QONCH: A System for Distributed Network Querying using Bounded Approximate Caching

Badrish Chandramouli Jun Yang
Dept. of Computer Science
Duke University
{badrish,junyang}@cs.duke.edu

Amin Vahdat
Dept. of Computer Science and Engineering
University of California, San Diego
vahdat@cs.ucsd.edu

ABSTRACT

As networks continue to grow in size and complexity, distributed network monitoring and resource querying are becoming increasingly difficult and costly. We have designed, built, and evaluated a scalable infrastructure for answering queries over distributed measurements, while reducing costs (in terms of both network traffic and query latency) and maximizing precision of results. In this infrastructure, each network node owns a set of numerical measurement values and actively maintains bounds on these values cached at other nodes. We can then answer queries approximately, using bounds from nearby caches to avoid contacting the owners directly. We argue that approximate results are acceptable for our target applications, as long as errors are quantified precisely and reported to the user, and there is a mechanism for the user to obtain results with a specified precision. We have designed, implemented, and evaluated two approaches: One, called QONCH-1, uses a recursive partitioning of the network space to place caches in a static, controlled manner, while the other, called QONCH-2, uses a locality-aware *distributed hash table* to place caches in a dynamic and decentralized manner. We use large-scale network emulation to demonstrate that our techniques are very effective in reducing query costs while generating an acceptable amount of background traffic. They are also able to exploit various forms of locality that are naturally present in queries, and adapt to volatility of measurements.

1. INTRODUCTION

Consider a network of nodes, each monitoring a number of numeric measurements. Measurements may be performance-related, e.g., per-node statistics such as CPU load and the amount of free memory available, or pairwise statistics such as latency and bandwidth between nodes. Measurements may also be application-specific, e.g., progress of certain tasks or rate of requests for particular services. We consider the problem of efficiently supporting set-valued queries of these distributed measurements. This problem has important applications such as network monitoring and distributed resource querying. For example, a network administrator may want to issue periodic monitoring queries from a workstation to a remote cluster of nodes over the wide-area network; a team of scientists may be interested in monitoring the status of an ongoing simulation running distributedly over the Grid [3]. As an example of distributed resource querying, suppose that researchers want to run experiments on PlanetLab [6], a testbed for wide-area distributed systems research. They can specify load or connectivity requirements on machines in the form of a query, and the system should return a set of candidate machines satisfying their requirements.

With increasing size and complexity of the network, the task of querying distributed measurements has become exceedingly difficult and costly in terms of time and network traffic. The naive approach to processing a query is by simply contacting the nodes responsible for the requested measurements. This approach is very expensive in terms of network cost and could interfere with measurements themselves. Our goal was to develop an infrastructure system with better support for distributed network queries, by exploiting a number of optimization opportunities that naturally arise in our target applications. First, exact answers are not needed for most of our target applications. Second, measurements that our applications are interested in do not vary in a completely chaotic manner. Third, many types of localities exist in our target applications: temporal locality among queries e.g. periodic querying, spatial locality among querying nodes, and spatial locality among nodes that own the measurements requested.

We have built a distributed querying system named QONCH (QONCH, pronounced *conch*, stands for Queries Over a Network of CacHes) that exploits the optimization opportunities discussed above. The first two opportunities can be exploited by *bounded approximate caching*, whose effectiveness is well established (e.g. in [4]). We have developed efficient and scalable techniques to place, locate, and manage bounded approximate caches across a large network, taking advantage of the localities.

We have developed two systems: The first system (QONCH-1) is described briefly in Section 2. It uses a recursive partitioning of the network space to place caches in a static, controlled manner. The second system (QONCH-2) is described in Section 3. It uses a *distributed hash table* (e.g., [8]) to place caches in a dynamic and decentralized manner.

