MASTERS PROJECT REPORT
Cache Consistency Techniques for Peer-to-Peer File Sharing Networks

Jiang Lan
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
jiang@cs.umass.edu

Advised by Prashant Shenoy and Brian Levine

June 26, 2002

1 Introduction

1.1 Cache Consistency Problems in Peer-to-Peer Networks

Peer-to-peer (P2P) file sharing systems provide infrastructure for communities to share storage space (e.g., Napster, Gnutella [1], Kazza [2]). Unlike traditional distributed systems, P2P networks aim to aggregate large numbers of computers that join and leave the network frequently and might not have permanent network (IP) addresses. In a pure distributed P2P system such as Gnutella, peers communicate directly with each other and share resources without dedicated servers. In essense, these P2P systems build, at the application level, a virtual overlay network with it own routing mechanisms.

Popular usage of P2P applications has focused simple sharing of music or video files. However, P2P file sharing systems are not limited to this task. Because P2P networks are application-level networks, new functionalities can be easily added and deployed. For example, the Gnutella protocol was designed to share any type of file, and new features have been continuously added or proposed to the existing model. These observations have motivated us to consider cache consistency issues in a P2P system, like Gnutella.

Using the web model as a comparison, we assume that each file in a Gnutella network has an *origin server*. And correspondingly, the file at its origin server is called its *master copy*. Any user in the network can download any file she finds through querying, and thus, files are replicated by user initiatives. We consider all the non-master-copies of a file as *replica* of the master copy, and assume that only the master copy of a file can be modified. Given this terminology, maintaining cache consistency in the Gnutella network means ensuring that when the master copy of a file changes, the peers that cache the file should be made aware of it. This has two consequences: firstly, local access of a cached copy should only be allowed if the copy is consistent with the master copy (not stale); and secondly, peers send back query result messages only if the matching replica file is consistent with the master copy.

1.2 Research Contributions of this Report

This report presents the results of our investigation of several cache consistency techniques, broadly divided into *push-based* or *pull-based* methods. In addition, we propose using a combination of push and pull (PAP), which our evaluations show improved consistency of replicas in the network. More specifically, we focused on the following techniques:

1. *Pull with Adaptive TTR*, our first algorithm, is presented in Section 4.1.2. The main idea is to poll the origin server of the replicas whenever the TTRs expire. TTR refers to Time-to-Expire, it is an estimation of the duration during which the file is less likely to change. We present an adaptive TTR algorithm where the clients calculate TTR values based on the polling history. We refer to this algorithm as *Adaptive Pull*.

2. *Push with Invalidation Messages*, our second algorithm, is presented in Section 4.2. Whenever a file gets modified at its origin server, we construct an *invalidation* message and broadcast it into the P2P network. Peers that replicated the same file check their local replicas upon receiving the invalidation message, and change their local copies' status to *stale* if they are older than the master copy. We refer to this algorithm as *Push*.

3. *Push with Adaptive Pull*, our third algorithm, is presented in Section 4.3. It simulaneously employs both *Push* and *Adaptive Pull* algorithms to provide replica consistency. The push part of the algorithm is exactly the same as the *Push* algorithm. However, the pull part of this algorithm is slightly modified from the *Adaptive Pull* algorithm such that it takes into account a peer's received invalidation messages and number of active connections to reduce the polling overhead. We refer to this algorithm as *Push with Adaptive Pull*.

We also described briefly another variant of the poll algorithm, *Pull with Static TTR*, in Section 4.1.2. However, we don't emphasize much on it because it's less flexible in a dynamically changing network environment, like Gnutella.

In order to evaluate the effectiveness and overhead of each technique, we performed a detailed simulation study. We performed simulation experiments for two types of network environments: one is a stable network where all the peers remain in the network throughout the simulation; the other is a dynamic network where peers join and leave frequently. The reason we are interested in both is because there are different P2P applications suitable for different environments, and the cache consistency techniques we studied can be applied to different situations. We derive the following conclusions from the simulation results.

For a stable network environment:

1. *Push* alone achieves almost perfect fidelity when the update rate is lower than query rate. *Push with Adaptive Pull* achieves only 40% better fidelity than pure *Push* when the update rate and query rate are comparable. With the assumption that there are normally much less updates than queries in a Gnutella-style network, we expect *Push* alone to be sufficient in guaranteeing strong consistency.

2. *Adaptive Pull* alone does not guarantee strong consistency. As updates rate decreases by 10 times, *Adaptive Pull* gives 67% to 92% better consistency. The poll message overhead is relatively consistent when the update rate changes and is in general at least one magnitude lower than the invalidation message overhead when using *Push* or *Push with Adaptive Pull*. In applications where the consistency requirements are less stringent, the *Adaptive Pull* algorithm is a good choice.

3. TTL value determines the reach of invalidation messages. The fidelity of replicas decreases as TTL value increases, when the TTL value gets large enough, both *Push* and *push with adaptive pull* achieve almost perfect fidelity. The invalidation message overhead increases almost exponentially as TTL value increases. However, it reaches a plateau after the TTL gets large enough (8 in our simulation). In common Gnutella implementations [12] [13], TTL is set to 7 and it allows the broadcast messages to cover approximately 95% [8] of all the peers in the Gnutella network.

4. Network size affects fidelity when using the *Push* algorithm. We get 5 to 10 times worse fidelity as the network size increases by 10 times. However, when using the *Push with Adaptive Pull* algorithm, we only get 3 to 5 times worse fidelity as the network size increases by 10 times. And more importantly, the fidelity achieved with pure *Push* is 3 to 7 times worse than that with *Push with Adaptive Pull*. We attribute these to the leverage effect of the pulling part of the *Push with Adaptive Pull* algorithm.

5. The average number of active connections each peer maintains is another important parameter in a Gnutella-style network. The larger this value, the more peers each invalidation message gets broadcasted to. The simulation results show that in both *Push* and *Push with Adaptive Pull* algorithms, we get better fidelity as this value increases. We achieve almost perfect fidelity when this value gets large enough (4 in our simulation). However, the *Push with Adaptive Pull* algorithm is more immune to this parameter, for example, it achieves 77% to 88% better fidelity than *Push* when the average

number of connections per peer is 1. On the other hand, the invalidation message overhead involved increases exponentially, by approximately 3 magnitudes when this parameter increases from 1 to 4. Therefore in practive, we suggest peers to maintain a moderate number of active connections that helps to achieve good fidelity and at the same time, avoids large overhead.

6. Query rate does not have much effects on fidelity when using the *Push with Adaptive Pull* algorithm. We achieved almost perfect fidelity even when the update rate was 10 times of the query rate.

For a dynamic network environment:

1. A Gnutella application fixes its number of active connections when it falls below some threshhold. This helps maintaining the connectivity of the Gnutella network and thus helps *Push* to achieve better fidelity. However, using *Push with Adaptive Pull* achieves $> 90\%$ better consistency than using pure *Push*. The topology fix process isn't necessary when using the *Push with Adaptive Pull* algorithm.

2. Both the *Adaptive Pull* and *Push* algorithms achieve worse fidelity when more disconnections occur in the network. *Push* gives 25% to 80% better fidelity than *Adaptive Pull* when there are moderate disconnections, and possibly worse fidelity when the network gets highly disconnected. In general, neither of the two algorithms can guarantee strong consistency in a dynamically changing network.

3. The *Push with Adaptive Pull* algorithm gives good fidelity (with QFVR and DFVR both below 0.002) even when the network gets highly disconnected. So if strong consistency is desired, this is the best algorithm among the three.

4. The invalidation message overhead depends on the update rate. If the update rate is low relative to the query rate, then the invalidation message overhead is also lower than the query message overhead. Both the query and invalidation message overheads decrease linearly as more disconnections occur in the network.

5. The poll message overhead is generally at least one magnitude lower than the invalidation message overhead. Our adaptive TTR algorithm adapts to the changes in the number of active connections of each peer, therefore a peer polls more often when it has fewer connections. This adaptiveness helps the polling to achieve good fidelity even in a highly disconnected network environment.

The remainder of this report is structured as follows: the next section briefly describes Gnutella protocol and application. Section 3 introduces the traditional cache consistency techniques for distributed (especially client-server) systems. In Section 4 we focus our attention on applying the traditional techniques to the

Gnutella P2P model, and propose our extensions to those algorithms. Then in Section 5, we descibe our simulation study and the results of the simulation experiments. We offer conclusions in Section 6.

2 Overview of Gnutella

Gnutella [1] is a fully decentralized Peer-to-Peer file sharing protocol. In essence, Gnutella is a routing protocol where messages are distributed by means of application-level flooding. Each node in a Gnutella network only knows its neighbor *nodes* (also called *peers*). When a node A sends out a query message, it first routes the query message to its neighbor peers, the neighbor peers then pass on this query message to their neighbors, and so on. This process stops when the hop count of the message reaches the specifed maximum, defined as the *Time-to-Live (TTL)* value. Nodes that can satisfy the query will pass results back to A following the reverse path of the query message. A peer forward new query messages to all its neighbor peers no matter it can satisfy the queries or not.

Each Gnutella node performs tasks normally assiociated with both servers and clients. As a client, they provide interfaces through which users can issue queries and view search results; as a server, they accept queries form other servents, check for matches against their local store, and respond with corresponding results. A node joins the Gnutella network by initially connecting to one of several known hosts that are intended to be always available (e.g., gnutellahosts.com). Once connected to the network, nodes send group membership messages (PING and PONG) to interact with each other. Users can then send queries and view search results through the client interface. Once a servent receives the search results, it may initiate direct download of one of the files in the result set. File download is through the HTTP protocol, and is referred to as “out-of-network” in the Gnutella protocol [1].

In the next section, we introduce some traditional cache consistency techniques for distributed systems. Most of them have been extensively studied and successfully deployed in client-server type systems, such as the Web.

3 Traditional Cache Consistency Techniques for Distributed Systems

Generally there are two broad approaches to achieve cache consistency in a distributed system. One is the client-initiated approach (*client-poll*, for short); the other is the server-initiated approach (*server-push*, for short). Most of the studies of these techniques have been confined to the client-server model. We use the general terms *client* and *server* to indicate the consumer and provider of resources, and use *file* to represent resourses that are hosted by servers. In a web environment, a server could be any web server, a client could be a user operating at a local machine with a web browser.

3.1 Client-initiated Algorithms

3.1.1 Poll-Every-Time

The simplest technique is *poll-every-time*. Each time a cached file is accessed by a client, an “if-modified-since” message is sent to the server that hosts the file. Upon receiving this message, the server will check if its local copy of the specified file is newer than the cached copy at the client, then send a reply indicating the result. If the two copies are the same, the client will proceed accessing the file, else it will fetch the newer copy from the server before further accessing.

The advantage of this approach is its simplicity and strong consistency (ignoring the inconsistency caused by network delay) guarantee. However polling every time will incur large amount of network traffic and vastly increase server load. In addition, the client response time includes an additional round-trip delay even when the local copy is the same as the server’s copy.

3.1.2 Poll with “Time-To-Refresh”

In the *time-to-refresh (TTR)* approach [3] [4], a client assigns a TTR value to every file it downloads from the server. This TTR value is an estimation of the duration during which the file is less likely change. The client will then assume the file remains valid (i.e., consistent with the master copy) in its cache for the period of TTR. Any request for the file before the expiration of its TTR will be served by the client’s local cache. Upon the expiration of the TTR, the client needs to poll the server to check the validity of the file, and updates its local cache copy and its TTR value basing on the server’s feedback.

The purpose of using TTR is to reduce network traffic and server load. Larger TTR results in less network traffic and lighter server load. However, larger TTR also increases the degree of inconsistency. If measured in time, the worst case inconsistency period is now TTR as opposed to strong consistency in the poll-every-time approach.

Ideally, perfect TTR values could be set if updates of objects are predictable. However, this is hardly true in real life. In order to achieve both the goals of reducing overhead and maintaining a high degree of consistency, TTR values must be chosen intelligently. Several techniques have been proposed by Sirnivasan et al. for choosing good TTR values [3], as described below:

1. *Static TTR*, based on priori assumptions. The simplest approach is to choose a low TTR value based on some heuristics and use it throughout. The difficulty of this approach is that TTR values are hard to predict and usually fluctuate over time. Thus this technique works well only for applications

where updates to files are predicable and more or less follow a uniform distribution, such as online newspapers.

2. *Adaptive TTR*. In adaptive TTR, a client can compute the TTR values based on the rate of observed changes at the server. Rapidly changing files result in smaller TTR, whereas infrequent changes require less frequent polls to the server, and hence, a larger TTR. Sirnivasan et al. summarize a set of techniques for computing the TTR values [3]. We present a variation of it for usage in the Gnutella style P2P system in Section 4.

   The main problem of adaptive TTR is that using past to predict future is not always accurate, thus it can't guarantee strong consistency. However, if some degree of inconsistency can be tolerated, this technique will suffice.

3.2 Server-initiated Algorithms

There are two basic approaches in this category: the first one is referred to as *server-based invalidation*, which requires the server to notify clients with invalidation messages when a file gets modified; the second approach is to let the server send the actual changes to the clients instead of invalidation messages at the time of updates. Both approaches are optimal in the number of control messages exchanged between the server and client (since messages are sent only when files get modified). However, the invalidation based approach saves more bandwidth if changes are large. It is also more reasonable for clients that aren't interested in the changes, in this case it costs less overhead than sending the actual updates, and still achieves strong consistency. An intermediate approach is to send invalidation messages when changes are large, and send the actual changes when they are small, but this adds more complexity into the system. For simplicity, we will only discuss the invalidation based approach in this report.

In the traditional push-based approach (which, for example, can be implemented in a distributed file system), the server must maintain per-file state consisting of a list of all clients that cache the file. In the case of popular files in a large distributed system, the amount of state maintained can be significant. In addition, if a client is unreachable due to network partition or other reasons, the server must either block on a write request until a timeout occurs, or risk violating consistency guarantees. These limitations are the main reasons that server-based invalidation is not adopted by the current Internet (HTTP servers are stateless). To overcome this limitation, Gary and Cheriton introduced an alternative approach call *lease* [5], which reduces server space overhead at the expense of more control message overhead. In lease-based approach, the server assigns a lease to each download request from a client. The lease duration indicates the interval of time during which the server will notify the client about any update of the file. Once a file's lease expires, the client needs to contact the server to renew the lease. Since the server only maintains for each file the list of clients with valid leases, shorter leases imply smaller server space overhead. Lease was first proposed in the context of cache consistency in distributed file systems [5]. Yin et al. [6] and Duvvuri, et. al. [7] have proposed techniques of using lease in the World Wide Web (WWW) environment.

