Consistency Maintenance In Peer-to-Peer File Sharing Networks

Jiang Lan, Xiaotao Liu, Prashant Shenoy and Krithi Ramamritham†
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
jiang.lan@labs.gte.com, {xiaotaol,shenoy,krithi}@cs.umass.edu

Abstract— **While the current generation of peer-to-peer networks share predominantly static files, future peer-to-peer networks will support sharing of files that are modified frequently by their users. In this paper, we present techniques to maintain temporal consistency of replicated files in a peer-to-peer network. We consider the Gnutella network and present techniques for maintaining consistency even when peers containing replicated files dynamically join and leave the network. We present extensions to the Gnutella protocol to incorporate our consistency techniques and implement them 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 P2P network, (ii) a hybrid approach based on push and pull achieves high fidelity in highly dynamic P2P networks and (iii) the run-time overheads of our techniques are small, making them a practical choice for P2P networks.**

Keywords: Consistency maintenance, Peer-to-peer file sharing, Gnutella, Push, Adaptive Pull

I. Introduction

A. Motivation

The past few years have seen a dramatic increase in the popularity and use of peer-to-peer (P2P) file sharing networks. Current P2P systems are specifically designed to share static content such as music and video files. The utility of P2P systems goes beyond sharing of static files—future P2P applications (e.g., collaborative P2P applications) can be expected to share dynamic files. In such applications, shared files are not necessarily static; files may be modified and updated during their lifetime. To handle such dynamic content, P2P networks must evolve from a predominantly read-only system to one where files can be both read and written. Since files may be widely replicated in a P2P system, handling dynamic files requires consistency techniques to ensure that all replicas of a file are temporally consistent with one another.

†Also with the Indian Institute of Technology Bombay

Consistency techniques have been widely studied in the context of web proxy caches and a variety of techniques have been proposed for maintaining consistency of time-varying web content in web proxies [1], [2]. However, these techniques are not directly applicable to P2P systems. P2P systems are known to be highly dynamic—peers can dynamically join and leave the network at any time and the mean session duration of a peer is only a few hours [3]. Since peers containing replicated content may not be part of the network when a file is modified, maintaining consistency is more challenging in these environments (unlike in web environments, where web proxy caches are mostly available and failures are rare). Due to this crucial difference, novel consistency techniques specifically designed to handle unavailable peers are required and is the focus of this paper.

B. Research Contributions

In this paper, motivated by the need to support modifications to replicated files, we propose three consistency maintenance techniques for P2P networks. Our first two techniques are based on push and pull, respectively, and have complementary tradeoffs. A flooding-based push approach can provide near-perfect fidelity but has high communication overheads and is suitable only for static networks. 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 the Gnutella protocol to incorporate all three techniques. We then implement our hybrid technique into Gtk-Gnutella—a public source implementation of the Gnutella file sharing protocol.

We evaluate our techniques using a combination of simulations and prototype implementation. Our results show that while push is more suitable for stable P2P networks, our hybrid approach can provide good fidelity even in highly dynamic environments. Our measurements from the prototype implementation indicate that this fidelity can be provided at a reasonable run-time cost.

The remainder of this paper is structured as follows. Sections II provides a brief overview of the Gnutella P2P file sharing network. We present our consistency maintenance techniques in Section III. Enhancements to the Gnutella protocol and details of our prototype implementation are presented in Section IV. Section V presents our experimental results, and finally, Section VI presents our conclusions.

II. Overview of the Gnutella File Sharing Network

Peer-to-peer file sharing networks provide an infrastructure for communities to share storage and popular files. P2P systems can be based on a centralized or distributed model.

In the centralized model, a small number of centralized servers maintain an index of files stored at the various peers. A peer registers itself and its shared contents with the indexing server upon joining the network. Queries for a file are sent to the indexing server, which returns a list of all matches and the addresses of the corresponding peers. A user can then select the desired file, which is directly downloaded from the corresponding peer (usually via a direct HTTP connection). Note that the indexing server only stores an index of all shared files in the system; the files themselves are never stored at the central server. 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 indexing information is only refreshed periodically).

