Reliable Peer-to-Peer Information Monitoring Through Replication

Buğra Gedik
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332, U.S.A.
bgedik@cc.gatech.edu

Ling Liu
Georgia Institute of Technology
College of Computing
Atlanta, GA 30332, U.S.A.
lingliu@cc.gatech.edu

Abstract

A key challenge in peer-to-peer computing systems is to provide a decentralized and yet reliable service on top of a network of loosely coupled, weakly connected and possibly unreliable peers. This paper presents an effective dynamic passive replication scheme designed to provide reliable service in PeerCQ, a decentralized and self-configurable peer-to-peer Internet information monitoring system. We first describe the design of a distributed replication scheme, which enables reliable processing of long-running information monitoring requests in an environment of inherently unreliable peers. Then we present an analytical model to discuss its fault tolerance properties. A set of initial experiments is reported, showing the feasibility and the effectiveness of the proposed approach.

1 Introduction

Peer-to-peer (P2P) computing is rising as a promising distributed computing paradigm that enables sharing of various resources among a large group of client computers (peers) over the Internet. P2P applications can be classified as pure P2P or hybrid P2P systems depending on whether a central server is used for resource discovery. Pure P2P systems are characterized by their totally decentralized architecture. Although the design of pure P2P systems are more challenging than the design of hybrid P2P systems, they provide several advantages over their hybrid counterparts: Adapting a serverless approach addresses the problem of forming a single point of failure and scalability bottleneck. Moreover, it provides cost effective deployment of services, since no server side infrastructure support is needed. And lastly, P2P systems are self-configuring, requiring no management cost.

Pure P2P systems are facing two main challenges. The first one is *providing a resource discovery mechanism with low communication cost while maintaining decentralization*. The second one is *providing reliable service over a large group of unreliable peers*. Much effort in P2P research have been contributed towards addressing the first problem [16, 23, 3, 17, 1, 22]. It is widely recognized that further deployment of P2P technology in other application domains other than simple file sharing demands practical solutions to the second problem.

Replication is a proven technique for masking component failures. However, designing replication scheme for P2P systems presents a number of its own challenges. First, large scale peer-to-peer systems are confronted with highly dynamic peer turnover rate [19]. For example, in both Napster and Gnutella, half of the peers participating in the system will be replaced by new peers within one hour. Thus, maintaining fault-tolerance in such a highly dynamic environment is critical to the success of a peer-to-peer system. Second, all nodes holding replicas must ensure that the replication invariant (at least a fixed number of copies exist at all time) is maintained. Third but not least, the rate of replication and amount of data stored at each node must be kept at levels that allow for timely replication without causing too much network overhead even when regular nodes join and leave the network.

In this paper we present an effective dynamic passive replication scheme designed to provide reliable service in PeerCQ [7], a decentralized and self-configurable peer-to-peer Internet information monitoring system. Many applications today have the need for tracking changes in multiple information sources on the web and notifying users of changes if some condition over the information sources is met. A typical example in business world is to monitor availability and price information of specific products, such as “monitor the price of 2 mega pixel digital cameras in next two months and notify me when one with price less than 100$ becomes available”, “monitor the weather in Mount Hood resort from October to December and notify me if there is a snow”.

PeerCQ is an Internet scale peer-to-peer information monitoring system where peers of the system are user machines. Due to the P2P nature, PeerCQ has to cope with several dynamic uncertainties: there may be a high rate of unexpected peer failures; peers’ willingness to participate in the network and their online time may vary significantly; peers may disconnect from the network arbitrarily. Due to the information monitoring nature, PeerCQ has to deal with long running tasks. Each information monitoring request can be seen as a standing query which runs continuously, whenever the trigger condition is met (such as the price of 2 mega pixel digital cameras has changed), it will send out notification. This process continues until the specified stop condition is met. As a result, guaranteeing that no information monitoring requests are lost once installed and each request is in process at any given time is not a trivial task.

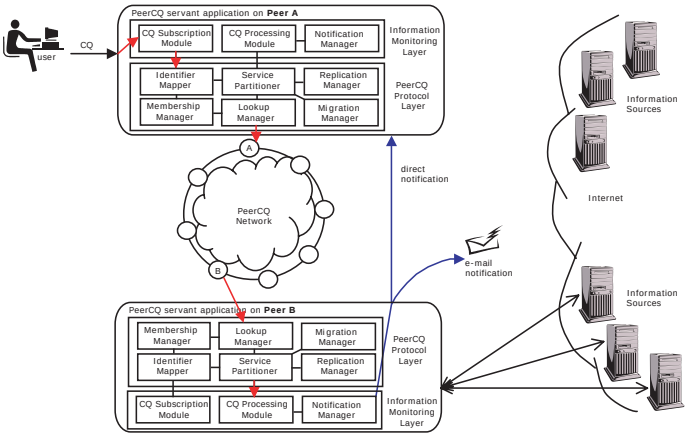

Figure 1: PeerCQ Architecture

The main contribution of this paper is two-fold. First we present the design of a dynamic passive replication scheme, which enables reliable processing of information monitoring requests in an environment of inherently unreliable peers. Second, we present an analytical model to study the fault tolerance properties of the PeerCQ replication scheme. We also report a set of initial experiments, showing the feasibility and the effectiveness of the proposed approach. In the rest of the paper, we first give an overview of the PeerCQ system and the PeerCQ protocol, emphasizing on the components that are directly relevant to the scope of this paper. Later we describe the design of our distributed replication scheme and the analytical modelling of its fault tolerance properties.

2 The PeerCQ System Overview

2.1 System Architecture

Peers in the PeerCQ system are user machines on the Internet that execute information monitoring applications. Peers act both as clients and servers in terms of their roles in serving information monitoring requests. An information-monitoring request is expressed as a Continual Queries (CQ) [14], which monitors changes in Internet data sources that are external to the PeerCQ system. CQs are standing queries that monitor information updates and return results whenever the updates reach certain specified thresholds. There are three main components of a CQ: query, trigger, and stop condition. Whenever the trigger condition becomes true, the query part is executed and the part of the query result that is different from the result of the previous execution is returned. The stop condition specifies the termination of a CQ. It is important to note that information monitoring jobs (CQs) are long running entities and might last days, weeks, even months. It is not acceptable to break a CQ execution and resume it at some arbitrary time or start it over from scratch again. Once started a CQ has to run until its stop condition is reached to maintain its continuous change tracking semantics.

A CQ can be posted by any peer in the system. Once it is posted, it is always in execution at some peer, independent of whether the peer posted it currently participates in the PeerCQ system or not. There is no scheduling node in the system. No peers have any global knowledge about other peers in the system. The decision on which peer to execute a CQ is done by a distributed *service partitioning* scheme (see Section 3.2 for detail), which takes into account several factors like peer resource diversity, load balance between peers and overall system utilization.

