network environment used in the simulations

Our simulations are conducted in dynamically changing overlay networks. We assume peers leave

Table 1
Parameters for the dynamic network environment

Parameter	Description	Default value
L_{sim}	Length of simulation	10 h for Gnutella, 48 h for Chord
F_{enable}	This flag turns on/off the failure mode	[FALSE, TRUE]
R_f	Percentage of maximum offline nodes	50%
I_f	Average time between successive disconnections	5 s
D_f	Average offline duration	2 h
$I_{topochk}$	Average time between successive topology checks (only in Gnutella)	5 min

and rejoin the network randomly, based on the three parameters R_f, I_f, and D_f, as defined earlier. This situation is close to the P2P overlay networks in use today. However, the size of the network used in our simulation is much smaller than a real network, since the memory and computation overhead required to run large-scale simulations are prohibitive. Unless specified otherwise, we assume a Gnutella overlay network consisting of 500 peers and 5000 objects, and a Chord overlay network consisting of 5000 peers and 50,000 objects.

We also show experiments with up to 5000 peers and 50,000 objects in a Gnutella overlay network and up to 15,000 peers and 150,000 objects in a Chord overlay network to demonstrate that our results apply to larger overlay networks as well. We note that it is very memory-intensive to simulate large P2P networks (simulating caches in a Gnutella overlay network consisting of 5000 peers and 50,000 objects requires more than 2 GB of memory). The memory limitation on our machines is the primary reason why we do not present results for Gnutella overlay networks larger than 5000 peers.

4.2. Simulation results

Our experiments are run with F_{enable} set to TRUE which means a dynamic network. All other parameters are set to their default values.

In a dynamic network, peers will frequently leave and rejoin the network. We simulate this behavior by incorporating a "failure" process that determines the lifetime of a peer session; a peer leaves the network when its session lifetime is exceeded. We simulate this behavior with three parameters: the maximum offline ratio, R_f, which is the maximum percentage of peers that are disconnected from the network at any given time; the session lifetime or the time between successive disconnections, I_f; and the average duration that a peer remains offline, D_f.

4.2.1. Impact of the dynamics of the overlay network

To understand the effects of the dynamics of the overlay network, we first vary the fraction of offline peers R_f from 5% to 50% in the Chord and Gnutella overlay networks, and measure the impact on the QFVR and DFVR. As shown in Figs. 1 and 2, push can provide better consistency guarantees than a pure pull approach, even in a dynamic network. The FVRs degrade as the fraction of offline peers increases. However, the hybrid approach outperforms both push and pull, since it employs a combination of the two and can employ pull in scenarios where push is ineffective. The approach can provide good fidelity and is relatively unaffected even when the fraction of offline peers is as high as 50%.

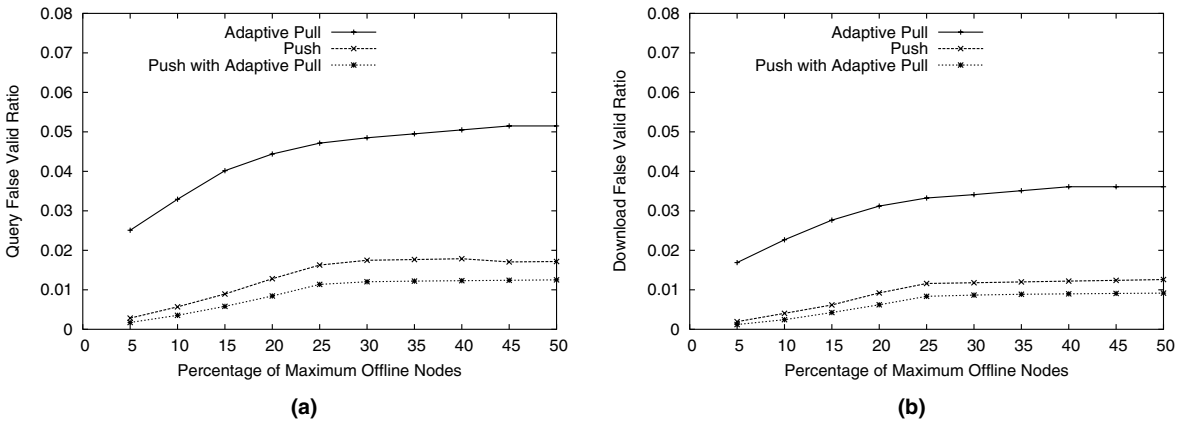


Fig. 1. Impact of maximum offline peers on false valid ratio—Chord network. (a) Query false valid ratio and (b) download false valid ratio.

