CQ-Buddy: Harnessing Peers For Distributed Continuous Query Processing

Wee Siong Ng[1], Yanfeng Shu[2], and Wee Hyong Tok[2]
[1]Singapore-MIT Alliance, 4 Engineering Drive 3, National University of Singapore, Singapore-117576
[2]Department Computer Science, 3 Science Drive 2, National University of Singapore-117543

***Abstract*— In this paper, we present the design and evaluation of CQ-Buddy, a peer-to-peer (p2p) continuous query (CQ) processing system that is distributed, and highly-scalable. CQ-Buddy exploits the differences in capabilities (processing and memory) of peers and load-balances the tasks across powerful and weak peers. Our main contributions are as follows: First, CQ-Buddy introduces the notion of pervasive continuous queries to tackle the frequent disconnected problems common in a peer-to-peer environment. Second, CQ-Buddy allows for inter-sharing and intra-sharing in the processing of continuous queries amongst peers. Third, CQ-Buddy peers perform query-centric load balancing for overloaded data source providers by acting as proxies. We have conducted extensive studies to evaluate CQ-Buddy's performance. Our results show that CQ-Buddy is highly scalable, and is able to process continuous queries in an effective and efficient manner.**

***Index Terms*— CQ-Buddy, continuous queries, CQ, P2P, peer-to-peer, distributed.**

I. Introduction

Peer-to-Peer (P2P) technology, also called peer computing, is emerging as a new paradigm that is now viewed as a potential technology that could re-architect distributed architectures. In a P2P distributed system, a large number of nodes (e.g., PCs connected to the Internet) can potentially be pooled together to share their resources, information and services. The nodes, which can be both a data consumer and provider, may join and leave the P2P network at any time, resulting in a truly dynamic and ad-hoc environment. Furthermore, the nodes could have idle resources (processing and memory) which can be exploited by other nodes in a secured manner to help process a portion of a distributed task.

Continuous queries are queries that are executed for a potentially long period of time, and are used in the monitoring of data semantics in the underlying data streams to trigger user-defined actions. Continuous queries transform a passive networked structure into an active environment, and are particularly useful in distributed environments where huge volumes of information are updated frequently and remotely. For example, users may be interested in monitoring the trading volume or price of a particular stock over a period of time. They could then express their request in a continuous query as follows:

Monitor the Singapore Stock Exchange indefinitely,
notify me when Straits Time Index
current value > 1300

Fig. 1. Example of a CQ Query.

In the literature on continuous queries, much of the existing work focuses on efficiently handling the processing of a large number of continuous queries by exploiting similarity in the queries, and subsuming a new incoming query into existing queries groups [1]. These existing techniques, however, are not expected to perform well in a highly distributed environment for several reasons. First, these techniques were designed mainly based on a centralized client-server architecture. Queries are routed and registered to a single continuous query system (CQS). Thus, much of the existing work focuses on supporting as many queries as possible against external data sources. However, it is clear that there is a limit to the number of queries that can be handled by a single server, no matter how efficient the CQS may be. Second, most of these techniques focus on the data stream consumer (i.e. the system processing the continuous queries), and neglect that the data providers themselves could be potential bottlenecks. A popular data provider may be easily overwhelmed by requests and consequentially delay the response of a CQS. Third, multiple continuous query systems do not share computations, and each function autonomously and is concerned with the efficient and effective execution of continuous queries within itself. Multiple CQSs also do not share any query processing task. In short, much of the work performed by individual CQSs is duplicated. Furthermore, resources at some CQSs could be under-utilized. For example, a large number of CQSs may be accessing the same data provider, thus overloading the data provider and causing it to become a bottleneck.

In this paper, we present the design and evaluation of CQ-Buddy, a peer-to-peer (P2P) continuous query (CQ) processing system that is distributed and highly-scalable. CQ-Buddy exploits the differences in capabilities (processing and memory) of peers and load-balances the tasks across powerful and weak peers. Furthermore, CQ-Buddy introduces the notion of pervasive continuous queries, to allow complex continuous queries to be processed by other buddy peers when a peer gets disconnected. Second, CQ-Buddy allows for inter-sharing and

Wee Siong Ng is with Singapore-MIT Alliance, National University of Singapore, S16 #04-19, 3 Science Drive 2, Singapore 117543 (email: smangws@nus.edu.sg).