The advantage of using leases is that it guarantees strong consistency with less server space overhead. The difficult part is how to choose a good lease value. Static lease values may inhibit systems from scaling. Adaptive lease algorithms have been proposed by Duvvuri, et. al. that address these problems [7].

In the next section, we investigate how to apply the traditional cache consistency techniques to the Gnutella network, and how well would they perform. We also propose variants of those algorithms that are promising to give better degree of consistency. The follow-up simulation study evaluates their performance in terms of the consistency level and control message overhead.

4 Cache Consistency Algorithms for the Gnutella Network

The Gnutella protocol is an open, decentralized group membership and search protocol. The term “Gnutella” also designates the virtual network of Internet accessible hosts running Gnutella-speaking applications, often referred to as the “Gnutella network”. The two main features of the Gnutella network are of a self-organizing topology and the use of flooding/back-propagation as its routing protocol. In theory, all the traditional cache consistency techniques described in the previous section could be implemented with some success in such an environment. However, the highly dynamic nature of the Gnutella network requires the algorithms to be fault tolerant and scalable. In addition, file downloads are user-initiated: a user may choose to download any copy among the search results, not necessarily the master copy. If we use the parent-child relationship to describe the two peers that uploads and downloads a file respectively, the distribution of a file then forms a tree-like hierarchy. And this tree structure is very unstable. All the above natures make it very hard to implement a *Lease* based technique in the Gnutella network, since any peer failure is hard to recover.

In the following sub-sections, we describe the advantages and disadvantages of applying the pull-based and push-based techniques in the Gnutella network, and we propose a new push and pull combined technique that is promising to give better degree of consistency.

4.1 Client-initiated Techniques

Implementing client-initiated techniques in a P2P network is no different than implementing it in a client-server style network. We require that the following information be stored with each file for this scheme to work correctly:

1. Version Number — the file version number is incremented upon each update.

2. Origin Server IP address — where to poll for the status of the master copy.

3. Consistency Status — we defined three values here:

   (a) *valid* — when a file is consistent with its master copy;

   (b) *stale* — when a file is older than its master copy;

   (c) *possible-stale* — when the origin server of a file is unavailable (e.g., left the network), we can't decide the definite consistency status of the file.

We maintain these information for each replica file. Each peer can maintain two local stores, one for the files it hosts (the master copies); the other for all the replica files.

4.1.1 Poll-Every-Time

A poll request is generated whenever there's a local access for a file, or there is a query for the file. The poll request includes the file name, the file's version number, and it is sent to the origin server of the file. The above file details are sufficient for a correct implementation of this algorithm.

The advantage of this technique is its simplicity and strong consistency guarantee. Its problem is every local access involves an extra round-trip delay (unless it is the master copy), and every query result is also delayed because of the polling. Because of the control message overhead and degradtion of performance, we consider this algorithm less appealing.

4.1.2 Poll with TTR

In order to implement polling with TTR, we need to maintain another piece of information for each downloaded file: the TTR value. The success of the pull-based technique hinges on the accurate estimation of the TTR value. There are various techniques to compute the TTR value, they can be broadly divided into static and adaptive TTR algorithm.

1. *Static TTR*. This is the simplest TTR algorithm. Basically it chooses a low TTR value (based on some heuristics) and use it throughout. It is easy to implement. The per-file state overhead is low since computing the TTR does not require any previous history. However, it is hard to choose a good TTR value that both gives a good consistency guarantee and has low control message overhead.

2. *Adaptive* (also called *Dynamic*) *TTR*. To overcome the problems of static TTR, Sirnivasan et al. [3] and Urgaonkar et al. [4] have developed adaptive TTR algorithms that compute TTR values based on

> the history of file updates and other heuristics. Sirnivasan et al. [3] summarizes a set of techniques for computing the TTR values. Urgaonkar et al. [4] proposed a *linear increase multiplicative decrease (LIMD)* algorithm to adapt the TTR value to the rate of change of the object. The *LIMD* algorithm gradually increases the TTR value as long as there are no updates at the master copy. On detection of updates, the TTR value is reduced by a multiplicative factor. In this manner it probes the server for the rate at which the object is changing and sets the TTR values accordingly.

We developed an algorithm similar to the LIMD algorithm. The precise details are described as follows.

For each object, the algorithm takes as input two parameters: TTR_{min} and TTR_{max}, which represents the lower and upper bounds on the TTR values. These bounds ensure that the TTR computed by the algorithm is neigher too large nor too small. TTR_{min} is typically set to the most-frequently-changed-objects minimal duration between successive updates. Although this value changes over time and systems, we can still use a statistically collected value as an estimate. The algorithm begins by initializing $TTR = TTR_{min}$. After each poll, the server will send back its version number of the file, the client then computes the next TTR value based on the following algorithm (we use VN to denote the file version number):

1. Each file maintains its most recent TTR value, TTR_{latest}.

2. The estimated TTR value for the next poll is calculated as follows:

   If $VN_{server} = VN_{client}$, then

   $$TTR_{estimate} = TTR_{latest} + C \tag{1}$$

   where C is a constant. Our assumption here is that if the file undergoes no change at the server during the previous TTR period, we should probably use a larger TTR for the next poll.

   If $VN_{server} > VN_{client}$,

   $$TTR_{estimate} = \frac{TTR_{latest}}{VN_{server} - VN_{client} + \alpha} \tag{2}$$

   where $\alpha > 0$ is a configurable parameter. It is necessary since when $VN_{server} - VN_{client} = 1$, we still want to reduce the next TTR value.

3. We compute a dynamic TTR value basing on both $TTR_{estimate}$ and TTR_{latest},

   $$TTR_{dyn} = w \times TTR_{estimate} + (1 - w) \times TTR_{latest} \tag{3}$$

where $0 < w < 1$ is a configurable parameter. We suggest using $w > 0.5$ to put more emphasis on the current poll result.

4. The final TTR value must be with the lower and upper bound,

$$TTR_{new} = Max(TTR_{min}, Min(TTR_{max}, TTR_{dyn})) \tag{4}$$

The *adaptive TTR* algorithm has the following features.

1. It provides several tunable parameters, namely C, α, and w that can be used to control its behavior.
2. Each cached file only need to store the last used TTR value and the file's version number in order to calculate the next TTR. This requires very small per-file state-space overhead.
3. Failure recover is easy. When a peer recovers, it can simply reset the TTRs of all cached objects to TTR_{min}.

4.2 Server-initiated Techniques — Broadcasting with Invalidation Messages

We have described previously that the main message routing protocol in a Gnutella network is flooding. Our push-style cache consistency technique utilizes this feature, and has the advantage of a stateless server.

The idea is simple: whenever a master copy (for example, of file A) gets updated at peer K, an *invalidation* message M is constructed and flooded into the network. Message M contains at least the following information: the origin server ID (e.g., IP address), the file name, and the new version number of the file. A peer receives M will check if it caches file A, if yes, it compares the local version number of A with that in the message, and invalidates its local copy if the message contains a newer version number. It passes on message M to its neighbors no matter if it contains the specifed file, just like the way it deals with *query* or *ping* messages.

The advantage of this push-style algorithm is its simplicity and stateless server. It guarantees strong consistency (ignoring inconsistencies caused by network delays) given that all the peers that caches the file are reachable from the origin server. However, it does add a lot of unnecessary control message overhead into the network, especially when an object is cached only at a few peers.

In order to implement this algorithm in the Gnutella network, the following changes need to be made to the protocol and its implementation:

- A new *descriptor*[1], *invalidation* descriptor, need to be added to the Gnutella protocol. This descriptor

[1] A descriptor is, in Gnutella terminology, a message with some fixed format.

contains the Origin Server ID, the File ID, and the new Version Number of the File. The origin server ID could simply be that node's IP address; the File ID must uniquely identify a file in the whole system, an example could be (Origin Server IP + Author Name + File Name); the new Version Number is necessary to identify if a cached copy is out of date.

- Whenever a file gets updated at its original server, the server broadcasts an *invalidation* message in the same manner as it will send out a *query* message. Upon receiving this message, every node in the network will compare the File ID with those in its cache, if a match is found, the cached copy's version number is compared with that in the message. The cached copy is marked stale if its local copy is older. Unlike a *query* messages, an *invalidation* message does not require reply messages when matches are found, therefore results less overhead.

- To further improve on the control message overhead, a peer can piggyback *invalidation* messages with outgoing *ping* or *query* messages.

Ideally, if all the peers are constantly connected in the Gnutella network and the TTLs of the flooding messages are large enough to reach every peer, the push-based algorithm described above should give us a strong consistency guarantee. However, things are not that simple in real life. And in particular, we are facing the following problems in the real Gnutella network.

1. Not all the peers in the network will receive the broadcast messages. There are two situations this can happen: one is when the network is partitioned; the other is when a peer is beyond the reach of the specified TTL.

2. Peers in the Gnutella network leave and join the network dynamically. After a peer leaves Gnutella, it won't be able to receive any further *invalidation* messages. Even after it rejoins the Gnutella network, it won't know if it has missed any *invalidation* message.

Given the above scenarios, we conclude that push alone is not sufficient for maintaining consistency in a large scale P2P environment such as the Gnutella network. To overcome this limitation, we propose a push and pull combined technique that is promising to give good consistency guarantees to all the peers in a large scale network.

4.3 Push and Pull Combined Technique — Push with Adaptive Pull

Push with Adaptive Pull (PAP) technique combines the *Push* (PUSH) and *adaptive poll* (PULL) algorithms described earlier in this section. Using *Push* alone gives strong consistency guarantees to all the peers that remain online and reachable by the broadcast messages. However, it does not handle well for those peers that are not reachable or when the network is highly dynamic (i.e., nodes leave and join frequently). Therefore we add *Adaptive Pull*, and we see that it succeeds where *Push* leaves off.

The push part of this algorithm is exactly the same as the invalidation-based push algorithm described in section 4.2.1. The pull part of the algorithm is similar to the adaptive poll algorithm. However, since we are now combining push and pull, ideally only those peers that could not receives the *invalidation* messages are supposed to poll the servers. It is hard to achieve this ideal case, but we can modify the adaptive poll algorithm to make the pulling less agressive. Particularly, we need to make the following changes:

1. Now we want to take into account the number of active connections a peer has. The heuristics is that the more connections a peer has, the better chance it will receive a future *invalidation* message. Let N_{conn} denotes the number of active connections a peer has, and $N_{avgconn}$ denotes the average number of connections per peer (this can be obtained from measurement studies of Gnutella network). We then need to change Eq. 2 to the following form:

$$TTR_{estimate} = TTR_{latest} + (1 + \frac{N_{conn} - N_{avgconn}}{N_{avgconn}}) \times C \quad (5)$$

   Based on Eq. 5, a file will be polled more actively if the peer that caches the file has less than the average number of connections, and less actively if that peer has more than the average number of connections.

2. All the other parts of the adapive TTR algorithm remain the same as the pure pull based approach.

3. Additionally, whenever a peer receives an *invalidation* message for a file B, it will mark the local copy of B as STALE if the message contains a newer version number for B. At the same time, the peer updates file B's next TTR value by adding to it a constant value, which we can use the same constant C as in the adaptive TTR algorithm. Although a peer will not poll for a STALE copy, when the copy is updated (for example, by user's initiative), the peer will restart polling for the object using the TTR stored with the STALE copy.

By combining server-push and client-poll, we provide strong consistency for most of the peers and bounded consistency for a small portion of them. In addition, the traffic overhead caused by the adaptive polling part is generally much less than the traffic overhead caused by the pushing part, given carefully chosen parameters for the adaptive TTR algorithm.

In the next section, we describe a simulation study of the three cache consistency techniques described previously, i.e., *adaptive pull*, *Push*, and *Push with Adaptive Pull*.

5 Simulation Study

We conducted a simulation study of the Gnutella-style P2P network. The simulator is written in C, and it uses the CSIM18 simulation library.

5.1 Overview of the Simulator

The basic scheme of the simulation is to initialize a P2P network based on observed statistical data and educated assumptions. Each peer is simulated as a seperate process, and each peer maintains a FIFO queue for incoming requests. A workload generator continues generating query and download requests, the intervals between successive requests follow an exponential distribution. A peer process is an infinite loop which continuously services incoming requests and performs appropriate actions based on the request type. To simulate cache consistency, we created other processes, for example, an update process, or a poll process. To account for the dynamic nature of the P2P network, we created a process that emulates the behavior of peers leaving and rejoining the network. Our simulation used a lot of statistical distributions to simulate the behavior or resource distributions within the network. These data are derived from measurement studies by Ripeanu et al. [8] and Saroiu et al. [9], and therefore our simulation is close to a real Gnutella network. In addition, we argue that the performance of our proposed cache consistency techniques are not strongly dependent on some of the distributions we choose, we will describe more about this in the result section.

5.2 Simulator Components

The simulator is composed of the following components: a network topology generator, a workload generator, a peer module, an update module, a poll module, a failure module, a download module, and a statistics module. Below we describe each module in details:

The network topology generator Although study by Ripeanu et al. and Saroiu et al. showed that the link distribution of the current Gnutella network is quasi-heavy-tail. We did not use this distribution since it hardly affects the cache consistency techniques we propose. To initialize a peer to peer network, the simulator requires two parameters, the number of peer nodes in the network, N_{peers}, and the number of active connections each peer maintains, N_{conn}. In addition, based on common Gnutella client implementations [12] [13], we defined two other parameters: the average number of links per node, N_{avg}, and the maximum number of links per node, N_{max}.

Currently we used three methods to initialize the P2P network:

1. The first one initializes the network *randomly*, and the links are all *uni-directional* links. The end result is that peers have different number of links. The N_{avg} parameter is not used here. The only guarantee in this case is that each peer has at least one outgoing link, and the total number of links of each peer does not exceed the N_{max}.
2. The second method initializes the network using *bi-directional* links, and all the peers have the *same number of links* defined by the N_{avg}. However, there's no guarantee that the result network will be fully connected.
3. The third method is similar to the second method except that the result network is guaranteed to be *fully connected*, i.e., there's a connected path between any pair of nodes in the network.

