Distributed Network Querying with Bounded Approximate Caching

Badrish Chandramouli[1], Jun Yang[1], and Amin Vahdat[2]

[1] Dept. of Computer Science, Duke University
{badrish, junyang}@cs.duke.edu
[2] Dept. of Computer Science and Engg., 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. Our aim is to design, build, and evaluate a scalable infrastructure for answering queries over distributed measurements, at reduced costs (in terms of both network traffic and query latency) while maintaining required precision. In this infrastructure, each network node owns a set of numerical measurements and actively maintains bounds on these values cached at other nodes. We can answer queries approximately, using bounds from nearby caches to avoid contacting the owners directly. We focus on developing efficient and scalable techniques to place, locate, and manage bounded approximate caches across a large network. We have developed two approaches: One uses a recursive partitioning of the network space to place caches in a static, controlled manner, while the other uses a locality-aware *distributed hash table* to place caches in a dynamic and decentralized manner. In this paper, we focus on the latter approach. Experiments over a large-scale emulated network show 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. These measurements may be related to performance, e.g., per-node statistics such as CPU load and the amount of free memory, or pairwise statistics such as latency and bandwidth between nodes. Measurements may also be application-specific, e.g., progress of certain tasks, rate of requests for particular services, popularity of objects in terms of number of recent hits, etc. Such measurements are of interest to distributed monitoring systems (e.g., Ganglia [8]) as well as systems requiring support for querying distributed resources (e.g., PlanetLab [12] and the Grid [6]).

We consider the problem of efficiently supporting relational-style queries over these distributed measurements. For example, a network administrator may want to issue periodic monitoring queries from a workstation over a remote cluster

M.L. Lee, K.L. Tan, and V. Wuwongse (Eds.): DASFAA 2006, LNCS 3882, pp. 374–388, 2006.
© Springer-Verlag Berlin Heidelberg 2006

of nodes; a team of scientists may be interested in monitoring the status of an ongoing distributed simulation running over the Grid. The results of these monitoring queries may be displayed in real time in a graphical interface on the querying node, or used in further analysis. As another example, consider relational-style querying of distributed resources. Suppose there are two sets of nodes. A query may request pairs of nodes (one from each set) satisfying the following condition: Both nodes have low load (which can be expressed as relational selection conditions), and the latency between them is low (which can be expressed as a relational join condition). Such queries are typical in resource discovery, e.g., when a Grid user wants to select a data replica and a compute server among candidate replicas/machines to perform a job, or when a distributed systems researcher wants to select some nodes on PlanetLab with desired load and connectivity requirements for running experiments.

With increasing network size and complexity, the task of querying distributed measurements has become exceedingly difficult and costly in terms of time and network traffic. Processing a query naively (by simply contacting the nodes responsible for the requested measurements) is very expensive, as we will demonstrate in our experiments. If kept unchecked, network activities caused by the queries could interfere with normal operations and lead to unintended artifacts in performance-related measurement values. These problems are exacerbated by periodic monitoring queries, by queries that request measurements from a large number of nodes, and by queries that return a large result set.

We seek to develop a better infrastructure for distributed network querying, by exploiting optimization opportunities that naturally arise in our target applications: (1) *Approximation*: For most network monitoring and resource querying applications, exact answers are not needed. Approximate values will suffice as long as the degree of inaccuracy is quantified and reported, and the user can control the degree of inaccuracy. Small errors usually have little bearing on how measurements are interpreted and used by these applications; at any rate, these applications already cope with errors that are inevitable due to the stochastic nature of measurements. (2) *Locality*: Many types of localities may be naturally present in queries. There is temporal locality in periodic monitoring queries and queries for popular resources. There may also be spatial locality among nodes that query the same measurements; for example, a cluster of nodes run similar client tasks that each check the load on a set of remote servers to decide which server to send their requests. Finally, there may be spatial locality among measurements requested by a query; for example, a network administrator monitors a cluster of nodes, which are close to each other in the network.

We have built a distributed querying infrastructure that exploits the optimization opportunities discussed above. The first opportunity can be exploited by *bounded approximate caching* [10] of measurement values. To ensure the quality of approximation, the system actively updates a cache whenever the actual value falls outside a prescribed bound around the cached value. The effectiveness of bounded approximate caching has been well established [10]. In this paper, we focus on developing efficient and scalable techniques to place, locate, and manage

bounded approximate caches across a large network, so that locality, the second opportunity mentioned above, is also exploited in an effective manner.