There are three main mechanisms that make up the PeerCQ system. The first mechanism is the overlay network membership. Peer membership allows peers to communicate directly with one another to distribute tasks or exchange information. A new node can join the PeerCQ system by contacting an existing peer (an entry node) in the PeerCQ network. There are several bootstrapping methods to determine an entry node. Here we assume that the PeerCQ service has a well-known set of bootstrapping nodes which maintain a short list of PeerCQ nodes that are currently alive in the system.

The second mechanism is the PeerCQ protocol, including the service partitioning and the routing query based service lookup algorithm. In PeerCQ every peer participates in the process of evaluating CQs, and any peer can post a new CQ of its own interest. When a new CQ is posted by a peer, this peer first determines which peer will process this CQ with the objective of utilizing system resources and balancing the load on peers. Upon a peer's entrance into the system, a set of CQs that needs to be redistributed to this new peer is determined by taking into account the same objectives. Similarly, when a peer departs from the system, the set of CQs of which it was responsible is re-assigned to the rest of peers, while maintaining the same objectives – maximize the system utilization and balance the load on peers.

The third mechanism is the processing of information monitoring requests in the form of continual queries (CQs). Each information monitoring request is assigned to an identifier. Based on an identifier matching criteria, CQs are executed at their assigned peers and cleanly migrated to other peers in the presence of failure or peer entrance and departure.

Figure 1 shows a sketch of the PeerCQ system architecture from a user's point of view. Each peer in the P2P network is equipped with the PeerCQ middleware, a two-layer software system. The lower layer is the PeerCQ protocol layer responsible for peer-to-peer communication. The upper layer is the information monitoring subsystem responsible for CQ subscription, trigger evaluation, and change notification. Any domain-specific information monitoring requirements can be incorporated at this layer.

3 The PeerCQ Protocol

The PeerCQ protocol specifies how to find peers that are considered best to serve the given information monitoring requests in terms of load balance and overall system utilization, how new nodes join the system, and how they recover from the failures or departures of existing nodes.

3.1 Overview

An information monitoring request (subscription) is described in terms of a continual query (CQ). A CQ is defined as a quadruple, denoted by $cq : (cq\_id, trigger, query, stop\_cond)$ [14]. cq_id is the unique identifier of the CQ, which is an m-bit unsigned value. $trigger$ defines the target data source to be monitored (mon_src), the data item to be tracked for

changes (mon_item), and the condition that specifies the update threshold (amount of changes) of interest (mon_cond). $query$ part specifies what information should be delivered when the mon_cond is satisfied. $stop\_cond$ specifies the termination condition for the CQ. For notational convenience, in the rest of the paper a CQ is referenced as a tuple of six attributes, namely $cq : (cq\_id, mon\_src, mon\_item, mon\_cond, query, stop\_cond)$.

The PeerCQ system provides a distributed service partitioning and lookup service that allows applications to register, lookup, and remove an information monitoring subscription using an m-bit CQ identifier as a handle. It maps each CQ subscription to a unique, effectively random m-bit CQ identifier. To enable efficient processing of multiple CQs with similar trigger conditions, the CQ-to-identifier mapping also takes into account the similarity of CQs such that CQs of the similar trigger conditions can be assigned to same peers.

Similarly, each peer in PeerCQ corresponds to a set of m-bit identifiers, depending on the amount of resources donated by each peer. A peer that donates more resources is assigned to more identifiers. A peer p is described as a tuple of two attributes, denoted by $p : (\{peer\_ids\}, (peer\_props))$. $peer\_ids$ is a set of m-bit identifiers. No peers share any identifiers, i.e. $\forall p, p' \in P, p.peer\_ids \cap p'.peer\_ids = \emptyset$, where P denotes the set of peers forming the PeerCQ network. $peer\_props$ is a composite attribute which is composed of several peer properties, including IP address of the peer, peer, resources such as connection type, CPU power and memory, and so on.

Identifiers are ordered in an m-bit identifier circle modulo 2^m, which forms a logical ring. The 2^m identifiers are organized in an increasing order in the clockwise direction. To guide the explanation of the PeerCQ protocol, we define a number of notations.

- The distance between two identifiers i, j, denoted as $Dist(i, j)$, is the shortest distance between them on the identifier circle, defined by $Dist(i, j) = min(|i - j|, 2^m - |i - j|)$.
- Let $path(i, j)$ denote the set of all identifiers on the clockwise path from identifier i to identifier j on the identifier circle. An identifier k is said to be *in-between* identifiers i and j, denoted as $k \in path(i, j)$, if $k \neq i$, $k \neq j$ and k can be reached before j going in the clockwise path starting at i.
- A peer p' with its peer identifier j is said to be an *immediate right neighbor* to a peer p with its peer identifier i, denoted by $(p', j) = IRN(p, i)$, if there are no other peers having identifiers in the clockwise path from i to j on the identifier circle. Formally the following condition holds: $i \in p.peer\_ids \wedge j \in p'.peer\_ids \wedge \nexists p'' \in P$ s.t. $\exists k \in p''.peer\_ids$ s.t. $k \in path(i, j)$. The peer p with its peer identifier i is referred to as the *immediate left neighbor*(ILN) of peer p' with its identifier j.
- A *neighbor list* of a peer p_0 associated with one of its identifiers i_0, denoted as $NeighborList(p_0, i_0)$, is formally defined as follows: $NeighborList(p_0, i_0) = [(p_{-r}, i_{-r}), \ldots, (p_{-1}, i_{-1}), (p_0, i_0), (p_1, i_1), \ldots, (p_r, i_r)]$, s.t. $\bigwedge_{k=1}^{r}((p_k, i_k) = IRN(p_{k-1}, i_{k-1})) \wedge \bigwedge_{k=1}^{r}(p_{-k}, i_{-k}) = ILN(p_{-k+1}, i_{-k+1})$. The size of the neighbor list is $2r + 1$ and we call r the neighbor list parameter.

3.2 Service Partitioning

Service partitioning can be described as the assignment of CQs to peers. In PeerCQ, this assignment is based on a matching algorithm defined between CQs and peers, derived from a relationship between CQ identifiers and peer identifiers. The PeerCQ service partitioning scheme can be characterized by the careful design of the mappings for generating CQ identifiers and peer identifiers, and the two-phase matching defined between CQs and peers.

In the *Strict Matching* phase, a simple matching criterion, similar to the one defined in Consistent Hashing [11], is used. In the *Relaxed Matching* phase, an extension to strict matching is applied to relax the matching criteria to include application semantics.