In addition to generating the link topology, we also initialize the object distribution within the network. We assume all the objects in the network are there from the beginning, objects can be replicated and the replicated (or cached) copies can be deleted, but no new objects are injected into the network during the simulation. Let N_{objs} denotes the total number of objects in the network. Given N_{peers} and N_{objs}, the objects are distributed following a $20/80$ rule, i.e., 20 percent of the peers owns 80 percent of the objects.

The peer bandwidth distribution is also simulated based on the study by Ripeanu et al. and Saroiu et al. We made a simplification by dividing peer bandwidth into two broad catagories, one as modem bandwidth (including 56K modems and ISDN), the other as broadband bandwidth (including cable modems, DSL, T1, and T3). For the purpose of studying cache consistency techniques, this simplification is sufficient.

The workload generator The *workload generator* is a process that continuously generates requests following an exponential distribution of the interarrival time. A request contains the following information:

- *Object ID*. This is the target object for the request. It is chosen according to a Zipf's distribution, which is based on the observation of the popularity of objects in the Gnutella network in [7].
- *Source peer ID*. This is the peer that sends the request. It is selected randomly based on the following criteria:
  - The peer must be connected in the network.
  - The peer does not have the specified object. Or the peer may have a *stale* or *possibly-stale* copy of the object.
- *Request type*. This can be QUERY, POLL, or REFRESH depends on if the target object is already in the source peer's local cache and its consistency state. There are other request types

used in the simulator, however, the workload generator only generates QUERY, POLL, or REFRESH requests, and a DOWNLOAD request may be generated following a QUERY request with certain possibility.

- *TTL*. This is necessary when the request is a broadcast message, such as a QUERY request.
- *Destination peer ID*. This is where the request is sent to. Initially this is the same as the source peer ID.
- *Arrival time*. This is the time when the request arrives at the destination peer.

The workload generator first selects a target object and the source peer of the request. If the source peer does not have the object, a query request is generated and inserted into the source peer's request queue, and broadcast into the network from there. With certain probability, $P_{download}$, a DOWNLOAD request is created some time after the query, the download server (i.e., the destination peer) is chosen among those peers that have valid copies of the object. If the source peer has a *stale* copy of the object, a REFRESH request will be generated and the new version of the object will be downloaded from the object's origin server, given that it's online. In the case the source peer has a *possibly-stale* copy of the object, a POLL request to the origin server of the object will be generated if polling is enabled, otherwise the request is discarded.

A request as described above carries all the information needed for processing itself and collecting statistics. There are totally six types of requests: QUERY, DOWNLOAD, INVALIDATION, FAILURE, REFRESH, POLL. Among them the FAILURE request is a bogus one, simply created for the simulation, it does not exist in real implementations. Its function is to notify a peer to stop functioning for a period of time, as a way to simulate a peer's leaving and rejoining the network. It will be described in more detail in the *failure module* section.

The peer module Peers are simulated as processes. Each peer maintains an incoming request queue. A peer process is essentially an infinite loop which continuously services new requests in FIFO order.

The update module An update process is created for simulating updates to master objects in the system. We adopt the object update distribution of the current web environment. Specifically, we divide the objects into four catagories: very fast mutable, very mutable, mutable, and immutable. Each catagory has its own *percentage* and *time between successive updates* respectively. Given the above sequence, the percentage and time between successive updates for each category are: (0.5%, 15 minute), (2.5%, 450 minutes), (7%, 1800 minutes), and (90%, 86400 minutes). Because the Gnutella network is primarily used for sharing immutable files to date, realistic distribution of object updates is unavailable and the web model is used as an approximation.

In the simulator, the average time between successive updates is a configurable parameter and the objects are chosen according to the distribution described above. Each update increments the master copy's version number by one and also updates the last-modified-time of the object accordingly. If push mechanism is used, an INVALIDATION message will be flooded into the network. The message contains the following fields:

- The object ID.
- The object's origin server ID.
- The new version number.
- The new last-modified-time.

Peers that receive this message will check their local caches. If a match is found, they will compare the version number of the local copy with that in the message, and mark the local copy *stale* if its version number is smaller.

The poll module A poll process is created to take care of all the pollings for the entire system. This module includes an ordered list where POLL requests are inserted according to the scheduled polling time. The poll process continously takes requests off the head of the list and services it.

In our implementation, a POLL request is not inserted into the corresponding origin server's incoming request queue and wait for the response being inserted back. Instead, we simplified the polling by directly comparing the object's version number with that of its master copy using global information. There are three types of poll results, as described below:

1. *Server Unavailable*. This happens when the origin server of the object leaves the Gnutella network. The polling peer will mark its cache copy *possibly-stale* and stop polling further. A *possibly-stale* copy will be polled again if there's a query request or local access for it.
2. *Object Unmodified*. This happens when the master copy has the same version number as the polling peer's cache copy. In this case, the polling peer will mark its cache copy *valid* if it is not, and calculates a new TTR value based on the TTR algorithm described earilier. A new POLL request will be constructed and inserted into the polling list based on the new TTR expiration time.

3. *Object Modified*. This happens when the master copy has a different version number than the polling peer's cache copy. The polling peer will simply mark its local copy *stale* and stop polling. A stale copy will be refreshed when there's a query request or local access for it.

The failure module A failure process is created to simulate the dynamic behavior of the Gnutella network. It simulates peers leaving and rejoining the network. There are three parameters involved in this process: the maximum offline ratio, R_f, which is the maximum percentage of peers that are disconnected from the network at any given time; the time between successive disconnection occurrences, I_f; and the average duration peers remain offline after disconnections, D_f.

The failure process is a loop that continuously generates failure requests. The interval between requests are simulated with an exponential distribution with an average I_f. Each failure request includes the following information:

- *Destination peer ID*. This is the peer that is about to leave the network. The peer is chosen following a $90/10$ rule, i.e., 90 percent of the nodes leave and rejoin the network frequently, the remaining 10 percent of the nodes are relatively stable in the network.
- *Offline duration*. This is the duration during which the peer will remain disconnected. It is chosen from an exponential distribution centered around D_f.
- *Arrival time*. Here the arrival time is when the peer gets disconnected from the P2P network.

After being generated, the failure request is inserted into the destination peer's incoming request queue. The destination peer (say B) processes this request by first marking its status as offline. Then all the un-serviced requests in B's incoming request queue will be discarded. All the links connected to B will be tear down. Then peer B's process will sleep for the specified duration. After that peer B will rejoin the network. The rejoining process consists of re-initializing its neighbors and marking its status as online.

When a peer becomes offline, all its scheduled downloads will be cancelled, its cached objects will become *possibly-stale* when their TTR expires.

The topology fix module This module logically belongs to the failure module.

When a peer leaves the network, it will tear down all its active connections, this will cause each of its neighbor nodes lose one of their active connections. If a lot of nodes leave the network, the network may become more or less disconnected. Real Gnutella client implementations [12] [13] solve this problem by letting a peer client pick new neighbors from its host cache when its number of active connections falls below some threshhold. Following the same idea, we created a topology fix process to simulate this functionality.

Basically what this process does is to check periodically if there are peers with less than average active connections, and add links between these peers.

The download module A download process is created to take care of all the scheduled downloads for the entire system. Basically, we created an ordered list where DOWNLOAD requests are inserted in the order of download completion time. The download process continously take off DOWNLOAD requests from the head of the list and services that request. Each download checks if the downloader already has the object or not. If not, it's a fresh download, the object is simply inserted into the downloader's local cache; otherwise the cache copy is updated.

If the state of a downloaded object is *valid*, the simulator will start polling that object. However, if the download updates an old cache copy, the old TTR value is used to calculate a new TTR value. In this manner the download process takes care of both fresh downloads and refreshes of stale objects.

The statistics module The statistics module is responsible for collecting statistical data in the simulation.

5.3 Simulation Parameters

Because our simulator synthesizes artificial traces instead of using a real one, it has a large set of configurable parameters for tuning the performance or testing against different hypothesis or realistic measurements. This also makes the simulator more extensible and versatile for different purposes.

In this section we list the important parameters used in the simulation, along with their descriptions and default values.

Table 1: General parameters for the simulation

Parameter	Description	Default Value
$Seed$	Random seed	N/A
N_{peers}	Number of peers in the network	500
N_{objs}	Number of unique objects in the network	5000
$N_{avgconn}$	Average number of active connections each peer maintains	4
$N_{maxconn}$	Maximum number of active connections each peer may have	8
L_{sim}	Length of simulation	10 hours
$Algo_{cons}$	Cache consistency algorithm used in the simulation	[NONE, PULL, PUSH, PAP*]
P_{modem}	Percentage of modem-type peers	8%
I_{query}	Average time between successive query requests	1 second
$P_{download}$	Probablity that a download request follows a query	70%
$I_{download}$	Average gap between a query and a download request	4 seconds
I_{update}	Average time between successive update requests	2 seconds
TTL	TTL of broadcast messages	8
$Algo_{ttr}$	TTR algorithm for PULL	[STATIC, DYNAMIC]
TTR_{static}	Static TTR value when TTR algorithm is STATIC	5 minutes
TTR_{min}	TTR low bound when using dynamic TTRs.	5 minutes
TTR_{max}	TTR high bound when using dynamic TTRs.	1 hour
C	Parameter for computing dynamic TTRs, see equation 1.	10 minutes
α	Parameter for computing dynamic TTRs, see equation 2.	0.5
w	Parameter for computing dynamic TTRs, see equation 3.	0.8

* PAP denotes PUSH with Adaptive Pull combined algorithm.

Table 2: Parameters for the dynamic network environment

Parameter	Description	Default Value
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

5.4 Metrics for Performance Analysis

To measure the effectiveness of our proposed cache consistency techniques, there are two main criteria we consider.

- *Fidelity*. Here we create the term False Valid Ratio (FVR) to measure the degree of fidelity. When a query reaches a peer and there's a local copy matching the query, if the local object's state is *valid*, we want to know the likelyhood that this object is actually NOT *valid*, i.e., the master copy has a newer version number. By collecting the total number of query hits that are false valid, N_{qfv}, and the total number of query hits where the object states are shown as *valid*, N_{qv}, we can define the query false valid ratio (QFVR) as

  $$N_{qfv}/N_{qv}$$

  In the same fashion, we define the download false valid ratio (DFVR) as

  $$N_{dfv}/N_{dv}.$$

- *Control message overhead*. Flooding invalidation messages into the network and/or polling with TTR both add control message overhead into the network. Comparatively, flooding with invalidation messages seems to incur more overhead than polling. However, we will show in the simulation results that the flooding overhead will depend on several factors, including the time between successive updates, the dynamic behavior of the network, and the topology of the network. Based on our study, we will propose situations where flooding is more appropriate. And we also argue that flooding is the basic message routing protocol in the Gnutella network, although it has poor network efficiency, currently there are no best solutions to solve this problem. With the rapid research and industry efforts in the P2P networking area, we believe that a better solution for efficient message routing in Gnutella network is right around the corner. One advantage of pushing invalidation messages is that the network overhead incured by it will be alleviated as soon as a better message routing scheme is available. Another advantage of it is that the servers are stateless, thus the technique is more fault-tolerant. In addition, implementation techniques can be applied to reduce the invalidation message overhead. For example, we can piggyback invalidation messages with PING or QUERY messages.

5.5 Network Environments Used in the Simulation

The simulations are conducted in two different network environments as described below:

- The first set of experiments are performed in a perfectly stable network environment. All the peers remain online throughout the simulation, there are no failures of any type. Although this situation is unrealistic for a global P2P network, it gives us some sense of how the techniques will perform in the ideal case.

- The second set of experiments are performed in a dynamically changing network environment. Here we assume peers leave and rejoin the network randomly, based on the three parameters R_f, I_f, and D_f, as defined previously. 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 the real one, this is due to the limitation of the simulator, where memory and time required to run large scale simulations are prohibitive. For most of the simulations, we use a network consisting of 500 peers and 5000 objects. In terms of cache consistency study, we believe such an environment is sufficient.

5.6 Simulation Results and Analysis

In the following two sections, we describe the simulation results for the two different network environments as described above.

5.6.1 Simulation Results of a Stable P2P Network

All the experiments described in this section were conducted with F_{enable} set to FALSE. The other parameters are set to the defaults unless explicitly spcified.

