Distributed Network Querying with Bounded Approximate Caching

ABSTRACT

As networks continue to grow in size and complexity, distributed network monitoring and resource querying are becoming increasingly difficult and costly. The aim of this work is to design, build, and evaluate 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. In this paper, we focus on developing efficient and scalable techniques to place, locate, and manage bounded approximate caches across a large network. We have developed and evaluated two approaches: One uses a recursive partitioning of the network space to place caches in a static, controlled manner, while the other uses uses a locality-aware *distributed hash table* to place caches in a dynamic and decentralized manner. Our experiments using large-scale network emulation 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. 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 [8]. 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. For an example of distributed resource querying, suppose that researchers want to run experiments on PlanetLab [19], 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, as we will later demonstrate with 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 monitoring queries that run periodically, and by queries that request measurements from a large number of nodes.

We seek to develop an infrastructure with better support for distributed network queries, by exploiting a number of optimization opportunities that naturally arise in our target applications:

1. For most of our target applications, exact answers are not needed. Approximate measurement values will suffice as long as the degree of inaccuracy is quantified and reported to the user, and the user has the ability to control the degree inaccuracy. The reason why bounded approximation is acceptable is that small errors usually have little bearing on how measurements are interpreted and used by the target applications; at any rate, these applications already need to cope with errors that are inevitable due to the stochastic nature of measurement.

2. In most cases, it is acceptable for a set-valued query not to return a perfect snapshot of the system in which all measurements are taken at the exact same point in time. Perfect snapshots are expensive to implement and bring little benefit to general-purpose network monitoring and resource querying. These applications are typically not interested in exact timings of measurements relative to each other, as long as they provide a loose snapshot of the system.

3. Oftentimes, measurement values do not vary in a completely chaotic manner. This behavior is especially common for

measurements that track some statistical properties of observation values over time, e.g., the average network latency over the last five minutes, or the standard deviation of CPU load over the last eight hours.

4. 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 the same query; for example, a network administrator monitors a cluster of nodes, which are close to each in terms of network distance.

We have built a distributed querying infrastructure that exploits the optimization opportunities discussed above. The first three opportunities can be exploited by *bounded approximate caching* 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 (e.g. in [18]); in this paper, we focus on developing efficient and scalable techniques to place, locate, and manage bounded approximate caches across a large network.

We present two approaches in this paper. The first approach, described in Section 3, uses a recursive partitioning of the network space to place caches in a static, controlled manner. The second approach, described in Section 4, uses a *distributed hash table* (e.g., [21]) to place caches in a dynamic and decentralized manner. We focus more on the second approach because it has a number of advantages over the first one (discussed in Section 5). Both approaches are designed to exploit the fourth optimization opportunity discussed above; namely, they are capable of capturing 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 [23], a scalable Internet emulation environment, we show in Section 6 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 Section 5, 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. The generalization described in Section 5 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 who 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 way 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. We will discuss how to handle locality in other spaces briefly in Section 5.

For a query that simply requests a set of measurements from a region, the result consists of the values of these measurements. As motivated in Section 1, in most situations we do not need accurate answers. 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).

Note that in general, measurement values returned in the same result may have been taken at different times, and their associated guarantees may also be different because the network delay varies across owners. In other words, unlike with traditional transactional database systems, our query result does not necessarily reflect an instant snapshot of the entire system. We do not find this issue limiting for our intended applications, which have long ago learned to cope with this issue.

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" answers 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 (this is also shown in our experments), and can cause interference with measurements. Caching is a natural and effective way to utilize previously obtained measurement values, especially for monitoring queries that repeat periodically. However, 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).

Let node N be the owner or a cache of a measurement. N may be responsible for maintaining 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).

Each *cache entry* contains the following information:

- The ID of the measurement being cached.

- The cached measurement value, and the time at which this measurement was taken (at the owner).
- A bound $[-\delta^-, \delta^+]$.
- The network address of the cache provider.

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, except that it records the network address of the child cache instead of the cache provider. We require the bound of a child cache to contain the bound of its provider cache.

Whenever the measurement value at a cache provider N changes (either because N is the owner who has detected the change, or because N has received an update from its provider), N compares the new value against each of its guarantee entry for this measurement. Suppose that the guarantee entry for child cache C currently records value v and bound $[-\delta^-, \delta^+]$. If the new value falls outside the range $[v - \delta^-, v + \delta^+]$, N will notify C of this new value and its timestamp. Both the cache entry at C and the guarantee entry at N are updated accordingly. In general, C can in turn provide for some other child caches, so this process continues from each provider to its child caches until we have updated all caches whose bounds are violated.

