A Scalable Peer-to-Peer Architecture for Distributed Information Monitoring Applications

Bugra Gedik, *Student Member*, *IEEE*, and Ling Liu, *Member*, *IEEE Computer Society*

Abstract—We present PeerCQ, a decentralized architecture for Internet scale information monitoring using a network of heterogeneous peer nodes. PeerCQ uses Continual Queries (CQs) as its primitives to express information-monitoring requests. The PeerCQ development has three unique characteristics. First, we develop a systematic and serverless approach to large scale information monitoring, aiming at providing a fully distributed, highly scalable, and self-configurable architecture for scalable and reliable processing of a large number of CQs over a network of loosely coupled, heterogeneous, and possibly unreliable nodes (peers). Second, we introduce an effective service partitioning scheme at the P2P protocol layer to distribute the processing of CQs over a peer-to-peer information monitoring overlay network while maintaining a good balance between system utilization and load balance in the presence of peer joins, departures, and failures. A unique feature of our service partitioning scheme is its ability to incorporate strategies for handling hot spot monitoring requests and peer heterogeneity into the load balancing scheme in PeerCQ. Third, but not least, we develop a dynamic passive replication scheme to enable reliable processing of long-running information monitoring requests in an environment of inherently unreliable peers, including an analytical model to discuss its fault tolerance properties. We report a set of experiments demonstrating the feasibility and the effectiveness of the PeerCQ approach to large-scale peer-to-peer information monitoring.

Index Terms—Distributed information monitoring, peer-to-peer networks, continual query systems.

1 Introduction

Peer-to-peer (P2P) systems are massively distributed computing systems in which peers (nodes) communicate directly with one another to distribute tasks, exchange information, or share resources. There are currently several P2P systems in operation and many more are under development. Gnutella [6] and Kazaa [8] are among the most prominent first generation peer-to-peer file sharing systems operational today. These systems are often referred to as unstructured P2P networks and they share two unique characteristics. First, the topology of the overlay network and the placement of the files within the network are largely unconstrained. Second, they use a decentralized file lookup scheme. Requests for files are flooded with a certain scope. There is no guarantee of finding an existing file within a bounded number of hops. The random topology combined with flooding-based routing is clearly not scalable since the load on each peer grows linearly with the total number of queries in the network, which in turn grows with the size of the system.

Chord [19], Pastry [16], Tapestry [20], CAN [14] are examples of the second generation of peer-to-peer systems. Their routing and location schemes are structured based on distributed hash tables. In contrast to the first generation P2P systems such as Gnutella and Kazaa, these systems provide guaranteed content location (persistence and availability) through tighter control of the data placement and the topology construction within a P2P network. Queries on existing objects are guaranteed to be answered in a bounded number of network hops. Their P2P routing and location schemes are also considered more scalable. These systems differ from one another in terms of their concrete P2P protocol design, including the distributed hash algorithms, the lookup costs, the level of support for network locality, and the size and dependency of routing table with respect to the size of the P2P overlay network.

Surprisingly, many existing P2P protocols [1], [19], [14], [16], [6] do not distinguish peer heterogeneity in terms of computing and communication capacity. As a result, these protocols distribute tasks and place data to peers assuming all peers participate and contribute equally to the system. Work done in analyzing characteristics of Gnutella in [18] shows that peers participating in these systems are heterogeneous with respect to many characteristics, such as connection speeds, CPU, shared disk space, and peers' willingness to participate. These evidences show that P2P applications should respect the peer heterogeneity and user (application) characteristics in order to be more robust [18].

In this paper, we describe PeerCQ [5], 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 to track changes in multiple information sources on the Web and notify users of changes if some condition over the information sources is met. A typical example in the business world is to monitor availability and price information of specific products, such as "monitor the price of 5 mega pixel digital cameras during the next two months and notify me when one with price less than $500 becomes available," "monitor the IBM stock price and notify me when it increases by 5 percent." In a large scale information monitoring system [9], many users may issue the same information monitoring request, such as tracking IBM stock price changes during a given period of time. We call such

• *The authors are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {bgedik, lingliu}@cc.gatech.edu.*

Manuscript received 21 Oct. 2003; revised 23 Nov. 2004; accepted 30 Nov. 2004; published online 15 Apr. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0187-1003.

phenomena the *hot spot* queries (monitoring requests). Optimizations for hot spot queries can significantly reduce the amount of duplicate processing and enhance the overall system utilization.

In general, offering information-monitoring service using a client/server architecture imposes two challenging requirements on the server side. First, the server should have the ability to handle tens of thousands or millions of distributed triggers firing over hundreds or thousands of Web sites. Second, the server should be scalable as the number of triggers to be evaluated and the number of Web sites to be monitored increase. It is widely recognized that the client/server approach to large-scale information monitoring is expensive to scale and expensive to maintain. The server side forms a single point of failure.

Compared to information monitoring in client-server systems, peer-to-peer information monitoring has a number of obvious advantages. First, there is no additional administrative management cost as the system grows. Second, there is no hardware or connection cost since the peers are the user machines and the connections to the data sources are the user connections. Third, there is no upgrade cost due to scaling since resources grow with clients. The only cost for PeerCQ information monitoring from the perspective of a service provider is the cost of developing the PeerCQ application and making it work effectively in practice.

PeerCQ uses continual queries as its primitives to express information monitoring requests. Continual Queries (CQs) [9] 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.

PeerCQ poses several technical challenges in providing information monitoring services using a P2P computing paradigm. The first challenge is the need for a smart service-partitioning mechanism. The main issue regarding service partitioning is to achieve a good balance between improving the overall system utilization and maintaining the load balance among peers of the system. 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 a large amount of duplicated computations or consume unnecessary resources such as the network bandwidth between the peers and the data sources. Several factors can affect the load balancing decision, including the computing capacity and the desired resource contribution of the peers, the willingness of peers to participate, and the characteristics of the continual queries. The second technical challenge is the reliability of CQ processing in the presence of peer departures and failures. The study reported in [18] shows that large scale peer-to-peer systems are confronted with high peer turnover rate.

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 in the system. There is no scheduling node in the system. No peers have any 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 contact with 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 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 for which it was responsible is reassigned to the rest of the 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.

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 partition scheme (see Section 3.2), `Peer A` is not responsible for this CQ. Thus, it triggers the PeerCQ's P2P 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. `Peer B` is referred to as the

executor peer of 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 peer that posts this CQ is notified with the newly updated information. The notification could be realized by e-mail or by sending it directly to `Peer A` if it is online at the time of notification. Note that, 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 P2P Protocol

The PeerCQ protocol specifies three important types of peer coordination: 1) how to find the peers that are best to serve the given information monitoring requests in terms of load balance and overall system utilization, 2) how new nodes join the system, and 3) how PeerCQ manages failures or departures of existing nodes.

