

# PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System

Buğra Gedik and Ling Liu  
College of Computing  
Georgia Institute of Technology  
{bgedik,lingliu}@cc.gatech.edu

## Abstract

*PeerCQ is a totally decentralized system that performs information monitoring tasks over a network of peers with heterogeneous capabilities. It uses Continual Queries (CQs) as its primitives to express information-monitoring requests. A primary objective of the PeerCQ system is to build a decentralized Internet scale distributed information-monitoring system, which is highly scalable, self-configurable and supports efficient and robust way of processing CQs. This paper describes the basic architecture of the PeerCQ system and focuses on the mechanisms used for service partitioning at the P2P protocol layer. A set of initial experiments is reported, demonstrating the sensitiveness of the PeerCQ approach to large scale P2P information monitoring and the effectiveness of the PeerCQ service-partitioning algorithms with respect to load balancing and system utilization.*

## 1 Introduction

With the emergence of successful applications like Gnutella [3] and Napster [7], peer-to-peer technology has received rapid and widespread deployment, and a striking visibility over the past few years.

There are currently several P2P systems in operation, and many more are under development. Gnutella [3], Napster [7] and Freenet [1] has been among the most prominent peer-to-peer file sharing systems. In these systems, files are stored at the end user machines rather than at a central server, and as opposed to the conventional client/server model. Files are transferred directly between peers. However, they differ from one another in terms of their lookup services, and the current P2P protocols in these systems are not scalable.

Chord [13], Pastry [10], Tapestry [14], CAN [9] are examples of a second generation of peer-to-peer systems. Their routing and location schemes are based on distributed hash tables. In contrast to the first generation P2P systems such as Gnutella and Freenet, these systems provide reliable content location (persistence and availability) through a tighter control of the data placement and topology within a P2P network. A query is guar-

anteed a definite answer in a bounded number of network hops. The second generation of routing and location schemes is also considered more scalable.

Surprisingly, most of the P2P protocols [1, 13, 9, 10, 3] to date make the assumption that all nodes tend to participate and contribute equally to the system. Thus, these protocols distribute tasks and place data to peers based on this assumption. However P2P applications should respect the peer heterogeneity and user characteristics in order to be more robust [12]. Another common weakness of the second generation P2P protocols is the lack of flexibility in optimizing the key to peer matching algorithms to incorporate important performance metrics such as load balance, system utilization, reliability, and trust.

In this paper we describe PeerCQ, a peer-to-peer information monitoring system, which utilizes a large set of heterogeneous peers to form a peer-to-peer information monitoring network. Many application systems 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 MP digital cameras in next two months and notify me when one with price less than 100\$ becomes available”, “monitor IBM stock price and notify me when it increases by 10%”.

PeerCQ uses continual queries (CQs) as its primitives to express information monitoring requests (subscriptions). Continual Queries [5] are standing (long running) 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.

In this paper, we focus on how PeerCQ addresses the problems related to service partitioning, and describe our proposed technical solutions. The paper has two main

contributions. First, we introduce a distinct approach to CQ systems, which enables processing of large number of CQs by harnessing the power at the edge of the Internet, without a need for centralized servers. Second, we introduce an effective service-partitioning scheme. A unique feature of the PeerCQ service-partitioning mechanism is its ability to integrate both the peer heterogeneity and the information monitoring characteristics of the users into the load balancing scheme, a challenge in large-scale, heterogeneous, and totally decentralized systems.

## 2 System Overview

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 job, expressed as a continual query (CQ), can be posted from any peer. There is no scheduling node in the system. No peers have global knowledge about other peers in the system.

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. We may assume that a PeerCQ service has an associated DNS domain name. It takes care of resolving the mapping of PeerCQ's domain name to the IP address of one or more PeerCQ bootstrapping nodes. A bootstrapping node maintains a short list of PeerCQ nodes that are currently alive in the system. To join PeerCQ, a new node looks up the PeerCQ domain name in DNS to obtain a bootstrapping node's IP address. The bootstrapping node randomly chooses several entry nodes from the short list of nodes and supplies their IP addresses. Upon contacting to an entry node of PeerCQ, the new node is integrated into the system through the PeerCQ protocol's initialization procedures.

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 re-distributed 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 reassigned to the rest of peers, while maintaining the same objectives—maximize the system utilization and balance the load of 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.

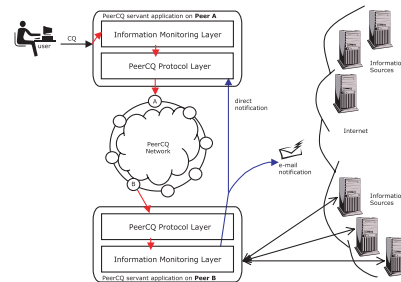


Figure 1: PeerCQ Architecture

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.