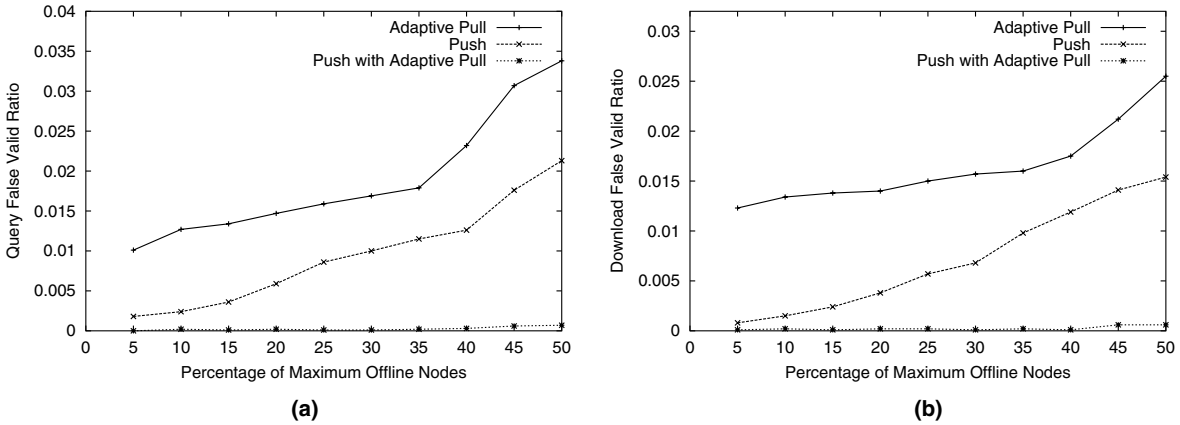


Fig. 2. Impact of maximum offline peers on false valid ratio—Gnutella network. (a) Query false valid ratio and (b) download false valid ratio.

Next, we vary the time between successive disconnections I_f in Chord and Gnutella overlay network. Intuitively, as I_f increases, fewer peers leave the network. The results are similar to the previous scenario (see Figs. 3 and 4). Push outperforms a pure-pull approach; both techniques yield better consistency guarantees in more stable networks. As before, the hybrid approach performs well and is relatively unaffected by the dynamics of the network.

Overall, our results demonstrate that a hybrid push–pull approach works well in highly dynamic overlay networks.

4.2.2. Impact of the interval between successive topology checks

When a peer leaves the Gnutella overlay network, it tears down all connections to its neighbors. This causes each of its neighbor to lose one of their active connections. In the scenario where many peers leave the network, the network may become partitioned. To overcome this drawback, actual Gnutella implementations [29,36,3] let a peer form new links with other active peers if a neighbor leaves the network. To simulate this behavior, we implement a topology checking process that periodically checks the connectivity of