2. SYSTEM OVERVIEW

Data and queries. Our system consists of a collection of nodes over a network. Each node monitors various numerical quantities (*measurements*), such as the amount of free memory on the node, or the latency between itself and another node. The monitoring node is called the *owner* of these measurements. A query can be issued at any node for any set of measurements over the network. The term *query region* refers to the set of nodes that own the set of requested measurements. By the way it is defined and used, a query region

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi t or commercial advantage and that copies bear this notice and the full citation on the fi rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi c permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

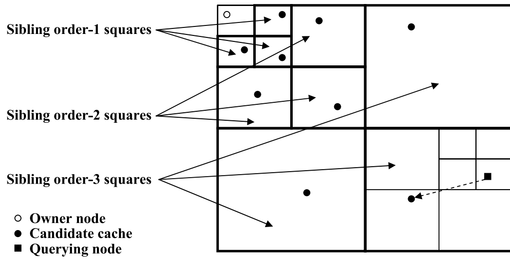

Figure 1: Recursive partitioning of network space into squares.

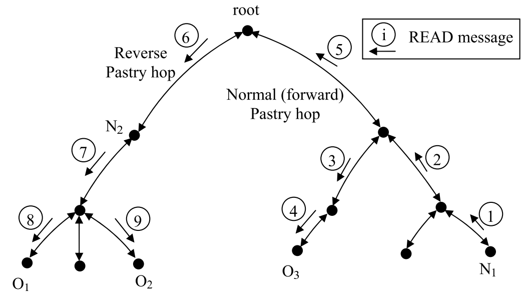

Figure 2: Two-way aggregation with Pastry.

often exhibits locality in some space. We will concentrate on the case where regions exhibit locality in terms of network proximity, which is common in practice.

For a query that simply requests a set of measurements from a region, the result consists of the values of these measurements. Our system allows a query to specify an *error bound*; a stale measurement value can be returned in the result as long as the system can guarantee that the "current" measurement value (taking network delay into account) lies within the specified error bound around the value returned.

Nodes can issue relational-style set-valued queries over measurements owned by any nodes in the network. Our query interface allows complex queries to be composed from operators such as region creation, selection, and join. For example, a query to select nodes from two regions, with low CPU load and high memory availability, such that they have a good connection between them (in terms of high available bandwidth and low node-to-node latency) is written as follows. A call to region() returns the set of nodes in a specified hypersphere in the network space. A numerical comparison can also specify a desired precision.

```
r1 = region(...);
r2 = region(...);
r3 = select(r1, CPU_LOAD($1) < 0.5 bound 0.1
            and AVAIL_MEM($1) >= 100 bound 10);
r4 = select(r2, CPU_LOAD($1) < 0.5 bound 0.1
            and AVAIL_MEM($1) >= 100 bound 10);
r5 = join(r3, r4, LATENCY($1,$2) < 50 bound 10
             and AVAIL_BW($1,$2) > 100 bound 10);
output r5;
```

Bounded approximate caching. Caching is a natural and effective way to utilize previously obtained measurement values. We use *bounded approximate caching*, where bounds on cached measurement values are actively maintained by the measurement owners (directly or indirectly).

Let node N be the owner or a cache of a measurement. N may be responsible for maintaining bounds for multiple other caches of the same measurement; we call these caches *child caches* of N, and we call N their *cache provider* (with respect to the measurement). A cache provider maintains a list of *guarantees*, one for each of its child caches. Whenever a guaranteed bound is violated, the cache provider updates the child with the new value and bounds. This repeats with the child informing its own guaranteed children and so forth. There are sophisticated techniques for setting bounds dynamically and adaptively (e.g., [5]); we do not consider them in our work. We focus on techniques for *selecting* bounded approximate caches across the network to exploit query locality and the trade-off between query and update traffic, and for *locating* these caches quickly and efficiently to answer queries.

Selecting and locating caches. We have developed two approaches to selecting and locating caches in the network. The first approach (QONCH-1) uses a hierarchy induced by recursive partitioning of the network to spread caches throughout the system in a controlled manner: Each owner preselects a number of nodes as its potential caches, such that nearby owners have a good probability of selecting the same node for caching, allowing queries to obtain cached values of measurements in large regions from fewer nodes. The selection scheme is shown in Figure 1. We recursively partition a d-dimensional network space (defined in terms of some distance metric such as latency) into successively smaller *squares* (d-dimensional hyper- rectangles). We allow each owner O to select a candidate cache in each of its *sibling squares*: an order-i sibling square of O is an order-i square that belongs to the same order-$(i+1)$ square as O, but does not contain O itself. This scheme ensures that the candidate caches provide reasonable coverage of the entire space. A cache locator function is used to select the cache – the function ensures that nearby owners tend to choose the same representative with high probability. We do not provide further details regarding this approach due to space considerations. This approach has a number of disadvantages. First, there is a need for centralized state to compute the cache locator functions. Second, this approach does not adapt to the workload and does not exploit potential locality among querying nodes at runtime. Finally, this approach restricts the amount of caching at any node (irrespective of node capability) by design. We will not focus on this approach in our demonstration; if visitors are interested we can demonstrate the comparison of the two approaches.