Yanfeng Shu is with School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543 (email: shuyanfe@comp.nus.edu.sg).

Wee Hyong Tok is with School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543 (email: tokwh@comp.nus.edu.sg).

intra-sharing in the processing of continuous queries amongst peers. In CQ-Buddy, intra-sharing of queries is achieved by grouping similar queries and processing them within the continuous query mechanism of the node, whereas inter-sharing is achieved when multiple CQ-Buddies help one another by processing continuous queries in a distributed manner. Third, we note that data providers may be overwhelmed with queries, and may become a bottleneck.CQ-Buddy peers help to alleviate overloaded data providers by performing query-centric load balancing for overloaded data source providers by acting as proxies.

II. Towards P2P Continuous Query Processing

In this section, we first provide scenarios on distributed continuous query processing on multiple sites. Next, we look at the features of P2P systems and provide examples on how P2P technology can be used to process continuous queries in a distributed manner. We also look at how peers can help and complement each other in processing queries and perform load balancing. This will also serve to motivate the need for continuous query processing using P2P technology. For this purpose, we shall refer to a node in the distributed P2P network as a *peer*.

A. Duplicate Processing of Similar Queries

Most existing continuous query systems [2], [3], [1], [4], [5] are designed to process continuous queries in an efficient manner at a single-site. In a network where there is a large number of computers (nodes), each CQS executing on each computer would process continuous queries independently. There is therefore a large possibility of duplicate processing of continuous queries in a network.

However, if the multiple CQSs executing at various peers could cooperate and "help" each other in processing the queries, the amount of duplicate processing can be significantly reduced, and thus improving overall system responsiveness. The grouping of similar queries to allow for sharing of computation has in fact been the focus of many CQSs. With increased opportunities for sharing, query processing can be further optimized holistically across all CQSs. Contrast this with a single CQS, where there are relatively lesser opportunities for similar queries.

B. Data Providers - Bottlenecks?

When a large number of peers access the same data source, the data provider itself becomes a bottleneck. Most of the existing work focuses on tackling adaptive query processing at the CQS end, but not at the data provider end. However, the data providers themselves may be overloaded by requests from multiple CQSs and hence their performance suffers. In our model, we consider two different configurations for data providers.

1) Data Provider with Multiple Nodes: In the first scenario, the data provider consists of multiple nodes, with each node providing the same set of data. The peers accessing the data providers are aware of the multiple data providers, and uses a selection policy to determine which data provider node would service a request.

2) Peers as Proxies: In the second scenario, the data provider consists of a single node. The node maintains a list of neighboring peers which it can delegate as *proxy peers*. Proxy peers fetch data on behalf of other peers, which must otherwise access the data provider node themselves. This cuts down the number of concurrent requests to the data provider node. As the load of the data provider node reduces significantly, the overall responsiveness of the system improves.

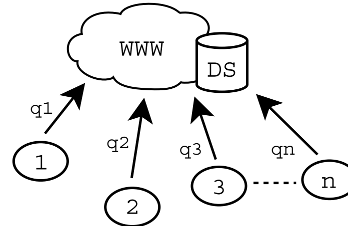
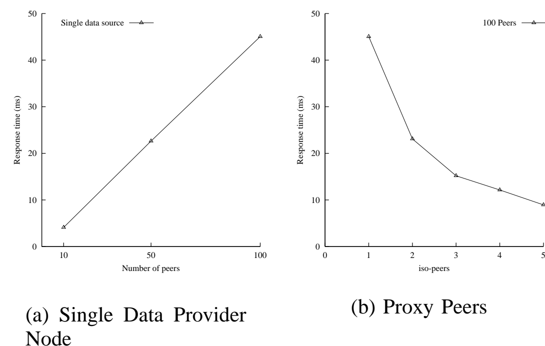


Fig. 2. Multiple peers accessing a popular data source.

Let us consider an example. In Figure 2, we have multiple peers each issuing continuous queries to a popular data provider. The data provider node quickly becomes a bottleneck, since it has to handle multiple query requests from multiple peers and send individual responses to each of them.

We conduct a simple experiment to validate this example. In the first experiment, we create a total of 100 peers (varies from 10 to 100). Each peer submits 50 queries on runtime to a CQS. In the first set of experiment, there is a single data provider node servicing the requests from the multiple peers. The average response time of peers (see Figure 3(a)) is recorded.