A user composes his or her information monitoring request in terms of a CQ and posts it to the PeerCQ system via an entry peer, say **Peer A**. Based on the PeerCQ's service partitioning scheme (see Section 3.2), **Peer A** is not responsible for this CQ. Thus it triggers the PeerCQ's service lookup function. The PeerCQ system determines which peer will be responsible for processing this CQ using the PeerCQ service partitioning scheme. Assume that **Peer B** was chosen to execute this CQ. After the CQ is assigned to **Peer B**, it starts its execution there. During this execution, when an interested information update is detected, the query is fired, and the owner of this CQ is notified with the newly updated information. The notification could be realized by e-mail or by directly sending it to **Peer A** if it is online at the time of notification. Even if a peer is not participating in the system at a given time, its previously posted CQs are in execution at other peers.

## 3 The PeerCQ Protocol

The PeerCQ protocol specifies how to find peers that are 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. In this section we first give an overview of the protocol, including the system model. We then introduce the PeerCQ service partitioning scheme and its lookup service. We also discuss how PeerCQ initialization module works when a new peer joins the network and how the departure or failure of existing peers is handled.

### 3.1 Overview

Similar to most of the second generation of P2P protocols [1, 13, 10, 14, 8], PeerCQ provides a fast and distributed computation of a hash function, mapping information monitoring requests (in form of continual queries) to nodes responsible for them. It extends consistent hashing [4] to take into account of peer heterogeneity and characteristics of user subscriptions in the distribution of CQs to peers, aiming at achieving load balance and efficient processing of large number of information monitoring requests.

An information monitoring request (subscription) is described in terms of a continual query (CQ). Formally, a CQ is defined as a quadruplet, denoted by  $cq : (cq\_id, trigger, query, stop\_cond)$  [5].  $cq\_id$  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 (see Section 3.2 for details). This property of the PeerCQ is referred to as *CQ-awareness*.

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. We refer to this property as *Peer-awareness*. It addresses the service partitioning problem by taking into account of peer heterogeneity and by distributing CQs over peers such that the load of each peer is commensurate to the peer capacities (in terms of cpu, memory, disk, and network bandwidth). Formally, let  $P$  denote the set of all peers in the system. 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.  $\forall p, p' \in P, p.peer\_ids \cap p'.peer\_ids = \emptyset$ . The identifier length  $m$  must be large enough to make the probability of two nodes or two CQs hashing to the same identifier negligible.  $peer\_props$  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. The concrete resource do-

nation model may be defined by PeerCQ applications (see Section 3.2 for details).

Identifiers are ordered in an  $m$ -bit identifier circle modulo  $2^m$ . The  $2^m$  identifiers are organized in an increasing order in the clockwise direction. To guide the explanation of the protocol, we first define our notation:

- 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 it 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:  $NeighborList(p_0, i_0) = [(p_{-r}, i_{-r}), \dots, (p_{-1}, i_{-1}), (p_0, i_0), (p_1, i_1), \dots, (p_r, i_r)]$ , where  $\bigwedge_{k=1}^r ((p_k, i_k) = IRN(p_{k-1}, i_{k-1})) \wedge \bigwedge_{k=1}^r (p_{-k}, i_{-k}) = ILN(p_{-k+1}, i_{-k+1})$ . The size of the neighbor list is  $2r + 1$  and we call  $r$  the neighbor list parameter.

### 3.2 Capability-Sensitive Service Partitioning

The PeerCQ protocol extends the existing routed-query based P2P protocols, such as Chord [13] or Pastry [10], to include a capability-sensitive service partitioning scheme. Service partitioning can be described as the assignment of CQs to peers. By capability-sensitive, we mean that the PeerCQ service partitioning scheme extends a randomized partition algorithm, commonly used in most of the current P2P protocols, with both *peer-aware* and *CQ-aware* capability. As demonstrated in [13, 10, 14, 8], randomized partitioning schemes are easy to implement in decentralized systems. However they do not perform well in terms of load balancing in heterogeneous peer-to-peer environments. We implement *peer-awareness* based on peer donation. Each peer donates a self-specified portion of its resources to the system. The scheduling decisions are based on the amount of donated resources. We implement *CQ-awareness* by distributing CQs having similar information monitoring requests to same peers. CQ-awareness is an important consideration

for reducing or minimizing redundant processing and consumption of network resources when multiple users monitor the same or similar information updates.