3.2.1 Strict Matching

The idea of strict matching is to assign a CQ to a peer such that the chosen peer has a peer identifier that is numerically closest to the CQ identifier among all peer identifiers on the identifier circle. Formally, strict matching can be defined as follows: The function $strict\_match(cq)$ returns a peer p with identifier j, denoted by a pair (p, j), if and only if the following condition holds:

$$strict\_match(cq) = (p, j), \text{ where } j \in p.peer\_ids \wedge \forall p' \in P, \forall k \in p'.peer\_ids,\ Dist(j, cq.cq\_id) \leq Dist(k, cq.cq\_id)$$

Peer p is called the *owner* of the cq. This matching is strict in the sense that it does not change unless the set of peers in the network changes. To guide the understanding of the strict matching algorithm, we first describe how CQ identifiers and peer identifiers are generated.

Mapping peers to identifiers

In PeerCQ a peer is mapped to a set of m-bit identifiers, called the peer's identifier set ($peer\_ids$). m is a system parameter and it should be large enough to ensure that no two nodes share an identifier or this probability is negligible. To balance the load of peers with heterogeneous resource donations when distributing CQs to peers, the peers that donate more resources are assigned more peer identifiers, so that the probability that more CQs will be matched to those peers is higher.

The number of identifiers to which a peer is mapped is calculated based on a peer donation scheme. We introduce the concept of *ED* (effective donation) for each peer in the PeerCQ network. *ED* of a peer is a measure of its donated resources effectively perceived by the PeerCQ system. For each peer, an effective donation value is first calculated and later used to determine the number of identifiers that peer is going to be mapped. The calculation of *ED* is described in [7]. The mapping of a peer to peer identifiers needs to be as uniform as possible. This can be achieved via feeding the base hashing functions like MD5 or SHA1 by node specific information, like the IP address of the peer.

Mapping CQs to identifiers
This mapping is intended to map CQs with similar trigger conditions to the same peers as much as possible, in order to achieve higher overall utilization of the system. Two CQs, cq and cq', are considered *similar* if they are interested in monitoring updates on the same item from the same source, i.e. $cq.mon\_src = cq'.mon\_src \wedge cq.mon\_item = cq'.mon\_item$.

A CQ identifier is composed of two parts. The first part is expected to be identical for similar CQs and the second part is expected to be uniformly random. This mechanism allows similar CQs to be mapped into a contiguous region on the m-bit identifier circle. The length of a CQ identifier is m. The length of the first part of an m-bit CQ identifier is a, which is a system parameter called *grouping factor*. The first part of the CQ identifier is generated by hashing the concatenation of the data source and the item of interest being monitored, again by using a message digest function. The second part of the CQ identifier is generated in a similar way to a peer identifier.

According to the parameter a (grouping factor), the identifier circle is divided into 2^a contiguous regions. The CQ-to-identifier mapping implements the idea of assigning similar CQs to the same peers by mapping them to a point inside a contiguous region on the identifier circle. Introducing smaller regions (i.e., the grouping factor a is larger) increases the probability that two similar CQs are matched to the same peer. Taking into account the non-uniform nature of the monitoring requests, there is a trade-off between reducing redundancy in CQ evaluation and forming hot-spots. Thus, the grouping factor should be chosen carefully.

3.2.2 Relaxed matching

The goal of Relaxed Matching is to fine tune the performance of PeerCQ service partitioning by incorporating additional characteristics of the information monitoring applications. Concretely, in the Relaxed Matching phase, the assignments of CQs to peers are revised to take into account factors such as the network proximity of peers to remote data sources, whether the information to be monitored is in the peer's cache, and how peers are currently loaded. By taking into account the network proximity between the peer responsible of executing a CQ and the remote data source being monitored by this CQ, the utilization of the network resources is improved. By considering the current load of peers and whether the information to be monitored is already in the cache, one can further improve the system utilization.

The idea behind the relaxed matching is as follows: The peer that is matched to a given CQ according to the strict matching, i.e. the owner of the CQ, has the opportunity to query its neighbors to see whether there exists a peer that is better suited to process the CQ in terms of the three additional factors described above. In case such a neighbor exists, the owner peer will assign this CQ to one of its neighbors for execution. We call the neighbor chosen for this purpose the *executor* of the CQ. The three factors used for deciding on a CQ executor are combined into a single value by a *utility function*. This function is computed by each possible CQ executor and the owner peer is responsible for selecting the peer that has the highest utility function value as the executor.

As it will be described in Section 4, the CQ owner should replicate each CQ in order to mask failures. As a result the selection of executor peer for a CQ is a replica selection problem. Let us define $ReplicationList(cq)$ as the set of peers which hold a replica for cq and as a result are considered for being the executor peer for cq. And let $UtilityF(p, cq)$ be the utility function that assigns a utility value by considering p as the executor peer of cq. Then the function $relaxed\_match(cq)$ is formally defined as follows. It returns a peer identifier pair (p, i) if and only if the following condition holds:

$$relaxed\_match(cq) = (p, i), \text{ where } (p, i) \in ReplicationList(cq) \wedge \forall (p'', k) \in ReplicationList(cq),\ UtilityF(p, cq) \geq UtilityF(p'', cq)$$

3.3 PeerCQ P2P Service Lookup

The PeerCQ service lookup implements the two matchings described so far. Given a CQ (forming a lookup query), lookup is able to locate its *owner* and *executor* using only $O(\log N)$ messages in a totally decentralized environment, where N is the number of peers. PeerCQ's lookup operation is described in [7] and is based on identifier routing, like several distributed hash table based lookup approaches introduced in the literature [22, 17, 23].

The lookup is performed by routing the lookup queries towards their destination peers using routing information maintained at each peer. The routing information consists of a *routing table* and a *neighbor list* for each identifier possessed by a peer. The routing table is used to locate a peer that is more likely to answer the lookup query, where a neighbor list is used to locate the owner peer of the CQ and the executor peer of the CQ. The routing table is basically a table containing information about several peers in the network together with their identifiers. The structure of the neighbor list is already described in Section 3.1. A naive way of answering a lookup query is to iterate on the identifier circle using only neighbor lists until the matching is satisfied. The routing tables are simply used to speed up this process. Initialization and maintenance of the routing tables and the neighbor lists do not require any global knowledge.

In this section we will lastly mention that, communicating with a peer p', that is in one of the neighbor lists of peer p, is a localized operation for p; however a service lookup implementing the described matchings is an operation that requires several messages to be exchanged between peers.

4 Reliability in PeerCQ

4.1 Departures and Failures

A *proper departure* in PeerCQ is a volunteer disconnection of a peer from the PeerCQ network. During a proper departure, the PeerCQ P2P protocol updates its routing information. In addition to this, if there is no CQ replication mechanism employed in the system, the PeerCQ application on the departing peer migrates its currently owned CQs before departing. Such a scenario is very vulnerable to failures. A *failure* in PeerCQ is a disconnection of a peer from the PeerCQ network without notifying the system. This can happen due to a network problem, computer crash or improper program termination. Failures are assumed to be detectable (a fail-stop assumption), and are captured by the

PeerCQ P2P protocol's neighbor list polling mechanisms. However, in order to recover the lost CQs a replication mechanism is needed. Notice that once there is a replication mechanism, which will enable the continuation of the CQ executions from the replicated copies, then the proper departures are very similar to failures in terms of the action that needs to be taken. This will enable the elimination of the explicit CQ migration process during departures. The main difference between a proper departure and a failure is that, a properly departing peer will explicitly notify other peers of its departure. In the rest of the paper we use the term departure to mean either proper departure or failure.