3.1 Overview

Similar to most of the distributed hash table (DHT) based P2P protocols [7], [1], [19], [16], [20], [12], PeerCQ provides a fast and distributed computation of a hash function, mapping information monitoring requests (in the form of continual queries) to nodes responsible for them. PeerCQ protocol design differs from other DHT-based protocols, such as Chord [19], Pastry [16], Tapestry [20], and CAN [14], in a number of ways. First, PeerCQ provides two efficient mapping functions as the basic building blocks for distributing information monitoring requests (CQs) to peers with heterogeneous capabilities. The mapping of peers to identifiers takes into account peer heterogeneity and load dynamics at peers to incorporate peer awareness into the service partitioning scheme. The mapping of CQs to identifiers incorporates CQ grouping optimization [12] for hot spot CQs found frequently in large scale information monitoring applications, striving for efficient processing of a large number of information monitoring requests and minimizing the cost for duplicate processing of hot spot CQs. Second, PeerCQ introduces relaxed matching algorithms on top of the strict matching based on numerical distance between CQ identifiers and peer identifiers when distributing CQs to peers, aiming at achieving good load balance and good system utilization.

In PeerCQ, an information monitoring request (subscription) is described in terms of a continual query (CQ). Formally, a CQ is defined as a quadruplet, denoted by $cq : (cq\_id, trigger, query, stop\_cond)$ [10]. 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 items 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 seven attributes, namely,

$$cq : (cq\_id, mon\_src, mon\_item, mon\_cond, query, notification, stop\_cond).$$

Consider the example monitoring request "monitor Nasdaq index and tell me IBM stock price when Nasdaq index value increases by 5 percent in the next three months." One way to express this request is to use the following continual query: $\langle$ cq_id, mon_src: http://www.quote.com, mon_item: Nasdaq index (/quotes.aspx?symbols=NASDAQ), mon_cond: increase by 5 percent, $query$: IBM stock price (/quotes.aspx?-symbols=NYSE:IBM), $notification$: my email, $stop\_cond$: next 3 months $\rangle$.

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 with similar trigger conditions can be assigned to the 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 peer heterogeneity and by distributing CQs over peers such that the load of each peer is commensurate with 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 the IP address of the peer, peer resources such as connection type, CPU power, and memory, and so on. The concrete resource donation model may be defined by PeerCQ applications (see Section 3.2 for further 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 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 $Dist(i, j) = min(|i - j|, 2^m - |i - j|)$. By this definition, $Dist(i, j) = Dist(j, i)$ holds.
- 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

TABLE 1
Basic API of the PeerCQ System

Function	Description
p.join(out: $status$)	a node p adds itself to the PeerCQ system.
p.leave(out: $status$)	a node p departs the PeerCQ system.
p.post(in: cq, out: cq_id)	a node p posts a cq to the system for execution.
p.terminate(in: cq_id)	a node p posts a termination request for a CQ with identifier cq_id.

path from i to j on the identifier circle. Formally, the following condition holds:

$$i \in p.peer\_ids \ \wedge j \in p'.peer\_ids \wedge \nexists\, p'' \in P \text{ s.t. } \exists k \in p''.peer\_ids \text{ s.t. } k \in path(i,j).$$

The peer p with its peer identifier i can also be referred to as the *immediate left neighbor* (ILN) of peer p' with its identifier j. The definition is symmetric.

- 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. Informally, the neighbor list of a peer p_0 with identifier i_0 consists of three components: 1) the first r number of identifiers on the clockwise path starting from i_0, 2) the first r number of identifiers on the counter clockwise path starting from i_0, and 3) i_0 itself. For each identifier, the neighbor list also stores the peer who owns this identifier.

The basic application interface (API) provided by the PeerCQ system consists of four basic functions, shown in Table 1. The first two API calls are functions for nodes to join or leave the PeerCQ system. The $p.\text{join}(status)$ function adds a node p to the PeerCQ system and returns status information regarding the result of this operation. The $p.\text{leave}(status)$ function departs a node p from the system and returns a status value indicating whether an error has occurred or not. Given a peer p, when $p.\text{post}(cq, cq\_id)$ is called, PeerCQ finds the destination peer that should be responsible for executing the cq posted by peer p and ships cq to that destination peer for execution. We call p the *initiator* peer of the given CQ and the destination peer the *executor* peer of the CQ. $p.\text{post}(cq, cq\_id)$ returns the cq_id as a result of the operation. The function $p.\text{terminate}(cq\_id)$ is used by the issuer peer of a CQ to terminate the processing of its CQ specified by the identifier cq_id.

3.2 Capability-Sensitive Service Partitioning

The PeerCQ protocol extends the existing routed-query-based P2P protocols, such as Chord [19] or Pastry [16], 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 DHT-based protocols, with both *peer-awareness* and *CQ-awareness* capability. As demonstrated in [19], [16], [20], [12], randomized partitioning schemes are easy to implement in decentralized systems. However, they perform poorly in terms of load balancing in heterogeneous peer-to-peer environments.

PeerCQ capability-sensitive service partitioning manages the assignment of CQs to appropriate peers in three stages: 1) mapping peers to identifiers to address peer-awareness, 2) mapping CQs to identifiers to address CQ-awareness, and 3) matching CQs to peers in a two-phase matching algorithm. The main objective for the PeerCQs capability-aware service partitioning is twofold. First, we want to balance the load of peers in the system while improving the overall system utilization. Second, we want to optimize the hot spot CQs such that the system as a whole does not incur a large amount of redundant computations or consume unnecessary resources such as the network bandwidth between the peers and the data sources.

We implement *peer-awareness* based on peer donation and dynamic mapping of peers to identifiers. Each peer donates a self-specified portion of its resources to the system and can dynamically adjust the amount of donations through on-demand or periodical revision of donations. The scheduling decisions are based on the amount of donated resources. We implement *CQ-awareness* by distributing CQs having similar triggers to the same peers. 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.,

$$cq.mon\_src = cq'.mon\_src \wedge cq.mon\_item = cq'.mon\_item.$$

These CQs share the same monitoring source (a stock quote Web site) and the same monitoring item (IBM stock price). By CQ-awareness, we mean that CQs with similar triggers will be assigned to and processed by the same peers.

3.2.1 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. Fig. 1 shows an example of mapping of two peers, say p' and p'', to their peer identifiers. Based on the amount of donations, peer p' has three peer identifiers, whereas peer p'' has six. The example

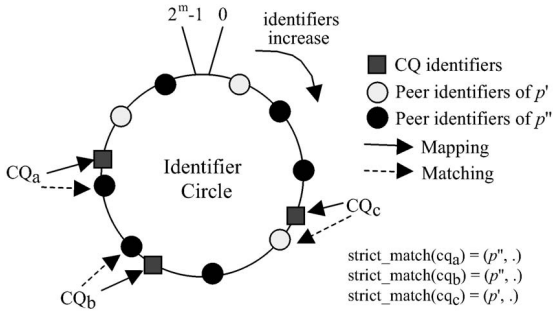

Fig. 1. Example peer-to-peer identifier set mapping.

shows that p'' is assigned more CQs than p' using the strict matching defined in Section 3.2.3.

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. The *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 onto which this peer is going to be mapped. The calculation of *ED* is given in [5]. The mapping of a peer to peer identifier needs to be as uniform as possible. This can be achieved by using base hashing functions like MD5 or SHA1 (or any well-known message digest function). The following algorithm explains how the peer identifier set is formed given the effective donation of a peer:

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

The function `donation_to_ident` is responsible for mapping the effective donation value of a peer to the number of m-bit identifiers in the 2^m identifier space, which forms the peer's peer identifier set. `SHA1` is a message digest function. The first parameter of this digest function is the input message that will be digested. The input message is formed by concatenating the IP address of the peer and a counter which is initialized to a one second real time clock at the initialization time and incremented each time a peer identifier is generated. This concatenation forms a unique input message for each peer identifier. The second parameter, m, is the length of the output in bits.