Concretely, capability-sensitive service partitioning manages the assignment of CQs to appropriate peers, with the objective of balancing the load of the peers in the system and improving the overall system utilization. By balanced load we mean there are no peers that are overloaded. By system utilization, we mean that when taken as a whole the system does not incur large amount of duplicated computations or consume unnecessary resources such as the network bandwidth between the peers and the data sources.

In PeerCQ, the assignments of CQs to peers are 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 creating 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 [4], is used. A distinct feature of the PeerCQ strict matching algorithm is the two identifier mappings that are carefully-designed to achieve some level of peer-awareness and CQ-awareness, namely the mapping of CQs to CQ identifiers that enables the assignment of CQs having similar triggers to same peers, and the mapping of peers with heterogeneous resource donations to a varying set of peer identifiers. In the *Relaxed Matching* phase, an extension to strict matching is applied to relax the matching criteria to include application semantics in order to achieve the desired level of peer-awareness and CQ-awareness.

### 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  $cq$ . To guide the understanding of the strict matching algorithm, we first describe how CQ identifiers and peer identifiers are generated, and why they play a vital role in achieving peer-awareness and CQ-awareness. Then we discuss the properties of the algorithm.

#### 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. Figure 2 shows an example of mapping two peers, say  $p'$  and  $p''$ , to their peer identifiers on an identifier circle modulo  $2^m$ . Based on the amount of donations, peer  $p'$  has 3 peer identifiers, whereas peer  $p''$  has 6. The example shows that  $p''$  is assigned more CQs than  $p'$  using the defined strict matching.

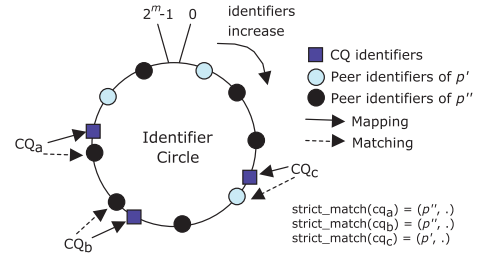


Figure 2: Example of peer to peer identifier set mapping

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 omitted in this paper due to space restriction and readers may refer to our technical report [2] for detail. The mapping of a peer to peer identifiers needs to be as uniform as possible. This can be achieved by using the base hashing functions like MD5 or SHA1 (or any well-known message digest function). The following algorithm<sup>1</sup> explains how the peer identifier set is formed given the effective donation:

```

generatePeerIDs(p, ED)
  p.peer_ids = empty
  for i = 1 to donation_to_ident(ED)
    add SHA1(concat(p.peer_props.IP, counter), m) into p.peer_ids
    increment counter

```

#### Mapping CQs to identifiers

This mapping function is intended to address the CQ-awareness objective. It maps 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

<sup>1</sup>The function  $donation\_to\_ident$  used in the algorithm is responsible of mapping the effective donation value of a peer to the number of  $m$ -bit identifiers, which forms the peer's peer identifier set.

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*. Given  $m$  and  $a$ , the method that maps CQs to the CQ identifiers uses two message digest functions. A sketch of the method is described as follows:

```

calculateCQID(p, cq)
  part1 = SHA1(concat(cq.mon_src, cq.mon_item), a)
  part2 = SHA1(concat(p.peer_props.IP, counter), m-a)
  cq.cq_id = concat(part1, part2)
  increment counter

```

According to the parameter  $a$  (grouping factor) of the first digest function, 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. As the number of CQs is expected to be larger than the number of peers, the number of CQs mapped inside one of these regions is larger than the number of peers mapped. Introducing smaller regions (i.e., the grouping factor  $a$  is larger) increases the probability that two similar CQs are matched to the same peer. This by no means implies that the peers within a contiguous region are assigned only to CQs that are similar for two reasons. First, if the grouping factor  $a$  is not large enough, then two non-similar CQs might be mapped into the same contiguous region by the hashing function used (SHA1 in our case). Second, peers might have more than one identifier possibly belonging to different contiguous regions. 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 (some peers may be responsible for too many CQs). Thus, the grouping factor  $a$  should be chosen carefully.

We refer to the grouping provided by the CQ-to-identifier mapping as the *level-one grouping*. A fine tuning of the level-one grouping will be introduced in the relaxed matching phase in Section 3.2.2.

### 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.

We calculate these three measures for each match made between a CQ and a peer at the strict matching phase. Let  $p$  denote a peer and  $cq$  denote the CQ assigned to  $p$ .

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

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

**Peer load factor** is denoted as  $PLF(p.peer\_props.load)$ . It is a measure of a peer  $p$ 's willingness to accept one more CQ for execution, considering its current load. It is defined as:

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

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

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

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 relaxed matching algorithm enables efficient CQ processing through the use of cache-awareness, decreases overall bandwidth requirement through the use of data source-awareness, and fine tunes the load balancing through the use of load-awareness.