Decentralized peer-to-peer systems attempt to overcome these drawbacks. Gnutella is a widely used decentralized system, so we primarily focus on the Gnutella model. In this model, each peer is simultaneously a server and a client. Peers are organized into an overlay network, where each peer is connected to some number of neighboring peers over logical links. The overlay network is used for searching shared content. A peer initiates a query by sending the query message to all its neighbors. Each neighbor in turn sends the query to all its neighbors and so on. Thus, 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. If the requested file is found at a peer, it transmits the file information (name, size, peer ID, etc) back to the initiator on the reverse path. Upon receiving all matches, the user can select the desired file and download it from the corresponding peer (via a direct HTTP connection). Thus, each peer maintains its own index in Gnutella and queries are processed by flooding the overlay network. A peer can join or leave the network at any time. A peer can join the overlay by running a discovery protocol to actively discover peers and form logical links with a subset of these peers. Similarly, if one or more neighbors leave the network, a peer can form a new logical links with other active peers using the discovery protocol. In essence, Gnutella-based P2P systems build, at the application level, a virtual overlay network with its own routing mechanisms. The set of supported messages in the Gnutella protocol [4] are described in Table I.

Descriptor	**Description**
Ping	Used to actively discover hosts on the network. A peer receiving a Ping descriptor is expected to respond with one or more Pong descriptors.
Pong	The response to a Ping. Includes the address of a connected Gnutella peer and information regarding the amount of data it is making available to the network.
Query	The primary mechanism for searching in Gnutella. A peer receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHit	The response to a Query. This descriptor provides the recipient with enough information to acquire the data matching the corresponding Query
Push	A mechanism that allows a fire-walled peer to contribute file-based data to the network.

TABLE I

Currently Supported Message Descriptors in the Gnutella Protocol

III. Consistency Techniques for P2P Gnutella File Sharing Networks

In this section, we present techniques for maintaining consistency of replicated files in a P2P network. We assume that each file in the system has a unique owner; the owner of a file is simply the peer where the file originated (i.e., the peer where the file was created or first shared). Modifications to a file can only be made by its owner. While this assumption may seem overly restrictive, it is not—any user (peer) may modify a file, 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 file at all times. Observe that additional mechanisms such as distributed locking are required to prevent multiple peers from simultaneously updating a file; these techniques can be implemented separately and are beyond the scope of this paper. Each file is also associated with a version number; the version number is incremented by the owner upon each update. Since a file may be arbitrarily replicated at different peers, upon a modification, the replicas will need to be made consistent with the version at the owner peer. We present three different techniques for doing so in the remainder of this section.

A. Push: Owner-initiated Consistency

In the owner-initiated approach, the owner broadcasts an invalidation message upon each update to a file (alternately, the new version of the file may be broadcast). The broadcast message propagates through the P2P overlay 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. Upon receiving an invalidation message, a peer checks its shared cache and invalidates the file if the version number of the cached copy is smaller than the version number specified in the invalidation message (if updates are sent instead of invalidations, the cached copy is replaced by the new version).

The main advantage of such a push-style approach is its simplicity and stateless nature. Since invalidations are propagated via flooding, the owner need not maintain a list of peers holding a replica of the file. Further, the approach guarantees a strong notion of consistency so long as all peers holding a replica are reachable from the owner (i.e., are no more than TTL hops away). The limitation though is that the broadcast nature of the technique increases the control message overhead substantially, especially for objects cached only at a few peers. While a push based approach is suitable for a static P2P network, the following limitations arise in dynamic networks:

1) Not all the peers in the network may receive the broadcast messages. There are two scenarios when this can happen: one is if the network is partitioned; the other is if a peer is beyond the reach of the specified TTL limit.
2) Peers in the Gnutella network can join and leave the network dynamically. After a peer leaves the network, it won't be able to receive any further *invalidation* messages. Upon a subsequent rejoin, the peer will share a stale copy of the file.

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

B. 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 peers. Implementing a pull-based consistency technique in Gnutella is no different from implementing it in a client-server system such as the Web. In this approach, a peer polls the owner to determine if a replica is stale. A peer can employ different policies to determine when and how frequently to poll the owner to check for consistency: we outline three such policies in this section. 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 file currently cached at the peer. The last modification time of the file can also be used to determine this information instead of explicit version numbers.
2) *Owner IP address:* This allows a peer to locate the owner of a file.
3) *Consistency status:* The consistency status of a file 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 file could possibly be stale but the peer is unable to determine the actual status since the owner peer is unavailable (i.e., has left the P2P network).

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

1) Poll Every Time: In this policy, the peer polls the owner every time a query or a download request for the file is received. This is a lazy polling policy since a poll is triggered only when necessary. The advantage of the technique is its simplicity and strong consistency guarantee—since the peer checks the consistency status prior to responding to query messages, no stale information is returned. The drawback though is that the policy adds a round trip delay to each query or download message (unless the peer is itself the owner), which in turn degrades query response times. We note since file downloads are via a HTTP, a poll can simply be implemented by an *if-modified-since* HTTP message to the owner, which allows the peer to check if the replica was modified since the last poll.