In summary, the peer-to-identifier mapping algorithm maps a peer to a set of randomly chosen m-bit identifiers on the m-bit identifier circle. The number of identifiers each peer will be mapped to is determined in terms of the effective donation (ED) of the peer, initially computed upon the entry of the peer into the PeerCQ overlay network. In order to handle runtime fluctuations of workload at each peer node, a peer can revise its effective donation through periodic updates or by an on-demand revision upon sudden load surge experienced at the peer node.

3.2.2 Mapping CQs to Identifiers

This mapping function maps a CQ to an m-bit identifier in the identifier space with modulo 2^m. An important design goal for this mapping function is to implement CQ-awareness by mapping CQs with similar triggers (the same monitoring sources and same monitoring items) to the same peers as much as possible in order to produce the CQ-to-peer matching that achieves higher overall utilization of the system.

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 to ensure the uniqueness of the CQ identifiers. 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)
  d = concat(cq.mon_src,cq.mon_item)
  part1 = SHA1(d,a)
  d = concat(p.peer_props.IP, counter)
  part2 = SHA1(d, m-a)
  cq.cq_id = concat(part1,part2)
  increment counter
```

The first digest function generates the same output for similar CQs and the second digest function generates a globally unique output for each CQ posted by a peer. Although there is a small probability of generating the same identifier for two different CQs, the collisions can be detected by querying the network with the generated identifier before posting the CQ. A similar argument is valid for peer identifiers. The first parameter of the first digest function is the concatenation of the data source and the item of interest being monitored. The second parameter is the length of the output in bits. The second digest function generates a random number of length $m - a$ for each CQ. The first parameter of the second digest function is the concatenation of the IP address of the peer posting this CQ and a counter which is initialized to a one second real-time clock at the initialization time and incremented each time a CQ identifier is generated. The second parameter is the length of the output in terms of bits. The CQ-to-identifier mapping returns an m-bit CQ identifier cq_id by concatenating the outputs of these two digest functions.

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 nonsimilar 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. Given the

nonuniform nature of the monitoring requests, there is a trade-off between reducing redundancy in CQ evaluation and balancing load. By setting larger values for the grouping factor a, two extreme situations may occur. On one hand, there may be regions on the identifier circle in which peers are responsible for too many CQs and, on the other hand, there are some other regions in which peers may be assigned too few CQs and are starving. Thus, the grouping factor a should be chosen carefully to optimize the processing of similar CQs while keeping a good balance of the peer loads. 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 described later in the relaxed matching discussion.

3.2.3 Assignment of CQs to Peers: The Two Phase Matching Algorithm

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 matching algorithm consists of two phases: the strict phase and the relaxed phase.

In the *Strict Matching* phase, a simple matching criterion, similar to the one defined in Consistent Hashing [8], 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 the 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.

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), iff the following condition holds:

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

Peer p is called the *owner* of the cq. This matching is strict in the sense that it follows the absolute numerical closeness between the CQ identifier and the peer identifier to determine the destination peer. In other words, suppose two adjacent peer identifiers (together with their peers) (p, i) and (q, j) on the identifier circle such that the CQ identifier lies between i and j. If $Dist(i, cq\_id) \leq Dist(j, cq\_id)$, then the peer p associated with identifier i is the owner peer of this CQ.

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 for 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 a measure of the availability of a CQ that is being executed at a peer p, which monitors the same data source and the same data item as cq. It is defined as follows:

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

- **Peer load factor** is a measure of a peer p's willingness to accept an additional CQ to execute, considering its current load. The PLF factor provides an opportunity for reassigning a CQ to a less loaded peer whenever the executor peer of this CQ needs to be determined. It is defined as follows:

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

- **Data source distance factor** 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:

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

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:

$$\begin{aligned} &UtilityF(p, cq) = PLF(p.peer\_props.load) * \\ &(CAF(p.peer\_props.cache, cq.mon\_item) + \\ &w * SDF(p.peer\_props.IP, cq.mon\_src)). \end{aligned}$$

Note that the peer load factor PLF is multiplied by 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 be selected to execute the CQ if it is heavily loaded. w is used as a constant to adjust the importance of the data source distance factor with respect to the cache affinity factor. For instance, a newly entered peer, which does not have a cache ready for the given CQ but is much 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 w value.

Although the PLF (peer load factor) used in the relaxed matching helps assigning a CQ to a less loaded peer, an increase in the load of a peer after CQs are assigned is not considered by the PLF. However, the PeerCQ design provides easy mechanisms to address such situations. The dynamic changes in the load of a peer can be handled by adjusting the number of peer identifiers. For instance, an increase in the load due to other processes executed by the peer can be handled by decreasing the number of peer identifiers possessed by the peer. This will offload the CQs associated with the dropped peer identifiers to other less loaded peers (as determined by the PLF component of relaxed matching).

Formally, relaxed matching can be defined as follows: The function $relaxed\_match(cq)$ returns a peer p with identifier i, denoted by a pair (p, i), if and only if the following holds:

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

The idea behind the relaxed matching can be summarized 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 load awareness, cache awareness, and network proximity of the data sources being monitored. In case such a neighbor exists, the owner peer will assign this CQ to one of its neighbors for execution. We call the neighbor node chosen according to the relaxed matching the *executor* of the CQ.

It is interesting to note that the cache-awareness property of the relaxed matching provides an additional level of CQ awareness by favoring the selection of a peer as a CQ's executor if the peer has a cache ready for the 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*, which can be seen as an enhancement to the mapping of similar CQs to the same peer in the strict matching phase.

An extreme case of relaxed matching is called *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*. Unless otherwise specified, the terms relaxed matching and optimized relax matching are used interchangeably.

3.3 PeerCQ Service Lookup