Note that by establishing measurement caches at the querying node with bounds specified by the query, we can support *continuous queries* (in addition to periodic queries), whose results are continuously updated whenever they fall out of query bounds.

It is possible that when a provider or part of the network fails, child caches would wrongly assume that their cached values lie within bounds in the absence of any updates. To handle this situation, we use a timeout mechanism. If no update has been sent to a child cache over a prescribed timeout period, the cache provider will send an update to the child cache even if its bound is not violated. If any cache does not receive any update from its provider over the prescribed timeout period, this cache is dropped, and so are all caches that depend on it. The system then notifies all query clients who have received answers based on any of the dropped caches during the timeout period. Although this possibility of invalidating recent query results does complicate semantics, it is acceptable to our target applications because our system is guaranteed to detect a failure shortly after the fact.

The choice of bounds is up to the application issuing the query. Tighter bounds provide better accuracy, but may require more update traffic in the system. There are sophisticated techniques for setting bounds dynamically and adaptively (e.g., [17]); such techniques are outside the scope of this paper and largely orthogonal to the contributions of this paper. In this paper, we focus on techniques for *selecting* bounded approximate caches across the networks to exploit query 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 approach 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 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. This approach is discussed further in Section 3.

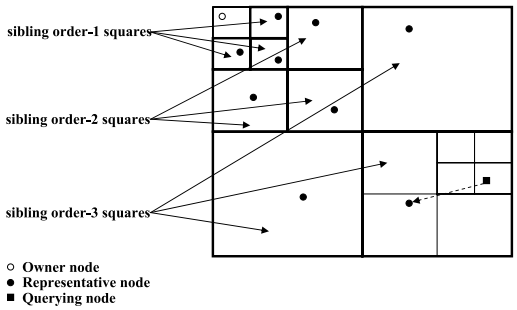

Figure 1: Recursive partitioning of the GNP space into squares.

In this paper, we focus more on the second approach, which uses a locality-aware DHT to achieve locality- and workload-aware caching in an adaptive manner. This approach has a number of advantages over the controlled approach. 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, which does not happen in the controlled approach. With the use of a DHT, the system is also 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 4, including a discussion on the cost/benefit analysis for making caching decisions.

3. GNP-BASED CONTROLLED CACHING

Partitioning of the GNP space. In addition to its IP address, each node can be identified by its position within the network, described as a set of coordinates in a geometric space such as the one proposed by *Global Network Positioning* (*GNP*) [15]. GNP assigns coordinates to nodes such that their geometric distances in the GNP space approximate their actual network distances. A 7-dimensional Euclidean space is enough to map all nodes on the Internet with reasonable accuracy, using network latency as the distance metric [15]. We note that GNP could be replaced by any other network positioning technique such as Vivaldi [5], PIC [4], or NPS [16].

Our controlled caching approach is based on a hierarchy produced by recursively partitioning the GNP space. For ease of exposition, we use a simple, grid-based partitioning scheme identical to that of [13]; it can be replaced by any other recursive partitioning scheme without affecting other aspects of our approach. We recursively partition a d-dimensional GNP space into successively smaller *squares* (d-dimensional hyper-rectangles), as shown in Figure 1 for $d = 2$. The smallest squares are referred to as order-1 squares. In general, each order-$(i + 1)$ square is partitioned into 2^d subsquares of order i. A node in the GNP space is located in exactly one square of each order.

Candidate cache selection. Each owner selects a number of other nodes in the network as its candidate caches. We allow each owner O to select a candidate cache in each of its *sibling squares*: As illustrated in Figure 1, 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 ensure that the candidate caches provide reasonable coverage of the entire GNP space, with better coverage closer to the owner.