2) Periodic Polls: An alternate approach is to poll the owner periodically to check if the file has been updated. The periodicity of the polls is statically determined by the owner and depends on how frequently the file is expected to be modified. The approach provides weaker guarantees than the poll-every time—if the file is updated between two polls, the peer will serve stale data until its next poll. Hence, the success of such a pull-based approach hinges on the accurate estimation of the polling frequency. Unless carefully determined, the technique may result in an excessive poll overhead (if the polling frequency is over-estimated and file does not change frequently) or may result in stale data (if the polling frequency is under-estimated and the file is modified frequently).

3) Adaptive Polls: Rather than determining the poll frequency statically, a third approach is to dynamically vary the polling frequency based on the update rate for the file. A peer can observe that rate of updates to a file and poll more frequently when the file 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 file. A history of modification times can maintained and used to determine the update rate to the file. Instead of 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 frequency of polls is reduced if the file was not modified since the last poll and made more aggressive if the file was modified. The notion of adaptive polling has been explored in the context of web cache consistency [6], [7] and we use a similar idea here.

A time-to-refresh (TTR) value is associated with each file. The TTR denotes the next time instant the peer must poll the owner, and thus, determines the polling frequency. The TTR value is varied dynamically based on the results of each poll message. The TTR value is increased by an additive amount if the file doesn't change between successive polls. Increasing the TTR value results in a reduction in the polling frequency. In the event the file is updated since the last poll, the TTR value is reduced by a multiplicative factor. In essence, an additive increase multiplicative decrease (AIMD) algorithm 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.

To precisely define this technique, if a file has not changed between two polls, we set

$$TTR = TTR_{old} + C \tag{1}$$

where C, $C > 0$, is an additive constant. If the file was modified, then

$$TTR = TTR_{old}/D \tag{2}$$

where D, $D > 1$ 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,

$$TTR = \max(TTR_{min}, \min(TTR_{max}, TTR)) \tag{3}$$

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

1) It provides 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 technique can handle dynamic joins and leaves. Upon rejoining the network, the peer simply resets the TTRs of all cached files to TTR_{min}. This enables the peer to poll each owner quickly to determine the consistency information.

C. Hybrid Push and Adaptive Pull Technique

A push-based technique can provide good consistency guarantees for peers that are online 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—owners broadcast invalidations upon each update. In addition, the hybrid technique requires peers to occasionally poll the owner to check if the file was updated. Ideally, only those peers who are unable to receive invalidation messages should poll the owners of files. 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. In either case, a poll at a subsequent time allows the peer to refresh a file 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 reach of the owner. We make the following modifications to the adaptive pull technique:

In addition to adapting the TTR to the update rate, we take into account the number of active neighbors of a peer 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. 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 (most P2P systems use $N_{avgconn}$ 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.

Thus, after computing the TTR in Equation 2, the TTR is further tuned as

$$TTR = TTR + \left(1 + \frac{N_{conn} - N_{avgconn}}{N_{avgconn}}\right) \times \alpha \qquad (4)$$

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

The TTR value can also be updated upon receiving a push-based invalidation message. Since an invalidate message is an indicator of an update, the peer can mark the stored copy as stale and decrease the TTR based on Equation 2. This TTR is used for future polls if the file is subsequently refreshed by the user.

By combining push and poll, the approach is able to provide good consistency guarantees to reachable peers, while accommodating the needs of distant peers via pull. Further, our modified TTR computation technique can adapt the TTR value to network dynamics and poll more frequently in the presence of frequent joins and leaves. Last, since the modified adaptive pull adds a modest amount of communication overhead, the overall overheads are not significantly greater than a pure push approach.

D. Discussion

Our consistency mechanisms suffer from one drawback. Recall that our mechanisms associate an owner with each file and store the IP address of the owner with the file. However, peers connecting over dial-up connections use dynamic IP addresses assigned by DHCP. The IP address of such peers can potentially change every time they rejoin a network, making it harder to identify owners by their IP addresses. This is problematic, since the IP address of the owner is necessary to poll for changes in the adaptive pull and the hybrid approaches. A possible solution to this problem is to restrict owner responsibilities to those peers with static IP addresses (e.g., peers connected over cable modems or T1 lines that have static IP addresses or long-lived DHCP leases). In such a scenario, upon creating a new file, the user must search for a peer with a static IP connection and hand over owner responsibilities to that peer (thus, the creator is no longer responsible for consistency maintenance). It is easy to determine which peers have static IP addresses and which ones don't—the user needs to specify the type of the connection (dial-up, broadband, T1 etc) in the configuration options while installing a P2P client. This information can be used to infer the type of IP address used by the peer.