The naive approach is to cache queried measurements just at the querying node. Unfortunately, this approach is not very effective in our setting. First, queries from other nodes have no efficient way of locating these caches. Second, bounded approximate caches are more expensive to maintain than regular caches, because nodes with the original measurements must actively update bounded approximate caches when their bounds are violated. For regular caches, because of low cache maintenance overhead, one can take an aggressive approach of caching every miss and discard it later if it turns out to be not beneficial. The naive approach may well work if such an aggressive approach is feasible. However, we do not have such luxury for bounded approximate caching; we must carefully weigh its cost and benefit before deciding to cache a measurement, because of the costs incurred in establishing, maintaining, and tearing down a bounded approximate cache. With the naive approach of caching only at the querying node, since caching only benefits the querying node itself, it is unlikely that this benefit will outweigh the cost of caching.

Therefore, we need to find an effective way to aggregate the benefits of caching by making caches easier to locate and more accessible to querying nodes. We would also like to exploit locality in query workload by encouraging the same node to cache measurements that are frequently queried together, and by encouraging a measurement to be cached close to nodes that are querying it. Moreover, we need to base our caching decision on a cost/benefit analysis that seeks to minimize the overall foreground traffic (for queries) and background traffic (for cache updates and maintaining statistics for caching decisions) in the system. Accomplishing these goals in a scalable manner, without relying on central servers and access to global knowledge of the system, is a challenging task.

We have developed two approaches. The first approach uses a recursive partitioning of the network space to place caches in a static, controlled manner, and is described briefly in Section 2. The second approach (described in Section 3) uses a *distributed hash table* (DHT) such as [14] to place caches in a scalable, dynamic and decentralized manner. Both approaches are designed to capture various forms of locality in queries to improve performance. We show how to make intelligent caching decisions using a cost/benefit analysis, and we show how to collect statistics necessary for making such decisions with minimum overhead. Using experiments running on ModelNet [16], a scalable Internet emulation environment, we show in Section 4 that our solution significantly reduces query costs while incurring low amounts of background traffic; it is also able to exploit localities in the query workload and adapt to volatility of measurements.

Although we focus on network monitoring and distributed resource querying as motivation for our work, our techniques can be adapted for use by many other interesting applications. In [3], we briefly describe how to generalize the notion of a "query region" from one in the network space to one in a semantic space. For example, a user might create a live bookmark of top ten Internet discussion forums about country music, approximately ranked according to some popularity

measure (e.g., total number of posts and/or reads during the past three hours), and have this bookmark refreshed every five minutes using a periodic query. In this case, the query region is "discussion forums about country music," and the popularity measurements of these sites are requested. Generalization would allow our system to select a few nodes to cache all data needed to compute this bookmark, and periodic queries from users with similar bookmarks will be automatically directed to these caches.

2 System Overview

Data and queries. Our system consists of a collection of nodes over a network. Each node monitors various numerical quantities, such as the CPU load and the amount of free memory on the node, or the latency and available bandwidth between this and another node. These quantities can be either actively measured or passively observed from normal system and network activities. We call these quantities *measurements*, and the node responsible for monitoring them 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 measurements requested. Our system allows a query to define its region either by listing its member nodes explicitly, or by describing it semantically, e.g., all nodes in some local-area network, or all nodes running public HTTP servers. By the manner in which it is defined and used, a query region often exhibits locality in some space, e.g., one in which nodes are clustered according to their proximity in the network, or one in which nodes are clustered according to the applications they run. For now, we will concentrate on the case where regions exhibit locality in terms of network proximity, which is common in practice. In [3], we briefly discuss how to handle locality in other spaces.

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* $[-\delta_q^-, \delta_q^+]$; 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. To be more precise, suppose that the current time is t_{curr} and the result contains a measurement value v_{t_0} taken at time t_0. The system guarantees that v_t, the value of the measurement as monitored by its owner at time t, falls within $[v_{t_0} - \delta_q^-, v_{t_0} + \delta_q^+]$ for any time $t \in [t_0, t_{curr} - lag]$, where lag is the maximum network delay from the querying node to the owner of the measurement (under the routing scheme used by the system). More discussion on the consistency of query results in our system can be found in [3].

Beyond simple queries, our system also supports queries involving relational selections or joins over bounded approximate measurement values. Results of such queries may contain "may-be" as well as "must-be" answers. The details of the query language and its semantics are beyond the scope of this paper.