To select a candidate cache in a sibling square, we use a *cache locator function*. This function takes as input a sibling square and the IP address of the owner, and returns the IP address of the owner's

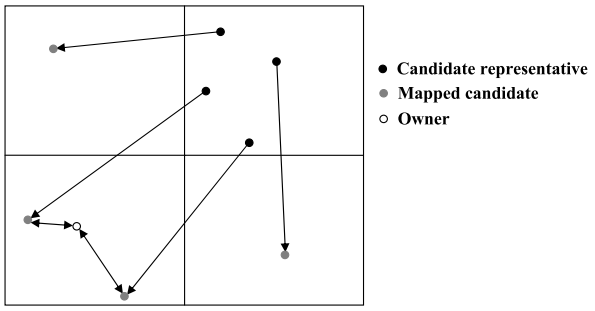

Figure 2: k-nearest mapped cache locator.

candidate cache within the given sibling square. A good cache locator function should be quick to compute, consistent in its result, and should ensure that nearby owners have a good probability of selecting the same candidate cache. The last requirement allows us to exploit locality in a query region to reduce processing costs: A query can obtain cached measurements in a large region by contacting just a few nodes.

We have considered several possible definitions of the cache locator function. The simplest approach is to hash each node identifier into a circular space, and select the candidate cache to be the node whose hash value is closest to the owner's hash value in this circular space. Although this approach provides good load balancing, it fails to ensure that owners close to each other in the GNP space tend to select same candidate caches.

We have developed a cache locator function, *k-nearest mapped cache locator*, which considers locality in query regions. Suppose that we wish to determine the representative for an owner in a sibling square S of particular order. We map all nodes within S, randomly and uniformly, into points in the other $2^d - 1$ squares that belong to the same higher-order square as S, as shown in Figure 2. We find the k (a small integer) points that are nearest to the owner, and order them by their distance to the owner. The candidate cache is selected to be the node corresponding to the i-th point, where i is obtained by hashing the owner IP to an integer in $[1, k]$. Since nearby owners may share many of their k-nearest points, there is a good chance that the same candidate cache will be selected.

GNP servers. We need a mechanism to accomplish two basic tasks required by our caching scheme: (1) A node should be able to determine the GNP coordinates of any other node given its IP. (2) Given an owner, a querying node should be able to locate the closest candidate cache of the owner. To this end, we use a hierarchy of *GNP servers*. Within each square (of any order), a node is designated as the GNP server responsible for this square. Each node in the system remembers the IP of the GNP server responsible for its order-0 square. Each GNP server remembers the IP of the GNP server responsible for each of its subsquares, and vice versa. In addition, each GNP server maintains the IP and GNP coordinates for all nodes in its square, which raises the concern of scalability at higher-order squares. Indeed, this concern is one of the reasons that led us to develop the alternative DHT-based approach (Section 4). Nevertheless, because of its simplicity, the GNP-based approach is still viable for small- to medium-sized systems.

To look up the GNP coordinates of a node X given its IP, a querying node first contacts the GNP server for its order-0 square. If the GNP server does not find X in its square, it forwards the request to a higher-order GNP server. The process continues until X is found at a GNP server; in general, it will be the GNP server for the lowest-order square containing both X and the querying node.

To locate the closest candidate cache of an owner O, the querying node follows the same procedure as looking up O. The GNP server that finds O can evaluate the k-nearest mapped cache locator function to find the candidate cache of O in the subsquare containing the querying node. This candidate cache is the closest in the sense that it is the only candidate cache of O in that subsquare.

GNP servers also support declarative specification of query regions in the GNP space, e.g., "all nodes within a distance of 10 from a given node." We omit the details for lack of space.

We aggressively cache the results of GNP-related lookups to improve performance and prevent overload of higher-order GNP servers. This technique is reminiscent of DNS caching.

Operational details. To answer a query for a set of measurements, the querying node first looks up the closest candidate cache for each owner of the requested measurements using GNP servers, as discussed earlier. The lookup requests and replies are aggregated, so regardless of the number of measurements requested, there are no more than $2h$ such messages per query, where h is the number of levels in the GNP server hierarchy. Next, the querying node contacts the set of candidate caches; there are hopefully much fewer of them than the owners, because our cache locator function exploits locality in query regions. If a measurement is not found in the candidate cache or the bound on the cached value is not acceptable, the request will be forwarded to the owner.

Each candidate cache decides on its own whether to cache a measurement and what bound to use. The decision is made using a cost/benefit analysis based on the request and update rates. We omit the details here because a similar (and more complex) analysis used by the DHT-based approach will be covered in detail in Section 4.2.5.

The owner is directly responsible for maintaining all caches of its measurement, using the procedure described in Section 2. As also noted in Section 2, we use a timeout mechanism to handle failures.