4.2 PeerCQ Replication Scheme

In order to ensure smooth CQ execution and to prevent failures interrupting CQ processing and threatening CQ durability, we need to replicate each CQ. In addition to replicating a CQ, some execution state of the CQ has to be also replicated together with it and updated as needed, in order to enable correct continuation of the CQ execution after a departure. (This is discussed in more detail in Section 4.3.3.) Since CQs should be available any time for processing, PeerCQ requires a strong replication mechanism.

The PeerCQ replication scheme has to be dynamic. This means that at any time each CQ should have certain number of replicas available in the system, and this property should be maintained dynamically as the peers enter and exit the system. As a result, our replication consists of two phases. In the first phase a CQ is replicated to a certain number of other peers. This phase happens immediately after a CQ is injected into the system. In the second phase the number of replicas existing in the system is kept constant, and the existing replicas are kept consistent. The second phase is called the *replica management* phase and lasts until the CQ is explicitly removed from the system or the CQ's termination condition is met.

One important decision at this point is where to replicate CQs. In order to preserve correctness of the location mechanism which will also enable the update and deletion of CQs, and to preserve good load-balance we select the peers that will contain the replicas of a CQ from the peers inside the neighbor list of its owner peer. Moreover, choosing these peers from the neighbor list localizes the replication process (no search is required for locating peers that will hold the replicas), which is an advantage in a totally decentralized system. Furthermore, peers that are neighbors on the identifier circle are not necessarily close to each other geographically, which decreases the probability of collective failures. We describe PeerCQ replication formally as follows: A CQ, denoted as cq, is replicated at peers contained in the set:

$ReplicationList(cq) =$

$[(p_{-\lfloor rf/2 \rfloor}, i_{-\lfloor rf/2 \rfloor}), \ldots, (p_{-1}, i_{-1}), (p_0, i_0), (p_1, i_1), \ldots, i_{\lceil rf/2 \rceil})]$, where

$\bigwedge_{k=1}^{\lceil rf/2 \rceil} p_{i_k} = IRN(p_{k-1}, i_{k-1}) \wedge \bigwedge_{k=1}^{\lfloor rf/2 \rfloor} p_{i_{-k}} = ILN(p_{-k+1}, i_{-k+1}) \wedge$

$(p_0, i_0) = strict\_match(cq)$

This set is called the *replication list*, and is denoted as $ReplicationList(cq)$. Size of the replication list is $rf + 1$, where rf is called the *replication factor*. Replication list size should be smaller than or equal to the neighbor list size ($rf + 1 \leq 2 * r + 1$) to maintain the property that replication is a localized operation, i.e. $ReplicationList(cq) \subset NeighborList(p, i)$, where $(p, i) = strict\_match(cq)$. Figure 2 illustrates the described replication scheme with $rf = 4$.

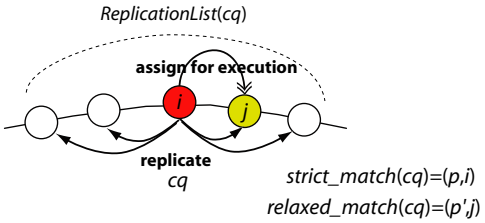

Figure 2: CQ replication with $rf = 4$

The job of dealing with replica management of a CQ is the responsibility of the CQ's owner in PeerCQ. Owner of CQs change as peers enter or depart. Moreover an ownership change possibly causes an executor change, since the relaxed matching is defined on top of strict matching. As a result, our discussion of PeerCQ replica management includes a detailed discussion of CQ owner and executor changes (See Section 4.3). The PeerCQ replication scheme is similar to the well known primary/backup approach [2] (also called passive replication) with the executor acting as the 'primary server' and the peers holding replicas as the 'backup servers' with respect to a CQ. However there are two main differences:

- There is an important difference between the executor and the owner of a CQ. The executor of a CQ is simply the peer that is assigned by the CQ owner to execute the CQ. On the other hand, the owner of a CQ is responsible for the replication of the CQ and the selection of the CQ executor, in the presence of node joins and departures.
- The executor of a CQ do not only changes when the executor peer departs. It can also change when the set of replica holders change. This is done in order to improve load balance and/or to make better use of the system resources. We deal with the problem of selecting executor peers in Section 4.2.2.

4.2.1 Fault Tolerance

Given the description of the PeerCQ replication scheme, we define two different kinds of events that result in loosing CQs. One is the case where the existing peers that are present in the system are not able to hold (either for replication or for execution) any more CQs due to their heavy load. There is nothing to be done for this if the system is balanced in terms of peer loads. Because this indicates insufficient number of peers present in the system. The other case is when all replica holders of a CQ (or CQs) fail in a short time interval, not letting the dynamic replica management algorithm to finish its execution. We call this time interval the *recovery time*, denoted by Δt_r. We call the event of having all peers contained in a replication list fail within the interval Δt_r, a *deadly failure*. We first analyze the cases where we have deadly failures and then give an approximation for the probability of having a deadly failure due to a peer's departure. We assume that peers depart by failing with probability pf and the time each peer stays in the network, called the *service time*, is exponentially distributed with mean st.

Let us denote the subset of CQs owned by a peer p that satisfies $strict\_match(cq) = (p, i)$ as $O_{p,i}$. Let $RL_{p,i}(t)$ be the set of peers in the replication list of CQs in $O_{p,i}$ at time t. Assume that the peer p fails right after time t_a. Then $RL_{p,i}(t_a)$ consists of the peers that are assumed to be holding replicas of CQs in $O_{p,i}$ at time t_a. Let us denote the time of the latest peer failure in $RL_{p,i}(t_a)$ as t_l and the length of the shortest time interval which covers the failure of peers in $RL_{p,i}(t_a)$ as Δt where $\Delta t = t_l - t_a$. If Δt is not sufficient enough, i.e. $\Delta t < \Delta t_r$, then p's failure at time t_a together with the failures of other peers in $RL_{p,i}(t_a)$ will cause a deadly failure. This will result in loosing some or all CQs in $O_{p,i}$.