Bounded approximate caching. As discussed in Section 1, the brute-force approach of contacting each owner to obtain measurement values is unnecessary, expensive, and can cause interference with measurements. Caching is a natural and effective solution but classic caching is unable to bound the error in stale cached values. Instead, we use *bounded approximate caching*, where bounds on cached measurement values are actively maintained by the measurement owners (directly or indirectly).

The owner (or a cache) of a measurement is referred to as a *cache provider* (with respect to that measurement) if it is responsible for maintaining one or more other caches, called *child caches*, of that measurement. Each *cache entry* contains, among other information, the cached measurement value and a bound $[-\delta^-, \delta^+]$. A cache provider maintains a list of *guarantee entries*, one for each of its child caches. A guarantee entry mirrors the information contained in the corresponding child cache entry, and is used to ensure that the guaranteed bounds of child caches are maintained. We require the bound of a child cache to contain the bound of its provider cache.

Whenever the measurement value at a cache provider changes, it checks to see if any of its child caches need to be updated with a new value and bound. If yes, the provider notifies the affected child caches. The cache entries at these child caches and the guarantee entry at the provider are updated accordingly. This process continues from each provider to its child caches until we have contacted all the caches that need to be updated. This update of bounded approximate caches is similar to the update dissemination techniques described in [15]. We use a timeout mechanism to handle network failures (see [3] for details).

The choice of bounds is up to the application issuing queries. Tighter bounds provide better accuracy, but may cause more update traffic. There are sophisticated techniques for setting bounds dynamically and adaptively (e.g., [11]); such techniques are largely orthogonal to the contributions of this paper. Here, we focus on techniques for *selecting* bounded approximate caches to exploit locality and the tradeoff between query and update traffic, and for *locating* these caches quickly and efficiently to answer queries. These techniques are outlined next.

Selecting and locating caches. We have developed two approaches to selecting and locating caches in the network. The first is a controlled caching approach and is described in [3]. The idea is to use a coordinate space such as the one proposed by *Global Network Positioning* (*GNP*) [9] for all nodes in the network, and perform controlled caching based on a hierarchical partitioning of the GNP space. 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 also ensures that no single node is responsible for caching too many measurements, and that the caches are denser near the owner and sparser farther away; therefore, queries from nearby nodes get better performance. We show in [3] that this approach does quite well compared to the naive approach of contacting the node responsible for the requested measurements. This approach, however, exploits some but not all types of locality that

we would like to exploit and also restricts the amount of caching at any node by design. There is also a concern of scalability because some nodes carry potentially much higher load than other nodes. Nevertheless, because of its simplicity, the GNP-based approach is still viable for small- to medium-sized systems.

This led us to develop a new approach which has a number of advantages over the first one and is the focus of this paper. This second approach uses a locality-aware DHT to achieve locality- and workload-aware caching in an adaptive manner. Not only do nearby owners tend to select the same nodes for caching (as in the controlled approach), queries issued from nearby nodes for the same measurements also encourage caching near the querying nodes. With the use of a DHT, the system is also more decentralized than in the controlled approach. We use DHTs because the technology scales to a large number of nodes, the amount of state at each node is limited, it uses no centralized directory, and it copes well with changing network conditions. The downside is a lesser degree of control in exploiting locality, and more complex protocols to avoid centralization. This approach is presented next.

3 DHT-Based Adaptive Caching

Background on DHTs. An *overlay network* is a distributed system whose nodes establish logical *neighbor* relationships with some subset of global participants, forming a logical network overlayed atop the IP substrate. One type of overlay network is a *Distributed Hash Table* (*DHT*). As the name implies, a DHT provides a hash table abstraction over the participating nodes. Nodes in a DHT store data items; each data item is identified by a unique key. An overlay routing scheme delivers requests for a key to the node responsible for storing the data item with that key. Routing proceeds in multiple hops and is done without any global knowledge: Each node maintains only a small set of neighbors, and routes messages to the neighbor that is in some sense "closest" to the destination.

Pastry [14] is a popular DHT that takes network proximity into account while routing messages. A number of properties of Pastry are relevant to our system. The *short-hops-first* property, a result of locality-aware routing, says that the expected distance traveled by a message during each successive routing step increases exponentially. The *short-routes* property says that the average distance traveled by a Pastry message is within a small factor of the network distance between the message's source and destination. The *route-convergence* property concerns the distance traveled by two messages sent to the same key before their routes converge. Studies [14] show that this distance is roughly the same as the distance between the two source nodes. These properties provide us a natural way to aggregate messages originated from close-by nodes.

Overview of 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 1. 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 based on demand, according to a cost/benefit analysis using only locally maintained information. The operational details of our system are presented next. We do not discuss cache updates because the process is similar to that described in Section 2 (see [3] for details).