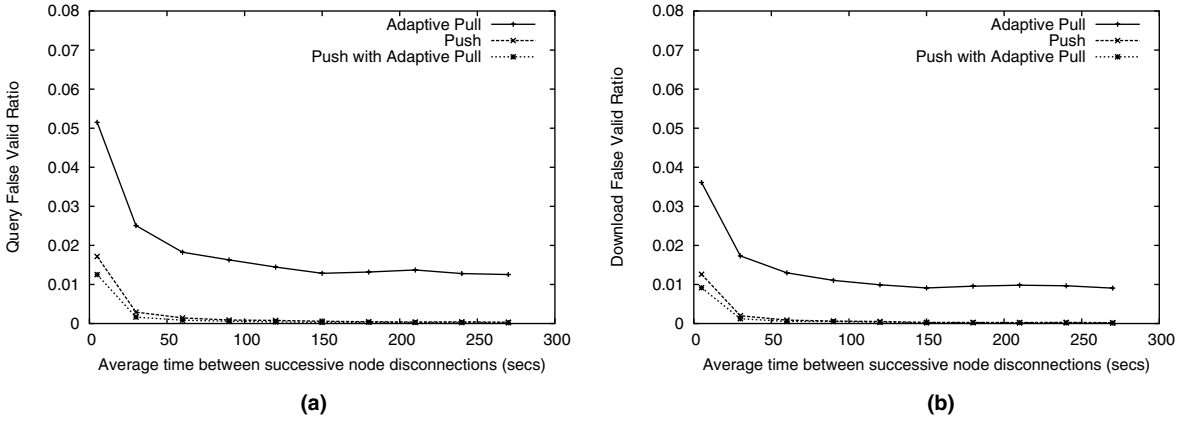


Fig. 3. Impact of time between successive peer disconnections on false valid ratio—Chord network. (a) Query false valid ratio and (b) download false valid ratio.

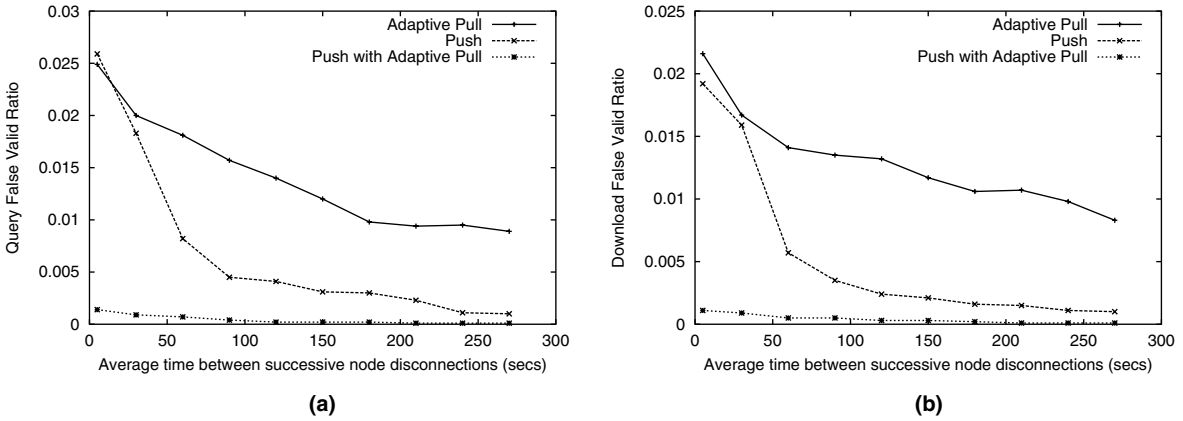


Fig. 4. Impact of time between successive peer disconnections on false valid ratio–Gnutella network. (a) Query false valid ratio and (b) download false valid ratio.

each peer and constructs new logical links if a peer has fewer neighbors than a threshold.

In this section, we study the effects of topology checking process on the Gnutella overlay network by varying the I_{topochk} values from 5 s to 170 s. This parameter effectively determines the delay between a broken connection and the instant when a new connection is formed by a peer. Observe that this parameter only impacts push, since the pull technique does not rely on the Gnutella overlay network and then is not affected by the connectivity of the overlay network. Hence, we focus on the push and the hybrid push–pull approach. Fig. 5 shows the query FVR and the download FVR for the two approaches. As shown, the longer the delay for replacing broken links with new neighbors, the worse the performance of push. In contrast, the hybrid approach is unaffected by the topology changes, since the approach can resort to pulls when invalidates do not reach a peer.

4.2.3. Impact of the network size

We also conduct experiments to investigate the effects of network size on fidelity of *Push* and *Push with Adaptive Pull*. We vary the network size from 200 to 5000 nodes, I_{query} and I_{update} from 1 s to 200 ms in Gnutella overlay network. The TTL