Let $Pr_{df}(p)$ denote the probability of a peer p's departure to result in a deadly failure. Then we define $Pr_{df}(p, i)$ as: $Pr_{df}(p, i) = Pr\{$All peers in $RL_{p,i}(t)$ has failed within a time interval $< \Delta t_r$, where p failed at time $t\}$

Then $Pr_{df}(p)$ can be formulated as:

$$Pr_{df}(p) = 1 - \prod_{i \in p.peer\_ids} (1 - Pr_{df}(p, i))$$

If we assume $\bigcap_{i \in p.peer\_ids} RL_{p,i}(t) = p$, then $\forall_{i,j \in p.peer\_ids}\ Pr_{df}(p, i) = Pr_{df}(p, j)$. Then we have:

$$Pr_{df}(p) = 1 - (1 - Pr_{df}(p, i))^{p.identifier\_count} \quad (1)$$

Let t_0 denote a time instance at which all peers in $RL_{p,i}(t)$ was alive. Furthermore let us denote the amount of time each peer in $RL_{p,i}(t)$ stayed in the network since t_0 as random variables $A_1, \ldots, A_{rf+1}$. Due to the memorylessness property of the exponential distribution, $A_1, \ldots, A_{rf+1}$ are still exponentially distributed with $\lambda = 1/st$. Then, we have:

$$Pr_{df}(p, i) = pf^{rf+1} * Pr\{MAX(A_1, \ldots, A_{rf+1}) - MIN(A_1, \ldots, A_{rf+1}) < \Delta t_r\}$$
$$Pr_{df}(p, i) = pf^{rf+1} * Pr\{MAX(A_1, \ldots, A_{rf}) < \Delta t_r\}$$
$$Pr_{df}(p, i) = pf^{rf+1} * \prod_{i=1}^{rf} Pr\{A_i < \Delta t_r\}$$

Then we have:

$$Pr_{df}(p, i) = pf^{rf+1} * \prod_{i=1}^{rf} (1 - e^{-\Delta t_r / st}) \quad (2)$$

Equations 1 and 2 are combined to give the following equation:

$$Pr_{df}(p) = 1 - \left(1 - pf^{rf+1} * \prod_{i=1}^{rf} (1 - e^{-\Delta t_r / st})\right)^{p.identifier\_count}$$

In a setup where $rf = 4$, $pf = 0.1$, $\Delta t_r = 30$secs and $st = 60$mins, $Pr_{df}(p)$ turns out to be $\simeq 2.37 * 10^{-13}$ where $p.identifier\_count = 5$. However when we set rf = 2, $\Delta t_r = 120$secs and $st = 30$mins, $Pr_{df}(p)$ becomes $\simeq 2 * 10^{-5}$, which can be considered as an unsafe value in PeerCQ context. We further investigate the effects of failures in PeerCQ, through a simulation study in Section 5. Note that the greater the replication factor rf is, the lower the probability of loosing CQs. However having a greater replication factor increases the cost of managing the replicas, which we will explore in more detail in Section 4.3.4.

4.2.2 Replica Selection Policy

In this section we describe the details of the replica selection policy used in PeerCQ for deciding the executor peer of a CQ. Recall Section 3.2.2 that the utility function that is used to evaluate the replica holders' suitability for executing a CQ was composed of three factors, namely cache affinity factor, peer load factor and data source distance factor. We define each of these factors as follows:

Let p denote a peer and cq denote the CQ considered to be assigned to p.

Cache affinity factor is denoted as $CAF(p, cq)$. It is a measure of the affinity of a CQ to execute at a peer p with a given cache. It is defined as follows:

$$CAF(p, cq) = \begin{cases} 1 & \text{if } cq.mon\_item \text{ is in } p.peer\_props.cache \\ 0 & \text{otherwise} \end{cases}$$

Peer load factor is denoted as $PLF(p)$. It is a measure of a peer p's willingness to accept one more CQ to execute considering its current load. It is defined as follows:

$$PLF(p) = \begin{cases} 1 & \text{if } p.peer\_props.load \leq tresh * \text{MAX\_LOAD} \\ 1 - \frac{p.peer\_props.load}{\text{MAX\_LOAD}} & \text{if } p.peer\_props.load > tresh * \text{MAX\_LOAD} \end{cases}$$

Data source distance factor is denoted as $SDF(cq, p)$. It is a measure of the network proximity of the peer p to the data source of the CQ specified by identifier cq. SDF is defined as follows:

$$SDF(cq, p) = \frac{1}{ping\_time(cq.mon\_src, p.peer\_props.IP)}$$

Let $UtilityF(p, cq)$ denote the utility function, which returns a utility value for assigning cq to peer p, calculated based on the three measures given above:

$$UtilityF(p, cq) = PLF(p.peer\_props.load) * (CAF(p.peer\_props.cache, cq.mon\_item) + \alpha * SDF(p.peer\_props.IP, cq.mon\_src))$$

Note that the peer load factor PLF is multiplied with the sum of cache affinity factor CAF and the data source distance factor SDF. This gives more importance to the peer load factor. For instance a peer which has a cache ready for the CQ, and is also very close to the data source will not selected to execute the CQ if it is heavily loaded. α is used as a constant to adjust the importance of data source distance factor with respect to cache affinity factor. For instance a newly entered peer, which does not have a cache ready for the given CQ but is much more closer to the data source being monitored by the CQ, can be assigned to execute the CQ depending on the importance of SDF relative to CAF as adjusted by the α value. In case the data source of the CQ at hand is not reachable from the peer, the value of the $UtilityF$ function is set to zero.

The three factors used for defining the utility function influence the average CQ processing cost of a peer, the average network cost for a peer to fetch data items from data sources in order to process its assigned CQs, and the balance in CQ processing costs for peers. We investigate these effects in conjunction with the effect of changing replication factor rf through a simulation study in Section 5.

4.3 Replica Management

In this section we explain how the described dynamic replication scheme is maintained as the peers enter into and depart (either properly or by failure) from the system. For brevity, in the following discussions we assume that each peer has only one identifier. The case of peers with multiple identifiers is very similar, except that the events we describe happens for all neighbor lists of a peer associated with its peer identifiers. Furthermore, our discussion will be based on the case where $rf = 2 * r$. In other words, we consider a special case of the replication scheme, in which every CQ assigned to a peer p according to strict matching is replicated to all peers in p's neighbor list.

When a CQ is posted to the system, it is replicated to appropriate peers (the ones in its owner's neighbor list) as soon as the owner of the CQ is determined. The dynamic replication scheme is maintained by reacting to entering and departing peers. PeerCQ P2P protocol is responsible for detecting entering and departing peers and presenting a consistent view of the neighbor list to the upper level modules dealing with replica management. It achieves this by serializing events and reporting them as atomic changes on the neighbor list. When a change that affects the neighbor list of a peer occurs, the PeerCQ P2P protocol notifies the upper level through calling a function in the form of $neighborListChange(Peer\ p', Peer\ p'', Reason\ r, NeighborList\ nl)$. In this function r is the reason of this neighbor list change, which can be a peer departure or a peer entrance. A departure is either a proper departure, or it is due to failure. p' and p'' are, the peer that is added into the neighbor list nl and the peer that is removed from the neighbor list nl respectively. If the reason of this change is a peer departure then p'' is the departed peer. Similarly, if the reason of this change is a peer entrance then p' is the newly entered peer. In the following subsections we describe how PeerCQ reacts these changes to maintain reliable CQ processing.