These disadvantages are overcome in the second approach, which uses a locality-aware DHT to achieve locality- and workload-aware caching in an adaptive manner. With the use of DHT, the system is much more decentralized than in the controlled approach. The downside is a lesser degree of control in exploiting locality, and more complex protocols to avoid centralization. This approach is presented in detail in Section 3.

3. DESIGN OF QONCH-2

To combat the problems with the controlled caching approach, we propose a dynamic, DHT-based approach to placing and locating caches that adapts well to a changing query workload. There are several high-level reasons for using DHTs: The technology scales to a large number of nodes, the amount of state maintained by each node is limited, the system uses no centralized directory, and it copes well with changing network conditions. We use Pastry [8], a popular DHT that provides a scalable distributed object location and routing substrate for P2P applications. Pastry's properties provide us a natural way to aggregate messages originated from close-by nodes.

3.1 Caching with Pastry

Our basic idea is to leverage a locality-aware DHT such as Pastry in building a caching infrastructure where two types of aggregation naturally take place. One type of aggregation happens on the owner side: Close-by owners select same caching nodes nearby, allowing us to exploit the spatial locality of measurements involved in region-based queries. The other type of aggregation happens on the querying node side: Close-by querying nodes can also find common caches nearby, allowing us to exploit the spatial locality among querying nodes.

Suppose that all nodes route towards a randomly selected root using Pastry. The Pastry routes naturally form a tree $\mathcal{T}$ (with bidirectional edges) exhibiting both types of aggregation, as illustrated in Figure 2. Queries first flow up the tree following normal (forward) Pastry routes, and then down to owners following reverse Pastry routes. Nodes along these routes are natural candidates for caches. Our system grows and shrinks the set of caches organically based on demand, according to a cost/benefit analysis using only locally maintained information. The operational details of our system are presented next.

3.1.1 Initialization, Querying, and Updating

A primary objective of the initialization phase is to build the structure $\mathcal{T}$. While Pastry itself already maintains the upward edges (hops in forward Pastry routes), our system still needs to maintain the downward edges (hops in reverse Pastry routes). These are maintained as a representation of the set of nodes found in each subtree, which we call a *subtree filter*. Subtree filters are used to forward messages on reverse Pastry paths. Subtree filters for large sets are implemented with *Bloom filters* [1].

When a query is issued for a set of measurements, the querying node routes a `READ` message towards the root via Pastry. When a node N receives a `READ` message, it checks to see if it can provide any subset of the measurements requested and if yes it responds to the querying node. The unanswered measurement requests are forwarded towards the owners of those measurements. The requests are either forwarded along reverse paths (for measurements whose owners are located below) or up along forward paths (for the remaining measurements). Figure 2 shows the flow of `READ` messages when node N_1 queries measurements owned by O_1, O_2, and O_3, assuming that no caching takes place. If node N_2 happens to cache measurements owned by O_1 and O_2, then messages 7 through 9 will be saved.

Updates due to bound violations are processed as described in Section 2. In essence, the tree shown by dotted arrows in Figure 3 provides a scalable structure for multicasting `CACHE_UPDATE` messages.

3.1.2 Adding and Removing Caches

Each node in our system has a *cache controller* thread that periodically wakes up and makes caching decisions. Suppose that a node N decides to start caching a particular measurement m. Let P_m denote the first node that can be N's cache provider on the shortest path from N to the owner of m in $\mathcal{T}$. Let $\mathcal{C}_m$ denote the subset of P_m's child caches whose shortest paths to P_m go through N. An example of these nodes is shown in Figure 3. The `SPLICE_IN` operation causes P_m to be responsible for updating N, and N to take over the responsibility of updating $\mathcal{C}_m$, as illustrated in Figure 3 on the right. Now if N decides to stop caching m, the reverse of the above operation occurs, and is called a `SPLICE_OUT` operation. We do not discuss the details of these operations due to space considerations.