Let  $UtilityF(p, cq)$  denote the utility function of relaxed matching, 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.  $\alpha$  is used as a constant to adjust the importance of data source distance factor with respect to cache affinity factor.

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:

$$\begin{aligned} relaxed\_match(cq) &= (p, i), \text{ where} \\ (p', j) &= strict\_match(cq) \wedge (p, i) \in NList(p', j) \wedge \\ \forall (p'', k) \in NList(p, j), & UtilityF(p, cq) \geq UtilityF(p'', cq) \end{aligned}$$

It is interesting to note that the cache-awareness property of the relaxed matching provides additional level of CQ grouping by favoring the selection of a peer as a CQ's executor if the peer has a cache ready for this CQ (which means that one or more similar CQs are already executing at that peer). We refer to the cache-awareness based grouping as *level-two grouping*.

An extreme case of relaxed matching is called the *random relaxed matching*. Random relaxed matching is similar to relaxed matching except that instead of using a value function to find the best peer to execute a CQ, it makes a random decision among the neighbors of the CQ owner. In the rest of the paper we call the original relaxed matching *optimized relaxed matching*.

### 3.3 PeerCQ P2P Lookup

In this section we shortly describe PeerCQ's ability to efficiently find peers to execute a CQ from a potentially huge number of peers.

Inspired by the lookup operations described in Pastry [10], Tapestry [14], Plaxton Routing [8], and Chord [13], the PeerCQ P2P lookup service is designed to find peers that are most appropriate to execute a CQ in a PeerCQ network in terms of good load balance and better system utilization. It provides two basic functions to implement the matching algorithms described in the previous section:

$p.lookup(i)$ : The *lookup* function takes an  $m$ -bit identifier  $i$  as its input parameter, and returns a peer -identifier pair  $(p, j)$  satisfying the matching criteria used in strict matching, i.e.  $j \in p.peer\_ids \wedge \forall p' \in P, \forall k \in p'.peer\_ids, Dist(j, cq.cq\_id) \leq Dist(k, cq.cq\_id)$ .

$p.get\_neighbors(i)$ : This function takes an identifier from the peer identifier set of  $p$  as a parameter. It returns the neighbor list of  $2r + 1$  peers associated with the identifier  $i$  of the peer  $p$ , i.e.,  $NeighborList(p, i)$ .

The  $p.lookup(i)$  function implements a routed query based lookup algorithm. Lookup is performed by recursively forwarding a lookup query containing a CQ identifier to a peer which has a peer identifier closer to the CQ identifier in terms of the strict matching, until it reaches the owner peer of this CQ. PeerCQ maintains two types of routing information – 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.

Due to the space restriction, we omit the details of the lookup algorithm. Readers who are interested in further details may refer to our technical report [2].

### 3.4 Peer Joins, Departures, and Failures

A key issue in PeerCQ is the maintenance of the CQ-to-peer matchings defined by the PeerCQ service partitioning scheme in a dynamic environment where peers join and depart at any time and peers may fail without notice. To facilitate the understanding of the mechanisms used during peer joins and departures, we first illustrate how to maintain strict matching during peer joins and departures, then we extend the discussion to the maintenance of relaxed matching before we describe how to handle node failures.

#### Joins, Departures with Strict Matching

Assume that after a new peer  $p$  joins the PeerCQ network, its routing table and neighbor list information is initialized. For each identifier  $i \in p.peer\_ids$ , a set of CQs, owned by  $p$ 's immediate right and left neighbors before  $p$  joins the system, are migrated to  $p$  if they meet the strict matching criteria. The departure of a peer  $p$  requires a similar but reverse action to be taken. Again for each identifier  $i \in p.peer\_ids$ ,  $p$  distributes all CQs that it owns to its immediate left and right neighbors associated with  $i$  according to strict matching.

#### Joins, Departures with Relaxed Matching

For each CQ migrated to a new peer  $p$ ,  $p$  becomes the owner of these CQs. By applying the relaxed matching, the executor peer can be located from  $p$ 's neighbor list. Concretely, each peer keeps two possibly intersecting sets of CQs, namely *Owned CQs* and *Executed CQs*. Owned CQs set is formed by the CQs that are assigned to a peer identifier according to strict matching and the executed CQs set is formed by the CQs that are assigned to a peer identifier according to relaxed matching. CQs in the executed CQs set of a peer are executed by that peer, where the CQs in the owned CQs set are kept for control purpose.