1. The first set of experiments measure the effects of time between successive updates, $I_{updates}$, on the query/download FVR. More specifically, we fix the time between successive query requests, I_{query}, to 1 second, and change the time between successive update requests, I_{update}, such that the ratio of I_{update}/I_{query} ranges from 1 to 10. We did three experiments in this batch, the first one uses *Adaptive Pull*) alone as the cache consistency technique, the second one uses *Push* alone, and the third one combines *Push* and *Adaptive Pull*. Figures 1.1 and 1.2 show our observations:

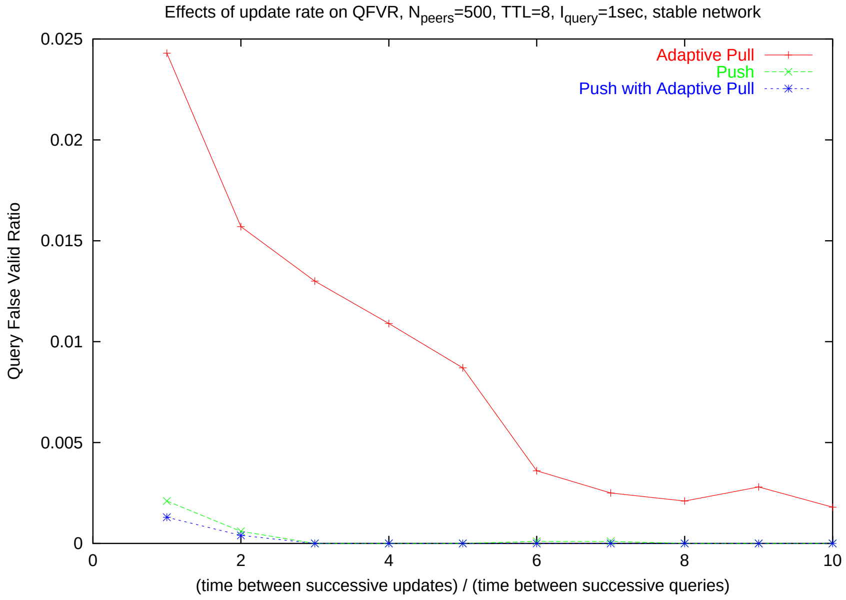

Figure 1.1: Effects of update rate on query false valid ratio

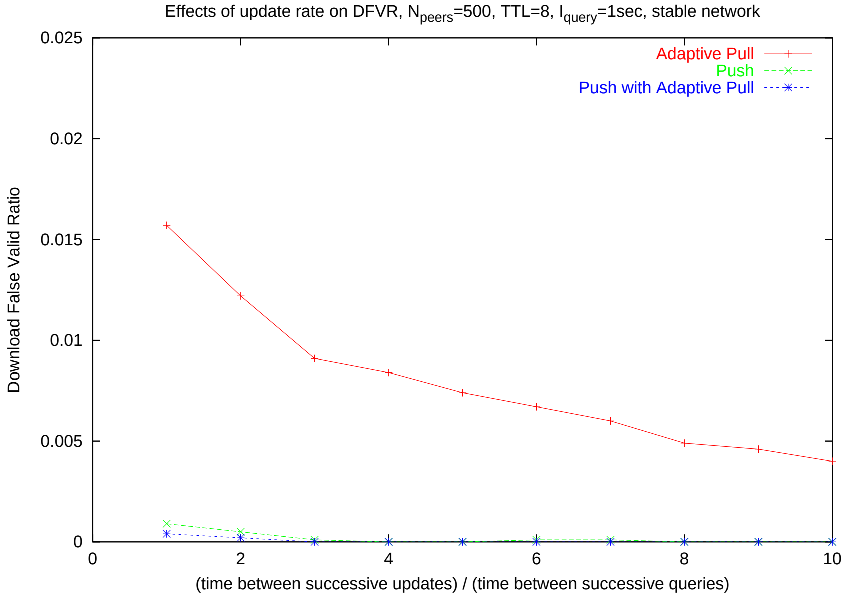

Figure 1.2: Effects of update rate on download false valid ratio

The above two figures imply the following:

(a) Observe that in both graphs, the FVRs of using *Push with Adaptive Pull* are about 60% of that of using pure *Push* when the ratio I_{update}/I_{query} is 1. *Push* alone achieves almost perfect fidelity when the ratio is greater than 2. Therefore if there are less updates than queries in the system, we expect *Push* alone to be sufficient.

(b) *Adaptive Pull* alone does not guarantee strong consistency. The FVRs of using *Adaptive Pull* are 10 to 30 times higher than using *Push with Adaptive Pull*. As update rate decrease by 10 folds, the FVRs of using *Adaptive Pull* decrease by 67% to 92%. From these facts we recommend using *Adaptive Pull* only when the consistency requirements are less stringent, and it is better to use it when the update rate is low relative to the query rate.

The control message overhead (measured in terms of number of control messages) of the above experiements are also collected and showed in the following graphs:

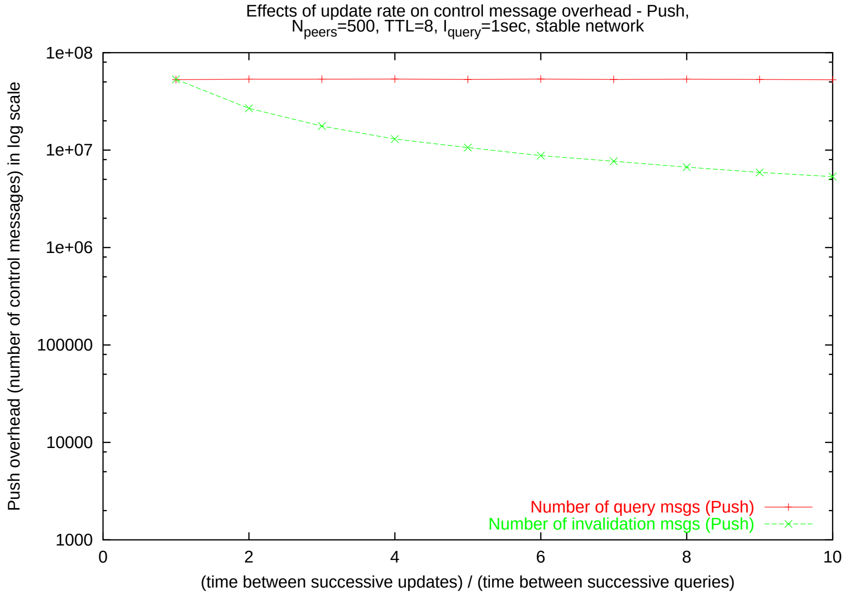

Figure 1.3: Effect of update rate on control message overhead – PUSH

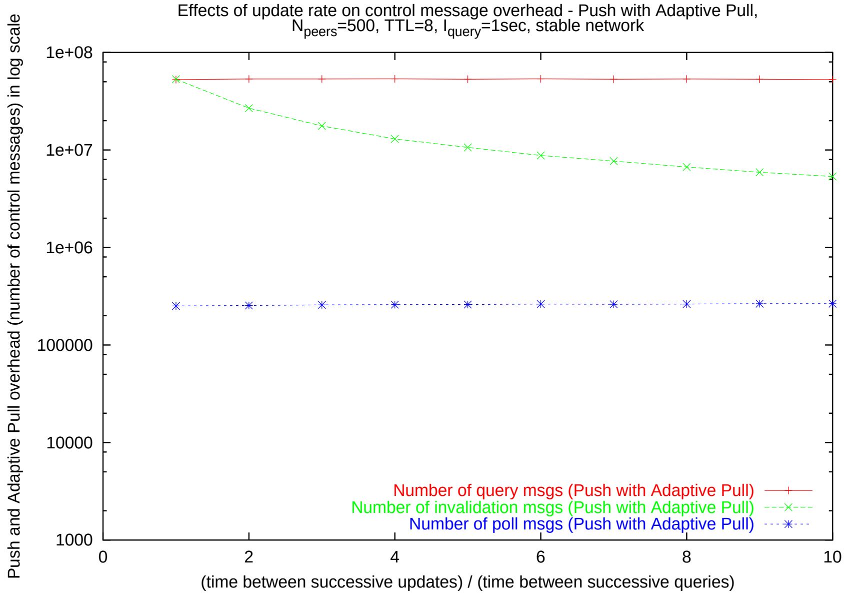

Figure 1.4: Effect of update rate on control message overhead – PAP

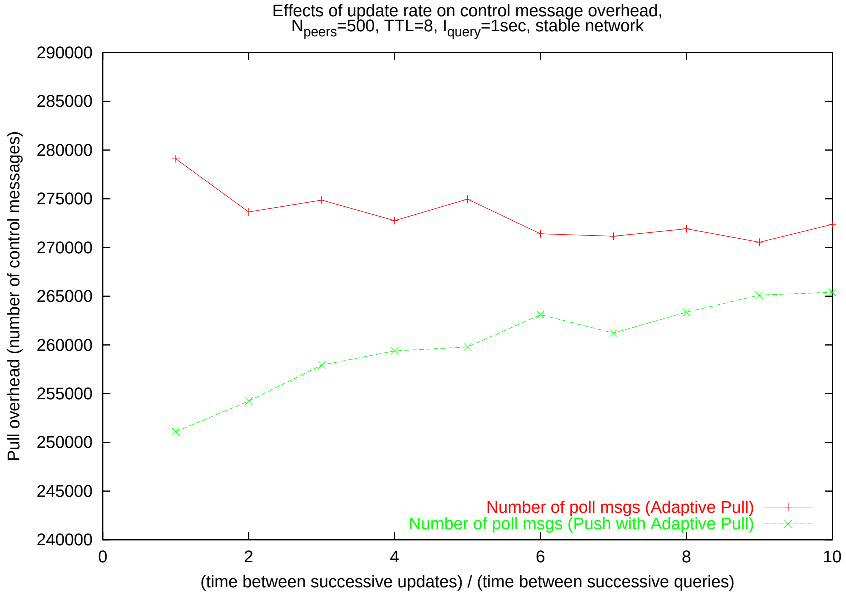

Figure 1.5: Comparison of PULL overhead

Figures 1.3 and 1.4 show the control message overhead (in log scale) of using *Push* and *Push with Adaptive Pull* respecitively. Figure 1.5 compares the poll message overhead when using *pull* alone and when using *push and adaptive pull*. We can derive the following conclusions from the obervations:

(a) Invalidation message overhead is dominant over poll message overhead for example, two magnitudes higher in our experiments. The invalidation message overhead in pure *Push* is roughly the same as that in *push and adaptive pull*. In addition, the invalidation message overhead decreases linearly as the update rate decreases. This suggests that the *Push* technique is more appropriate for systems where update rates are lower than query rates.

(b) Although poll message overhead is relatively trivial comparing to invalidation message overhead, we are still interested in a smart pull algorithm where poll message overhead is reduced when push and pull are used together; Figure 1.5 above shows such a comparison. Observe that when *pull* is used alone, the poll message overhead decreases by about 2% as the update rate decreases by 10 folds. This is expected because when updates are more often, the adaptive TTR algorithm tends to produce smaller TTR values which results in slightly more polls. In comparison, when push and pull are used together, each invalidation hit will cancel or delay a currently scheduled pull, and also increase TTR for future pulls. Therefore in Figure 1.5, the poll message overhead increases by 6% when the update rate decreases by 10 folds. Also the poll message overhead in *Push with Adaptive Pull* is 1.5% to 10% less than that in *Adaptive Pull*.

2. The second set of experiments study the *pull with static TTR* algorithm. We set I_{query} to 1 second and the I_{update} to 2 seconds. We varied the static TTR values from 1 minute to 19 minutes across the experiments. The results are shown in Figure 2.1 below:

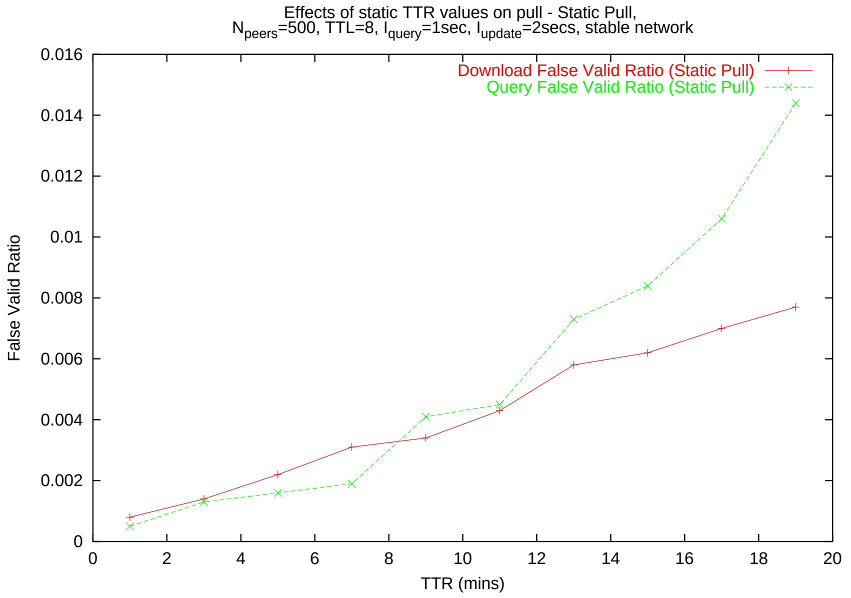

Figure 2.1: Effects of static TTR values on fidelity

As the static TTR values increase, there are less polls for the objects in the system, thus the QFVR and DFVR both increase. The QFVRs increase by approximately 20 folds from TTR 1 minute to TTR 19 minutes, the DFVRs increase only by about 10 folds. Our explanation to the slower increase of the DFVRs is that downloads are picked from those query hits of *valid* replica, since the majority of these replica are indeed valid, the DFVRs tend to be smaller than QFVRs.

The control message overhead of this set of experiments are shown in Figure 2.2 below:

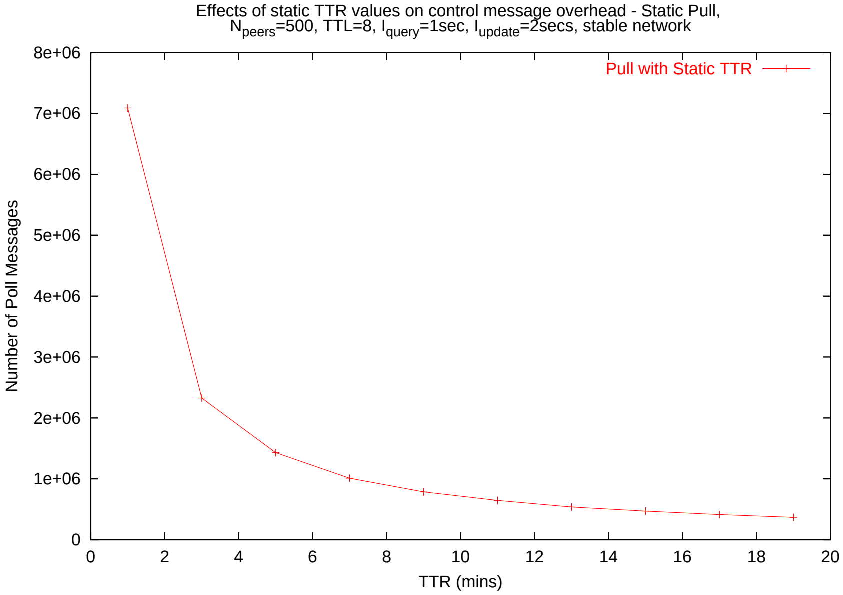

Figure 2.2: Effects of static TTR values on control message overhead

The control message overhead decreases by 7 folds as the static TTR values increase from 1 minute to 19 minutes. Thus it is a tradeoff when selecting static TTR values: large TTR values give worse consistency but less overhead; small TTR values give better consistency but more overhead. In practice, people tend to emphasize more on consistency and use small TTR values.