In the second experiment, we allow the data provider node to delegate several proxy peers to service the requests. Queries are submitted to these peers in a random manner. Figure 3(b) shows that the response time improves significantly. Thus, we can observe that by introducing proxy peers, we are able to improve the overall responsiveness of the system.



(a) Single Data Provider Node

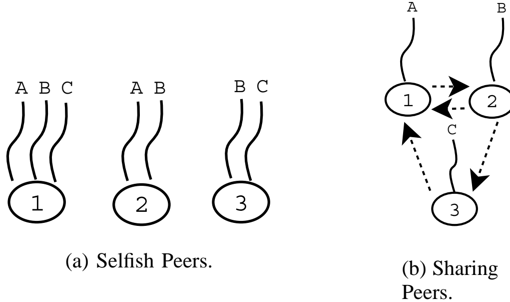
(b) Proxy Peers

Fig. 3. Experiment to show the advantages of introducing intermediate proxies

C. Resource Sharing Strategies

P2P technology facilitates the sharing of data and computing resources. Intuitively, if we can harness peers in a P2P network

to service continuous queries, there is immense potential for enhancing the reliability and performance of all the CQS participating in the P2P network. Figure 4(a) illustrates a scenario where several "selfish" peers do not share the processing of continuous queries with their neighbors, and choose to process them by themselves. Let us now consider the scenario in Figure 4(b). Each peer does not handle the entire CQ processing of its own query. Instead, it shares the processing workload with other peers in its neighborhood. Intuitively, the workload can thus be evenly distributed amongst the peers, instead of having several single overloaded peers.



(a) Selfish Peers.

(b) Sharing Peers.

Fig. 4. Peers' relation.

Furthermore, peers may not have equal resources. Peers can be running on a variety of devices, ranging from a Personal Digital Assistants (PDA) to a laptop or a desktop. The basic idea behind CQ-Buddy is to allow peers that are weaker than its neighboring peers to seek help from buddy peers in processing similar continuous queries.

D. Frequent Connection/Disconnection of Peers

Before leaving this section, let us look at an example that motivates the need for pervasiveness in continuous queries.

Let us consider a business traveller, who wishes to perform a long running computation (based on a complex financial model) on real-time updates of the STI Composite index. Furthermore, he needs the computed results upon arrival at the destination. When the traveller boards the plane, his PDA is disconnected from the network of peers. However, prior to disconnecting, his peer software asks for help from its buddy peers to perform the query. When he arrives at his destination, he powers up his PDA and immediately, the buddy peers provide him (rather his PDA) with the computed results from the complex, long running function that has been applied to an underlying continuous data stream (i.e. from the Singapore Stock Exchange).

We refer to this class of continuous queries that are processed by a peer on behalf of another peer, and retrieved at a later time period as *pervasive continuous queries*. It is useful when a peer can leverage on other buddy peers to process a long running processing during its absence from the network.

III. CQ-Buddy: A Distributed CQS Using Peer Technology

In this section, we shall present CQ-Buddy, a peer-to-peer (P2P) continuous query system. We shall first look at the CQ-Buddy network and the architecture of a CQ-Buddy node. For illustration, Figure 5 shows a CQ-Buddy network with several heterogeneous peers, including a handheld device (Peer 1), laptop (Peer 6), PCs and a server-type peer.

A. CQ-Buddy Network

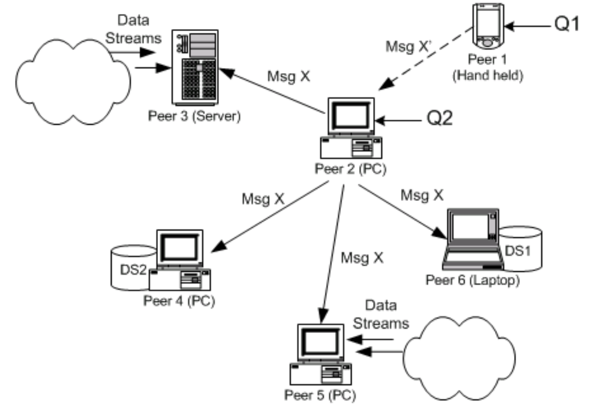


Fig. 5. Overview of CQ-Buddy Network.