Initialization. A primary objective of the initialization phase is to build the structure $\mathcal{T}$. While Pastry itself already maintains the upward edges (forward Pastry hops), our system still needs to maintain the downward edges (reverse Pastry hops). To this end, every node in $\mathcal{T}$ maintains, for each of its child subtree in $\mathcal{T}$, a representation of the set of nodes found in that subtree, which we call a *subtree filter*. Subtree filters are used to forward messages on reverse Pastry paths, as we will discuss later in connection with querying. Nodes at lower levels can afford to maintain accurate subtree filters because the subtrees are small. Nodes at higher levels, on the other hand, maintain lossy subtree filters implemented with *Bloom filters* [1].

During the initialization phase, after the overlay network has been formed, each node in the system sends an `INIT` message containing its IP address towards the root. Each node along the path of this message adds the node IP to the subtree filter associated with the previous hop on the path. As an optimization, a node can combine multiple `INIT` messages received from its children into a single `INIT` message (containing the union of all IP addresses in the messages being combined), and then forward it.

Querying. When a query is issued for a set of measurements, the querying node routes a `READ` message towards the root via Pastry. This message contains the IP address of the querying node and the set of measurements requested (along with acceptable bounds). When a node N receives a `READ` message, it checks to

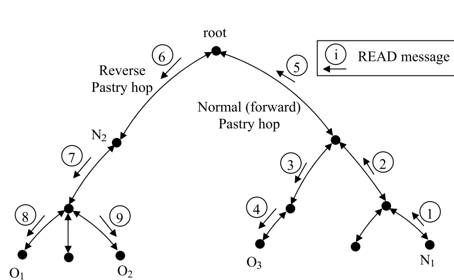

Fig. 1. Two-way aggregation with Pastry

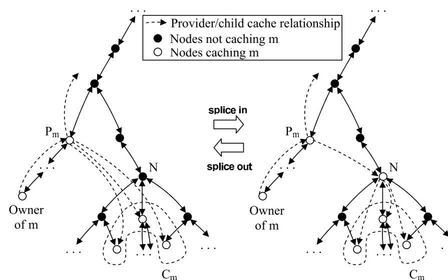

Fig. 2. Splicing: add/remove a cache

see if it can provide any subset of the measurements requested. If yes, N sends back to the querying node a READ_REPLY message containing these measurement values (with cached bounds and timestamp, if applicable). If all requested measurements have been obtained, we are done. Otherwise, let $\mathcal{O}$ denote the set of nodes that own the remaining measurements. N checks each of its subtree filters $\mathcal{F}_i$: If $\mathcal{O} \cap \mathcal{F}_i \neq \varnothing$, N forwards the READ message to its i-th child with the remaining measurements owned by $\mathcal{O} \cap \mathcal{F}_i$ (unless the READ message received by N was sent from this child in the first place). Note that messages from N to its children follow reverse Pastry routes. Finally, if the READ message received by N was sent from a child (i.e., on a forward Pastry route), N also forwards the READ message to its parent unless N is able to determine that all requested measurements can be found at or below it.

As a concrete example, Figure 1 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. It is possible to show that our system attempts to route queries towards measurement owners over $\mathcal{T}$ in an optimal manner. We do not discuss the effect of false positives in Bloom filters in this paper; the reader is referred to [3] for details.

Adding and removing caches. Each node in our system has a *cache controller* thread that periodically wakes up and makes caching decisions. We first describe the procedures for adding and removing a cache of a measurement.

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 2. After N caches m, we would like P_m to be responsible for updating N, and N to take over the responsibility of updating $\mathcal{C}_m$, as illustrated in Figure 2 on the right. Note that at the beginning of this process, N does not know what P_m or $\mathcal{C}_m$ is. To initiate the process, N sends a SPLICE_IN message over $\mathcal{T}$, along the same path that a READ request for m would take. Forwarding of this message stops when it reaches P_m, the first node who can be a cache provider for m. We let each cache provider record the shortest incoming path from each of its child caches; thus, P_m can easily determine the subset $\mathcal{C}_m$ of its child caches by checking whether the recorded shortest paths from them to P_m go through N. Then, P_m removes the guarantee entries and shortest paths for $\mathcal{C}_m$; also, P_m adds N to its guarantee list and records the shortest path from N to P_m. Next, P_m sends back to N a SPLICE_IN_OK message containing the current measurement value and timestamp stored at P_m, as well as the removed guarantee entries and shortest paths for $\mathcal{C}_m$. Upon receiving this message, N caches the measurement value, adds the guarantee entries to its guarantee list, and records the shortest paths after truncating their suffixes beginning with N. Finally, N sends out a SPLICE_IN_OK message to each node in $\mathcal{C}_m$ to inform it of the change in cache provider. The cache removal procedure uses SPLICE_OUT

and SPLICE_OUT_OK messages. It is similar to cache addition and slightly simpler (see [3] for a detailed description).