IV. Prototype Implementation

We have implemented our hybrid push-pull algorithm into Gtk-Gnutella ver 0.17, an open-source implementation of the Gnutella protocol. To do so, 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. Our implementation assigns a type code of 0x90 for invalidation messages; an even type code indicates a broadcast message in Gnutella. Each invalidation message contains a 16 byte file identifier (an MD-5 [14] hash of the file name), the file 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 file if present in its local cache, decrements the TTL, and forwards the message to its neighbors.

Gtk-Gnutella uses HTTP to download files 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 files. 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. The message format of an IMS message and a response is described in Figures 1 and 2, respectively. In addition to these modifications, the response to a HTTP file download also needs to be enhanced—the owner's IP address and the port number need to be included with each file download. This extension is described in Figure 3.

```
HEAD /head/File Index/File Name/ HTTP/1.0
If-Modified-Since: GMT Time
Connection: Keep-Alive
User-Agent: consistency-gnutella/Version.SubVersion
File-Identifier: File ID
```

Fig. 1. HTTP Format of Poll Request

```
Response when the file has been modified:
  HTTP/1.0 200 OK
  Server: consistency-gnutella/Version.SubVersion
  Content-type: application/binary
  Content-length: File Length
  Last-Modified: Last Modified Time
  File-Identifier: File ID

Response when the file has not been modified:
  HTTP/1.0 304 Not Modified Since Last Modified Time
  Server: consistency-gnutella/Version.SubVersion
  File-Identifier: File ID
```

Fig. 2. HTTP Response of Poll Request

```
HTTP/1.0 200 OK
Server: consistency-gnutella/Version.SubVersion
Content-type: application/binary
Content-length: File Length
Last-Modified: Last Modified Time
Origin-Server-IP: IP Address of origin-peer
Origin-Server-Port: Port of origin-peer
File-Identifier: File ID
```

Fig. 3. HTTP Response of Download Request

Out prototype sets the TTR value of a file to TTR_{min} when it changes from *Stale* or *Possible Stale* to *Valid* or if the file is newly downloaded. In order to be backward compatible with current Gnutella protocol, we always set the status of the files 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.

V. Experimental Evaluation

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

A. Experimental Methodology

1) Simulation Environment: We have designed an event-based simulator to evaluate our cache consistency techniques. Our simulator simulates an overlay network of Gnutella peers. The topology of the overlay and various network and P2P system parameters are initialized using using observed statistics. We borrow heavily from recent measurements studies [8], [9], [10] to initialize parameters such as link bandwidths, network diameter, node connectivity, session times, file popularities, etc. We will show in a subsequent section that the consistency guarantees are insensitive to several of these parameters. Each peer in our simulator is responsible for answering queries, propagating query messages to neighbors and for servicing file 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 file downloads. Our simulator also incorporates an update process that generates to files stored at owners. An update causes the last modification time and the version number of the file to be updated at the owner. The default values of various parameters used in our simulations is listed in Table II.

Parameter	**Description**	**Default Value**
L_{sim}	Length of simulation	10 hours
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 seconds
D_f	Average offline duration	2 hours
$I_{topochk}$	Average time between successive topology checks	5 minutes

TABLE II
PARAMETERS FOR THE DYNAMIC NETWORK ENVIRONMENT

The workload for generating queries, file downloads and updates to files is generated synthetically. While measurements of query rates and file downloads are available from recent studies, realistic distributions of file update rates are not available since current P2P systems only share static files. Consequently, we use update distributions of web pages [1], [2] to be a reasonable indicator of updates in P2P environments. Specifically, we assume four types of files: highly mutable, very mutable, mutable, and immutable. Each category has a different mean update rates. The percentage of the files in each category and the mean update rate in each category is (0.5%, 15 sec), (2.5%, 7.5 min), (7%, 30 min), and (90%, 1 day). Note that the mean lifetime of a immutable file is longer than our simulation duration of 10 hours.

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 files that return stale request (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 files. A query that returns some stale files 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 files that are stale. The false valid ratio for queries and downloads should be as close to zero as possible.
- *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 simply the number of invalidations that are broadcast. In the pull approach, the control message overhead is defined to be the number of poll messages (IMS HTTP messages). We note that, while flooding of invalidations 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 P2P 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.