CQ-Buddy is a P2P-enabled distributed CQS. In CQ-Buddy, we distinguish between two different roles of peers. First, peers can act as *proxies* to data providers and help to reduce the number of concurrent requests to the data provider nodes (e.g Peer 2 and 5 in Figure 5). Second, each peer implements a continuous query system that cooperatively interacts with other peers to process continuous queries (e.g. Peer 3 to 6 in Figure 5).

Let us consider the case where a new query Q2 = *select * from sti.stream where Stock.symbol ='Creative 50' or Stock.symbol = 'SIA'* is submitted to Peer 2. *sti.stream* retrieves the stock indexes from the Straits Time Index which is provided by the Singapore Stock Exchange. All incoming queries that are submitted by the user to a peer are first optimized. If a query is similar to one of the existing queries, it is subsumed into an existing query group.

If the query is not similar, the peer could either process it by itself or ask another peer (ie CQ-Buddy) which is already processing a similar query to help. In the second option, the peer sends a "help" message with the newly arrived query to other peers to see whether they are already processing a similar query. This hypothetical model is practical especially in a P2P environment, where some peers are more reliable and stable than the others, e.g., workstations as compared to PDAs, and dedicated network lines as compared to modem dial-ups. Stronger peers, with more resources (i.e. processing and memory) help weaker peer in processing continuous queries. Note that the objective is to locate peers which currently handle *similar processes* (i.e., monitoring data source sti.stream with projection attributes Stock.symbol), so no exact match of projection attributes is necessary.

When Peer 2 sends a "help" message to other peers, it has no advance knowledge of the number of peers that will respond. Instead, it relies on a predefined threshold (e.g., stop when 2

peers return results or when timeout sets in). In the case of an empty result, the query will be sent to the original CQS, e.g., Peer 3 and Peer 5. A new process will be created in the process pool of Peer 3 and Peer 5 since there are no similar queries that are currently running. Note that although Peer 3 and Peer 5 can always process the incoming query (either merge it into the existing local process pool for similar queries, or create a new process to handle it), that option will only be taken last in order to avoid building up a single data source bottleneck.

When a peer receives a request, it may either handle the query if it has similar queries running in its local process pool, or drop the message otherwise. *Msg X* keeps on propagating to neighboring peers and the live time is controlled by TTL (Time-to-Live). TTL indicates the maximum number of hops the message can be passed on before it expires, and this is used to avoid flooding the network. In order to break potential message loops, each peer keeps a queue of the recent messages and rejects the ones that have been processed before. Peers which are able to handle the query (i.e. able to merge the incoming query into their existing process groups) will send an acknowledgement directly to Peer 2 with its identity, BPID [1]. Peer 2 keeps the BPIDs, which may be used for further reference, e.g., to remove the query.

B. Architecture of a CQ-Buddy Node

Let us consider the architecture of a CQ-Buddy node. Figure 6 depicts the architecture of an autonomous peer in CQ-Buddy. CQ-Buddy is an extension of the BestPeer platform that provides low-level P2P facilities, e.g., communication, and search mechanism. The core of a peer in CQ-Buddy is the CQ-Manager that accepts user queries through a user interface and then invokes the underlying execution engine. Each query is optimized by the Query/Group optimizer, where it is integrated into a group of queries if it is similar to them. An incoming query will first be optimized internally with a peer. The queries or grouped queries that cannot be subsumed into an existing query will then be used as input for the P2P search engine to locate the other peers that can handle the queries. Note that the Data Manager module may not be operational in a peer, since it simply consumes data provided by the data provider and act as an intermediate proxy for other peers if there is a need for load balancing. The data manager in a peer monitors the data sources (i.e. a flat file, DBMS or data streams from devices in the network). Here, we assume that the data are read-only, and that there is an implicit time attribute tagged to all data. CQ-Manager invokes the execution engine to evaluate the installed continuous queries. Second, the CQ-Manager orchestrate the queries that are processed by other peers, and handles the return of the results to the CQ-Buddy GUI.

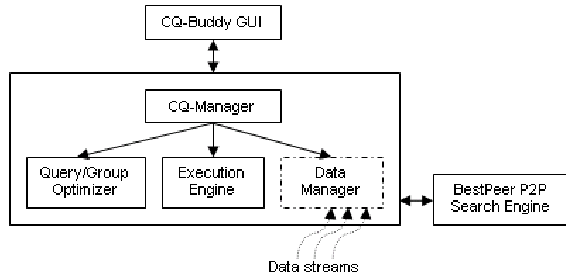


Fig. 6. Architecture of a Peer.