It can be shown that, in the absence of false positives in subtree filters, a cache update originated from the owner would be sent over a minimal multicast tree spanning all caches if update messages were routed over $\mathcal{T}$.

Caching decisions. Periodically, the cache controller thread at N wakes up and makes caching decisions. For each measurement m that N has information about, the thread computes the benefit and cost of caching m. We break down the benefit and cost of caching m into four components: (1) $B_{read}(m)$ is the benefit in terms of reduction in read traffic. For each READ message received by N requesting m, if m is cached at N, we avoid the cost of forwarding the request for m, which will be picked up eventually by the node that either owns m or caches m, and is the closest such node on the shortest path from N to m's owner in $\mathcal{T}$. Let d_m denote the distance (as measured by the number of hops in $\mathcal{T}$) between N and this node. The larger the distance, the greater the benefit. Thus, $B_{read}(m) \propto d_m \times H_m$, where H_m is the request rate of m at node N. (2) $B_{upd}(m)$ is the net benefit in terms of reduction in update traffic. It's computation requires the maintenance of a large number of parameters; hence we approximate it to be proportional to the reduction in update cost from the cache provider P_m's perspective (see [3] for details). (3) $C_{upd}(m)$ is the cost in terms of resources (processing, storage, and bandwidth) incurred by N for maintaining its child caches for m. (4) $C_{cache}(m)$ is the cost incurred by N for caching m (other than $C_{upd}(m)$). We omit the details of these last three components and refer the interested reader to [3].

Given a set $\mathcal{M}$ of candidate measurements to cache, the problem is to determine a subset $\mathcal{M}' \subseteq \mathcal{M}$ that maximizes $\sum_{m \in \mathcal{M}'} \Big(B_{read}(m) + B_{upd}(m) \Big)$ subject to the cost constraints that $\sum_{m \in \mathcal{M}'} C_{upd}(m) \leq T_{upd}$, and $\sum_{m \in \mathcal{M}'} C_{cache}(m) \leq T_{cache}$. Here, T_{upd} specifies the maximum amount of resources that the node is willing to spend on maintaining its child caches, and T_{cache} specifies the maximum cache size. This problem is an instance of the *multi-constraint* 0-1 *knapsack problem*. It is expensive to obtain the optimal solution because our constraints are not small integers; even the classic single-constraint 0-1 knapsack problem is NP-complete. So, we use a greedy algorithm by defining the *pseudo-utility* of caching m as

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

It is basically a benefit/weighted-cost ratio of caching m. The greedy algorithm simply decides to cache measurements with highest, non-negative pseudo-utility values above some threshold. Caches are added and removed as described earlier.

Maintaining statistics. We now turn to the problem of maintaining statistics needed for making caching decisions. For measurements currently being cached by N, we can easily maintain all necessary statistics with negligible overhead by piggybacking the statistics on various messages. A more challenging problem is how to maintain statistics for a measurement m that is not currently cached at N. Maintaining statistics for all measurements in the system is simply not scalable. Ignoring uncached measurements is not an option either, because we

would be unable to identify good candidates among them. In classic caching, any miss will cause an item to be cached; if it later turns out that caching is not worthwhile, the item will be dropped. However, this simple approach does not work well for our system because the penalty of making a wrong decision is higher: Our caches must be actively maintained, and the cost of adding and removing caches is not negligible.

Fortunately, from the cost/benefit analysis, we observe that a measurement m is worth caching at N only if N sees a lot of read requests for m or there are a number of frequently updated caches that could use N as an intermediary. Hence, we focus on monitoring statistics for these measurements, over each *observation period* of a tunable duration. For example, the request rate H_m is maintained by N for each m requested during the observation period; request rates for unrequested, uncached measurements are assumed to be 0. Our techniques to estimate update rates and d_m over the observation period are more complex. More details on scalable maintenance of statistics are described in [3].

Overall, the space needed to maintain statistics for uncached measurements is linear in the total number of measurements requested plus the total number of downstream caches updated during an observation period. Thus, the amount of required space can be controlled by adjusting the observation period length.