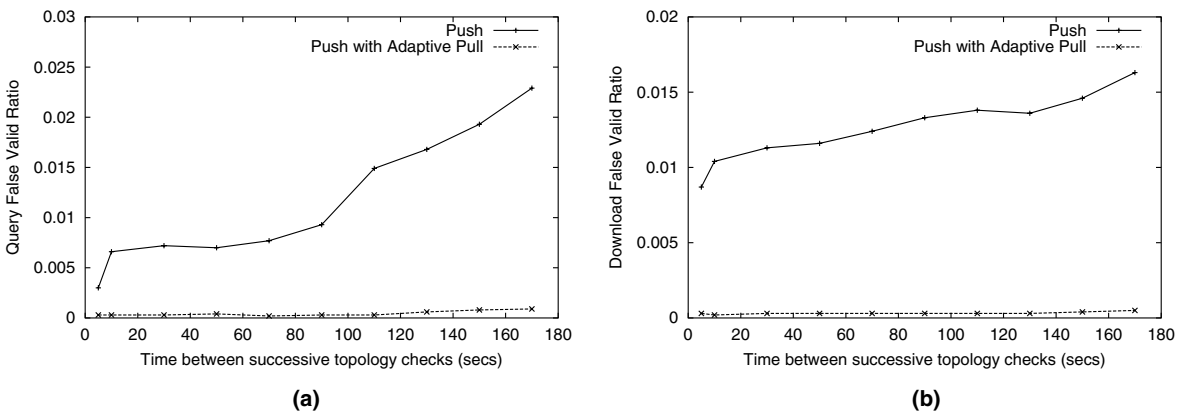


Fig. 5. Impact of time between successive topology checks on false valid ratio—Gnutella network. (a) Query false valid ratio and (b) download false valid ratio.

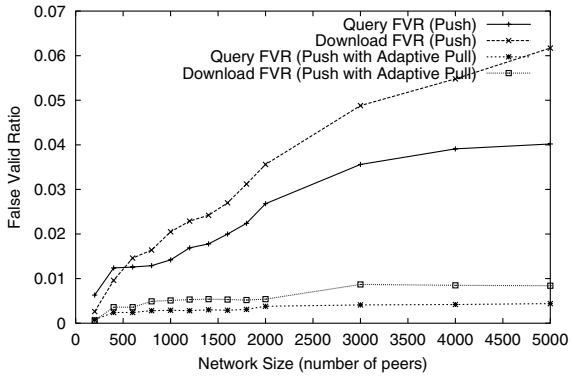


Fig. 6. Impact of network size on fidelity—Gnutella network.

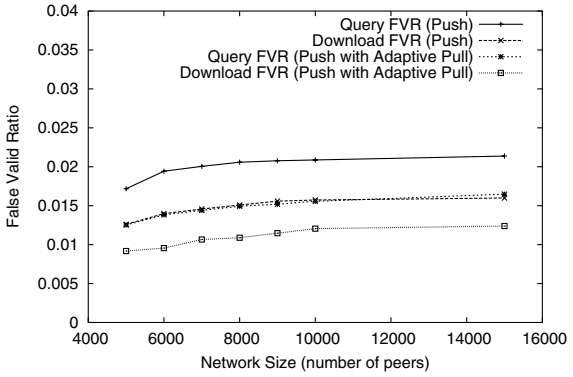


Fig. 7. Impact of network size on fidelity—Chord network.

for invalidations is set to six hops. Fig. 6 plots the QFVR and DFVR for the two techniques. Since the TTL value is fixed, invalidates reach fewer peers as the network size increases. Consequently, the QFVR and DFVR for push increases with increasing network size. In contrast, the hybrid approach provides significantly better fidelity, since the adaptive pull enables distant peers to maintain consistency when push is ineffective. The result shows that the effectiveness of push is crucially dependent on proper choice of the TTL value for invalid messages. The hybrid push–pull approach is less sensitive to the choice of this value, since it can fall back on the pull approach for consistency.

We also vary the network size from 5000 to 15,000 nodes, I_{query} and I_{update} from 1 s to 200 ms in the Chord overlay network. Different from the Gnutella overlay network, there does not exist a TTL limit in the Chord overlay network, and the Chord overlay network guarantees the delivery of invalidation messages given that the path from source to destination is online. Fig. 7 plots the QFVR and DFVR for the two techniques. Since we use the same value of *Maximum offline peers ratio* (R_f) in all network sizes, the invalidates reach a similar portion of peers as the network size increases. Consequently, the QFVR and DFVR for push and the hybrid approach do not vary much with increasing network size. The result shows that the hybrid push–pull approach outperforms the pure push approach in all cases.