3.1.3 Caching Decisions

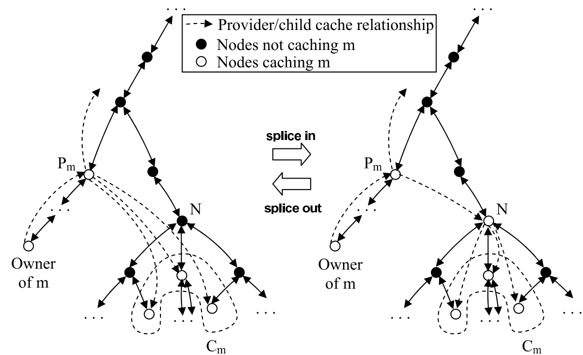

Figure 3: Splicing: adding and removing a cache.

For each measurement m that N has information about, the cache controller thread computes the benefit and cost of caching m. We have broken down the components of benefit and cost. $B_{read}(m)$ denotes the benefit in terms of reduction in read traffic. $B_{update}(m)$ is the net benefit in terms of reduction in update traffic. $C_{update}(m)$ is the cost in terms of resources (processing, storage, and bandwidth) incurred by N for maintaining its child caches for m. Finally, $C_{cache}(m)$ is the cost incurred by N for caching m (other than $C_{update}(m)$). We have derived formulas for each of these components. Let T_{update} specify the maximum amount of resources that the node is willing to spend on maintaining its child caches, and T_{cache} specify the maximum size of the cache. We use a simple greedy algorithm by defining the *pseudo-utility* of caching m as

$$\frac{B_{read}(m) + B_{update}(m)}{C_{update}(m)/T_{update} + C_{cache}(m)/T_{cache}}.$$

It is basically a benefit/weighted-cost ratio of caching m. The greedy algorithm decides to cache measurements with highest, non-negative pseudo-utility values above some threshold.

4. SYSTEM ARCHITECTURE

We have implemented both QONCH-1 and QONCH-2. The implementation of QONCH-1 consists of around 3000 lines of C++ code. For QONCH-2, we used the MACEDON [7] implementation of the Pastry DHT. MACEDON is an infrastructure for designing and implementing robust networked systems; it allows us to plug in different DHT implementations without changing the rest of the code. Our implementation of the DHT-based QONCH-2 on top of MACEDON consists of around 4500 lines of C++ code. We focus on QONCH-2 in this section.

Figure 4 shows the overall architecture of our evaluation methodology. We emulate a large network using ModelNet [9], a scalable Internet emulation environment. ModelNet enables researchers to deploy unmodified software in a configurable Internet-like environment and subject them to varying network conditions. A set of *edge emulation nodes* run the software code to be evaluated; all packets are routed through a set of *core emulation nodes*, which cooperate to subject the traffic to the latency, bandwidth, congestion constraints, and loss profile of a target network topology. These nodes are shown in the figure as the lowest layer. Experiments with several large-scale distributed services have demonstrated ModelNet's effectiveness.

For all our experiments, we use 20,000-node INET [2] topologies with a subset of 250 nodes participating in measurement and querying activities. These nodes are emulated by twenty 2.0GHz Intel Pentium 4 edge emulation nodes running Linux 2.4.27. All

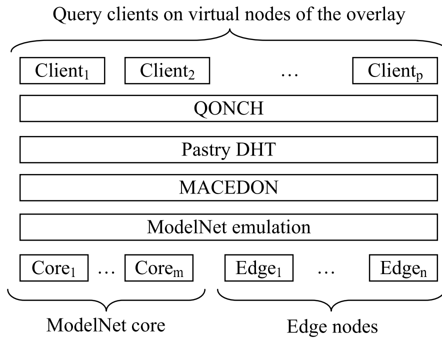

Figure 4: Evaluation architecture.