The PeerCQ service lookup implements the two-phase matching described in the previous section. Given a CQ, the lookup operation is able to locate its *owner* and *executor* using only $O(\log N)$ messages in a fully decentralized P2P environment, where N is the number of peers. Similarly to several existing design of the DHT lookup services [17], [21], [13], [20], the lookup operation in PeerCQ is performed by routing the lookup queries toward 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 and the executor peer of the CQ.

Two basic API functions are provided to find peers that are most appropriate to execute a CQ based on 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.,

$$
\begin{aligned}
&j \in p.peer\_ids \ \wedge \ \forall p' \in P, \\
&\forall k \in p'.peer\_ids, Dist(j, cq.cq\_id) \leq Dist(k, cq.cq\_id).
\end{aligned}
$$

- $p.get\_neighbors(i)$: 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 whose peer identifier is closer to the CQ identifier in terms of the strict matching until it reaches the owner peer of this CQ. A naive way of answering a lookup query is to iterate on the identifier circle using only the neighbor list until the matching is satisfied. The routing tables are used simply to speed up this process. Initialization and maintenance of the routing tables and the neighbor lists do not require any global knowledge. The number of messages used by our lookup operation is logarithmic with respect to the number of peers in the system. More importantly, neither the mappings introduced by PeerCQ nor the implementation of PeerCQ's relaxed matching increases the asymptotic complexity of the number of messages required by the lookup service to carry out CQ to peer matching. Readers may refer to [5] for details of the PeerCQ lookup protocol.

3.4 Peer Joins and Departures

In a dynamic P2P network, peers can join or depart at any time and peer nodes may fail without notice. A key challenge in implementing these operations is how to preserve the ability of the system to locate every CQ in the network and the ability of the system to balance the load when distributing or redistributing CQs. To achieve these objectives, PeerCQ needs to preserve the following two principles: 1) Each peer identifier's routing table and neighbor list are correctly maintained. 2) The strict matching and the relaxed matching are preserved for every CQ.

The maintenance of the routing information for DHT-based P2P systems in the presence of peer joins and departures is studied in the context of several P2P systems [17], [20], [12], [19]. Due to space restrictions, we focus on the mechanisms used in PeerCQ to ensure the second principle in the following subsection and refer the readers to our technical report [4] for detailed discussion and algorithms on the routing information maintenance. We first describe how to maintain strict matching during peer

joins or departures. Then, we extend the discussion to the maintenance of relaxed matching. We defer the discussion on how PeerCQ handles node failures to the next section.

3.4.1 Maintaining the Two-Phase Matching of CQs to Peers

It is important to maintain the two-phase matching criteria in order to preserve the ability of the system to sustain the installed CQs while balancing the peer load in the presence of peer joins, departures, and failures.

Joins, Departures with Strict Matching. Assuming that, after a new peer p joins the PeerCQ network, its routing table and neighbor list information are initialized, the subset of CQs that need to transfer their ownership to this newly joined peer p can be calculated as follows: For each identifier $i \in p.peer\_ids$, a set of CQs owned by p's immediate left and right neighbors before p joins the system is 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 it owns to the immediate left and right neighbors associated with i according to strict matching.

Joins, Departures with Relaxed Matching. For CQs 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*. The owned CQs set is formed by the CQs that are assigned to a peer according to strict matching and the executed CQs set is formed by the CQs that are assigned to a peer according to relaxed matching. CQs in the executed CQs set of a peer are executed by that peer, whereas the CQs in the owned CQs set are kept by the owner peer for control purposes.

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 relaxed matching. If peers different from 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 affected due to the entrance of peer p into the system also reevaluate the relaxed matching phase for their owned CQs since p's entrance might have caused the violation of relaxed matching for some peers.

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 will execute these CQs according to the relaxed matching. The departing peer p also returns CQs in its executed CQs set to their owners and these owner peers find the new executor peers of these CQs according to the relaxed matching.

Concurrent Joins & Departures. Concurrent joins and departures of peers introduce additional challenges in initializing routing information of newly joined peers, updating routing information of existing peers, and redistributing CQs. More concretely, the problem is how to guarantee concurrent updates of neighbor lists correctly and efficiently as the PeerCQ network evolves. In order to provide consistency in the presence of concurrent joins and departures, in the first prototype of PeerCQ, we enable only one join or one departure operation at a time within a neighbor list. This is achieved by a distributed synchronization algorithm executed within neighbor list boundaries which serializes the modifications to the neighbor list of each peer identifier. We use a mutual exclusion algorithm [15] to ensure the correctness instead of a weaker solution based on periodic polls to detect and correct inconsistencies, as is done in Chord [19].

3.5 Handling Node Failures with Dynamic Replication

It is known that failures are unavoidable in a dynamic peer-to-peer network where peer nodes correspond to user machines. 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. Byzantine failures that include malicious program behavior are not considered in this paper. We assume a fail-stop model where timeouts can be used for detecting failures. In PeerCQ, failures are detected through periodic pollings between peers in a neighbor list.

Failures threaten the system reliability in two aspects. First, a failure may result in incorrect routing information. Second, a failure of a peer will cause CQ losses if no additional mechanisms are employed. The former problem is solved using routing maintenance mechanisms similar to those in handling peer departure. The only difference is that the detection of a failure triggers the maintenance of the routing information instead of a volunteer disconnection notification. However, the latter problem requires a more involved solution.

There are two important considerations in PeerCQ regarding providing fault-tolerant reliable service. One is to *provide CQ durability* and the other is to *provide uninterrupted CQ processing*. CQ durability refers to the ability of PeerCQ to maintain the property that no CQs executed at a peer will get lost when it departs or fails unexpectedly. Uninterrupted CQ processing refers to the ability of PeerCQ to pick up those CQs dropped due to the departure or failure of an existing peer and continue their processing. Whenever CQ durability is violated, the uninterrupted CQ processing will be violated, too. Furthermore, when a peer p fails, if there are existing peers that hold additional replicas of the CQs that p runs as their executor, but that do not have sufficient information on the execution state of those CQs, then the execution of these CQs will be interrupted, possibly resulting in some inconsistent behavior. The proper resumption of the CQ execution upon the failure of its executor peer requires the replicas to hold both the CQs and their runtime state information.