A peer  $p$  upon entering the system first initializes its owned CQs set as described in the strict matching case. Then it determines where to execute these CQs based on the relaxed matching. If peers different than the previous executors are chosen to execute these CQs, then they are migrated from the previous executors to the new executors. Peers whose neighbor lists are effected from the entrance of the peer  $p$  into the system also re-evaluate the relaxed matching phase for their owned CQs, since the  $p$ 's

entrance might have effected the relaxed matching. The departure process follows a reverse path. A departing peer  $p$  distributes its owned CQs to its immediate neighbors in terms of strict matching. Then the neighbors determine which peers to execute these CQs according to the relaxed matching function. The departing  $p$  also returns CQs in its executed CQs set to their owners, and these owner peers will find peers to execute these CQs according to relaxed matching.

### Concurrent Joins, Departures

Concurrent joins and departures of peers introduces some problem both in initializing routing information and in redistributing CQs. The approach taken by PeerCQ to provide consistency in the presence of concurrent joins and departures is to enable only one join or one departure operation at a time inside a neighbor list. This is achieved by a distributed synchronization algorithm, which serializes the modifications to the neighbor lists. Instead of a weaker solution based on periodic polls to detect and correct inconsistencies, we use a locking scheme to ensure the correctness.

### Node 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 detected through periodic pollings between peers in a neighbor list. A dynamic replication mechanism is developed for recovering the lost CQs due to unexpected node failures. An important question to ask is where to replicate CQs. Currently we choose to select the peers to store replicas of a CQ from the peers within the neighbor list of its owner peer. Due to the space restriction, the replication mechanisms of the PeerCQ system are omitted.

## 4 Simulation Results

A unique characteristic of the PeerCQ protocol is its service-partitioning scheme, which distributes CQs among peers of diverse capacities, while maintaining balanced loads on peers as well as good overall utilization of the system. To evaluate PeerCQ's service partitioning scheme with respect to system utilization and load balancing, we have designed a series of experiments. In the subsequent sections we first describe our experimental setup, including the simulator, the list of system parameters, and the performance metrics. Then we report three sets of experiments.

### 4.1 Protocol Implementation and Simulator

We built a simulator that assigns CQs to peers using the service partitioning and lookup algorithms described in the previous sections. The system parameters to be set in the simulator include;  $m$ : length of identifiers in bits,  $a$ : grouping factor,  $r$ : neighbor list parameter,  $N$ : number of peers, and  $K$ : number of CQs. In all experiments reported in this paper, the length of the identifiers ( $m$ ) is

set to 128, and the size of the neighbor list  $2r + 1$  is set to 5, i.e.  $r = 2$ .

We model each peer with its resources, the amount of donation it has, and its IP address. The resource distribution is taken as normal distribution. The donations of peers are set to be a half of their resources. We model CQs with the data sources, the data items of interest, and the update thresholds being monitored. The distribution of the user interests on the data sources is selected to model the hot spots that arise in real-world situations due to the popularity of some triggers.

### 4.2 Sensitiveness to Peer Heterogeneity

The heterogeneity of peers in the PeerCQ system is captured by their effective donation ( $ED$ ) values. Based on different  $ED$  values, peers are mapped to different number of peer identifiers on the identifier circle. Therefore, peer heterogeneity is reflected by the number of peer identifiers that different peers possess. To understand sensitiveness of the PeerCQ service partitioning scheme with respect to peer heterogeneity, we model peers with different resources using a normal distribution. We measure the number of CQs distributed over peers having different number of peer identifiers. The results of this experiment where we consider a 10,000 node network ( $N = 10^4$ ) is plotted in Figure 3.

The  $x$ -axis of the graph in Figure 3 represents the number of identifiers possessed by peers, and the  $y$ -axis represents the average number of CQs assigned to peers having  $x$  number of peer identifiers. This graph shows that the number of CQs that a peer is assigned for processing is proportional to the number of identifiers it has on the identifier circle, which is in turn proportional to the effective donation value of the peer.

### 4.3 Effect of Grouping Factor

Another important factor that may affect the effectiveness of the service partitioning scheme is the grouping factor. The grouping factor  $a$  is designed to tune the probability of assigning similar CQs to the same peer. The larger the  $a$  value is, the higher the probability that two similar CQs will be mapped to the same peer, thus the fewer number of CQ groups per peer. However, increasing  $a$  has limitations as discussed in Section 3.2.1.

This experiment considers again a 10,000 node network ( $N = 10^4$ ), and the total number of CQs in the network is 100 times of  $N$ , i.e.,  $K = 10^6$ . Figure 4 compares the average group size and the average number of groups per peer. The values on the  $x$ -axis of Figure 4 are the grouping factors, where the two series represent average CQ group size (average number of CQs per CQ group) and average number of CQ groups per peer. When  $a = 0$ , there is nearly no grouping since the average CQ group size is close to one and the number of CQ groups is large (one CQ per group). As the grouping factor increases, the average size of CQ groups also increases, while the number of CQ groups decreases.