4. ADAPTIVE CACHING BASED ON DHT

While simple to implement, the controlled caching approach described in the previous section has a number of disadvantages. First, GNP servers carry potentially much higher load than other nodes in the system because GNP servers are needed in locating caches. Second, the controlled approach also fails to exploit the potential locality in querying nodes: It is possible for a number of close-by nodes to request the same faraway owner over and over again, yet still not find a cache nearby, because there are fewer caches farther from the owner, and the static, hash-based cache placement scheme does not adapt to the query workload.

To combat these problems, 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 will begin this section by reviewing the background on DHTs. Then, in Section 4.2, we describe the details of our adaptive caching approach.

4.1 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 networks 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, and each

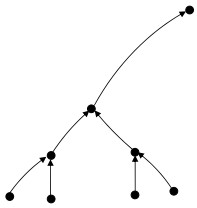

Figure 3: The message aggregation effect in Pastry.

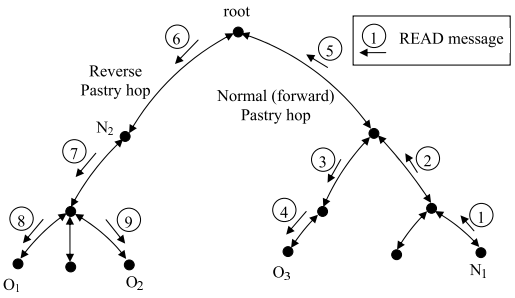

Figure 4: Two-way aggregation with Pastry.

data item is identified by a unique key. At the heart of a DHT is an overlay routing scheme that delivers requests for a given key to the node currently 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 [21] is a popular DHT that provides a scalable distributed object location and routing substrate for P2P applications. Pastry routes a message for a given key to the node whose Pastry ID (obtained by hashing its IP address) is numerically closest to the given key. An important feature that distinguishes Pastry from many other DHTs is that it takes network proximity into account. Specifically, Pastry seeks to minimize the network distance traveled by messages as measured by, for example, the number of IP routing hops. Pastry is described in detail in [21].

A number of properties of Pastry are relevant to our system. First of all, the *short-hops-first* property, a direct result of locality-aware routing in Pastry, says that the expected distance traveled by a message during each successive routing step increases exponentially. The *short-routes* property, as shown by studies, says that the average distance traveled by a Pastry message is within a small factor of the distance between the message's source and destination in the underlying Internet. The *route-convergence* property concerns the distance traveled by two messages sent to the same key before their routes converge. Studies [21] 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, as illustrated in Figure 3. This aggregation effect is used by SCRIBE [2] in building a scalable multicast system. The same effect is exploited by our system, although for a different purpose, as we will discuss next.

4.2 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.

Message Type	Meaning	Section
`INIT`	Sent during initialization to build reverse paths	4.2.1
`READ`	Request for measurement data	4.2.2
`READ_REPLY`	Answer (possibly partial) to a `READ` message	4.2.2
`CACHE_UPDATE`	Sent by a node to update a child cache	4.2.3
`SPLICE_IN`	Request to start caching a measurement	4.2.4
`SPLICE_IN_OK`	Reply to a `SPLICE_IN` request	4.2.4
`SPLICE_OUT`	Request to stop caching a measurement	4.2.4
`SPLICE_OUT_OK`	Reply to a `SPLICE_OUT` request	4.2.4

Table 1: List of messages in the system.

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 4. 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. For reference, Table 1 summarizes the major types of messages.

4.2.1 Initialization

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). 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 (by storing the entire content of each set), because the subtrees are small. Nodes at higher levels, on the other hand, maintain lossy subtree filters implemented with *Bloom filters* [1]. A Bloom filter is a simple, space-efficient approximate representation of a set that supports membership queries. Although Bloom filters allow false positives, for many applications such as ours the space savings outweigh this drawback when the probability of false positives is sufficiently low.

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 to the parent.

4.2.2 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, the set of measurements requested and acceptable bounds on them.

When a node N receives a `READ` message, it checks to see if it can provide any subset of the measurements requested, either because it owns some of these measurements, or it has them cached

within the requested bounds. 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 will also forward 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 4 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. The following proposition shows that our system attempts to route queries towards measurement owners over $\mathcal{T}$ in an optimal manner.