4.2.4. Control messages overhead

As shown in Section 2.4.1, the control message overhead introduced by the pull technique is only proportional to the number of replicas and the poll rate and is not affected by the routing mechanisms used in overlay networks. Therefore, we only evaluate the control message overhead introduced by push-based invalidations with different network size in this section. Fig. 8 plots the invalidation messages per *Update* for Chord and Gnutella overlay networks. The result shows: (i) the number of invalidation messages per Update in Chord varies from 17 to 32 which is comparable to $\log N$ (N is the network size) as the network size varies from 5000 to 15,000; (ii) the number of invalidation messages per Update in Gnutella varies from 500 to 1300 which is comparable to N (N is the network size) as the network size varies from 200 to 5000. This result is consistent with the discussion about control message overhead in Section 2.4.1.

4.3. Results from the prototype implementation using Gtk-Gnutella

In this section, we study the implementation overheads of various operations need for consistency maintenance in our hybrid push–pull approach using our Gtk-Gnutella prototype. We conduct two sets of experiments, one on our laboratory testbed and another on PlanetLab [37], a distributed testbed of experimental machines. We only summarize our key results here and refer interested readers to our technical report [38] for more details.

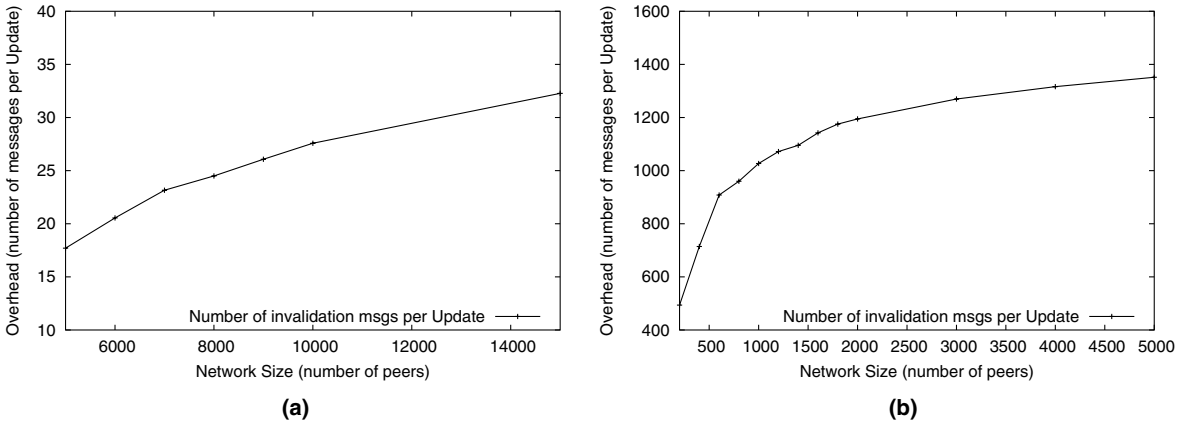


Fig. 8. Number of control message per Update. (a) Chord overlay network (b) Gnutella overlay network.

Our measurements in the laboratory testbed show that the overhead of event processing for incoming poll, outgoing goll, incoming invalidation, and outgoing invalidation are 181.83 μs, 1.36 ms, 150.24 μs, and 265.34 μs, respectively. The results from our PlanetLab experiments show that the overhead of event processing for incoming poll, outgoing goll, incoming invalidation, and outgoing invalidation are 172.46 μs, 1085.66 ms, 93.27 μs, and 98.78 μs, respectively.

Overall, these results indicate that the overhead of incoming poll processing, incoming invalidation processing and outgoing invalidation processing is very small, and the overhead of an outgoing poll highly depends on the round trip time of the route to the owner.

5. Related work

Several research efforts have investigated the design of push-based, pull-based, and push–pull-based cache consistency techniques in Web environments. Push-based techniques that have been developed include leases [22,18,21]. Pull-based techniques include time-to-live (TTL) values [39] and adaptive polling [40]. Adaptive combinations of push and pull have also been considered in [40]. Whereas these efforts have focused on Web proxy environments where proxies and/or servers are seldom off-line, they are not directly applicable to peer-to-peer environments that use their own routing techniques using overlays and where peers can freely join and leave the network at any time.

Push-based techniques developed for Web caching requires a server to maintain state of all proxies that cache an object. Further, it has been shown that push provides strong guarantees and is more efficient than pull. In contrast, our push-based invalidations are propagated using broadcast and require no state at the origin peer. Further, unlike the case of Web proxies, push, due to its broadcast nature, incurs two orders of magnitude more overhead than pull. We also note that push-based invalidations are less effective in dynamic networks or in networks with large diameters (thus, the guarantees provided by push for P2P networks are weaker than in Web environments).