3.5.1 PeerCQ Replication Scheme

In order to ensure smooth CQ execution and to prevent failures from interrupting CQ processing and threatening CQ durability, we need to replicate each CQ. We describe PeerCQ replication formally as follows: A CQ, denoted as cq, is replicated at peers contained in the set:

$$ReplicationList(cq) = [(p_{-\lfloor rf/2 \rfloor}, i_{-\lfloor rf/2 \rfloor}), \ldots,$$
$$(p_{-1}, i_{-1}), (p_0, i_0), (p_1, i_1), \ldots, (p_{\lceil rf/2 \rceil}, i_{\lceil rf/2 \rceil})], \text{ where}$$
$$\bigwedge_{k=1}^{\lceil rf/2 \rceil} p_{i_k} = IRN(p_{k-1}, i_{k-1}) \ \wedge \ \bigwedge_{k=1}^{\lfloor rf/2 \rfloor} p_{i_{-k}}$$
$$= ILN(p_{-k+1}, i_{-k+1}) \ \wedge \ (p_0, i_0) = strict\_match(cq).$$

This set is called the *replication list* and is denoted as $ReplicationList(cq)$. The 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 to maintain the property that replication is a localized operation, i.e.,

$$ReplicationList(cq) \subset NeighborList(p, i), \text{ where}$$
$$(p, i) = strict\_match(cq).$$

In addition to replicating a CQ, some execution states of the CQ need to be replicated together with the CQ and be updated when the CQ is executed and its execution state changes in order to enable correct continuation of the CQ execution after a failure. Recall from Section 2, the executor peer of a CQ needs to maintain three execution states about this CQ: 1) the evaluation state of the trigger, which contains monitoring source, monitoring item, and trigger condition evaluation result, 2) the query result returned since the last CQ evaluation, and 3) the notification state of the CQ. In PeerCQ, changes on the states associated with each CQ are propagated to replicas of the CQ in two steps, with either an eager mode or a deferred mode: 1) Whenever an executor peer of a CQ updates the related states of a CQ, it notifies the CQ owner immediately to ensure that such updates will be propagated to all replicas of the CQ. 2) Upon receiving update notification from the executor peer of a CQ, the owner peer of the CQ may choose to send update notifications to all other peers holding replicas of the CQ immediately (eager mode) or to propagate the update to rest of the replicas using a deferred strategy that considers different trade-offs between performance and reliability. The propagation of the state update to the owner ensures that at least two peers hold the update state of each CQ and the probability of both executor peer and owner peer failing together is relatively low.

Since CQs should be available for processing at any time once they are installed in the system, PeerCQ requires a strong and dynamic replication mechanism. By strong and dynamic replication, we mean that, at any time, each CQ should have a 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 Phase 1, a CQ is replicated at a certain number of peers. This phase happens immediately after a CQ is installed into the system. In Phase 2, the number of replicas existing in the system is kept constant and all replicas are kept consistent. The second phase is called the *replica management* phase and lasts until the CQ's termination condition is met or the CQ is explicitly removed from the system.

One important decision for the PeerCQ replication scheme is where to replicate CQs. In order to preserve the correctness of the lookup mechanism and preserve good load-balance, we select the peers to host the replicas of a CQ from the peers in the neighbor list of the owner peer of this CQ. Moreover, choosing these peers from the neighbor list localizes the replication process (no search is required for locating replica holders), which is an advantage in a fully decentralized system. Furthermore, peers that are neighbors on the identifier circle are not necessarily close to each other geographically, thus the probability of collective failures is low.

3.5.2 Fault Tolerance

Given the description of the PeerCQ replication scheme, we define two different kinds of events that result in losing 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 an 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 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 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, where the replication list is a subset of the neighbor list and has size $rf + 1$. Assume that 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 large 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 losing some or all CQs in $O_{p,i}$.

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

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

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:

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

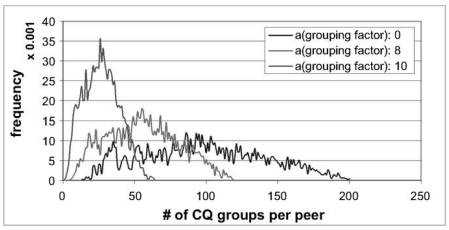

Fig. 2. Effect of grouping for $a = 0$, $a = 8$, and $a = 10$.

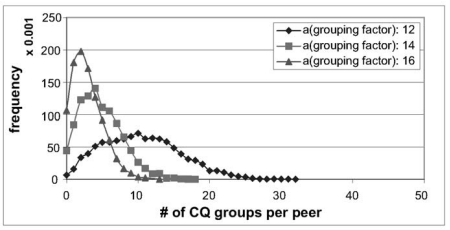

Fig. 3. Effect of grouping for $a = 12$, $a = 14$, and $a = 16$.

Let t_0 denote a time instance at which all peers in $RL_{p,i}(t)$ were 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:

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

which leads to

$$\begin{aligned} Pr_{df}(p,i) &= pf^{rf+1} * \prod_{i=1}^{rf} Pr\{A_i < \Delta t_r\}, \\ Pr_{df}(p,i) &= pf^{rf+1} * \prod_{i=1}^{rf} (1 - e^{-\Delta t_r/st}). \end{aligned} \quad (2)$$

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

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

In a setup where $rf = 4$, $pf = 0.1$, $\Delta t_r = 30$ secs and $st = 60$ mins, $p.identifier\_count = 5$, $Pr_{df}(p)$ turns out to be $\simeq 2.37 * 10^{-13}$. We further investigate the fault tolerance capability of PeerCQ in Section 4.4. Note that the greater the replication factor rf is, the lower the probability of losing CQs. However, having a greater replication factor increases the cost of managing the replicas. As described earlier, the job of dealing with the replica management of a CQ is the responsibility of the CQ's owner. We omit the detailed replica management in this paper and refer readers to [4] for further discussion.

4 Simulation-Based Experiments and Results

To evaluate the effectiveness of PeerCQ's service partitioning scheme with respect to system utilization and load balancing, we have designed a series of experiments. Here, we first describe our experimental setup.

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; K, number of CQs. With this simulator, we conduct our experiments under different stabilization states of the system as well as under unstable states. The system states were modeled with different numbers of peers, different workloads of CQs, and different configurations of some system parameters. The measurements were taken on these snapshots. In all experiments reported in this paper, the length of the identifiers (m) is set to 128.

We model each peer with its resources, the amount of donation, the reliability factor, and its IP address. The resource distribution is taken as normal distribution. The donations of peers are set to be half of their resources. We model CQs with the data sources, the data items of interest, and the update thresholds being monitored. There are $D = 5 * 10^3$ data sources and 10 data items on each data source. 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. Both normal distribution for modeling the user interests on the data sources and a zipf distribution (Section 4.2.4) are considered in the experiments.

4.1 Effect of Grouping Factor

An important factor that may affect the effectiveness of the service partitioning scheme is the grouping factor. Recall from Section 3.2.2 that the grouping factor a is introduced at the protocol level to promote the idea of grouping similar CQs to optimize the processing of similar information monitoring requests. 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 and, thus, the fewer number of CQ groups per peer. However, increasing a has limitations, as discussed in Section 3.2.2.