3) Network Environment Used in the Simulations: Our simulations are conducted for two different network environments as described below:

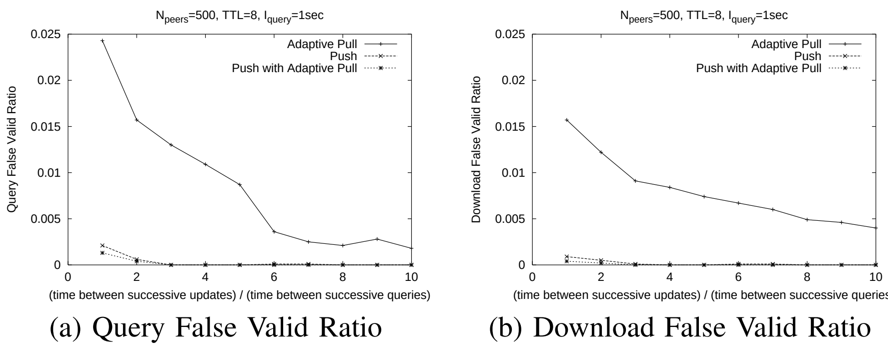
- The first set of experiments are performed in a stable P2P network environment. All the peers remain online throughout the simulation, and there are no failures of any type. Although this assumption is unrealistic for real P2P networks, it gives us an indication of how our techniques will perform in the ideal case.
- The second set of experiments are performed in a dynamically changing P2P network. We assume peers leave 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 Gnutella network 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 network consisting of 500 peers and 5000 objects. While a 500 peer network is small for many P2P studies, it is adequate for studying consistency techniques.

B. Simulation Results from a Stable P2P Network

All the experiments described in this section were conducted with F_{enable} set to FALSE. All other parameters are set to their defaults values.

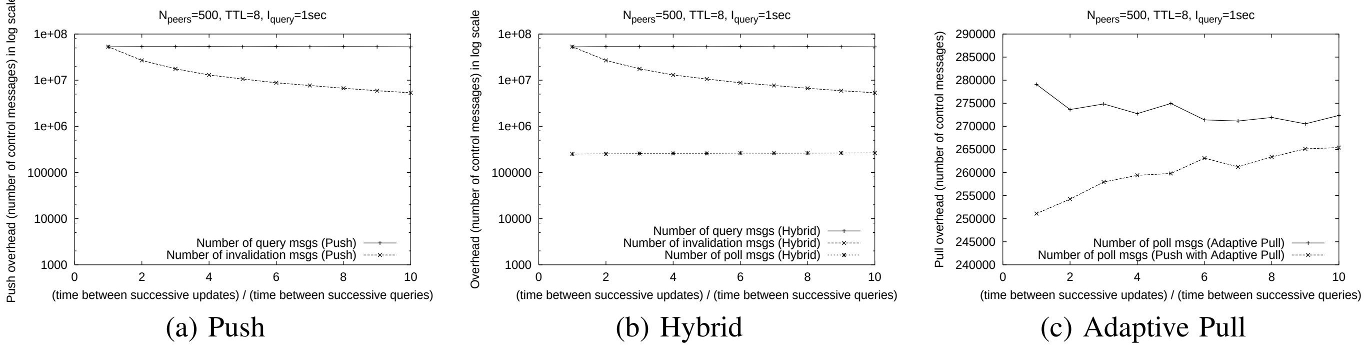
To evaluate the efficacy of different cache consistency mechanisms presented in Section 3 for a stable network environment, we conduct several experiments by varying the interval between successive updates $I_{updates}$, the time between successive query requests, I_{query}, the TTL values, the network size and the average number of network connections of each peer. We discuss each experiment in turn.

1) Impact of the Update Rate: First, we analyze the false valid ratio of different cache consistency mechanisms by varying the update rate $I_{updates}$. We fix the query inter-arrival time I_{query} to 1 second, and vary I_{update} from 1 sec to 10 sec. We plot the query false valid ratio as well as the download false valid ratio for push, adaptive pull and the hybrid push-pull approaches in Figure 4. We note that *Push* can achieve near-perfect FVRs in a stable network environment. Thus, push alone is sufficient when the network is stable and updates are less frequent than queries. In contrast, adaptive pull yields weaker consistency guarantees. For frequent updates, about 2.5% of the query responses are stale and 1.5% of the downloaded files are stale. The FVR falls as the updates become less frequent. The control message overhead for these experiments is shown in Figure 5. The figures show that the push-based invalidations



(a) Query False Valid Ratio (b) Download False Valid Ratio

Fig. 4. Impact of Update Rate on False Valid Ratio



(a) Push (b) Hybrid (c) Adaptive Pull

Fig. 5. Impact of Update Rate on Control Message Overhead

impose two orders of magnitude larger overhead when compared to pull. Due to the relatively small overheads of pull when compared to push, the total overhead of the hybrid technique is comparable to push. Additional experiments quantifying these overheads may be found in [11]. Based on these results, we recommend pull only if the consistency requirements are less stringent or if the control message overhead is a major consideration, and only for scenarios where updates are less frequent than queries.