3. The third set of experiments investigate the effects of different TTL values on fidelity when using *Push* only. Since TTL values set the scope each broadcast message can reach, an invalidation message can reach more replicas with larger TTL values. Thus we expect the query/download FVRs to decrease with the increase of TTL values, this is demonstrated in Figure 3.1 below:

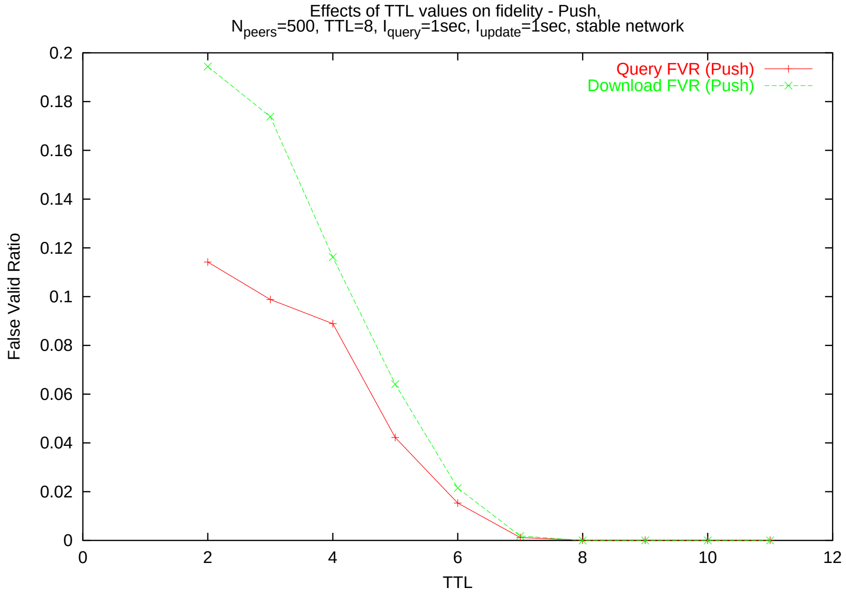

Figure 3.1: Effects of TTL values on fidelity when using push

In Figure 3.1, we observe that both the QFVR and DFVR fall to zero when the TTL values increase from 2 to 8. Setting the TTL larger than 8 is thus unnecessary.

The control message overhead of this set of experiments are shown in Figure 3.2 below:

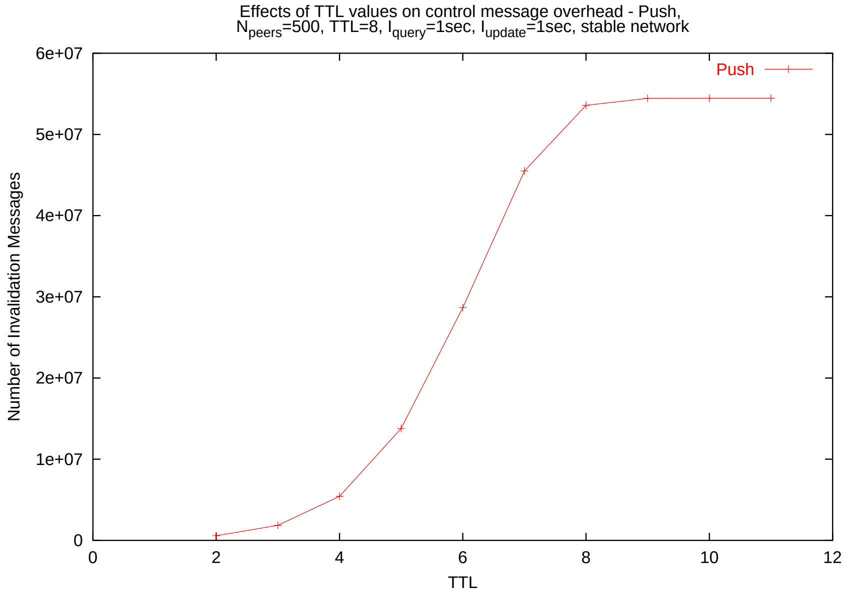

Figure 3.2: Effects of TTL values on control message overhead when using push

From Figure 3.2, we observe that the invalidation message overhead increases by almost 100 folds as TTL values increase from 2 to 8. This is close to an exponential increase. However, for a certain network configuration (here $N_{peers} = 500, N_{avgconn} = 4$), it reaches a plateau after TTL 8. This is due to the Gnutella routing algorithm where a peer will not route a message it has routed before.

4. We also conducted experiments to investigate the effects of network size on fidelity. The first experiment uses *Push*, and the second one uses *Push with Adaptive Pull*. In both the experiments, the network size ranges from 200 to 2000. The TTL value for all the experiemnts in this group is set to six. The results are shown in the following graph:

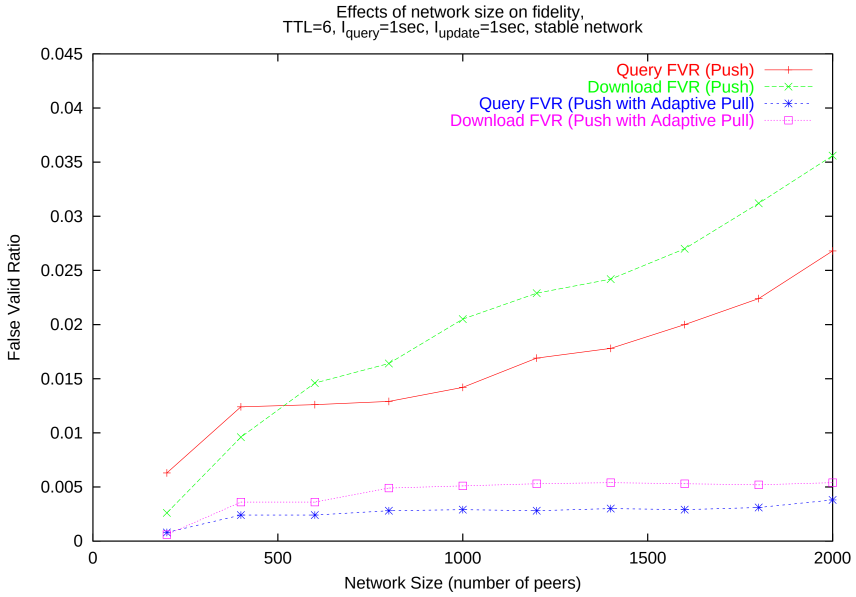

Figure 4.1: Effects of network size on fidelity

Figure 4.1 implies that as network size increases, the QFVR and DFVR in pure *Push* both increase by 5 and 10 folds respectively. However, in *push with adaptive pull*, the increase is only 3 to 5 folds. More importantly, the fidelity achieved with pure *Push* is 3 to 7 times worse than that with *Push with Adaptive Pull*. These suggest that *pull* leverages the effects of network size in terms of fidelity.

The control message overhead associated with the above experiments are shown in Figures 4.2 and 4.3 below:

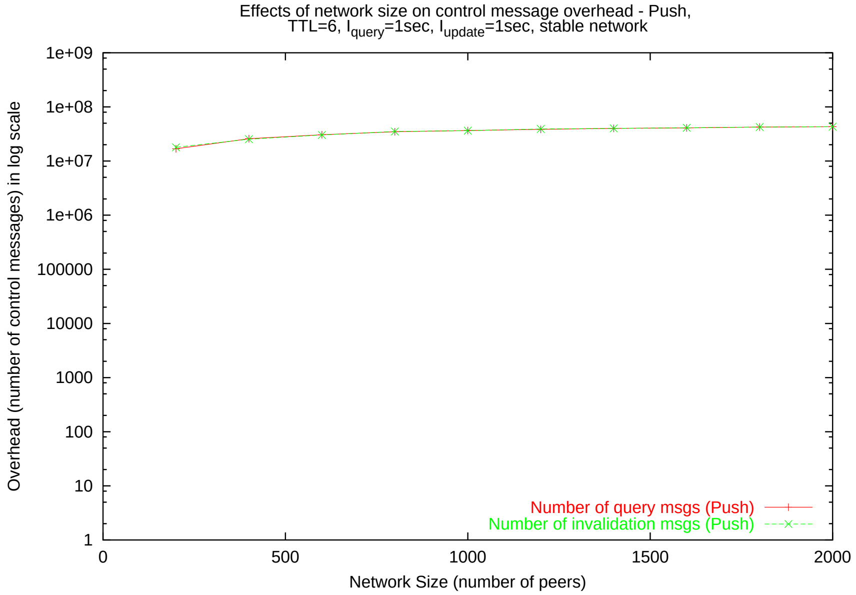

Figure 4.2: Effects of network size on control message overhead – PUSH

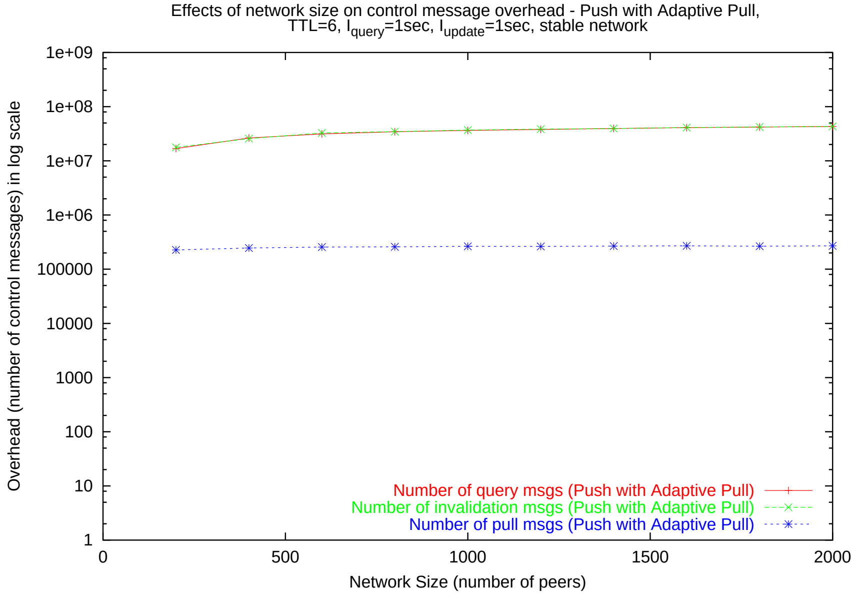

Figure 4.3: Effects of network size on communication overhead – PAP

In both the graphs, the invalidation message overhead increases by only 2.4 folds as the network size increases by 10 folds. The slow increase is due to the fixed TTL value. The query message overhead is roughly the same as the invalidation message overhead since $I_{update} = I_{query}$.

5. In a Gnutella-style network, the average number of active connections a peer maintains ($N_{avgconn}$) is an important factor on how well peers are connected. To evaluate this metric, we performed two

groups of experiments using *Push* and *push with adaptive pull* respectively. In both groups, the $N_{avgconn}$ values range from 1 to 5. The results are shown in Figure 5.1 below:

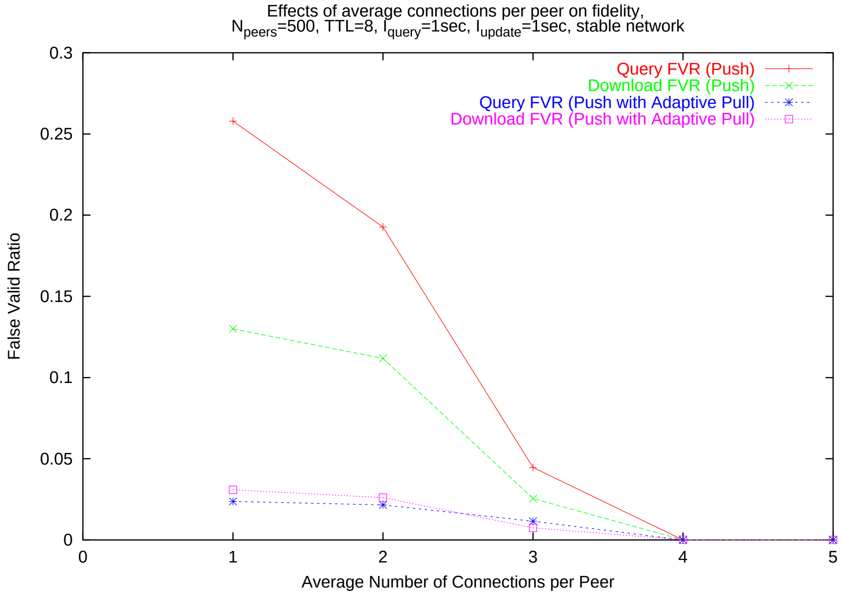

Figure 5.1: Effects of the average number of connections per peer on fidelity

From the above graph, we observe that both the query and download FVR decrease to zero as $N_{avgconn}$ increases to 4. *Push with Adaptive Pull* achieves 5 to 10 times better fidelity than *Push* when $N_{avgconn}$ is small. For these reasons we recommend P2P clients to maintain at least 4 active connections to help achieve better consistency. If the number of active connections differs greatly among peers, *Push with Adaptive Pull* is the better algorithm than pure *Push* .