The effectiveness of adaptive pull-based techniques employed by Web proxies is crucially dependent on proper choice of the TTR value for dynamic files. These techniques employ prediction capabilities at the clients/peers. An alternative of course is to leave the prediction to the servers/*origin-peer*. Such schemes are discussed in [41,21]. These schemes work as follows:

- The client does not use a TTR or prediction algorithm, but instead depends on some meta-data associated with the data to decide the time at which to poll the server.
- Since the server has access to all the data, it can use a prediction algorithm to predict a time when the data is going to change. The server

then attaches this time value to outgoing data. The client will use this meta-data to decide when to poll next.
- Since the server has better access to data than the client, server predictions will be in general more "accurate" than using a TTR algorithm at the client.

Though the *Server-Prediction* approach looks like a better option than the TTR algorithm in PAP, it runs into the following problems:

- The approach requires previous history for the relevant data to be maintained at the origin-peer. This implies increased state information and computational needs at the origin-peer which in turn affects scalability.
- The approach is more suitable for data that changes in a predictable manner. We are interested in updates that are inherently unpredictable. For dynamic data, the performance will be slightly better than adaptive TTR, but at a cost of origin-peer resources and scalability.

The Bayou system [24,42] and the Coda file system [43,44] also maintain consistency in weakly connected environments similar to P2P overlay networks (e.g., disconnections of mobile users). Bayou uses an eventual consistency model based on the theory of epidemics; our push–pull consistency techniques provide stronger consistency semantics than eventual consistency. Consistency in Coda is concerned with write–write conflicts that occur as a result of disconnections. Since we assume that all updates are only made at the origin peer, write–write conflicts are not an issue in the system model considered in this paper. Lastly, P2P overlay networks can have thousands or tens of thousands of nodes [28] and it is important for the consistency techniques to scale to these sizes.

6. Concluding remarks

In this paper, we presented techniques to maintain temporal consistency of replicated objects in a peer-to-peer overlay network. We considered Chord and Gnutella and presented techniques for maintaining consistency in these two overlay networks even when peers containing replicated objects dynamically join and leave the network. We presented extensions to the Chord and Gnutella protocol to incorporate our consistency techniques and implemented the extensions of the Gnutella protocol into a Gtk-Gnutella prototype. An experimental evaluation of our techniques showed that: (i) a push-based approach achieves near-perfect fidelity in a stable overlay network, (ii) a hybrid approach based on push and pull achieves high fidelity in highly dynamic networks and (iii) the run-time overheads of our techniques are small, making them a practical choice for overlay networks.

As part of future work, we plan to extend our techniques to the BitTorrent peer-to-peer system and evaluate their effectiveness in maintaining the consistency of traces disseminated from our BitTorrent-based trace repository [16].

References

[1] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, Oceanstore: an architecture for global-scale persistent storage, in: Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00), Cambridge, MA, 2000, pp. 190–201.
[2] L.P. Cox, C.D. Murray, B. Noble, Pastiche: Making backup cheap and easy, in: Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02), Boston, MA, 2002.
[3] Gnutella, Gnutella Org. Available from: <http://www.gnutella.org>.
[4] I. Clarke, O. Sandberg, B. Wiley, T.W. Hong, Freenet: a distributed anonymous information storage and retrieval system, in: Proceedings of the ICSI Workshop on Design Issues in Anonymitty and Unobservability, Berkeley, CA, 2000, pp. 311–320.
[5] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, Wide-area cooperative storage with cfs, in: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01), Chateau Lake Louise, Banff, Canada, 2001, pp. 202–215.
[6] C.G. Plaxton, R. Rajaraman, A.W. Richa, Accessing nearby copies of replicated objects in a distributed environment, in: Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97), Newport, RI, 1997, pp. 311–320.
[7] K. Hildrum, J.D. Kubiatowicz, S. Rao, B.Y. Zhao, Distributed object location in a dynamic network, in:

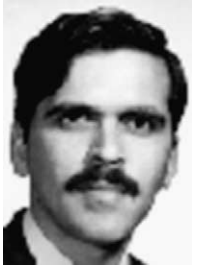
Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02), Winnipeg, Manitoba, Canada, 2002, pp. 41–52.
[8] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, J.D. Kubiatowicz, Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination, in: Proceedings of the 11th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'01), Port Jefferson, NY, 2001, pp. 11–20.
[9] M. Castro, P. Druschel, A. Kermarrec, A. Rowstron, Scribe: a large-scale and decentralized application-level multicast infrastructure, IEEE Journal on Selected Areas in communications (JSAC) 20 (8) (2002) 1489–1499.
[10] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for Internet applications, in: Proceedings of the 2001 Annual ACM Conference of the Special Interest Group on Data Communication (SIGCOMM'01), San Diego, CA, 2001, pp. 149–160.
[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, in: Proceedings of the 2001 Annual ACM Conference of the Special Interest Group on Data Communication (SIGCOMM'01), San Diego, CA, 2001, pp. 161–172.
[12] A. Rowstron, P. Druschel, Pastry: scalable, distributed object location and routing for largescale peer-to-peer systems, in: Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01), Heidelberg, Germany, 2001, pp. 329–350.
[13] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, IEEE Journal on Selected Areas in Communications 22 (1) (2004) 41–53.
[14] P. Maymounkov, D. Mazieres, Kademlia: A peer-to-peer information system based on the xor metric, in: In Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS'02), Cambridge, MA, 2002, pp. 53–65.
[15] Napster. Available from: <http://www.napster.com>.
[16] Umass Trace Repository. Available from: <http://trace.cs.umass.edu>.
[17] Bittorrent. Available from: <http://www.bittorrent.com>.
[18] V. Duvvuri, P. Shenoy, R. Tewari, Adaptive leases: A strong consistency mechanism for the world wide Web, in: Proceedings of the 19th IEEE Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom'00), Tel Aviv, Israel, 2000, pp. 834–843.
[19] J. Yin, L. Alvisi, M. Dahlin, C. Lin, Hierarchical cache consistency in a wan, in: Proceedings of the 2nd USENIX Symposium on Internet Technologies (USITS'99), Boulder, CO, 1999, pp. 13–24.
[20] B. Urgaonkar, A. Ninan, M. Raunak, P. Shenoy, K. Ramamritham, Maintaining mutual consistency for cached Web objects, in: Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, AZ, 2001, pp. 371–380.
[21] C. Liu, P. Cao, Maintaining strong cache consistency in the world wide Web, in: Proceedings of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS-97), Baltimore, Maryland, 1997, pp. 12–21.
[22] P. Cao, S. Irani, Cost-aware www proxy caching algorithms, in: Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems (USITS'97), Monterey, CA, 1997, pp. 193–206.
[23] J. Chu, K. Labonte, B.N. Levine, Availability and locality measurements of peer-to-peer file systems, in: Proceedings of the 2002 SPIE ITCom: Scalability and Traffic Control in IP Networks II Conference, Boston, MA, 2002.
[24] A.J. Demers, K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, B.B. Welch, The Bayou architecture: support for data sharing among mobile users, in: Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, pp. 2–7.
[25] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser, Managing Update conflicts in Bayou, a weakly connected replicated storage system, in: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95), Copper Mountain Resort, Colorado, 1995, pp. 172–183.
[26] R. Srinivasan, C. Liang, K. Ramamritham, Maintaining temporal coherency of virtual data warehouses, in: Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98), Madrid, Spain, 1998, pp. 60–70.
[27] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, I. Stoica, Towards a common api for structured peer-to-peer overlays, in: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, CA, 2003.
[28] M. Ripeanu, I. Foster, A. Iamnitchi, Mapping the Gnutella network: properties of large scale peer-to-peer systems and implications for system design, IEEE Internet Computing 6 (1) (2002) 50–57.
[29] Kazaa. Available from: <http://www.kazaa.com>.
[30] Dynamic dns, DynDNS.org. Available from: <http://www.dyndns.org/services/dyndns>.
[31] Gtk-Gnutella. Available from: <http://gtk-gnutella.sourceforge.net>.
[32] R. Rivest, The md5 message-digest algorithm, Tech. Rep. RFC 1321, MIT LCS and RSA Data Security Incorporation, 1992.
[33] Chord Simulator, The Chord Project. Available from: <http://www.pdos.lcs.mit.edu/chord/>.
[34] S. Saroiu, P. Gummadi, S. Gribble, A measurement study of peer-to-peer file sharing systems, in: Proceedings of the ACM/SPIE Multimedia Computing and Networking 2002 (MMCN'02), San Jose, CA, 2002, pp. 156–170.
[35] K. Sripanidkulchai, The popularity of Gnutella queries and its implications on scalability, Tech. rep., 2001.
[36] Limewire. Available from: <http://www.limewire.com>.
[37] Planet lab. Available from: <http://www.planet-lab.org>.
[38] X. Liu, J. Lan, P. Shenoy, K. Ramamritham, Consistency maintenance in dynamic peer-to-peer overlay networks, Tech. rep., University of Massachusetts, Amherst.