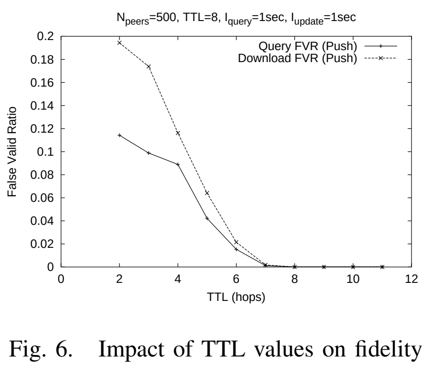


Fig. 6. Impact of TTL values on fidelity

2) Impact of TTL values: Since TTL values determine the reach of each invalidation broadcast, the query/download FVR will decrease for larger TTL values. To quantify the effect on fidelity, we varied the TTL from 2 to 12 hops and measure the QFVR and DFVR for the push approach. Figure 6 plots the resulting fidelity. As shown, both QFVR and DFVR fall to zero beyond a TTL of 8, indicating that a TTL of 8 hops is sufficient to reach most peers in a 500 node P2P network.

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 2000 nodes. The TTL for invalidations is set to 6 hops. Figure 7 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 invalidate 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.

4) Impact of Average Number of Active Connections: In a Gnutella-style network, the average number of neighbors of a peer ($N_{avgconn}$) is an important factor on how well the peers are connected. To evaluate this metric, we vary the average connectivity from 1 to 5 logical links per peer and measure its impact on the fidelity of push

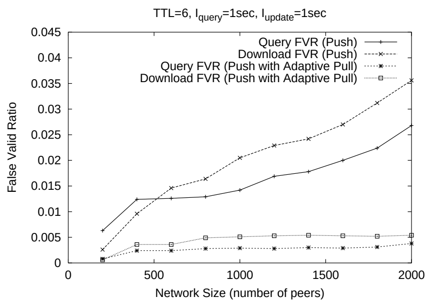


Fig. 7. Impact of network size on fidelity

and the hybrid push-pull approach. Figure 8 depicts our results.

From Figure 8, we observe that both the query and download FVR decrease to zero as $N_{avgconn}$ increases beyond 4. Fewer neighbors indicates a larger diameter for the network; hence, push becomes less effective when a peer has fewer neighbors, since invalidates do not reach some peers. The fidelity improves as the network connectivity is increased. Like before, this parameter has less impact on the hybrid push-pull approach. The approach provides 5 to 10 times better fidelity than push when $N_{avgconn}$ is small. Again, this is because distant peers can resort to polling when invalidates are ineffective. We note that the default minimum value of $N_{avgconn}$ in Gtk-gnutella is 4 peers, and our result shows that this is sufficient for providing good consistency guarantees in moderate-size P2P networks.

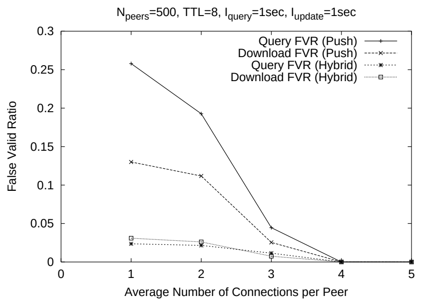


Fig. 8. Impact of the average number of connections per peer on fidelity

5) Insensitivity to the Query Rate: Our final experiment studies the impact of the query rate on the fidelity. A higher query rate will result in more downloads and replication of content. Greater replication can have an impact on fidelity, especially if the replicated content is stored at distant peers. We fix the update rate I_{update} to one update per second and vary the query rate from one query per second to one query every 10 seconds. The resulting FVRs for the hybrid approach is shown in Figure 9. The figure suggests that the query rate does not have an impact on the fidelity. Thus, the fidelity depends on the topology of the network, and for a fixed topology, is not sensitive to the query rate or the amount of replication.

In the remaining experiments, we will fix the I_{update} to 2 seconds and I_{query} to 1 second.