The control message overhead of these experiments are shown in the following graphs:

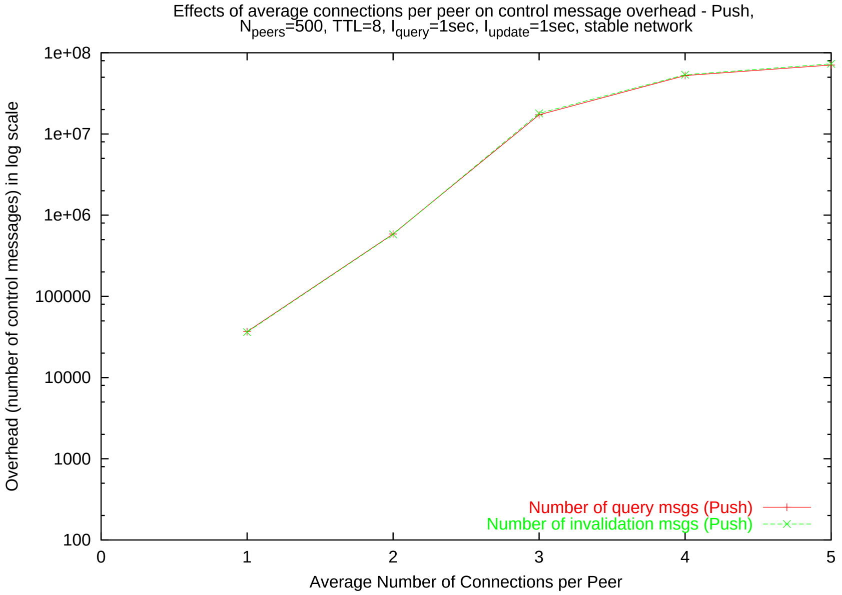

Figure 5.2: Effects of the average number of connections per peer on control message overhead – PUSH

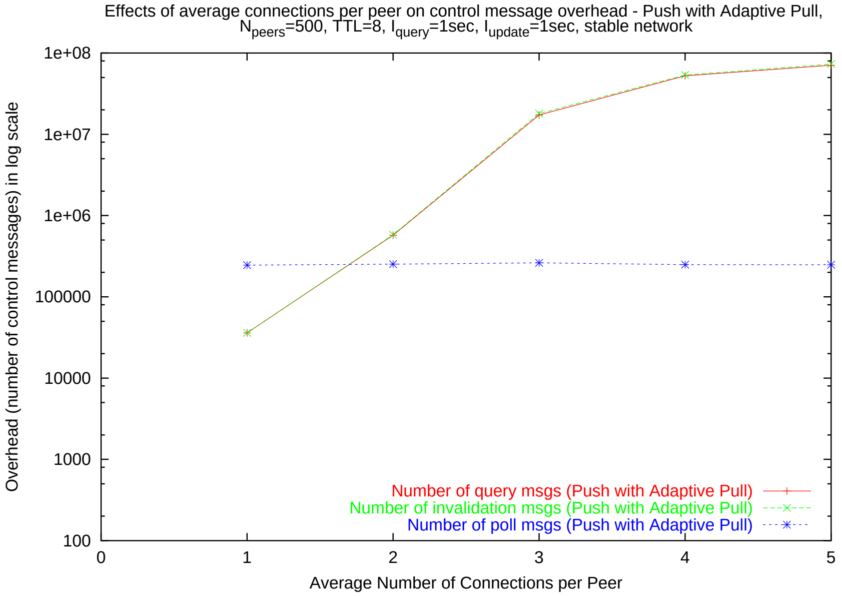

Figure 5.3: Effects of the average number of connections per peer on control message overhead – PAP

Figures 5.2 and 5.3 show the control message overhead of using *Push* and *Push with Adaptive Pull* respectively. Observe that both graphs have roughly the same invalidation message overhead, and it increases exponentially as $N_{avgconn}$ increases, which is as expected due to the broadcast routing protocol. The increase becomes slower after $N_{avgconn}$ is above 3, this is because nodes stop forwarding messages they have seen before. The poll message overhead is not affected by this parameter since all polls are made directly towards the origin servers through HTTP.

6. Before we went on to study the dynamic P2P network environment, we conducted a set of experiments to study the effects of query rate on fidelity and control message overhead. We only performed these experiments with the *Push with Adaptive Pull* algorithm. The I_{update} in these experiments are set to 1 second, and the I_{query} values range from 1 second to 10 seconds. The results are shown below.

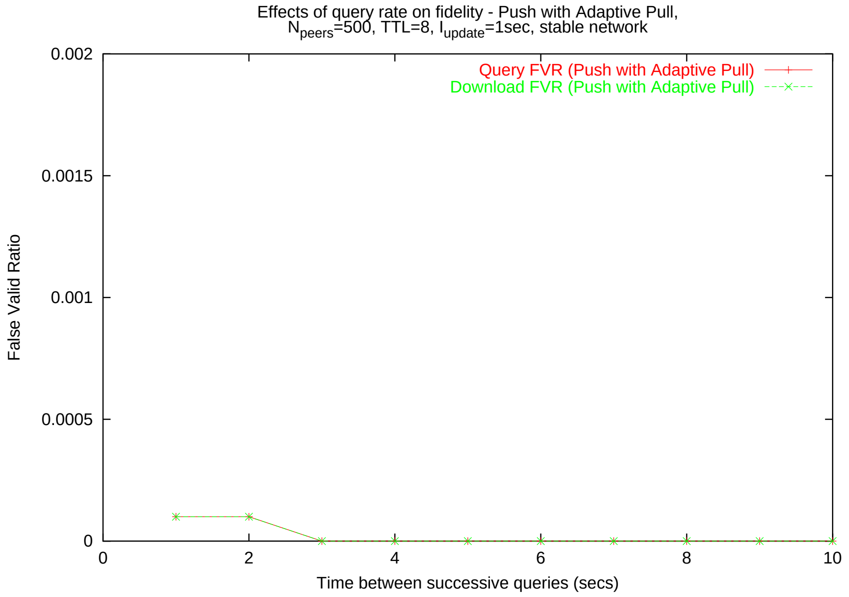

Figure 6.1: Effects of query rate on fidelity

The plot shown above suggests that the query rate does not affect fidelity when using *Push with Adaptive Pull*. This implies that even when the query rate is ten times larger than the update rate, *Push with Adaptive Pull* algorithm still achieves almost perfect fidelity.

Below is the control message overhead associated with this set of experiments.

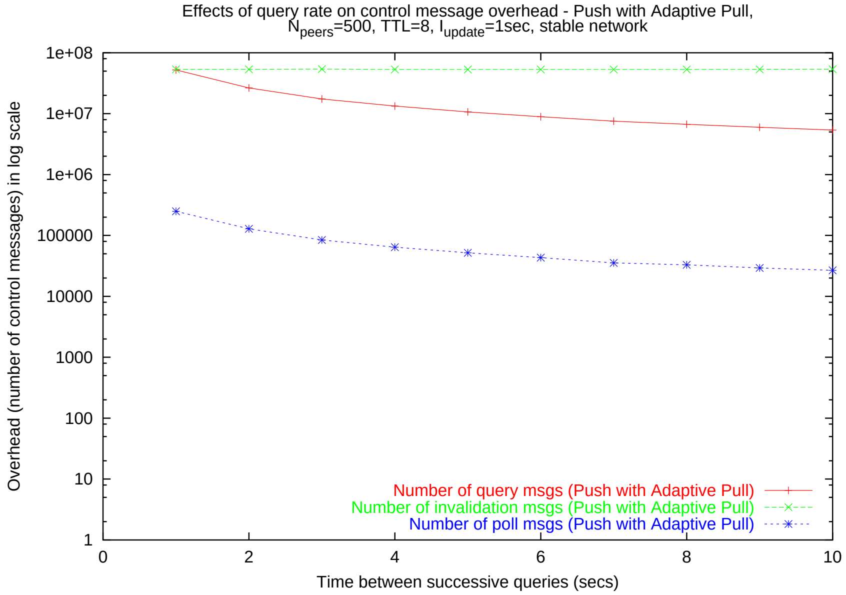

Figure 6.2: Effects of query rate on communication overhead

The query message overhead decreases almost linearly as query rate decreases, it deceases by 10 folds when the query rate decrease by 10 folds. The poll message overhead also decreases with the same trend. This is due to the fact that when query rate decreases, there are also less downloads such that the number of replications decreases, which in turn causes less poll messages.

5.6.2 Simulation Results of a Dynamic P2P Network

All the experiments in this section were conducted with F_{enable} set to TRUE. The rest of the parameters are set to the defaults unless explicitly specified.

1. In our simulation, we created a process that fixes a peer's active connections when this number drops below $N_{avgconn}$. This process checks the network periodically and adds more connections between peers that have less than average connections.

   Our first set of experiments study the effects of this process. Presumably, this process only affects the *Push* algorithm or the push part of the *Push with Adaptive Pull* algorithm. Therefore the experiments are performed for both the *Push* and *Push with Adaptive Pull* algorithms. In both experiments, the $I_{topochk}$ values range from 5 seconds to 170 seconds. The results are shown in Figures 7.1 and 7.2 below.

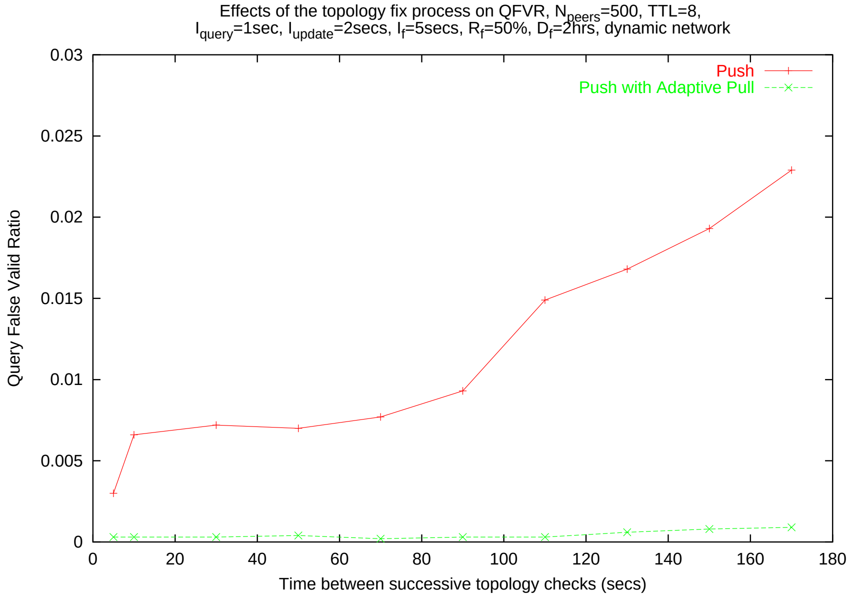

Figure 7.1: Effects of time between successive topology checks on query false valid ratio

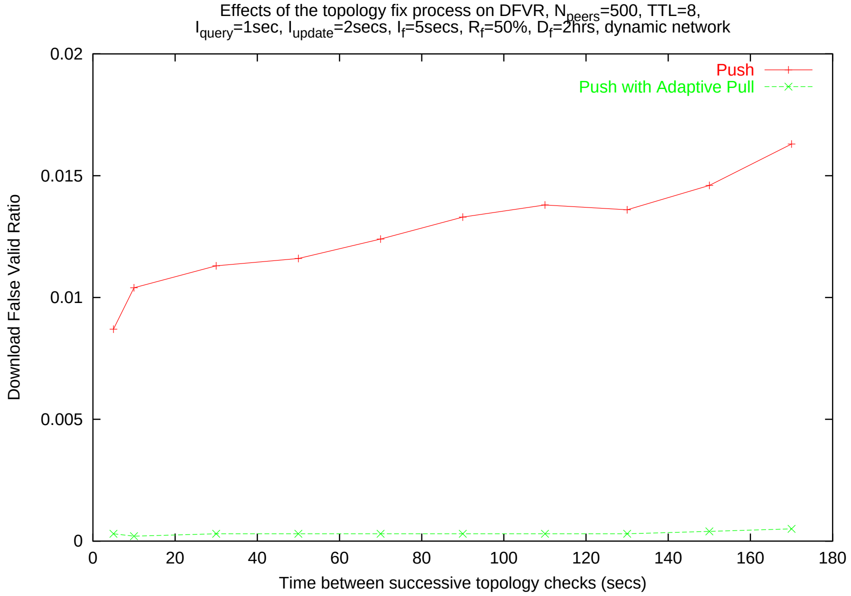

Figure 7.2: Effects of time between successive topology checks on download false valid ratio

In Figure 7.1, the QFVR increases by 6 folds when the $I_{topochk}$ increases from 5 seconds to 170 seconds. In Figure 7.2, the DFVR increases by 2 folds when the $I_{topochk}$ increases from 5 seconds to 170 seconds. Both cases tell us that *Push* achieved much better consisency when the topology fix process ran more regularly. However, when using *Push with Adaptive Pull*, the QFVR and DFVR only increase by 3 folds and 67% when the $I_{topochk}$ increases from 5 seconds to 170 seconds. In addition, the QFVRs and DFVRs of using *Push* are 10 to 30 times higher than those of using *Push with Adaptive Pull*. All these imply that the pull part of the *Push with Adaptive Pull* algorithm is effective enough such that the topology fix process is not necessary.

The invalidation message overhead of running these experiments is shown in Figures 7.3 and 7.4 below.

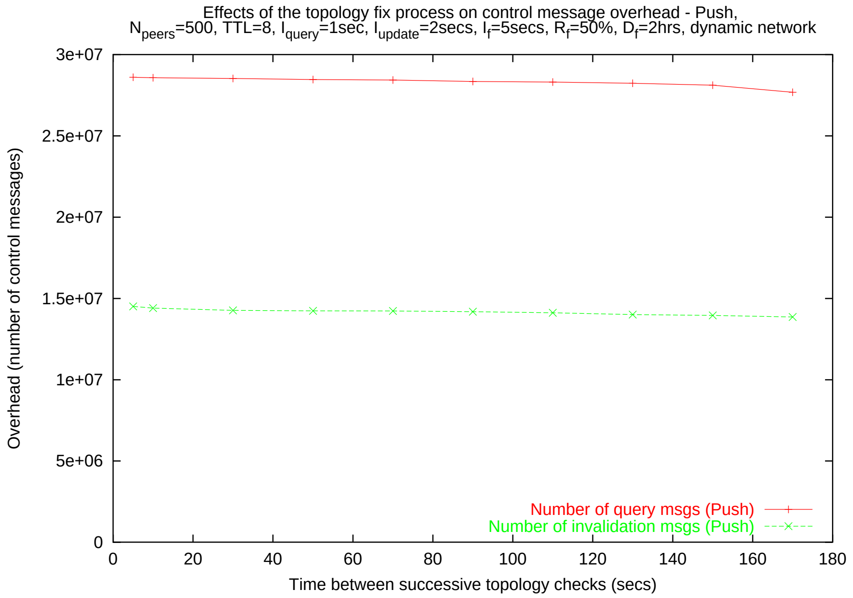

Figure 7.3: Effects of time between successive topology checks on control message overhead – PUSH

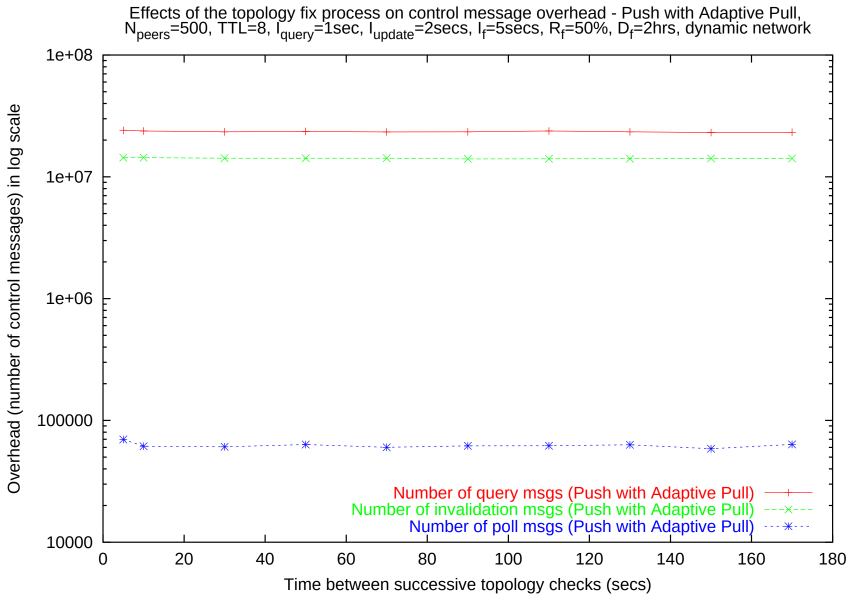

Figure 7.4: Effects of time between successive topology checks on control message overhead – PAP

Notice that the plots in Figure 7.4 are in log scale but those in Figure 7.3 are not. As we expected, the invalidation message overhead decreases slowly ($\sim 5\%$) as $I_{topochk}$ increases by 32 folds.