This experiment considers a 10,000 node network ($N = 10^4$) and the total number of CQs in the network is 100 times N, i.e., $K = 10^6$. Fig. 2 shows the effects of increasing a on grouping when a is 0, 8, and 10. Fig. 3 shows the effects of increasing a on grouping when a is 12, 14, and 16. The values on the x-axis are the number of CQ groups that the peers have and the corresponding values on the y-axis are the frequencies of peers having x number of CQ groups. From these two figures, one can observe that, when a is set to 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 and the number of CQs a peer is responsible for may reach up to 200). The number of groups decreases as a is set to a larger value. The average size of the CQ groups also increases as a is set to a larger value. Consider the simulation results from Fig. 2 and Fig. 3, when $a = 8$, the largest number of groups a peer may have is decreased to 120 or so. When $a = 10$, the largest number of groups a peer may have is dropped to less than 70. When the grouping factor a is set to be 16, the largest number of groups a peer has is less than 20. A small number of CQ groups implies larger sizes of the CQ groups.

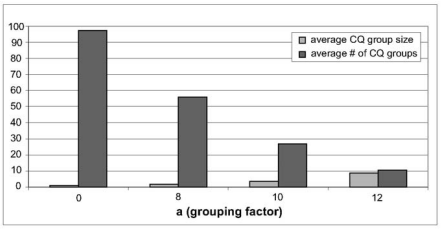

Fig. 4. Influence of a on average CQ group sizes and average number of CQ groups.

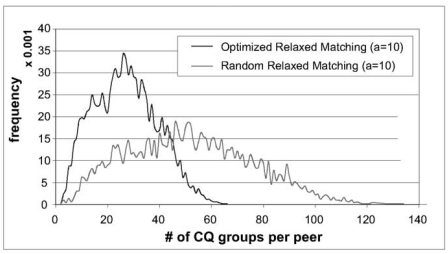

Fig. 5. Optimized relaxed matching compared to strict matching.

Fig. 4 compares the average group size and the average number of groups per peer. The values on the x-axis of Fig. 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, respectively. 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 the CQ groups also increases, while the number of CQ groups decreases.

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 causes a lot of peers getting no CQs!

To provide an in-depth understanding of the effect of the grouping factor, we compare the *optimized relaxed matching* algorithm with the *random relaxed matching* algorithm under a given grouping factor. Fig. 5 compares the two matching algorithms when $a = 10$. 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* supported through its cache-awareness.

4.2 Effectiveness with Respect to Load Balancing and System Utilization

This section presents a set of experiments to evaluate the 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 joins or departure of other peers or due to the increase of requests to monitoring data sources that are hot spots at times.

4.2.1 Load Balance versus System Utilization

It is interesting to point out that, by incorporating the grouping optimization into PeerCQ, we observe that the goal of balancing the load over peers may not always be consistent with the goal of maximizing the overall system utilization in PeerCQ.

To illustrate this observation, consider a simple example: Assume we have two peers, p and p', which are identical in terms of their capacities. Assume that there are seven CQs that need to be distributed to these two peers. One CQ of type a denoted as cqa_1 and six CQs of type b denoted as $cqb_1, \ldots, cqb_6$. Furthermore, assume that CQs of the same type are similar and thus can be grouped together. The scenario that shows a better *overall* utilization of system resources is the case where p is assigned only one CQ, which is cqa_1, and p' is assigned six CQs, namely, $cqb_1, \ldots, cqb_6$. By using the full power of CQ grouping, one can minimize the repeated computation and duplicated consumption of network resources. However, the optimal system utilization may not necessarily imply a good balance of loads on peers for two reasons. First, the cost of processing a CQ consists of three main components: the cost of evaluating its trigger, the cost of executing its query component upon the truth evaluation of triggers, and the cost of notification. Grouping of similar CQs only saves the repeated evaluation of trigger conditions that are shared among groups of CQs. Second, even though the most expensive computation of the CQ processing is the continued testing of CQ triggers, our experience with the continual query systems shows that the cost of grouping, querying, and notification is not negligible [10]. Therefore, it is likely that the scenario where the system is best utilized may not be the same as the scenario where the load of the system is best balanced among peers.

4.2.2 CQ Load of a Peer

We define the CQ load of a peer to be the number of CQs processed by the peer divided by the number of identifiers it has. Due to the support of grouping in the CQ processing, the CQ load of a peer should not be used as a measure to compare peer loads. To illustrate this observation, consider a case where a peer, say p, is assigned 10 CQs and another peer, say p', which has twice the number of peer identifiers that p has, is assigned 20 CQs. Note that their CQ loads are equal. Furthermore, assume that the 10 CQs assigned to p are partitioned into five CQ groups of sizes 4, 2, 2, 1, 1, and the 20 CQs assigned to p' are partitioned into two CQ groups of sizes 12, 8. In this case, it is quite possible that the peer p' is loaded less than peer p, although their CQ loads are equal. This is due to the fact that the number of CQ groups in p' (which is two) is smaller than the number of CQ groups in p (which is five), although their CQ loads are equal.

4.2.3 Computing Peer Load

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

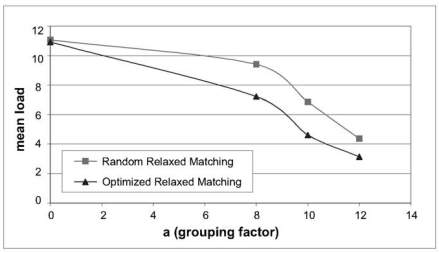

Fig. 6. Effect of a and relaxed matching on mean peer load.

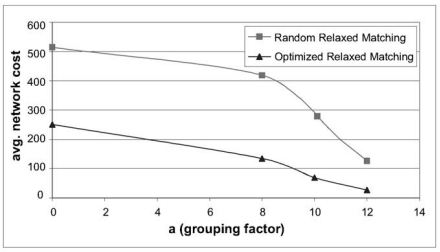

Fig. 7. Effect of a and relaxed matching on average network cost.

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:[1]

Let G_p represent the set of groups that peer p has, denoted by a vector $\langle g_1, \ldots, g_n \rangle$, where n is the number of CQ groups that peer p has. We refer to n as the size of G_p denoted by $size(G_p)$. 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. The size of a group g_i, which is the number of CQs it contains, is denoted by $size(g_i)$. 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:

$$\begin{aligned} cost(G_p) &= \sum_{i=1}^{size(G_p)} cost(g_i) \\ &= \sum_{i=1}^{size(G_p)} (monCost(g_i) + gCost(size(g_i))). \end{aligned}$$

In our experiments, we assume that the cost of grouping increases linearly with group size. In particular, if processing one CQ costs one unit, then processing k similar CQs costs $1 + x * k$ units, where $x * k$ corresponds to the cost of grouping k CQs. x is taken as 0.25 in our simulations. This setting was based on the grouping effect and cost study we have done on WebCQ [12].

Given that the cost of detecting changes in 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 $monCost$, then the cost of processing all CQs on a peer p can be reduced to:

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

In order to calculate the load on a peer, the cost is normalized via dividing it by the effective donation because the notion of load on a peer in our system is relative to the effective donation of the peer. Let ED_p be the effective donation of peer p. We calculate the load on a peer as: $load(p) = cost(G_p)/ED_p$.

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 by the mean load. This measure is called the *balance in peer 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 purposes, 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 net_cost be the function that assigns costs to (peer, data source) pairs, then the resulting value named as *average network cost* and denoted by $avgNetCost$ is equal to: $avgNetCost = \frac{1}{N} \sum_{p \in P} \sum_{i=1}^{size(G_p)} net\_cost(p, g_i.mon\_src)$.

4.2.4 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.

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