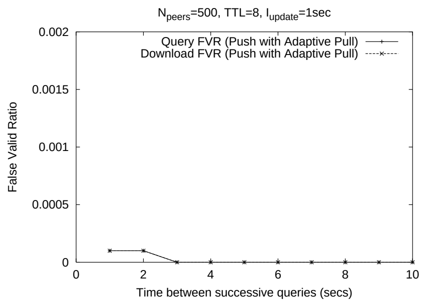


Fig. 9. Impact of query rate on fidelity

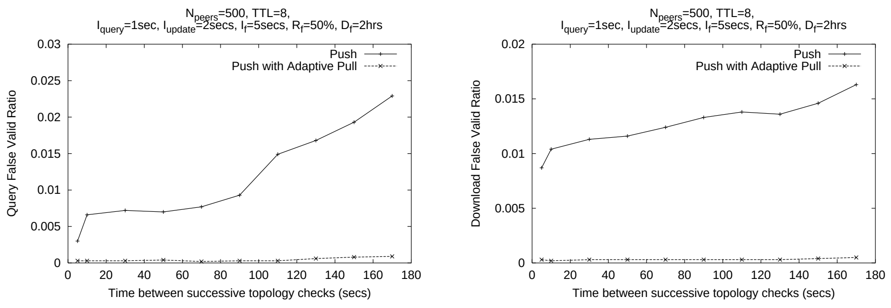
C. Simulation Results for a Dynamic P2P Network

Our experiments for a dynamic network are run with F_{enable} set to TRUE. 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.

When a peer leaves the Gnutella network, it tears down all connections to its neighbors. This causes each of its neighbor to lose on 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 [12], [5], [13] 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 each peer and constructs new logical links if a peer has fewer neighbors than a threshold.

1) Impact of Interval Between Successive Topology Checks: In this section, we study the effects of topology checking process by varying the $I_{topochk}$ values from 5 seconds to 170 seconds. This parameter effectively determines the delay between a broken connection and



(a) Query False Valid Ratio (b) Download False Valid Ratio

Fig. 10. Impact of time between successive topology checks on False Valid Ratio

the instant when a new connection is formed by a peer. Observe that this parameter only impacts push, since an earlier experiment showed that the connectivity of the overlay does not impact pull. Hence, we focus on the push and the hybrid push-pull approach. Figure 10 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.

2) Impact of the Dynamics of the Gnutella Network: To understand the effects of the dynamics of the Gnutella network, we first vary the fraction of offline peers R_f from 5% to 50% and measure the impact on the QFVR and DFVR. As shown in Figure 11(a) and (b), push can provide better consistency guarantees than a pure pull approach, even in a dynamic network. The FVRs degrade as the fraction of offline nodes 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 nodes is as high as 50%.

Next, we vary the time between successive disconnections I_f. Intuitively, as I_f increases, fewer nodes leave the network. The results are similar to the previous scenario (see Figure12). Push outperforms a pure-pull approach; both techniques yield better consistency guarantees in more stable networks. Like 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 P2P networks and can provide good fidelity at a cost that is comparable to a pure push approach.

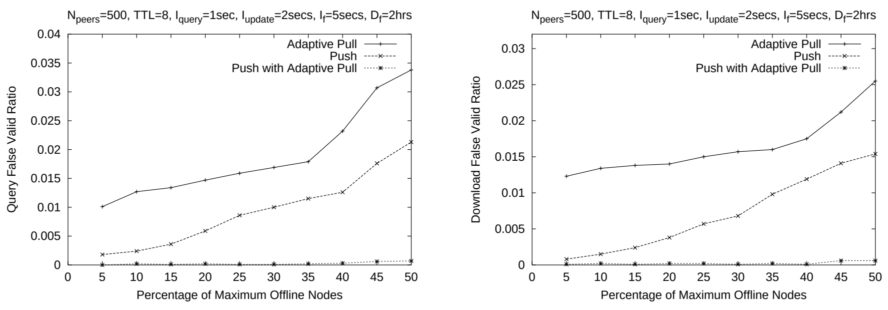
D. Results from Prototype Implementation

In this section, we study the implementation overheads of various operations need for consistency maintenance in the hybrid push-pull approach. The testbed of our experiments consists of 18 peers, all of which run on a cluster of six Linux PCs. Five PCs in our experiment are 1.5GHZ Pentinum IV with 256MB RAM, the other PC is a 933MHZ Pentinum III with 512MB RAM, all of them are interconnected by 100Mb/s switched ethernet. Each PC runs 3 peers. Each peer shares 50 files to all the other peers. We assume 90% of files have a lifetime of 1 day, while the other 10% files have a mean lifetime of 5 minutes. Initially, each peer caches 10% of the files from other peers. Since our focus is to study consistency maintenance overheads, we do not inject any query or download messages. Thus, the system only sees invalidations and poll messages from peers for consistency maintenance. We measure the overhead of various cache consistency operations at each peer over a 24 hour duration. Table III lists our results. As shown in the table, the overhead of incoming poll processing, incoming invalidation processing and outgoing invalidation processing is very small (in the order of hundreds of microseconds). Similarly outgoing poll can be processed efficiently (clearly this overhead depends on the round trip time of the route to the owner.