2. Among the three parameters that define the dynamics of the network, the first one we studied is the maximum failure ratio R_f, which defines the maximum percentage of offline nodes at any given time. Intuitively, the larger this value, the more disconnections occur in the network. We performed experiments for all the three cache consistency algorithms. In these experiments, the I_f is fixed at 5 seconds, the D_f is fixed at 2 hours, and the R_f values range from $5\% - 50\%$. Figures 8.1 and 8.2 below show the results.

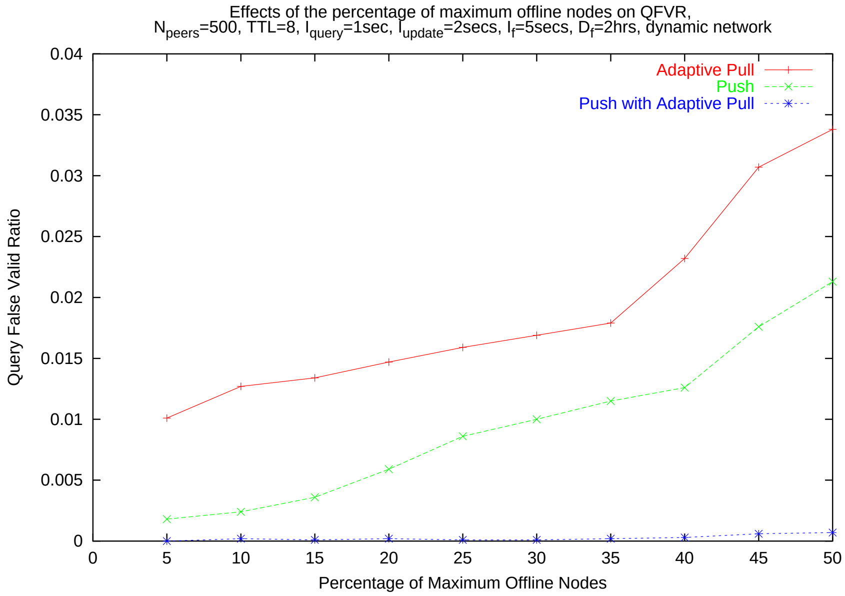

Figure 8.1: Effects of percentage of maximum offline nodes on query false valid ratio

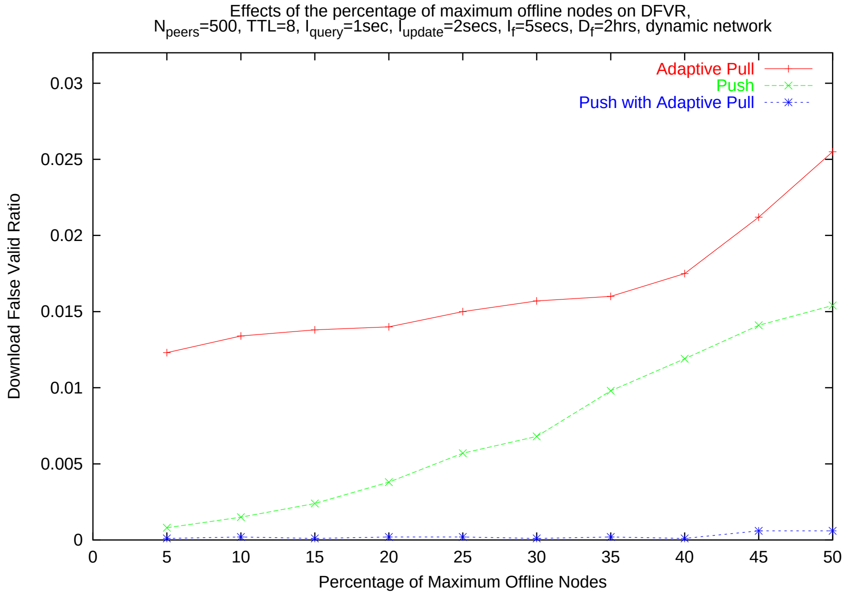

Figure 8.2: Effects of percentage of maximum offline nodes on download false valid ratio

From the above two graphs, we observe that when using *Adaptive Pull*, both the QFVR and DFVR increase by 3 and 2 folds as R_f increases from 5% to 50%; the same trend occurred when using *Push*, and the increase in *Push* is roughly linear, and both the QFVR and DFVR increase by 10 folds as R_f increases from 5% to 50%. The Q/DFVRs are at least 50% higher when using *Adaptive Pull* than using *Push*, which suggests that even in a highly dynamic network environment, *Push* alone still achieves better consistency than that of *Adaptive Pull*. Using *Push with Adaptive Pull* gives good and consistent fidelity across different R_f values, for example, the maximum QFVR when using this algorithm at $R_f = 50\%$ is 0.001, while *Adaptive Pull* and *Push* give 0.022 and 0.034 respectively.

The control message overhead of these experiments is shown in Fig 8.3 and 8.4 below.

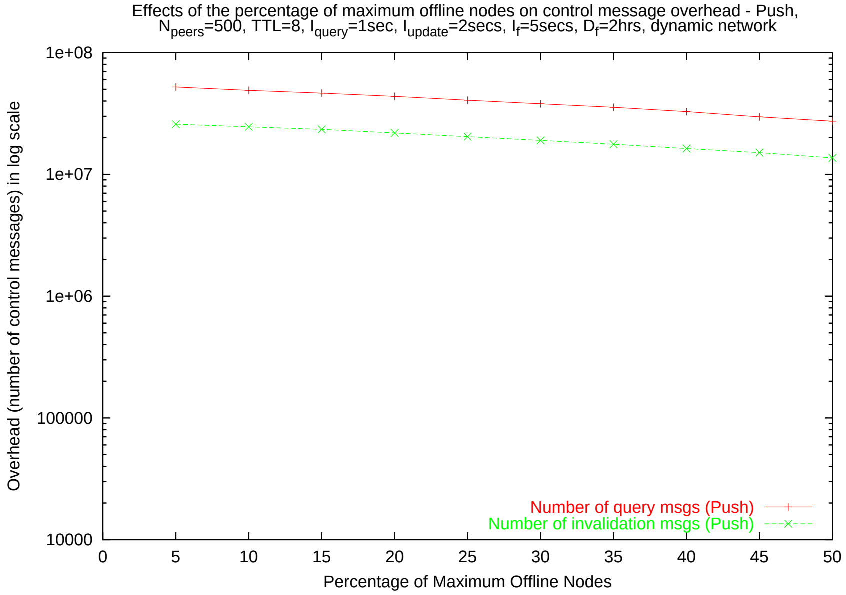

Figure 8.3: Effects of percentage of maximum offline nodes on control message overhead – PUSH

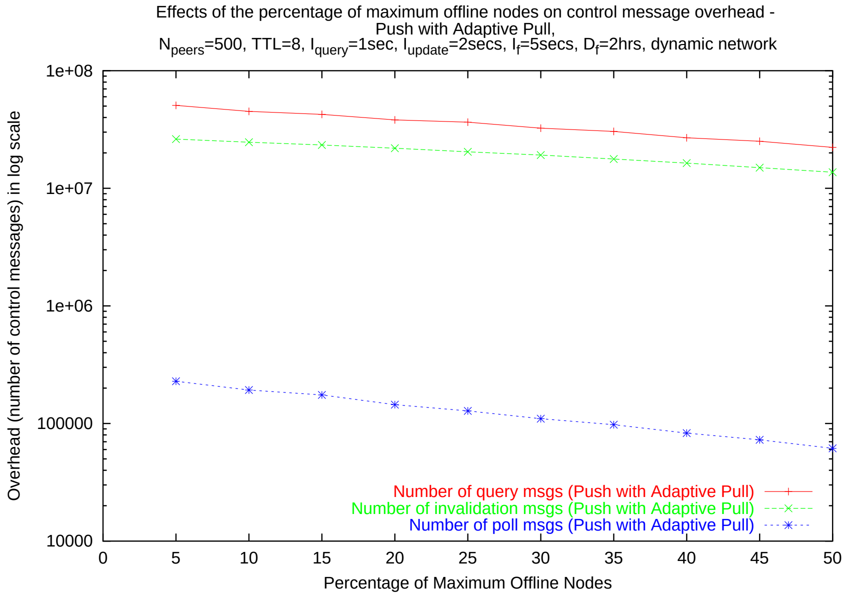

Figure 8.4: Effects of percentage of maximum offline nodes on control message overhead – PAP

Observe that as R_f increases from 5% to 50%, the invalidation and poll message overhead decrease by $1/2$ and $1/4$ respectively. The decrease of the invalidation messages is consistent with the assumption that there are more disconnections in the network as R_f increases. The decrease of the poll messages is due to the fact that when a peer leaves the network, it discards all the scheduled and future polls. Alghough we tries to adapt the TTR algorithm such that it polls more actively when there are more disconnections, since a peer can only judge this by local information, this adaptiveness to network changes is limited. We also observe that the poll message overhead is insignificant comparing to the invalidation message overhead, and as shown previously, we achieved much better consistency with this little overhead on top of the invalidation message overhead.

Our TTR algorithm's adaptiveness to network conditions is shown in the following graph.

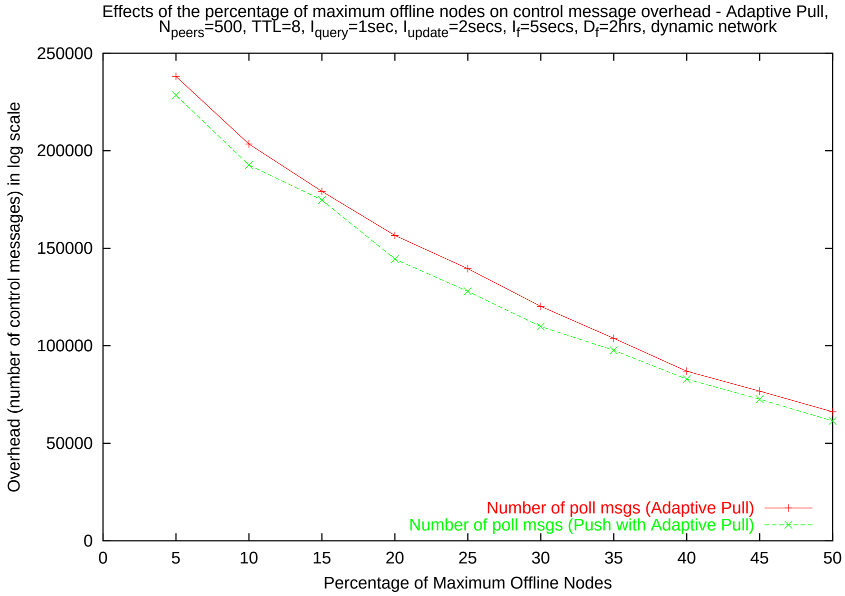

Figure 8.5: Comparison of PULL overhead

In Figure 8.5, the plot on top is the poll message overhead using pure *Adaptive Pull* algorithm, the one below it is the poll message overhead using the *Push with Adaptive Pull* algorithm. Observe that in the *Push with Adaptive Pull* algorithm, we do poll less than that in the pure *adaptive pull*. Also notice that the gap between the two plots become smaller as the degree of disconnection increases, for example, the poll message overhead in *Adaptive Pull* is 10% more than that in *Push with Adaptive Pull* at $R_f = 50\%$, it is only 4% more at $R_f = 5\%$. This is due to the fact that the adaptive TTR algorithm used in the *Push with Adaptive Pull* takes into account network dynamics and adjusts to it. However, this adaptiveness is limited due to the fact that it only comes from the change in the number of active connections a peer has, see equation 5.

3. The second parameter we studied is the time between successive disconnection occurrences, I_f. The R_f is fixed at 50%, the D_f is fixed at 2 hours, and I_f values range from 5 seconds to 270 seconds. Intuitively, as I_f increases, there are less disconnections in the network. Thus we expect this set of experiments to give us similar results as the previous one with the R_f parameter. The results

turned out to be consistent with our prediction, therefore we only list the graphs below without further explanations.