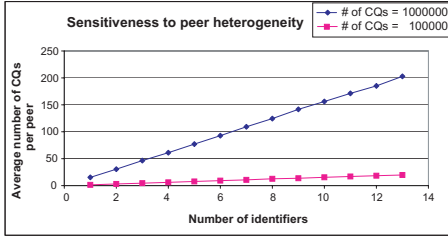


Figure 3: Sensitivity to Peer Heterogeneity

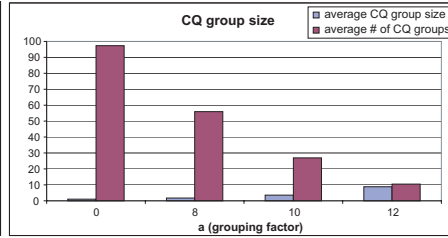


Figure 4: Influence of grouping factor on grouping

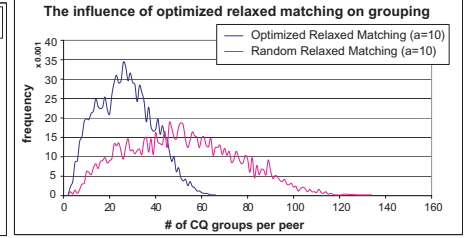


Figure 5: Influence of relaxed matching on grouping

These observations have an important implication. Assignment of CQs to peers that try to achieve better grouping (setting the grouping factor  $a$  to be higher) will *decrease* the number of CQ groups processed by a peer, while *increasing* the number of CQs contained in each CQ group (CQ group size). As a result, the average load of peers will be decreased and the overall system utilization will be better. However, increasing the grouping factor too much will degrade the load balance as it will be described in Section 4.4.

To provide an in-depth understanding of the effect of grouping factor, we also compare the *optimized relaxed matching* algorithm with the *random relaxed matching* algorithm under a given grouping factor. Figure 5 compares the two matching algorithms when  $a = 10$ . The graph shows that the number of CQ groups per peer is lower with optimized relaxed matching. It is clear that the optimized relaxed matching is more effective in its ability to group CQs, which is due to its cache-awareness. We can say that random relaxed matching has only *level-one grouping* which is the grouping provided by the grouping factor, where the optimized relaxed matching algorithm also has *level-two grouping* which is supported through its cache-awareness.

#### 4.4 Effectiveness with respect to Load Balancing and System Utilization

This section presents a set of experiments to evaluate effectiveness of the PeerCQ service partitioning scheme with respect to load balance and system utilization. By better system utilization, we mean that the system can achieve higher throughput and lower overall consumption of resources in terms of processing power and network bandwidth. By load balancing, we mean that no peer in the system is overloaded due to increase of requests to monitoring data sources that are hot spots at times. The notion of load on a peer we use in our performance evaluation is relative to the peer capacities.

##### 4.4.1 Computing Peer Load

An effective measure to evaluate the load balance is the load on peers. In order to analyze the load on peers we first formalize the load on a peer. In PeerCQ, the cost associated with the P2P protocol level processing is considered to be proportional to peer capacities, since the protocol level processing is proportional to the number of

identifiers a peer has. Based on this understanding, we consider the continued monitoring of remote data sources and data items of interest to be the dominating factor in computing the peer load. We formalize the load on a peer  $p$  as follows:

Let  $G_p$  represent the set of groups that peer  $p$  has, denoted by a vector  $\langle g_1, \dots, g_n \rangle$ , where  $n$  is the number of CQ groups that peer  $p$  has. Each element  $g_i$  represents a group in  $p$ , which can be identified by the data source being monitored and the data items of interest. Let  $cost(g_i)$  be the cost of processing all CQs in a group  $g_i$ ,  $monCost(g_i)$  be the cost of monitoring a data item, and  $gCost(size(g_i))$  be the cost of grouping for group  $g_i$ , which is dependent on the number of CQs in  $g_i$ . Then the cost of processing all CQs in a peer, denoted as  $cost(G_p)$ , can be calculated as follows:

$$cost(G_p) = \sum_{i=1}^{size(G_p)} (monCost(g_i) + gCost(size(g_i)))$$

Given that the cost of detecting changes on the data items of interest from remote data sources is the dominating factor in the overall cost of processing a CQ, we assume that the cost of monitoring is the same for all data items independent of the monitoring conditions defined by CQs, and is equal to  $monCost$ :