4 Experiments and Results

Experimental setup. We have implemented the GNP- and the DHT-based approaches. We conduct our experiments over ModelNet [16], a scalable and highly accurate Internet emulation environment. We emulate 20,000-node INET [4] topologies with a subset of nodes participating in measurement and querying activities. We report results for subsets with 250 nodes acting as both owners and querying nodes. These nodes are emulated by twenty 2.0GHz Intel Pentium 4 edge emulation nodes running Linux 2.4.27. All traffic passes through a 1.4GHz Pentium III core emulation node running FreeBSD-4.9.

While all results in this paper use an emulated network, we have also deployed our system (with around 50 nodes) over PlanetLab [12]. Note that the number of owners and querying nodes in our experiments is not constrained by the system's scalability, but rather by the hardware resources available for deploying it over an emulated network. The advantage of deploying a full system over an emulated network is that it ensures that all costs are captured and we do not inadvertently miss out any important effects or interactions. As future work, we plan to develop a simpler simulation-based evaluation, which would allow us to demonstrate larger experiments at the expense of some realism.

Workloads. We wish to subject our system to workloads with different characteristics that may represent different application scenarios. To this end, we have designed a workload generator to produce a mix of four basic types of "query groups." The four types of query groups are: (1) *Near-query-near- owner* (*NQNO*): A set of n_q nearby nodes query the same set of n_o owners that are near one another (not necessarily close to the querying nodes). This group should

benefit most from caching, since there is locality among both querying nodes and queried owners. (2) *Near-query-far-owner* (*NQFO*): A set of n_q nearby nodes query the same set of n_o owners that are randomly scattered in the network. There is good locality among the querying nodes, but no locality among the queried owners. (3) *Far-query-near-owner* (*FQNO*): A set of n_q distant nodes query the same set of n_o owners that are near one another. This group exhibits good locality among the queried owners, but no locality among the querying nodes. (4) *Far-query-far-owner* (*FQFO*): A set of n_q nodes query the same set of n_o owners; both the querying nodes and the queried owners are randomly scattered. This group should benefit least from caching.

A workload $[a, b, c, d]$ denotes a mix of a NQNO query groups, b NQFO query groups, c FQNO query groups, and d FQFO query groups. All query groups are generated independently. Each workload is further parameterized by n_q and n_o, the number and the size of queries in each group, and p, the period at which the queries will be reissued.

In this paper, we experiment with synthetic measurements, each generated by a random walk where each step is drawn from a normal distribution with mean 0 and standard deviation σ. If σ is large, bounds on this measurement will be violated more frequently, resulting in higher update cost. Synthetic measurements allow us to experiment with different update characteristics easily. Experiments with real node-to-node latency measurements demonstrate the effectiveness of bounded approximate caching, and are presented in [3].

4.1 Results for the DHT-Based Approach

Advantage of caching. To demonstrate the advantage of caching, we run a workload $W_1 = [1, 1, 1, 1]$ for 1000 seconds, with $n_q = 4$, $n_o = 10$, and $p = 16$ seconds. Effectively, during each 16-second interval, there are a total of 16 nodes querying a total of 40 owners, with each query requesting 10 measurements. This workload represents an equal mix of all four types of query groups, with some benefiting more than others from caching. The measurements in this experiment are synthetic, with $\sigma = 7$. Bounds requested by all queries are $[-10, 10]$. During the experiment, we record both *foreground traffic*, consisting of `READ` and `READ_REPLY` messages, and *background traffic*, consisting of all other messages including splice messages and `CACHE_UPDATE` messages.

Figure 3 shows the behavior of our system over time, with the size of each cache capped at 100 measurements (large enough to capture the working set of W_1). We also show the behavior of the system with caching turned off. The message rate shown on the vertical axes is the average number of messages per second generated by the entire system over the last 16 seconds (same as the period of monitoring queries). From Figure 3, for cache size 100 we see that after a burst of foreground traffic when queries start running, there is an increase in the background traffic as nodes decide to cache measurements. Once caches have been established, the foreground traffic falls dramatically due to the caches. As the set of caches in system stabilizes, the background traffic also reduces to mostly `CACHE_UPDATE` messages. On the other hand, with caching