4.3.1 Peer Departure

A peer's departure from the system causes several neighbor lists to change ($2r$ of them). A peer, say p, upon receiving a notification from the PeerCQ protocol that its neighbor list has changed due to a peer's departure, say p', performs the following actions:

1. p checks its replica store to see if there exist CQs that were owned by p' and should be owned by itself after p''s departure. If so, p notifies its neighbors regarding this ownership change, selects peers for executing these CQs using relaxed matching with its updated neighbor list, and sends these CQs to the selected peers for execution. It also notifies the previous executors of these CQs so that they can stop executing these CQs. Note that the whole step can happen only for immediate left and right neighbors of p'. This is due to the definition of strict matching.

2. p sends all CQs that it owns, to the new peer p'' that has entered into its neighbor list, as replicas.

3. p redetermines executors of its owned CQs based on relaxed matching. For CQs whose executors has changed, p ships them to their new executors, notifying the old ones. This is required due to the fact that p's neighbor list has changed (p' is removed and p'' is added), which might cause a change in the relaxed matching.

Note that these steps require no support from the departing peer, so that it doesn't matter whether we are dealing with a failing peer or a properly departing peer. However departing peers will coordinate their departs in order to prevent simultaneous departures within a neighbor list.

4.3.2 Peer Entrance

A peer's entrance into the system causes several neighbor lists to change ($2r$ of them). An already participating peer, say p, upon receiving a notification from the PeerCQ protocol that its neighbor list has changed due to a new peer entrance, say p', performs the following actions:

1. p checks whether it owns any CQs that should be owned by p' after p''s entrance. If so p sends these CQs to p' and notifies its neighbors regarding this ownership change. Note that the whole step can happen only for immediate left and right neighbors of p'.

2. p sends all CQs that it owns to p' as replicas.

3. p checks whether it is executing any CQs that are owned by the peer p'' that is removed from its neighbor list. If so, p stops executing these CQs. p also drops all replicas it stores that are owned by p''.

4. p redetermines executors of its owned CQs based on relaxed matching. For CQs whose executors has changed, p ships them to their new executors, notifying the old ones.

4.3.3 Updating Replicas

The problem of updating replicas arises due to the fact that flawless resumption of CQ processing after a CQ changes execution place, requires access to some maintained execution state. For instance, let us consider one of the queries presented before: "monitor the weather in Mount Hood resort from October to December and notify me if there is a snow". Let us assume that the peer that were executing this CQ has detected that it is snowing in Mount Hood, and has notified the user with the new weather condition. The peer also has to store the weather condition so that, the next time it evaluates the CQ it can determine whether an interested event (snow in this case) is a new one or not, in order to prevent redundant notifications. As a result, there exists some amount of state associated with the execution of CQs, such that changes on this state has to be propagated to the CQ replicas so that after a departure or failure of the peer executing a CQ, the new owner can continue the CQ processing without any problem.

In PeerCQ, changes on the state associated with CQs are immediately propagated to replicas as updates. Whenever an executor peer of a CQ updates the CQ related state that has to be propagated to replicas, it notifies the CQ owner, which in turn sends update notifications to all peers containing replicas (which are contained in the replication list of the CQ owner).

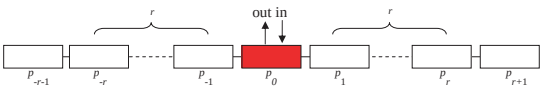

Figure 3: Peer departure and entrance

4.3.4 Analysis of Departure and Entrance Cost

In this section we analyze the cost of maintaining CQ replicas in PeerCQ by considering the number of CQ ownership changes, number of CQ execution place changes and number of CQs that has to be additionally replicated as peers enter and depart from the system. Figure 3 shows the scenario that will be used for the discussion, where peer p_0 enters into or departs from the network. Before analyzing departure and entrance costs, we formalize our setup (assuming each peer has one identifier) as follows to help the discussion:

Γ denotes the set of all CQs. $I(p)$ denotes the CQs owned by peer p. $E(p)$ denotes the CQs executed at peer p. $R(p)$ denotes the CQs replicated at peer p. $NL(p)$ denotes the peers in the neighbor list of peer p.

Properties that has to be kept consistent as peers enter into and depart from the system are:

1. Ownership: $\forall p, I(p) = \{cq : p = strict\_match(cq) \wedge cq \in \Gamma\}$
2. Execution: $\forall p, E(p) = \{cq : p = relaxed\_match(cq) \wedge cq \in \Gamma\}$
3. Replication: $\forall p, R(p) = \bigcup_{p \in NL(p)} I(p)$

Here we list the results obtained from our analysis. Interested reader may see our technical report [6] for the proofs.

Departure analysis:	
avg. # of ownership changes	$\lvert I(p_0)\rvert$
avg. # of exec. place changes	$\lvert E(p_0)\rvert + \frac{1}{2r+1} \sum_{p \in NL(p_0)} \lvert I(p) \backslash E(p_0)\rvert$
avg. # of new replications	$\sum_{p \in NL(p_0)} \lvert I(p)\rvert$
Entrance analysis:	
avg. # of ownership changes	$\frac{1}{3}(\lvert I(p_1)\rvert + \lvert I(p_{-1})\rvert)$
avg. # of exec. place changes	$\frac{1}{2r+1}(1 + \frac{2r}{2r+1}) \sum_{j \in [-r..r], j \neq 0} \lvert I(p_j)\rvert$
avg. # of new replications	$\frac{2}{3}(\lvert I(p_1)\rvert + \lvert I(p_{-1})\rvert) + \sum_{j=2}^{r} \lvert I(p_j)\rvert + \lvert I(p_{-j})\rvert$

If we assume that for any peer p, $I(p) = k$, $E(p) = k$, and $I(p) \bigcap E(p) = \frac{1}{2r+1}k$, then we can roughly approximate the costs for either entrance or departure as, average # of ownership changes $\simeq k$, average # of execution place changes $\simeq k$, average # of new item replications $\simeq (2r+1)k$. (or $(rf+1)k$ if we use a replication list different than the neighbor list) This shows that the number of execution place changes and the number of ownership changes for peer entrance or departure does not increase with the increasing replication factor. However, the number of new item replications that has to be performed as peers enter and exit, increases linearly with the replication factor. But when we consider the average # of new replications *per peer* in case of an entrance or departure, it is still $\simeq k$. On the other hand the total number of replicas a peer holds on average is $|R(p)| \simeq k * (rf+1)$, which increases linearly with rf. This implies that both the storage requirements for keeping replicas and the cost associated with keeping them consistent linearly increases with rf. As a result the cost associated with the replication of CQs is a limitation in increasing rf.