$$cost(G_p) = size(G_p) * monCost + \sum_{i=1}^{size(G_p)} gCost(size(g_i))$$

In order to calculate the load on a peer, the cost is normalized via dividing it by the effective donation. Let  $ED_p$  be the effective donation of peer  $p$ . We calculate the load on a peer, denoted as  $load(p)$ , as follows:

$$load(p) = cost(G_p) / ED_p$$

The load values of peers are used as both a measure of system utilization and a measure of load balance in our experiments. First, the *mean peer load*, which is the average of peer load values, is used as a measure of system utilization. The smaller the mean load is, the better the system utilization is. However, the system utilization is also influenced by the amount of network bandwidth consumed, which is captured by the *average network cost* defined below. Second, the *variation in peer loads* is used as a measure of load balance. To compare different scenarios, the load variance is normalized by dividing it into the mean load. This measure is called the *balance in peer*

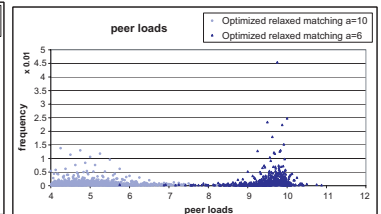
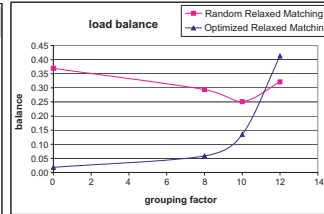
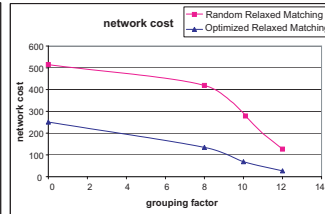
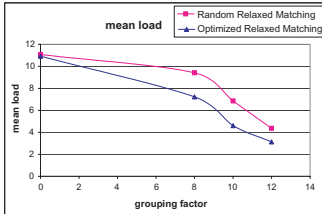


Figure 6: Effect of  $a$ , relaxed matching on mean load

Figure 7: Effect of  $a$ , relaxed matching on network cost

Figure 8: Effect of  $a$ , relaxed matching on load balance

Figure 9: Effect of grouping factor  $a$  on load distribution

loads. Small values of balance in peer loads imply a better load balance.

PeerCQ service partitioning makes use of network proximity between peers and data sources when assigning CQs to peers. It aims at decreasing the network cost of transferring data items from the data sources to the peers of the system. For simulation purpose, we assign a cost to each (peer, data source) pair in the range  $[10,1000]$ . We model such a cost by the ping times between peers and data sources. Then we calculate the sum of these costs for each CQ group at each peer and divide it by the total number of peers to get an average. Let  $P$  denote the network consisting of  $N$  peers, and the function that assigns costs to (peer, data source) pairs as  $net\_cost$ , then the resulting value named as *average network cost* and denoted by  $avgNetCost$  is equal to:

$$avgNetCost = \frac{\sum_{p \in P} \sum_{i=1}^{size(G_p)} net\_cost(p, g_i.mon\_src)}{N}$$

#### 4.4.2 Experimental Results

All experiments in this section were conducted over a network consisting of  $N$  peers and  $K$  CQs, where  $N = 10^4$  and  $K = 10^6$ . To evaluate the effectiveness of the optimized relaxed matching algorithm, we compare it with the random relaxed matching algorithm using the set of parameters discussed earlier, including the grouping factor  $a$ , the mean peer load, the variance in peer loads, balance in peer loads, average network cost, variance in CQ loads of peers.

Figure 6 shows the effect of the grouping factor  $a$  on the effectiveness of relaxed matching with respect to mean load. Similarly, Figure 7 shows the effect of the grouping factor  $a$  on the effectiveness of relaxed matching with respect to network cost. From Figures 6, 7, and 8, we observe a number of interesting facts:

First, as the grouping factor increases the mean peer load decreases. This is because, increasing the grouping factor reduces the redundant computation by enabling better group processing. Optimized relaxed matching provides more effective reduction in the mean peer load due to its level-two grouping. Level-two grouping works better as the grouping factor  $a$  increases.

Second, increasing the grouping factor also helps in decreasing the average network cost, since the cost of fetching data items of interest from remote data sources is incurred only once per CQ group, and served for all CQs

within the group. It is also clear that optimized relaxed matching provides more effective reduction in their average network cost, due to its level-two grouping and its data source awareness.

Third but not the least, the decrease in the mean peer load and in the average network cost is desirable, since it is an implication of better system utilization. However, if the grouping factor increases too much, then the goal of load balancing will suffer.