[1]CQ-Buddy is built on top of BestPeer [6]. BPID is a global identity used in BestPeer to uniquely identify different peers and their respective location in the dynamic network.

1) Strategies for Processing Similar Queries:

When a peer receives a new continuous query for processing, it first determines whether the continuous query is similar to any of the queries running in its existing pool. The similarity between a newly arrived continuous query and all the running queries is computed. If the newly arrived query is similar to one of the existing running queries, it will be added onto the existing query. If the newly arrived query is similar to none of the existing running queries, the peer can choose from two strategies.

In the first strategy, which we refer to as SELF-HELP, the peer initiates a new processing task to handle this new query itself. In this manner, the peer behaves exactly like a single CQS. In the second strategy, which we refer to as BUDDY-HELP, the peer asks its buddy peers for "help" in processing the query. The buddy peers then process the query on behalf of the peer, and provide the peer with the results of the continuous query. In Section IV, we perform an extensive study on the effectiveness of these two proposed strategies.

IV. A Performance Study

We have conducted detailed simulation to study the various CQ-Buddy features discussed in the previous sections. In this section, we present our extensive performance evaluation of CQ-Buddy. First, we show the benefits of CQ-Buddy in allowing multiple CQSs in a P2P network to cooperate and help each other. Second, we show how stronger peers can help weaker peers process continuous queries. Third, we consider the various proxy peer selection policies which can help a data provider reduce the number of simultaneous requests being sent to it. Finally, we look at the effects of the number of delegated peers on query response time.

A. Experiment Parameters

1) Data Sets:

We run our experiments against two different data sets, R and S. Relation R and S consists of 50,000 and 100,000 tuples respectively. We assume every join query in our experiments is a one-to-one, (i.e., each tuple in one relation finds a corresponding matching tuple in the other relation) binary join. The size of each tuple is about 1K bytes and the data values are uniformly distributed.

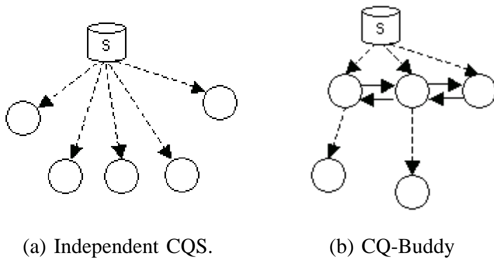
2) Queries:

In our experiments, we use three types of queries to represent the possible queries that users may submit to a CQS. We categorize queries into *Simple Selection Query*, *Range Selection Query* and *Join Query*. *Simple Selection Query* is a group of queries that have the same expression signature on the equal selection predicate on *Identity*. *Range Selection Query* is a group of queries that have the same expression signature on range selection predicate on *Change*

Ratio. *Join Query* is a class of queries that contain expression signature for both selection and join operators. Selection operators are pushed down under join operators.

B. CQ-Buddy vs. Independent CQS

In the first experiment, we compare the performance of existing CQSs with CQ-Buddy. Existing CQSs can generally be classified into two types. In the first type of CQSs, queries are shared (grouped sharing)[1], [5] techniques. In the second type of CQSs, queries are not shared [3]. We refer to the former CQS type as *GroupCQ* and the latter as *TraditionalCQ*. In the experiment, we shall consider GroupCQ, since the later is able to allow computation for similar queries to be shared and is thus more efficient and effective compared to TraditionalCQ.



(a) Independent CQS. (b) CQ-Buddy

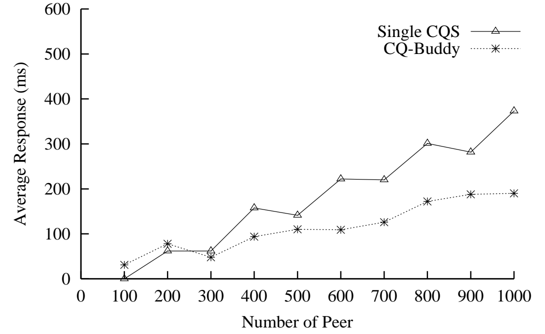
Fig. 7. Independent CQS vs CQ-Buddy

Each CQ peer consists of 10 basic queries, and another query set consisting of 10 queries following the 80-20 rule (i.e., 80% of the queries access a hot region representing 20% of the entire data stream) is introduced into the system at runtime. Queries are submitted to the peers.