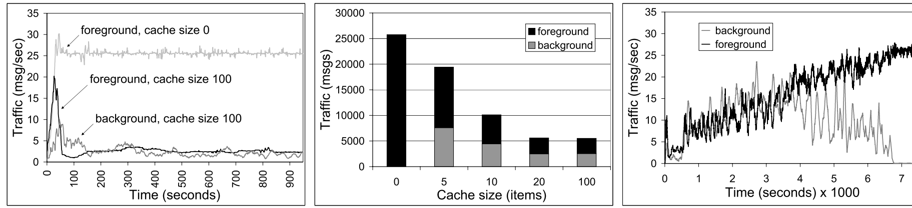

Fig. 3. Traffic vs. time **Fig. 4.** Traffic vs. cache size **Fig. 5.** Adapt to volatility

turned off (cache size 0) we see that the foreground traffic remains very high at all times (there is no background traffic). The high foreground traffic outweighs the benefit of having no background traffic. In sum, caching is extremely effective in reducing the overall traffic in the system.

Figure 4 compares the performance of the system under different cache sizes (in terms of the maximum number of measurements allowed in the cache of each node). We show the total number of foreground and background messages generated by the system over the length of the entire experiment (1000 seconds). As the cache size increases, the overall traffic decreases, although the benefit diminishes once the caches have grown large enough to hold the working set. Another interesting phenomenon is that for very small cache sizes, the background traffic is relatively high because of more splice operations caused by thrashing. Nevertheless, our system is able to handle this situation well.

Adapting to volatility in measurements. In this experiment, we use the same workload W_1 with cache size 100. We gradually increase the volatility of measurements by increasing the standard deviation σ of the random walk steps every 500 seconds. For the requested query bound of $[-10, 10]$, we effectively increase the update rate from 0.0 to 3.0 updates per second. The result of this experiment is shown in Figure 5. Initially, with a zero update rate, there is no cost to maintaining a cache, so all frequently requested measurements are cached, resulting in low foreground and background traffic. As we increase the update rate, however, the background traffic increases. This increase in cache update cost causes nodes to start dropping cached measurements; as a result, the foreground traffic also increases. Eventually, the update rate becomes so high that it is no longer beneficial to cache any measurements. Thus, the background traffic drops to zero, while the foreground traffic increases to the level when there is no caching (cf. Figure 3). To summarize, our system only performs caching if it leads to an overall reduction in total traffic; consequently, the total traffic in the system never rises above the level without caching. This shows that our system is able to adapt its caching strategy based on the volatility of measurements.

Aggregation effects. The next two sets of experiments demonstrate that our system can exploit locality in both querying nodes and queried owners. To illustrate aggregation on the querying node side, we perform a series of experiments using five workloads, $[0, 0, 2, 2]$, $[1, 0, 2, 1]$, $[2, 0, 2, 0]$, $[2, 1, 0, 1]$, and $[2, 2, 0, 0]$, where the percentage of queries issued from nearby nodes increases from 0% to

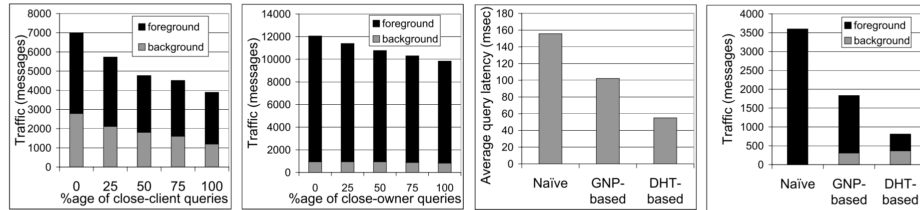

Fig. 6. Traffic vs. percentage of queries from nearby nodes.

Fig. 7. Traffic vs. percentage of queries to nearby owners.

Fig. 8. Comparison of average query latency.

Fig. 9. Comparison of total traffic.

100%. We set $n_q = 5$ and $n_o = 4$ for these five workloads. From the results in Figure 6, we see that the total traffic reduces as the percentage of queries from nearby nodes increases. Figure 7 shows the second set of experiments that illustrate owner-side aggregation by using five workloads where the percentage of queries requesting nearby owners increases from 0% to 100%. We again see that the total traffic reduces as the percentage of queries requesting nearby owners increases. These experiments show that our system derives performance benefits by exploiting locality both among querying nodes and in query regions.

Comparison with the naive and GNP approaches. Figure 8 compares the average query latency (as measured by the average time it takes to obtain the requested measurement, after all caches have been created) for a simple workload that exhibits locality among querying nodes. For baseline comparison, we also measure the average query latency of a naive approach, where each querying node simply contacts the owner directly for the measurement. From the figure, we see that the DHT-based approach has the lowest query latency, while the GNP-based approach performs a little worse, but both outperform the naive approach. Figure 9 compares the total network traffic generated by the system while processing a workload in which five querying nodes repeatedly query a faraway set of 12 nearby owners over 480 seconds, using the naive, GNP-based, and DHT-based approaches. Again, the DHT-based approach outperforms the other two approaches as it exploits querying node side locality effectively.