These results indicate that our techniques can be implemented efficiently in a *Gnutella* P2P file sharing network.

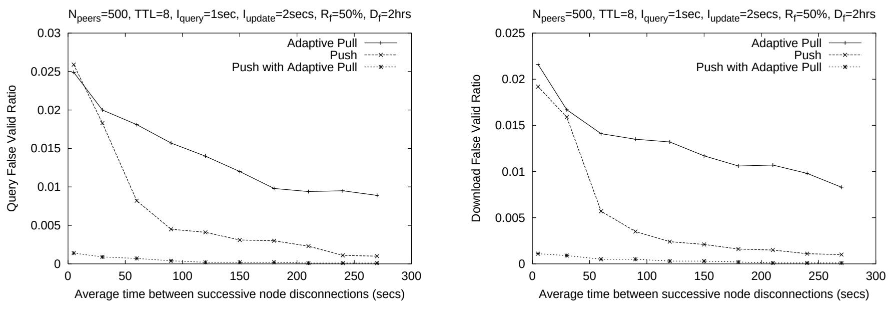
Event	**Time(μs)**
Incoming Poll Processing	181.827
Outgoing Poll Processing	1356.170(including the network transmission delay)
Incoming Invalidation Processing	150.242
Outgoing Invalidation Processing	265.341

TABLE III
PROTOTYPE IMPLEMENTATION OVERHEADS



(a) Query False Valid Ratio (b) Download False Valid Ratio

Fig. 11. Impact of maximum offline nodes on False Valid Ratio



(a) Query False Valid Ratio (b) Download False Valid Ratio

Fig. 12. Impact of time time between successive node disconnections on False Valid Ratio

VI. Concluding Remarks

While current of peer-to-peer systems share predominantly static files, we argued that future peer-to-peer networks will support sharing of files that are modified frequently by their users. We presented techniques to maintain temporal consistency of replicated files in a peer-to-peer network. We considered Gnutella and presented techniques for maintaining consistency in Gnutella even when peers containing replicated files dynamically join and leave the network. We presented extensions to the Gnutella protocol to incorporate our consistency techniques and implemented them 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 P2P network, (ii) a hybrid approach based on push and pull achieves high fidelity in highly dynamic P2P networks and (iii) the run-time overheads of our techniques are small, making them a practical choice for P2P networks.

References

[1] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of the IEEE Infocom'00, Tel Aviv, Israel*, March, 2000.

[2] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the USENIX Symposium on Internet Technologies (USEITS'99), Boulder, CO*, October 1999.

[3] J Chu, K Labonte, and B Neil Levine, Availability and Locality Measurements of Peer-to-Peer File Systems, *Proc. SPIE ITCom: Scalability and Traffic Control in IP Networks II Conference*, July 2002.

[4] The Gnutella Protocol Specification v0.4, Clips2 Distributed Search Solutions, http://dss.clip2.com.

[5] The KaZaA website, http://www.kazaa.com/

[6] R. Srinivasan, C. Liang, and Krithi Ramamritham, Maintaining Temporal Coherency of Virtual Data Warehouses, *The 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 2-4, 1998

[7] B. Urgaonkar, A. Ninan, M. Raunak, P. Shenoy and K. Ramamritham, Maintaining Mutual Consistency for Cached Web Objects, In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, AZ, April 2001

[8] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[9] S. Saroiu, P. Gummadi, and S. Gribble, A measurement study of peerto -peer file sharing systems, In *Proceedings of Multimedia Computing and Networking 2002*, January, 2002.

[10] K. Sripanidkulchai, The popularity of Gnutella queries and its implications on scalability, Technical Report, February 2001.

[11] Jiang Lan, Cache Consistency Techniques for Peer-to-Peer File Sharing Networks, M.S. thesis, University of Massachusetts, June 2002.

[12] Limewire web site, http://www.limewire.com/

[13] GTK-gnutella web site, http://gtk-gnutella.sourceforge.net/

[14] R. Rivest, The MD5 Message-Digest Algorithm, RFC 1321, MIT LCS & RSA Data Security, Inc., April 1992.