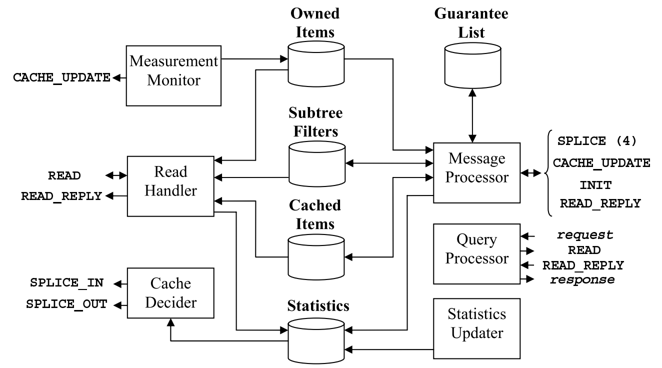

Figure 5: System architecture.

traffic passes through a 1.4GHz Pentium III core emulation node running modified FreeBSD-4.9. We deploy ModelNet on these nodes, and run MACEDON on the emulated overlay nodes. The Pastry DHT is deployed using MACEDON. Finally, QONCH-2 runs on top of the API exposed by the Pastry implementation of MACEDON. QONCH exposes an interface for issuing queries; this interface can be accessed via a local socket connection. Querying clients run on the virtual nodes in the overlay, and accept queries in the query language described earlier (either from the command line or via a batch interface). The requests for measurements of specified bound width are sent to QONCH via the socket interface.

Figure 5 shows the detailed architecture of QONCH-2. The *Request Processor* receives the measurement request, which consists of the set of owners and measurements desired, and the requested bound width. It sends a READ request to the *Read Handler*, which checks if the requested item is either owned locally or present in the cache at the requested bound width. If yes, the READ_REPLY is sent immediately. The unanswered part is forwarded as described in Section 3. The Read Handler at each step performs the same tasks until the request reaches either a cache or an owner. Each owner has a *Measurement Monitor* that monitors and updates measurements into the database of owned items. Whenever a guaranteed bound is violated, a CACHE_UPDATE is sent to the child cache, which in turn sends the same to its violated guarantees. The *Statistics Updater* updates the statistics periodically, e.g. it calculates the local hit rate as a moving average by using a hit counter that is periodically reset. The *Cache Decider* wakes up periodically and uses the collected statistics to make caching decisions. Based on the decisions, it send out SPLICE_IN and SPLICE_OUT messages. The *Message Processor* intercepts a number of messages, updates statistics as necessary, updates the cache if necessary, and forwards the messages or generates new messages (such as a SPLICE_IN_OK in response to a SPLICE_IN request that can be satisfied). It uses the *subtree filters* to perform reverse forwarding if the message is on the reverse route, otherwise normal Pastry routing is used in the forward direction.

5. DEMONSTRATION

The demonstration will show how QONCH-2 outperforms both the naive approach of contacting the owners directly, and QONCH-1. We will also show, by subjecting the system to a combination of different types of workloads, that the cache selection scheme used by QONCH-2 is more dynamic and workload-aware than QONCH-1. Although QONCH-1 is simple and does much better than the naive approach, its controlled approach does not exploit potential locality among querying nodes at runtime. For example, it is possible for a number of close-by nodes to request the same faraway owner over and over, and still not find a cache nearby, because by design there are fewer candidate caches farther from the owner, and the static cache selection scheme will not adapt to the query workload. In contrast, QONCH-2 will select a cache nearby as soon as the combined request rate from all querying nodes makes caching cost-effective. We will demonstrate the ability of the DHT-based QONCH-2 to take advantage of query region locality. We will also demonstrate the ability of QONCH-2 to adapt to volatility in measurements. For example, when the update rate is very high the system will revert to a no-caching mode because it is no longer beneficial to cache. QONCH-2 makes distributed network querying and network monitoring more scalable and less costly (in terms of both network traffic and query latency), and our demonstration will highlight the advantages of bounded approximate caching on a large scale network using our developed techniques.

6. REFERENCES

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, pages 13(7):422–426, 1970.

[2] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[3] I. Foster and C. Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., 1999.

[4] C. Olston. *Approximate Replication*. PhD thesis, Stanford University, 2003.

[5] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, 2001.

[6] PlanetLab. http://www.planet-lab.org.

[7] A. Rodriguez, C. Killian, D. Kostić, S. Bhat, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[8] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[9] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 2002.