First, as the grouping factor increases, both the mean peer load and the average network cost decreases. Increasing the grouping factor helps in decreasing the mean peer load since it reduces the redundant computation by enabling more group processing. Optimized relaxed matching provides more effective reduction in the mean peer load

1. For the purpose of our simulation, we have assumed that the frequency of changes is the same for all data items.

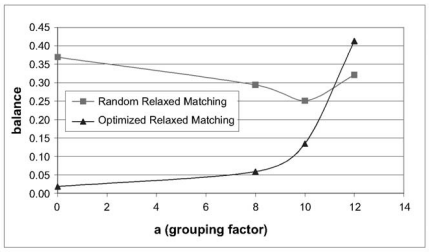

Fig. 8. Effect of a and relaxed matching on load balance.

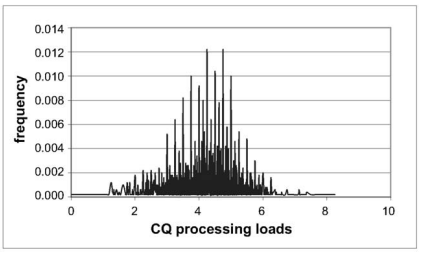

Fig. 10. Load distributions for normally distributed CQ interest.

due to its level-two grouping. Level-two grouping works better as the grouping factor a increases (i.e., the level-one grouping 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 serves 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 (which results in better grouping) and its data source awareness, which incorporates the network cost of accessing the data items of a CQ that are being monitored into the service partitioning decision.

Third, but not 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 over the peers of the system will suffer.

Fig. 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. In the case of $a = 0$, it provides the best load balance. However, as the grouping increases, peers having identifiers belonging to some hot spot regions of the identifier space match many more CQs than others (due to the nonuniform 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 is 8 or higher.

It is interesting to note that random relaxed matching shows an improvement in load balance for smaller values of the grouping factor and starts 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 underloaded peers in the system. Grouping helps decrease the loads of overloaded peers by enabling group processing. This effect decreases the gap between overloaded peers and underloaded peers, resulting in a 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 underloaded 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 underloaded 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. Fig. 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. By looking at the points, it is easy to see that the balance is better when $a = 6$ and load values are lower when $a = 10$.

Fig. 10 shows the distribution of the CQ processing loads over peers. Fig. 11 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 Fig. 10. The vertical line in Fig. 11 which crosses the x-axis at 10 marks

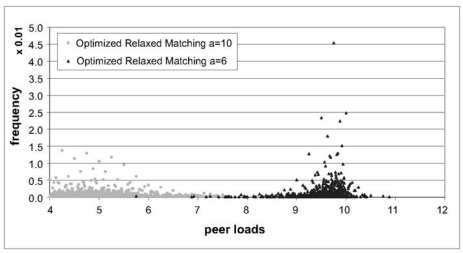

Fig. 9. Effect of a and relaxed matching on load distribution.

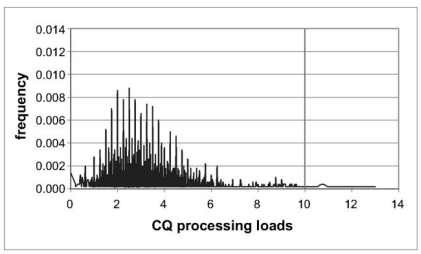

Fig. 11. Load distributions for zipf distributed CQ interest.

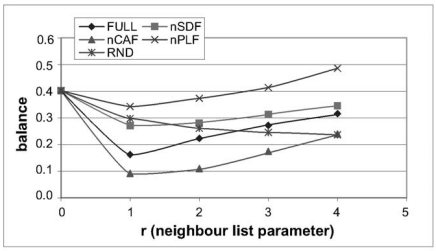

Fig. 12. Balance in loads for different utility functions.

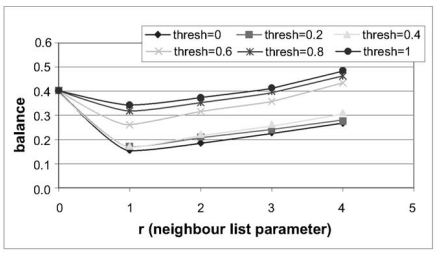

Fig. 13. Balance in loads for different threshold values in PLF.

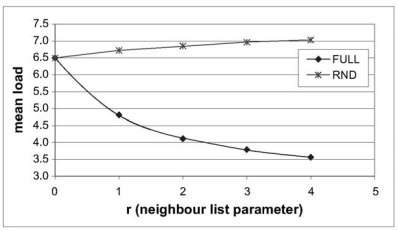

Fig. 14. Mean CQ processing load for different utility functions.

the maximum acceptable load, thus the region on the right of the vertical line represents overloaded peers. Comparing these two figures shows that more skewed distributions in information monitoring interests reduce the balance in CQ processing loads.

4.3 Effect of Relaxed Matching Criteria

The relaxed matching criteria, 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. The experiment is set up over a network of 10^4 nodes with 10^6 CQs and the grouping factor a is set to be 8.

Fig. 12 shows the effect of individual utility function components on the balance in CQ processing loads as a function of neighbor list size, r. The line labeled as FULL corresponds to the unmodified utility function. Lines labeled as $\mathrm{n}X$ correspond to utility functions in which the component X is taken out ($X \in \{PLF, CAF, SDF\}$). The line labeled as RND corresponds to a special utility function which produces uniformly random values in the range $[0, 1]$, resulting in randomized relaxed matching. The first observation from Fig. 12 is that, in all cases except RND, the balance shows an initial improvement with increasing r which is replaced by a degradation for larger values of r. For RND, the balance continuously but slowly improves with r. The degradation in balance is due to excessive grouping. When r is large, there is more opportunity for grouping and excessive grouping leads to less balanced CQ processing loads. Fig. 12 clearly shows that PLF is the most important factor in achieving good load balance. Since PLF is the most influential factor in achieving good load balance, a lower $thresh$ value used in PLF factor increases its impact, thus slowing down the r related degradation in the balance. This is shown in Fig. 13. Fig. 12 also shows that CAF is responsible for the degradation of balance with increasing r 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 $r \geq 3$, the mean CQ processing load of a peer is not decreasing with increasing r when RND is used as opposed to the case where FULL is used. The latter effect is shown in Fig. 14.

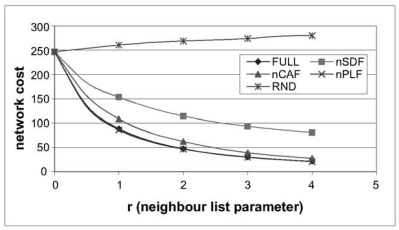

Fig. 15. Network cost as a function of r for different utility functions.

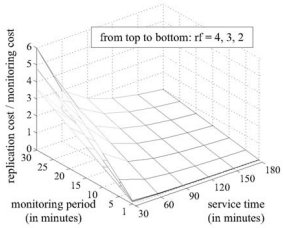

Fig. 16. Cost of replication relative to monitoring cost.

Fig. 15 shows the effect of individual utility function components on the network cost due to CQ executions. The increasing r values provide increased opportunity to minimize this cost due to the 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 network cost. Fig. 15 shows that CAF also helps decrease the network cost. This is because it provides grouping which avoids redundant fetching of the data items.

4.4 CQ Availability under Peer Failure

One situation that is crucial for the PeerCQ system is the case where 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 do not expect this kind of trend to continue for a long period, it can happen temporarily. In order to observe the worst case, we have set up 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, losing 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

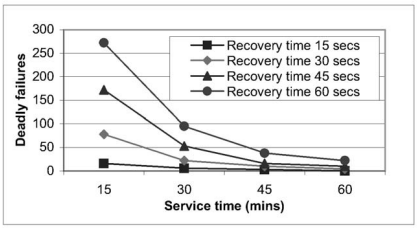

Fig. 17. Deadly failures $rf = 2$.

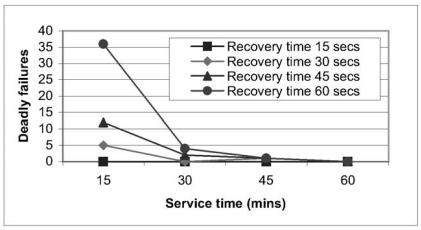

Fig. 18. Deadly failures $rf = 3$.

how gracefully the system degrades for different replication factors.

The graphs in Figs. 17, 18, and 19 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 worst-case 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 the dynamic replication provided by PeerCQ is able to achieve high reliability with moderate values for the replication factor. Although stronger reliability guarantees can be achieved through increasing the replication factor further, it has the side-effect of increasing the cost of replication. The formal analysis of this cost is given in [4]. Here, we provide a sample experimental result pertaining to cost of replication in order to give a general idea of the trade-offs involved in deciding an appropriate value for the replication factor.

Fig. 16 plots the network cost of replication relative to the network cost of monitoring as a function of mean peer service time and CQ monitoring period. Mean peer service time is the average time a peer stays in the network before it fails or departs. The monitoring period is the time between two successive pollings of the data sources for monitoring changes on data items. It is observed from the figure that the cost of replication is small compared to the monitoring cost when the mean service time values are high or when the monitoring period values are low. It is also observed that smaller replication list sizes (i.e., smaller rfs) help in decreasing the relative cost of replication. For applications with tighter latency requirements, the monitoring period should be small and the replication is less likely to be an issue in terms of network cost. On the other hand, for applications specifying larger monitoring periods, the cost of replication can be adjusted by changing the rf values and adjusting the desired level of reliability.