Figure 8 shows the effect of increasing the grouping factor  $a$  on load balance of both the optimized relaxed matching algorithm, and the random relaxed matching algorithm. As expected, the optimized relaxed matching provides better load balance, since optimized relaxed matching explicitly considers peer loads in its value function for determining the peer that is appropriate for executing a CQ (i.e., the *UtilityF* function in Section 3.2.2). In the case of  $a = 0$  it provides the best load balance. However, as the grouping increases, peers having identifiers belonging to some hot spotted regions of the identifier space are matched much more CQs than others (due to the non-uniform nature of information monitoring interests and the mechanisms used to match CQs to peers). Consequently, the load balance gets worse as the grouping increases. For our experiment setup, the load balance degrades quickly when  $a \geq 8$ .

It is interesting to note that random relaxed matching shows an improvement in load balance for smaller values of the grouping factor and start switching to a degradation trend when  $a$  is set to 10 or higher. This is mainly due to the fact that random relaxed matching only relies on randomized algorithms to achieve load balance in the system. Thus the load balance obtained in the case of  $a = 0$  is inferior when compared to optimized relaxed matching. This means that there are overloaded and under-loaded peers in the system. Grouping helps decreasing the loads of over-loaded peers by enabling group processing. This effect decreases the gap between overloaded peers and under-loaded peers, resulting in better balance to some extent.

Finally, it is important to note that, when we increase  $a$  too much, the optimized relaxed matching loses its advantage in terms of load balancing over the random relaxed matching. Intuitively this happens due to the fact that in optimized random relaxed matching there are two levels of grouping, whereas in random relaxed matching

there is only one level of grouping. More concretely, in overloaded regions of the identifier space, there is nothing to balance. In under-loaded regions, when  $a$  increases, the optimized relaxed matching maps more CQs to fewer peers due to the second-level grouping, causing even more unbalance since several peers get no CQs at all from the under loaded region.

In summary, to provide a reasonable balance between overall system utilization and load balance, it is advisable to choose a value for  $a$ , which is equal to or smaller than the value where the randomized relaxed matching changes its load balance trend to degradation, but is greater than half of this value. This results in the range [6, 10] in our setup. In this range, higher values are better for favoring overall system utilization, whereas lower values are better for favoring load balance. Figure 9 shows this trade-off. The values on the  $x$ -axis are the peer load values, and the corresponding values on the  $y$ -axis are the frequencies of peers having  $x$  amount of load. It is easy to see that the balance is better when  $a = 6$  and load values are lower when  $a = 10$ .

## 5 Related work

WebCQ [6] 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 client-server 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 infrastructure, the cost of administration, the implementation architecture, and the technical algorithms used to scheduling CQs. PeerCQ presents a large scale information monitoring system that is more scalable and less expensive to maintain due to the total decentralization and the self-configuring capability.

To our knowledge, the only P2P application that addresses information monitoring applications is Scribe [11]. Scribe is a publish/subscribe based large-scale, decentralized event notification system. It uses Pastry [10] as its underlying peer-to-peer protocol and builds application level multicast trees to notify subscribers from events published in their subscribed topic. Pastry's location algorithm is used to find rendezvous points for managing the group communication needed for a topic. It uses topic identifiers to map topics to peers of the system. In contrast to Scribe, which is a topic based event notification system, PeerCQ is a generic information monitoring and event notification system that demonstrates the benefits of the PeerCQ protocol in building a scalable information monitoring application.

There are several P2P protocols proposed so far [9, 13, 10, 14]. Similar to work done in Chord [13], Tapestry [14], and Pastry [10], the P2P protocol described in this paper is built based on distributed hash table and ideas

originated from Plaxton's routing algorithm [8].

## 6 Conclusion

We have described PeerCQ, a decentralized peer-to-peer Continual Query system for distributed information Monitoring at Internet-scale. PeerCQ is highly scalable, self-configurable and supports efficient and robust way of processing CQs.

**Acknowledgements** We would like to thank GueYoung Jung for his initial prototype PeerCQ implementation.

## References

- [1] 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*, San Diego, CA., July 2000.
- [2] B. Gedik and L. Liu. Peercq: A scalable and self-configurable peer-to-peer information monitoring system. Technical Report GIT-CC-02-32, Georgia Institute of Technology, February 2002.
- [3] Gnutella. The gnutella home page. <http://gnutella.wego.com/>, 2002.
- [4] 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*, pages 654–663, May 1997.
- [5] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, pages 610–628, July/August 1999.
- [6] 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., November 2000.
- [7] Napster. Napster home page. <http://www.napster.com/>, 2001.
- [8] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, June 1997.
- [9] 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*, August 2001.
- [10] 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*, November 2001.
- [11] 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*, pages 30–43, 2001.
- [12] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. Technical Report UW-CSE-01-06-02, University of Washington, July 2001.
- [13] 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*, August 2001.
- [14] 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, April 2001.