5 Related Work

Network monitoring. A large number of network monitoring systems have been developed by both the research community and commercial vendors. Astrolabe [17] is a system that continuously monitors the state of a collection of distributed resources and reports summarized information to the its users. Ganglia [8] is a system for monitoring a federation of clusters. While our work also considers the network monitoring problem, we focus on supporting set-valued queries approximately rather than aggregation queries. Our approach of bounded approximate caching and methods for locality-aware, cost-based cache management offer better flexibility and adaptability than these systems, which

are preset to either push or pull each piece of information. Our techniques can be used to enhance these and other existing network monitoring systems.

Data processing on overlay networks. PIER [7] is a DHT-based massively distributed query engine that brings database query processing facilities to new, widely distributed environments. For network monitoring, also one of PIER's target applications, we believe that bounded approximate caching meshes well with PIER's relaxed consistency requirement, and our DHT-based caching techniques can also be applied to PIER. Locality-aware DHTs have been used to build SCRIBE [2], a scalable multicast system, and SDIMS [18], a hierarchical aggregation infrastructure. Our DHT-based approach also uses a locality-aware DHT, but for the different purpose of selecting and locating caches; in addition, we use reverse DHT routes to achieve aggregation effects on the owner side.

Approximate query processing for networked data. The idea of bounded approximate caching has been explored in detail by Olston [10], along with techniques such as adaptive bound setting, source cooperation in cache synchronization, etc. We apply bounded approximate caching in this paper, but we focus on how to select caches across the network to exploit locality, and how to locate these caches quickly and efficiently to answer queries. We also extend the approximate replication scheme by allowing guarantees to be provided not only by the owner, but also by any other cache with a tighter bound.

Web caching and web replication. Web caching [13] is often done by ISPs using web proxy servers. Web replication [13] refers to data sources spreading their content across the network, primarily for load balancing. In both cases, the cache content is stored exactly and most often relatively stable content (e.g. images) is replicated at static locations. They do not deal with the problem of rapidly updating data; this means that they can afford to establish a large number of caches/replicas. Our system deals with replication of dynamic measurements and therefore update costs are high. We reduce update costs by caching bounded measurements, and balance update and query costs by caching at dynamically chosen nodes in the network.

6 Conclusions

In this paper, we tackle the problem of querying distributed network measurements, with an emphasis on supporting set-valued queries using bounded approximate caching of individual measurements. We focus on efficient and scalable techniques for selecting, locating, and managing caches across the network to exploit locality in queries and tradeoff between query and update traffic. We have proposed, implemented, and evaluated a DHT-based adaptive caching approach and compared it with a GNP-based controlled caching approach. Experiments over a large-scale emulated network show that our caching techniques are very effective in reducing communication costs and query latencies while maintaining the accuracy of query results at an acceptable level. The DHT-based approach is shown to adapt well to different types of workloads. In addition to temporal

locality in the query workload, the approach is able to exploit spatial localities in both querying nodes and measurements accessed by region-based queries.

Although the results are promising, techniques described in this paper represent only the first steps towards building a powerful distributed network querying system. As future work, we plan to investigate the hybrid approach of combining query shipping and data shipping, and consider more sophisticated caching schemes such as *semantic caching* [5].

Acknowledgement. We would like to thank Joe Hellerstein and David Oppenheimer for their ideas and suggestions. We would also like to thank Adolfo Rodriguez and Chip Killian for several discussions and help.

References

1. B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*,1970.
2. M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 2002.
3. B. Chandramouli, J. Yang, and A. Vahdat. Distributed network querying with bounded approximate caching. Technical report, Department of Computer Science, Duke University, June 2004.
4. H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, 2002.
5. S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.
6. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
7. R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, 2003.
8. M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 2004.
9. T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *IEEE INFOCOM*, 2002.
10. C. Olston. *Approximate Replication*. PhD thesis, Stanford University, 2003.
11. C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, 2001.
12. PlanetLab. http://www.planet-lab.org.
13. M. Rabinovich and O. Spatschek. *Webcachingandreplication*. Addison-Wesley, 2002.
14. 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.
15. S. Shah, K. Ramamritham, and P. J. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *VLDB*, 2002.
16. 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.
17. R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 2003.
18. P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, 2004.