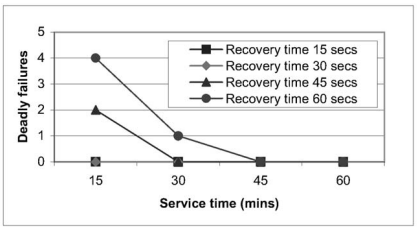

Fig. 19. Deadly failures $rf = 4$.

5 Related Work

WebCQ [11] 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 peer-to-peer architecture for large-scale information monitoring, which is more scalable and less expensive to maintain due to the total decentralization and the self-configuring capability.

Scribe [18] is a P2P application that is related to event monitoring and notification. It presents a publish/subscribe-based P2P event notification infrastructure. Scribe uses Pastry [17] 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. However, Scribe is a topic-based event notification system, where PeerCQ is a generic information monitoring and event notification system. In PeerCQ, notifications are generated based on the monitoring done on the web using the supplied CQs that encapsulate the interested information update requests. In Scribe, notifications are generated from publish events of the topic subscribers.

Several P2P protocols have been proposed to date, among which the most representative ones are CAN [15], Chord [20], Tapestry [21], and Pastry [17]. Similarly to these existing DHT-based systems, the PeerCQ P2P protocol described in this paper is developed by extending the Plaxton routing proposal in [13]. The unique features of PeerCQ are its ability to incorporate peer-awareness and CQ-awareness into the service partition scheme and its ability to achieve a good balance between load balance and overall system utilization. The peer awareness in PeerCQ is supported by virtual peer identifiers. Although our solution was developed independently [6], the concept of virtual servers in [14] also promotes the use of a varying number of peer identifiers for the purpose of load balancing. However, load balancing in PeerCQ has a unique characteristics. Its use of virtual peer identifiers is combined with CQ grouping and relaxed matching techniques, making the PeerCQ service partitioning scheme unique and more scalable in handling hot spot monitoring requests. Finally, our dynamic passive replication scheme shares some similarity to the replication techniques used in CFS [3]

with a number of major differences. First, in PeerCQ, there is a need for updating the state of the replicas as the CQs execute and change state. Thus, the replication scheme needs to maintain strong consistency among replicas. Second, PeerCQ provides dynamic relocation of CQ executors as better peers for executing them join the system to maintain the relaxing matching dynamically.

6 Conclusion

We have described PeerCQ, a fully decentralized peer-to-peer architecture for Internet-scale distributed information monitoring applications. The main contribution of the paper is the smart service partitioning scheme at the PeerCQ protocol layer, with the objective of achieving good load balance and good system utilization. This scheme has three unique properties: First, it introduces a donation-based peer-awareness mechanism for handling the peer heterogeneity. Second, it introduces the CQ-awareness mechanism for optimizing hot spot CQs. Third, it integrates CQ-awareness and peer-awareness through two phase matching algorithms into the load balancing scheme while maintaining decentralization and self-configurability.

Acknowledgments

This work is partially supported by the US National Science Foundation (NSF) under a CNS Grant, an ITR grant, a Research Infrastructure grant, and a US Department of Energy (DoE) SciDAC grant, an IBM SUR grant, an IBM faculty award, and an HP equipment grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or DoE.

References

[1] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," *Proc. ICSI Workshop Design Issues in Anonymity and Unobservability*, 2000.

[2] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2001.

[3] B. Gedik and L. Liu, "Building Reliable Peer-to-Peer Information Monitoring Service through Replication," Technical Report GIT-CC-02-66, Georgia Inst. of Technology, 2002.

[4] B. Gedik and L. Liu, "PeerCQ: A Scalable and Self-Configurable Peer-to-Peer Information Monitoring System," Technical Report GIT-CC-02-32, Georgia Inst. of Technology, 2002.

[5] B. Gedik and L. Liu, "PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System," *Proc. Int'l Conf. Distributed Computing Systems*, 2003.

[6] Gnutella, "The Gnutella Home Page," http://gnutella.wego.com/, 2002.

[7] 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," *Proc. ACM Symp. Theory of Computing Author Index*, 1997.

[8] KaZaa, "The Kazaa Home Page," http://www.kazaa.com/, 2003.

[9] L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 4, pp. 610-628, July/Aug. 1999.

[10] L. Liu, C. Pu, and W. Tang, "Detecting and Delivering Information Changes on the Web," *Proc. Int'l Conf. Information and Knowledge Management*, 2000.

[11] L. Liu, W. Tang, D. Buttler, and C. Pu, "Information Monitoring on the Web: A Scalable Solution," *World Wide Web J.*, 2003.

[12] C.G. Plaxton, R. Rajaraman, and A.W. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," *Proc. ACM Symp. Parallel Algorithms and Architectures*, 1997.

[13] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems," *Proc. Int'l Workshop Peer-to-Peer Systems*, 2003.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," *Proc. ACM SIGCOMM*, 2001.

[15] G. Ricart and A.K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Comm. ACM*, pp. 9-17, 1981.

[16] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Largescale Peer-to-Peer Systems," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms*, 2001.

[17] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel, "Scribe: The Design of a Large-Scale Event Notification Infrastructure," *Proc. Int'l Workshop Networked Group Comm.*, 2001.

[18] 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, Univ. of Washington, 2001.

[19] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. ACM SIGCOMM*, 2001.

[20] 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, Univ. of California Berkeley, 2001.

Bugra Gedik is currently a PhD student in the College of Computing at the Georgia Institute of Technology. Prior to that, he received the BS degree from the Computer Science Department of Bilkent University, Turkey. His research interests are in data intensive distributed computing systems, including mobile and sensor based data management, peer-to-peer computing, and data stream processing. He is a student member of the IEEE.

Ling Liu is an associate professor in the College of Computing at the Georgia Institute of Technology. Her research involves both experimental and theoretical study of distributed data intensive systems, including distributed middleware, advanced Internet systems, and Internet data management. Her current research interests range from performance, scalability, reliability, to security and privacy of Internet systems, mobile, wireless, and sensor network systems, and pervasive computing applications. Dr. Liu has done pioneering work on continual query systems for Internet-scale information morning (OpenCQ, WebCQ, PeerCQ) and automated wrapper code generation systems (XWRAP Original, XWRAPElite, XWRAPComposer). She has published more than 100 articles in international journals and international conferences. She is currently a member of the ACM SIGMOD Executive Committee, the editor-in-chief of *ACM SIGMOD Record*, and on the editorial boards of several international journals, including the *IEEE Transactions on Knowledge and Data Engineering*, *VLDB Journal*, *International Journal of Web Services*, *International Journal of Grid and Utility Computing*. Her current research is partially funded by government grants from the US National Science Foundation, US Defense Advanced Research Projects Agency, US Department of Energy, and industry grants from IBM and HP. She is a member of the IEEE Computer Society.