5 Experimental Results

We have designed a simulator that implements the mechanisms explained in this paper. In the following subsections we investigate two main subjects using results obtained from experiments

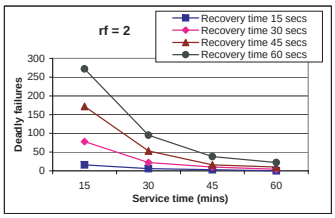

Figure 4: Deadly failures, $rf = 2$

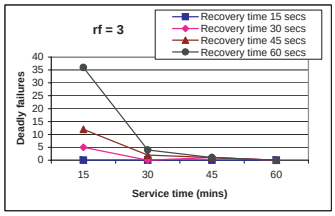

Figure 5: Deadly failures, $rf = 3$

carried out on our simulator. We first study the CQ availability in the existence of peer failures. Then we study the effect of replica selection policy on several performance measures. Some of the parameters used for the simulator in these experiments are: N, total number of peers; K, total number of CQs; rf, replication factor; a, grouping factor; Δt_r, recovery time; st, average service time. We do not experiment with the grouping factor a in our simulations. The value of the grouping factor is set to optimal values that are obtained from our previous research [7].

5.1 CQ Availability under Peer Failure

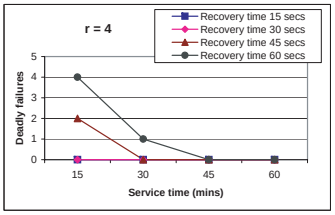

Figure 6: Deadly failures, $rf = 4$

An important measure for evaluating our proposed replication scheme for providing reliable CQ processing in PeerCQ is the CQ availability under various failure scenarios. In this section we present our simulation results on how the proposed replication scheme increases the reliability of the PeerCQ system by increasing CQ availability.

One of the situations that is rather crucial for the PeerCQ system is the case where the peers are continuously leaving the system without any peers entering; or the peer entrance rate is too low when compared to the peer departure rate, so that the number of peers present in the system decreases rapidly. Although we don't expect this kind of trend to continue for a long period, it can happen temporarily. In order to observe the worst case, we have setup our simulation so that the system starts with $2 * 10^4$ peers and 10^6 CQs and each peer departs the system by failing after certain amount of time. The time each peer stays in the system is taken as exponentially distributed with mean equal to 30 mins, i.e. $st = 30$ mins. It is clear that in such a scenario the system will die loosing all CQs, since all peers will depart eventually. However, we want to observe the behavior with different rf values under a worst case scenario to see how gracefully the system degrades for different replication factors.

The graphs in Figures 4, 5 and 6 plot the total number of deadly failures that have occurred during the whole simulation for different mean service times (st), recovery times (Δt_r), and replication factors (rf). These graphs show that the number of deadly failures is smaller when the replication factor is larger, the recovery time is smaller and the mean service time is longer. Note that our simulation represents a worse scenario, where every peer leaves the system by a failure and no peer enters into the system. However, a replication factor of 4 presents a very small number of or even no deadly failures.

These experiments show that, although the cost of replication

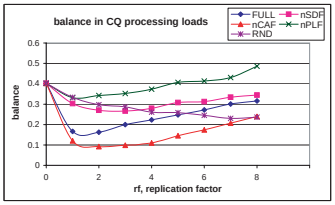

Figure 7: Balance in CQ processing loads for different utility functions.

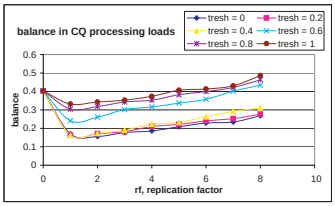

Figure 8: Balance in CQ processing loads for different threshold values used in PLF.

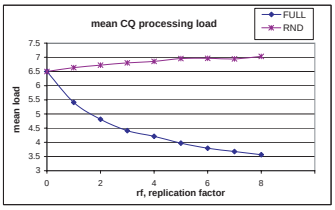

Figure 9: Mean CQ processing load for different utility functions.

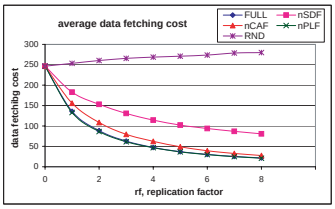

Figure 10: Data fetching cost as a function of rf for different utility functions.

grows with the increasing replication factor as described in Section 4.3.4, the dynamic replication provided by PeerCQ is able to achieve reasonable reliability with moderate values for the replication factor.

5.2 Effect of Replica Selection Policy

The replica selection policy, which is characterized by the utility function used in selecting CQ executors, has influence on several performance measures. In this section we examine the effect of each individual component of the utility function on some of these measures.

In all of the experiments in this section, the number of peers in the system is set to 10^4 and the number of CQs is set to 10^6. There are $5 * 10^3$ data sources and 10 data items on each data source. The information monitoring interests of CQs follow a normal distribution unless otherwise stated.

One measure we are interested in studying is the balance in CQ processing loads of peers. CQ processing load of a peer is the normalized cost of processing the CQs assigned to it for execution. The cost of processing all CQs assigned to a peer consists of the cost of processing each CQ group. A CQ group is a set of similar CQs (recall Section 3.2.1) and the cost of processing a CQ group consists of the cost of processing the shared data item of the CQ group that is being monitored and a grouping cost that increases linearly with the group size. Balance in CQ processing loads of peers is the variance of CQ processing loads. Another measure we are interested in studying is the average data fetching cost induced for periodically fetching data items in order to execute CQs that are assigned to a peer. It is important to notice that the cost of fetching a data item is incurred only once per CQ group.

Figure 7 shows the effect of individual utility function components on the balance in CQ processing loads as a function of replication factor, rf. The line labelled as FULL corresponds to the unmodified utility function. Lines labelled as nX correspond to utility functions in which the component X is taken out ($X \in \{PLF, CAF, SDF\}$). The line labelled as RND corresponds to a special utility function which produces uniformly random values in the range [0,1]. The first observation from Figure 7 is that in all cases except RND, the balance shows an initial

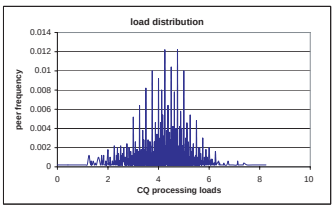

Figure 11: Mean CQ processing load distributions for normal CQ interest distribution

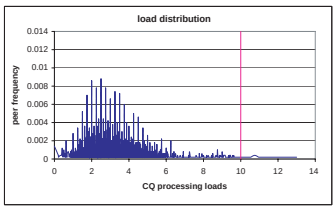

Figure 12: Mean CQ processing load distributions for zipf CQ interest distribution

improvement with increasing rf which is replaced by a linear degradation for larger values of rf. For RND the balance continuously but slowly improves with rf. The degradation in balance is due to excessive grouping. When rf is large, there is more opportunity for grouping and excessive grouping leads to less balanced CQ processing loads. Figure 7 clearly shows that PLF is the most important factor in achieving a good load balance. Since PLF is the most influential factor in achieving good load balance, a lower $tresh$ value used in PLF factor increases its impact thus slows down the rf related degradation in the balance. This is shown in Figure 8. Figure 7 also shows that CAF is responsible for the degradation of balance with increasing rf values. However CAF is an important factor for decreasing the mean CQ processing load of a peer by providing grouping of similar CQs. Although RND provides a better load balance than FULL for $rf \geq 6$, the mean CQ processing load of a peer is not decreasing with increasing rf when RND is used as opposed to the case where FULL is used. The latter effect is shown in Figure 9.