[39] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, K.J. Worrel, A hierarchical Internet object cache, in: Proceedings of the 1996 annual USENIX Technical Conference, San Diego, CA, 1996, pp. 153–164.
[40] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, P. Shenoy, Adaptive push–pull of dynamic Web data, IEEE Transactions on Computers 51 (6) (2002) 652–688.
[41] J. Gwertzman, M. Seltzer, The case for geographical push caching, in: Proceedings of the 5th Annual IEEE Workshop on Hot Topics in Operating Systems (HOTOS-V), Orcas Island, WA, 1995, pp. 51–55.
[42] M.J. Spreitzer, M.M. Theimer, K. Petersen, A.J. Demers, D.B. Terry, Dealing with server corruption in weakly consistent, replicated data systems, in: Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'97), Budapest, Hungary, 1997, pp. 234–240.
[43] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, D.C. Steere, Coda: a highly available file system for a distributed workstation environment, IEEE Transactions on Computers 39 (4) (1990) 447–459.
[44] J.J. Kistler, M. Satyanarayanan, Disconnected operation in the coda file system, ACM Transactions on Computer Systems 10 (1) (1992) 3–25.

Xiaotao Liu received his Bachelor of Engineering in Computer Science from the South China University of Technology, Guangzhouu, China in 1998. He worked as a Technical Consultant in Computer Associates Co., Ltd. from 1998 to 2001. He joined the Department of Computer Science at the University of Massachusetts, Amherst in 2001. He is currently a Ph.D. candidate there. His research interests include operating systems, computer networks and distributes systems.

Jiang Lan received his Master of Science in Computer Science from the University of Massachusetts, Amherst in 2002. He currently engineerings at Verizon Labs.

Prashant Shenoy received his B. Tech in Computer Science and Engineering from IIT Bombay, India in 1993, and his M.S. and Ph.D. in Computer Sciences from the University of Texas at Austin in 1994 and 1998, respectively. He is currently an Associate Professor in the Department of Computer Science at the University of Massachusetts, Amherst. His research interests are operating systems, computer networks and distributed systems. Over the past few years, he has been the recipient of the National Science Foundation CAREER award, the IBM Faculty Development Award, and the UT Computer Science Best Dissertation Award. He is a member of the ACM and the IEEE.



Krithi Ramaritham received the Ph.D. degree in Computer Science from the University of Utah and then joined the University of Massachusetts. He is currently at the Indian Institute of Technology Bombay as the Head in the Kanwal Rekhi School of Information technology and the Vijay and Sita Vashee Chair Professor in the Department of Computer Science and Engineering. He was a Science and Engineering Research Council (United Kingdom) visiting fellow at the University of Newcastle upon Tyne and has held visiting positions at the Technical University of Vienna and at the Indian Institute of Technology Madras. His areas of interest include database systems, real-time systems and Internet computing. He is a fellow of the IEEE and a fellow of the ACM. His conference chairing duties include the Real-Time Systems Symposium—as Program Chair in 1994 and as General Chair in 1995, the Conference on Data Engineering—as a Vice-Chair in 1995 and 2001 and as a Program Chair in 2003, and the Conference on Management of Data—as Program Chair in 2000 and General Chair in 2005. He has also served on numerous program committees of conferences and workshops. His editorial board contributions include IEEE Transactions on Knowledge and Data Engineering, IEEE Transaction on Parallel and Distributed Systems, IEEE Internet Computing, the Real-Time Systems Journal, the WWW Journal, the Distributed and Parallel Databases journal, and the VLDB Journal. He has co-authored two IEEE tutorial texts on real-time systems, a text on advances in database transaction processing, and a text on scheduling in real-time systems.