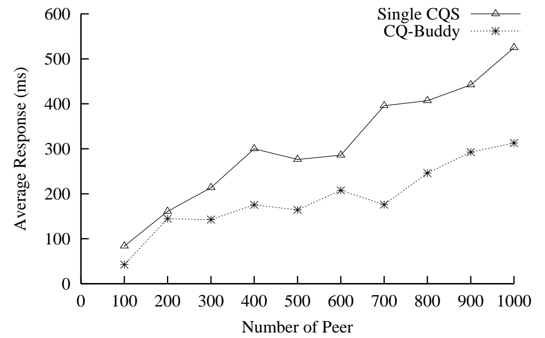
Similar to the case of a single CQS, a new query is checked to determine whether it can be shared with one of the basic queries. If the incoming queries cannot be shared, they are processed separately from the existing queries. We set the degree of overlap, for similar queries to be $\alpha = 0.4$. We vary the number of peers from 100 to 1000 peers.

In the GroupCQ case, we make use of several peers each running a CQS, independent of each other. Peers in the GroupCQ case do not interact with each other, and process the continuous queries with no knowledge of the continuous query that are being processed in other CQSs. In the CQ-Buddy case, peers help one another in processing similar queries. We compare the performance of these two cases.

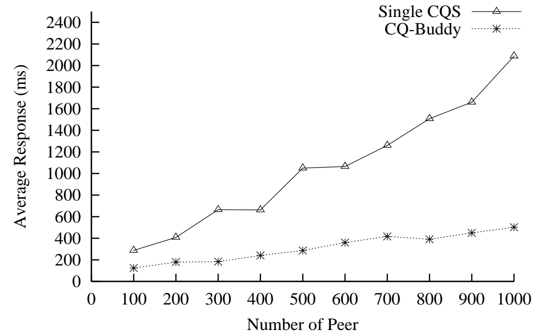
We study the performance of GroupCQ and CQ-Buddy using three types of queries: *Simple Selection Query*, *Range Selection Query* and *Join Query*. Figure 8 shows the results of the experiments. From Figure 8(a), we note that GroupCQ performs slightly better than CQ-Buddy when the number of peers is small (less than 250). This is due to cost of passing message to explore which other peers can process a similar query. However, when the number of peers increases, it can be observed that CQ-Buddy outperforms GroupCQ. In Figure 8(b) and Figure 8(c), as the nature of the operations get more complex (i.e. join queries), the benefits of being able to cooperatively process similar queries amongst peers become apparent.



(a) Selection Queries



(b) Range Queries



(c) Join Queries

Fig. 8. Traditional CQS vs. CQ-Buddy for Different Query Types

It can be observed from Figure 8 (a)-(c) that when the number of peers is small, the cost of message passing between CQ-Buddy peers dominates, and hence the performance of CQ-Buddy suffers. However, in a large P2P network, the number of peers participating is potentially large, and hence substantial benefits can be reaped from being able to cooperatively process queries.

V. Related Works

Continuous queries are used extensively as a useful tool for the monitoring of updated information. The concept of continuous queries was first introduced by Terry et al. [2] who

implemented timer-based continuous queries over append-only database. The approach is too restricted, i.e., it is confined to append-only systems and disallows deletions and modifications. Hence it is not adaptable to dynamic environments such as those found in a distributed or P2P context.

There has been considerable research done in continuous queries processing. More recently, there are several CQ systems developed or proposed for monitoring and delivering information on the Internet. OpenCQ [3] employs an SQL like query language and runs on top of a distributed information mediation system that integrates heterogeneous data sources. The NiagaraCQ system [1] and Xyleme system allow the monitoring of XML documents found on the web. In addition, both CACQ [4] and AdaptiveCQ [5] take note of the need for adaptivity and propose techniques based on the eddies mechanism to facilitate adaptive continuous query processing.