Figure 10 shows the effect of individual utility function components on the data fetching cost due to CQ executions. The increasing rf values provide increased opportunity to minimize this cost due to larger number of peers available for selecting an executor peer with respect to a CQ. Since SDF is explicitly designed to decrease the network cost, its removal from the utility function causes increase in the data fetching cost. Figure 10 shows that CAF also helps decreasing the data fetching cost. This is because it provides grouping which avoids redundant fetching of the data items.

Figure 11 shows the distribution of the CQ processing loads over peers. Figure 12 plots the same graph except that the information monitoring interests of CQs that are used to generate the graph follow a zipf distribution which is more skewed than the normal distribution used in Figure 11. The vertical line in Figure 12 which crosses the x-axis at 10 marks the maximum acceptable load, thus the region on the right of the vertical line represent overloaded peers. Comparing these two figures show that more skewed distributions in information monitoring interests reduce the balance in CQ processing loads.

6 Related Work

WebCQ [15] is a system for large-scale web information monitoring and delivery. It makes heavy use of the structure present in hypertext and the concept of continual queries. It is a server-based system, which monitors and tracks various types of changes to static and dynamic web pages. It includes a proxy cache service in order to reduce communication with the original information servers. PeerCQ is similar to WebCQ in terms

of functionality but differs significantly in terms of the system architecture, the cost of administration, and the technical algorithms used to schedule CQs. PeerCQ presents a large scale information monitoring system that is more scalable, highly reliable and has self-configuring capability.

There are several distributed hashtable based P2P protocols proposed so far [16, 22, 17, 23]. These protocols are used in building several applications including a distributed DNS service [4], a cooperative file storage system [5], a topic based publish subscribe system [18], and a co-operative web cache [10]. However the reliability requirements of P2P systems and the effect of dynamic nature of the network on the achieved reliability are not well studied topics so far. PeerCQ provides a dynamic replication mechanism to achieve high reliability in the presence of a highly dynamic and heterogenous peer network. Security related issues for distributed hashtable based lookup protocols are discussed in [21].

P2P file sharing systems like Gnutella [8] do not provide an explicit mechanism for replication. Files are replicated on demand by users and availability increases with the popularity of a file. If the replica holders of a file are all off-line, then the file is not accessible. Freenet [3], which is a P2P anonymous storage and retrieval system, replicates data items on the retrieval paths for providing improved access. However Freenet does not give strong guarantees on the life times of the data items. Oceanstore [12], which is aimed to build a global-scale persistent storage, is supported through a P2P location scheme [23] and provides replicated storage together with efficient replica management. However Oceanstore assumes high server availability and is not designed for highly unreliable systems. DDNS [4] which is already discussed in Section 4.2, uses a similar replication scheme to PeerCQ, where each data item has certain number of replicas in the system and this property is maintained as servers come and go. However the DDNS servers are expected to be more reliable than the peers of PeerCQ, and replica consistency is not an issue in DDNS.

PeerCQ is different from the described systems highlighted by two main distinctions. Firstly, PeerCQ distributes large number of long running jobs to nodes of a highly dynamic and unreliable P2P network, in contrasts to distributing only data. Secondly, PeerCQ cannot tolerate unavailability of CQs, which is regarded as a catastrophic event. This is due to the fact that PeerCQ has no dependence on a departing peer's reentrance into the system. Whenever a peer departs the system, whatever it has from its last session is assumed to be gone, since that peer may never reenter the system in the future. As a result, CQ unavailability is a once true always true property in PeerCQ, that results in definite CQ loss.

There exist two different classes of well-known replication techniques in the distributed systems literature, active replication and passive replication. In active replication [13, 20], each request is processed by all replicas. In passive replication (i.e. primary-backup) [2, 9], one replica processes the request, and sends updates to the other replicas. Active replication ensures a fast reaction to failures, whereas passive reaction usually has a slower reaction to failures. On the other hand, active replication uses more resources than passive replication. The latter property of the passive replication is the main motivation in selecting a variation of primary-backup approach in PeerCQ.

7 Conclusion

We have described the mechanisms used in PeerCQ, a decentralized peer-to-peer Continual Query system for Internet-scale distributed information monitoring, to support reliable service. Reliability is one of the most important issues in P2P systems that has to be addressed in order to support a wide range of P2P applications. We have presented a dynamic replication solution to provide increased CQ durability and uninterrupted CQs processing in PeerCQ, which leads to a more reliable system. We presented a set of initial experiments, demonstrating the effectiveness of our approach.

References

[1] K. Aberer, M. Hauswirth, M. Punceva, and R. Schmidt. Improving data access in p2p systems. *IEEE Internet Computing*, pages 58–67, 2002.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd international conference on Software engineering*, 1976.

[3] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[4] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving dns using a peer-to-peer lookup service. In *In the proceedings of the First International Workshop on Peer-to-Peer Systems*, 2002.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *ACM Symposium on Operating Systems Principles*, 2001.

[6] B. Gedik and L. Liu. Building reliable peer-to-peer information monitoring service through replication. Technical Report GIT-CC-02-66, Georgia Institute of Technology, 2002.

[7] B. Gedik and L. Liu. PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system. In *The 23rd International Conference on Distributed Computing Systems*, 2003.

[8] Gnutella. The gnutella home page. http://gnutella.wego.com/, 2002.

[9] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4), 1997.

[10] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *ACM Symposium on Principles of Distributed Computing*, 2002.

[11] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing Author Index*, 1997.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, 2000.

[13] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2, 1978.

[14] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[15] L. Liu, C. Pu, and W. Tang. Detecting and delivering information changes on the web. In *Proceedings of International Conference on Information and Knowledge Management*, Washington D.C., 2000.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, 2001.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for largescale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[18] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *International Workshop on Networked Group Communication*, 2001.

[19] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, 2002.

[20] F. Schneider. *Replication management using the state-machine approach*. Addison-Wesley, second edition, 1993.

[21] E. Sit and R. T. Morris. Security considerations for peer-to-peer distributed hash tables. In *In the proceedings of the First International Workshop on Peer-to-Peer Systems*, 2002.

[22] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, 2001.

[23] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, 2001.