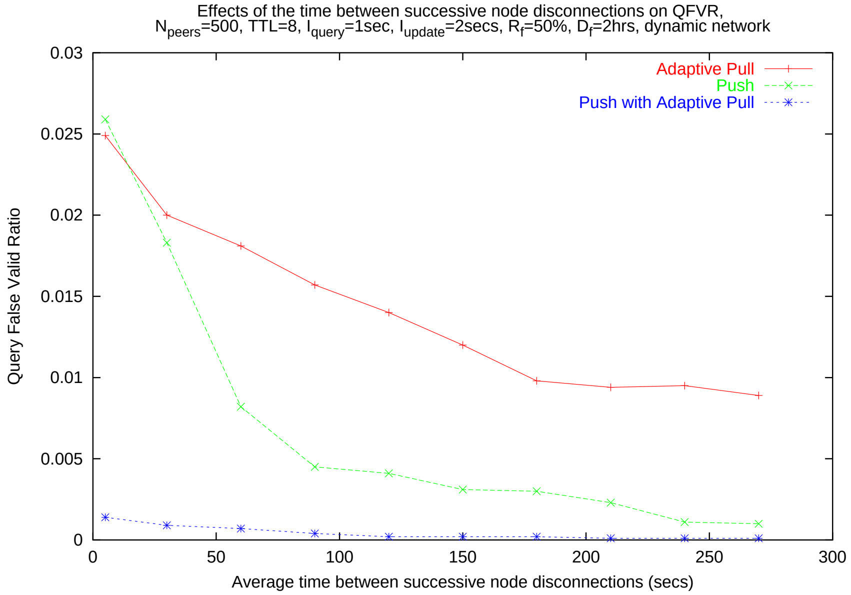

Figure 9.1: Effects of time between successive node disconnections on query false valid ratio

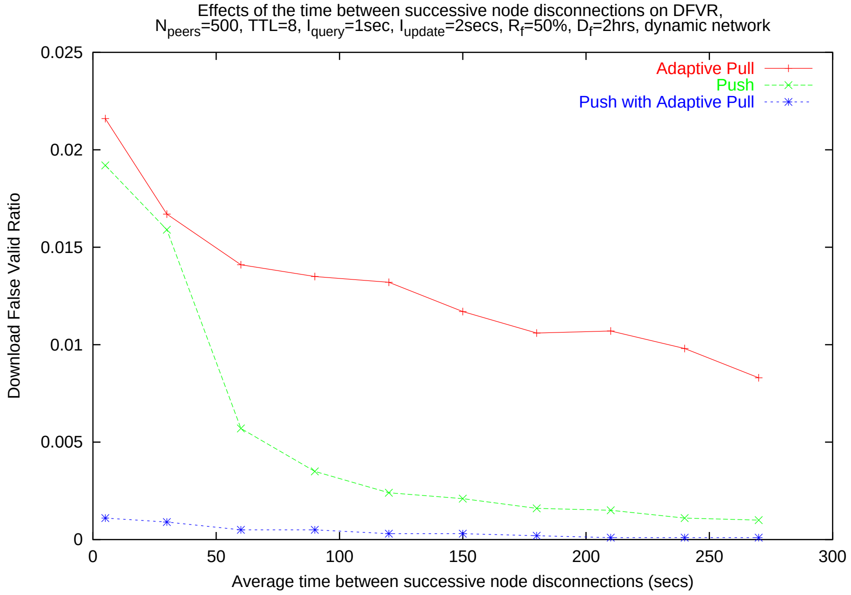

Figure 9.2: Effects of time between successive node disconnections on download false valid ratio

The above two graphs both show that *Push with Adaptive Pull* is the ideal cache consistency algorithm if strong consistency is desired.

The control message overhead of the experiments is shown in Fig 9.3 and 9.4 below.

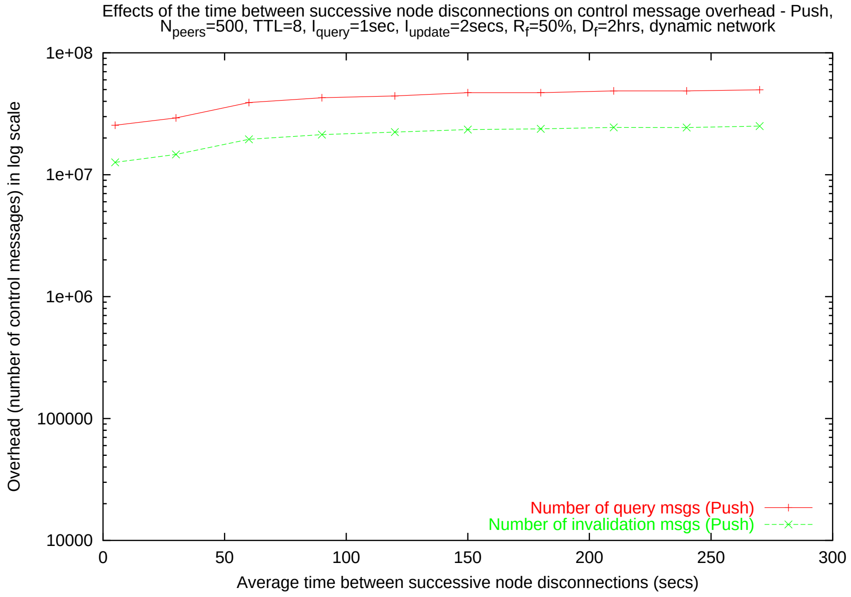

Figure 9.3: Effects of time between successive node disconnections on control message overhead – PUSH

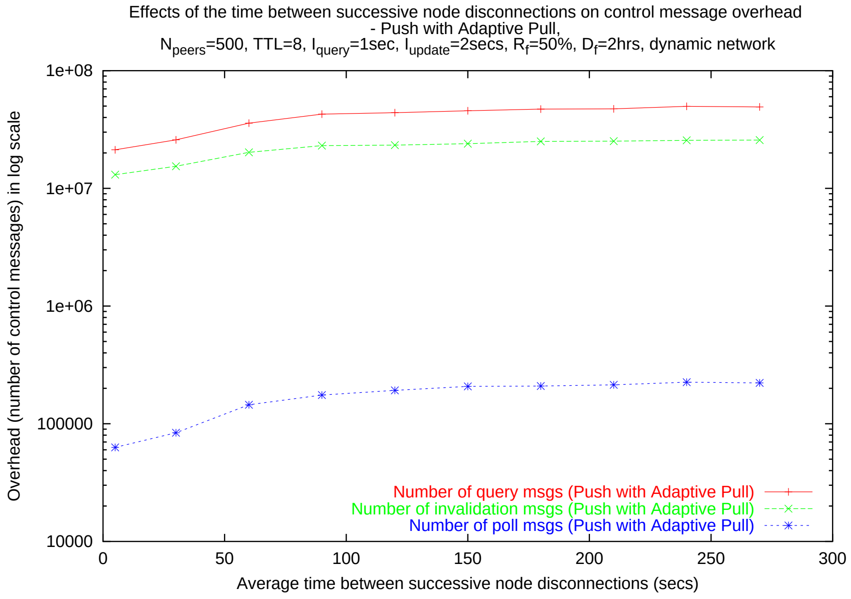

Figure 9.4: Effects of time between successive node disconnections on control message overhead – PAP

The adaptiveness of our TTR algorithm to network conditions is shown in Figure 9.5.

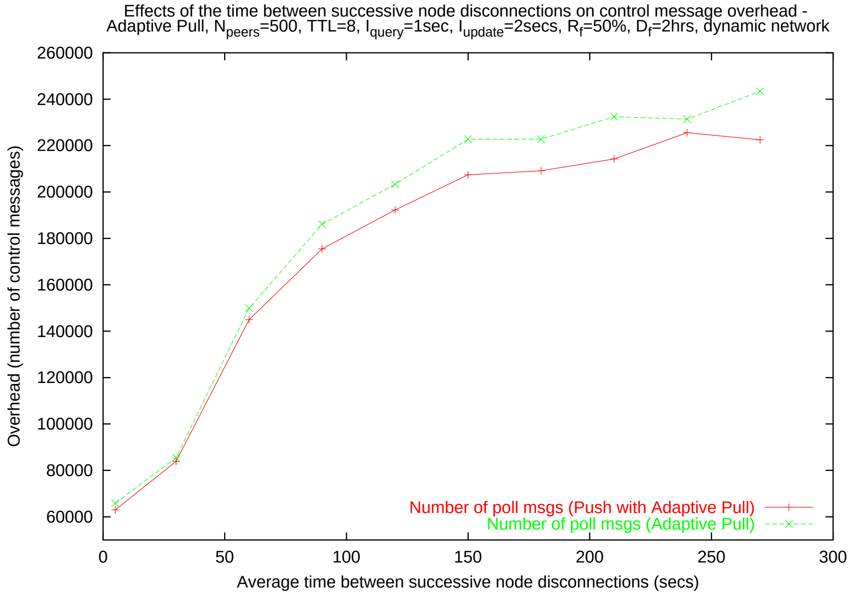

Figure 9.5: Comparison of PULL overhead

6 Summary and Conclusions

In this report we investigated different cache consistency techniques for peer-to-peer file sharing systems. Particularly, we used Gnutella as our target framework and proposed and studied three different cache consistency algorithms: *Push*, *Adaptive Pull*, and *Push with Adaptive Pull*.

We conducted a detailed simulation study on the performance of the proposed algorithms. Our evaluation criteria are fidelity and control message overhead. Below is a list of the observations and our conclusions.

For a stable network environment:

1. *Push* alone achieves almost perfect fidelity when the update rate is lower than query rate. *Push with Adaptive Pull* achieves only 40% better fidelity than pure *Push* when the update rate and query rate are comparable. With the assumption that there are normally much less updates than queries in a Gnutella-style network, we expect *Push* alone to be sufficient in guaranteeing strong consistency. In addition, we didn't count the query reply messages in our simulation. Given low update rates, the invalidation message overhead could be magnitudes lower than the query (and reply) message overhead, for example, the invalidation overhead was 10 times lower when the update rate is $1/10$ of the query rate. It is also worth mentioning that the large invalidation message overhead is due to the intrinsic message passing scheme of Gnutella, it would be improved if a better message routing scheme replaces the existing one.

2. *Adaptive Pull* alone does not guarantee strong consistency. As updates rate decreases by 10 times, *Adaptive Pull* gives 67% to 92% better consistency. The poll message overhead is relatively consistent when the update rate changes and is in general at least one magnitude lower than the invalidation message overhead when using *Push* or *Push with Adaptive Pull*. In applications where the consistency requirements are less stringent, the *Adaptive Pull* algorithm is a good choice.

3. TTL value determines the reach of invalidation messages. The fidelity of replicas decreases as TTL value increases, when the TTL value gets large enough, both *Push* and *push with adaptive pull* achieve almost perfect fidelity. The invalidation message overhead increases almost exponentially as TTL value increases. However, it reaches a plateau after the TTL gets large enough (8 in our simulation). In common Gnutella implementations [12] [13], TTL is set to 7 and it allows the broadcast messages to cover approximately 95% [8] of all the peers in the Gnutella network.

4. Network size affects fidelity when using the *Push* algorithm. We get 5 to 10 times worse fidelity as the network size increases by 10 times. However, when using the *Push with Adaptive Pull* algorithm, we only get 3 to 5 times worse fidelity as the network size increases by 10 times. And more importantly, the fidelity achieved with pure *Push* is 3 to 7 times worse than that with *Push with Adaptive Pull*. We attribute these to the leverage effect of the pulling part of the *Push with Adaptive Pull* algorithm.

5. The average number of active connections each peer maintains is another important parameter in a Gnutella-style network. The larger this value, the more peers each invalidation message gets broadcasted to. The simulation results show that in both *Push* and *Push with Adaptive Pull* algorithms, we get better fidelity as this value increases. We achieve almost perfect fidelity when this value gets large enough (4 in our simulation). However, the *Push with Adaptive Pull* algorithm is more immune to this parameter, for example, it achieves 77% to 88% better fidelity than *Push* when the average number of connections per peer is 1. On the other hand, the invalidation message overhead involved increases exponentially, by approximately 3 magnitudes when this parameter increases from 1 to 4. Therefore in practive, we suggest peers to maintain a moderate number of active connections that helps to achieve good fidelity and at the same time, avoids large overhead.

6. Query rate does not have much effects on fidelity when using the *Push with Adaptive Pull* algorithm. We achieved almost perfect fidelity even when the update rate was 10 times of the query rate.

For a dynamic network environment:

1. A Gnutella application fixes its number of active connections when it falls below some threshhold. This helps maintaining the connectivity of the Gnutella network and thus helps *Push* to achieve better

fidelity. However, using *Push with Adaptive Pull* achieves $> 90\%$ better consistency than using pure *Push*. The topology fix process isn't necessary when using the *Push with Adaptive Pull* algorithm.

2. Both the *Adaptive Pull* and *Push* algorithms achieve worse fidelity when more disconnections occur in the network. *Push* gives 25% to 80% better fidelity than *Adaptive Pull* when there are moderate disconnections, and possibly worse fidelity when the network gets highly disconnected. In general, neither of the two algorithms can guarantee strong consistency in a dynamically changing network.

3. The *Push with Adaptive Pull* algorithm gives good fidelity (with QFVR and DFVR both below 0.002) even when the network gets highly disconnected. So if strong consistency is desired, this is the best algorithm among the three.

4. The invalidation message overhead depends on the update rate. If the update rate is low relative to the query rate, then the invalidation message overhead is also lower than the query message overhead. Both the query and invalidation message overheads decrease linearly as more disconnections occur in the network.

5. The poll message overhead is generally at least one magnitude lower than the invalidation message overhead. Our adaptive TTR algorithm adapts to the changes in the number of active connections of each peer, therefore a peer polls more often when it has fewer connections. This adaptiveness helps the polling to achieve good fidelity even in a highly disconnected network environment.

To sum up, *Push with Adaptive pull* algorithm achieves the stongest consistency. With this algorithm, we achieve very good consistency (for example, both the QFVR and DFVR measured in the simulations are below 0.002) even in highly dynamic network environments. However, this algorithm has the most control message overhead, for example, its control message overhead could be two magnitudes higher than poll message overhead. But we argue that in most Gnutella-style P2P applications, the update rate is generally much lower than the query rate. In this scenario, the invalidation message overhead could be magnitudes lower than the query (and reply) message overhead. On the other hand, *Adaptive Pull* alone provides less strong consistency, but has the advantage of the least control message overhead. Based on these, we conclude that both the *Push with Adaptive Pull* and *Adaptive Pull* algorithms could be good choices for a P2P system, which one to choose depends on the requirements of the application and user demands.

References

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

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

[3] R. Sirnivasan, 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

[4] 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

[5] C. Gary and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202-210, 1989.

[6] 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.

[7] 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.

[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] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrel, A hierarchical Internet Object Cache, *USENIX'96*, January 1996.

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

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

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