All the systems mentioned above are fundamentally different from CQ-Buddy in several ways. First, most of these existing systems utilize a centralized approach in which the server performs the processing and treat the clients as simply receiving and presenting the information to the end-user. This is typical of a client-server approach. B. Gedik and L. Liu have proposed a distributed CQ system (PeerCQ) for information-monitoring [7]. Although PeerCQ is similar to our approach on supporting CQ processing using peers, the fundamental architectures are different. PeerCQ is built on a structured DHT-based topology. In order to balance the load among the distributed peers, PeerCQ requires a careful design on mapping CQ identifiers and peer identifiers. A poor decision might form hop-spots in the network. In contrast, CQ-Buddy does not require a structured network which avoids hop-spots generation.

The requirements of CQ-Buddy match the characteristics of the P2P technology perfectly. In a pure P2P environment there are no global services, resource or schema control. P2P systems, like Napster [8], Gnutella [9] , ICQ [10] and SETI@Home provide for content sharing, communication and sharing of computational power. An evaluation of P2P systems can be found in [11]. These systems are limited to transferring content at the object level and cannot support the execution of complex queries across multiple sources, nor use intermediate results in order to answer consecutive queries.

VI. Conclusion

In this paper, we have presented a distributed system that processes continuous queries using Peer-to-Peer technology, called CQ-Buddy. We have shown that CQ-Buddy is able to provide significant performance gains by sharing continuous queries with other peers in an efficient and effective manner. The system is fully distributed and highly scalable as there is no single-point failure and single-source bottleneck. The CQ-Buddy network is dynamic and it does not require any specific network structure to be defined. Peers in the CQ-Buddy network also turn their heterogeneity to their advantage, so that "weaker" peers such as PDAs and other mobile devices are helped by "stronger" peers for complex query processing.

As shown in the evaluation, CQ-Buddy achieves significant performance gains with respect to traditional CQ systems. This is accomplished by (i) Allowing inter-sharing and intra-sharing in the processing of continuous queries amongst peers. (ii) Performing query-centric load balancing for overloaded data source providers by allowing peers to act as proxies.

References

[1] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," in *ACM SIGMOD Intl. Conf. on Management of Data*, 2000, pp. 379–390.

[2] D. Terry, D. Holdberg, D. Nichols, and B. Oki, "Continuous queries over append-only database," in *ACM SIGMOD Intl. Conf. on Management of Data*, 1992, pp. 321–330.

[3] L. Liu, C. Pu, and W. Tang, "Continual queries for internet scale event-driven information delivery," in *IEEE Knowledge and Data Engineering, Special Issue on Web Technology*, vol. 11, No.4, 1999, pp. 610–628.

[4] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *ACM SIGMOD Intl. Conf. on Management of Data*, Madison, USA, 2002, pp. 49–60.

[5] W. H. Tok and S. Bressan, "Efficient and adaptive processing of multiple continuous queries," in *Intl. Conf. on Extending Database Technology (EDBT)*, Prague, Italy, 2002, pp. 25–27.

[6] W. Ng, B. Ooi, and K. Tan, "Bestpeer: A self-configurable peer-to-peer system," in *Intl. Conf. on Data Engineering (Poster) (ICDE)*, 2002, p. 272.

[7] B. Gedik and L. Liu, "Peercq: A decentralized and self-configuring peer-to-peer information monitoring system," in *Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2003.

[8] Napster Home Page, *http://www.napster.com/*.

[9] Gnutella Development Home Page, *http://gnutella.wego.com/*.

[10] ICQ Home Page, *http://www.icq.com/*.

[11] B. Yang and H. Garcia-Molina, "Comparing hybrid peer-to-peer systems," in *Intl. Conf. on Very Large Data Bases (VLDB)*, 2001, pp. 561–570.

Wee Siong Ng Ng Wee Siong is a research fellow in the National University of Singapore under the Singapore-MIT Alliance (SMA). His current research interests cover Peer-to-Peer data management, distributed query processing and database performance issues. He has received BIT (Bachelor of Information Technology) from University Malaysia Sarawak (UNIMAS). He has submitted his Ph.D. thesis recently.

Yanfeng Shu got her MSc from SouthEast Univery, china, in 1996, and her BSc from Harbin Institute of Technology, China, in 1993. She is now a Phd student in Computer Science Department, NUS. Her research interests includes query processing and optimization in relational databases, and P2P.

Wee Hyong obtained his MSc, BSc (Hons) from School of Computing (SoC), National University of Singapore (NUS) in 2002, 2000 respectively. Currently, he is a teaching assistant in the Computer Science Department, NUS. His research interests includes adaptive query processing and optimization issues in continuous query (CQ) systems, XML.