PROPOSITION 1. *If no subtree filters produce any false positives, then all nodes involved in processing a request for measurements owned by a set of nodes $\mathcal{O}$ belong to the minimal subgraph of $\mathcal{T}$ (in terms of number of edges) spanning both $\mathcal{O}$ and the querying node.*

On false positives. As discussed in Section 4.2.1, nodes at lower levels of $\mathcal{T}$ can afford to maintain accurate subtree filters without false positives. However, at higher levels, Bloom filters may produce false positives, so it is possible that $\mathcal{S} \cap \mathcal{F}_i \neq \varnothing$ even though the i-th subtree actually does not contain any node in $\mathcal{S}$. In that case, some extraneous READ messages are forwarded, but they do not affect the correctness of the query result. Furthermore, there are few such messages because Bloom filters are only used at higher levels, and the rate of false positives can be effectively controlled by tuning the size of these filters.

A more subtle point is that false positives can cause unnecessary bottlenecks near the root. Ideally, a READ message should never go above the lowest common ancestor of the querying node and the nodes being queried. However, if the subtree filters of this ancestor node are Bloom filters with false positives, the message will always need to be forwarded up the tree, because there is no guarantee that the subtrees own all requested measurements. This issue is problematic for monitoring queries that are executed repeatedly. Caching alleviates the problem somewhat, because a node can definitely stop forwarding READ if its cache provides all remaining measurements. However, caching cannot always solve this problem because the system may decide not to cache some measurements for whatever reason (e.g., the cache is already full, or the measurement values fluctuate too much for caching to be cost-effective). Fortunately, we have a simple trick that avoids this problem completely for a repeating query: We can compute and remember the lowest common ancestor node, say L, at the first time when this query is executed; subsequently, READ messages will carry the ID of L so that when L sees these messages it knows not to forward them up the tree.

4.2.3 Cache Updates

Let node N be a cache provider. As discussed in Section 2, whenever the measurement value at N violates the bound for a child cache N', N sends a CACHE_UPDATE message to N' to update its cached value. The message contains the new value of the measurement and the timestamp when it was taken. Meanwhile, N also updates its corresponding guarantee entry locally. Also, as noted in Section 2, we use a timeout mechanism to handle failures.

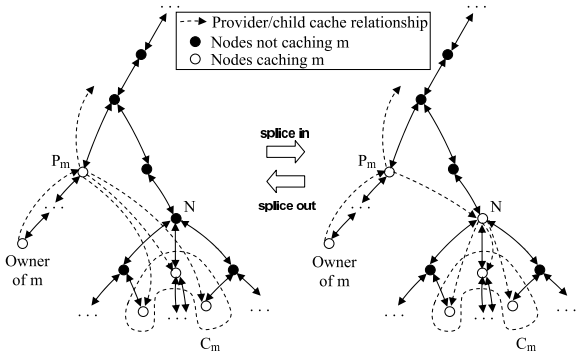

Figure 5: Splicing: adding and removing a cache.

In general, a child cache may have its own child caches, and all caches for the same measurement form a tree rooted at the owner of the measurement. A CACHE_UPDATE message originates from the owner, and may or may not trigger more CACHE_UPDATE messages down this tree depending on whether child cache bounds are violated. In essence, this tree provides a scalable structure for multicasting CACHE_UPDATE messages. The two operations that we describe next will ensure that this tree is properly maintained.

4.2.4 Adding and Removing Caches

Each node in our system has a *cache controller* thread that periodically wakes up and makes caching decisions. Before discussing how to make such 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 who 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 5. 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 5 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.

Now suppose that N decides to stop caching m. The procedure is similar and slightly simpler. Let P_m denote the cache provider for N, and let $\mathcal{C}_m$ denote N's child caches. We would like P_m to take over the responsibility of updating $\mathcal{C}_m$ after N stops caching

m. To this end, N sends out a SPLICE_OUT message containing its guarantee entries and recorded shortest paths for $\mathcal{C}_m$. This message is routed as if it is a READ request for m, until it reaches P_m. Upon receiving this message, P_m removes N from its guarantee list, adds all guarantee entries in the message, and records the shortest paths after appending them with the shortest path from N to P_m (which can be easily obtained by having the SPLICE_OUT message record each hop as it is being routed). Then, P_m sends back a SPLICE_OUT_OK message to N, so that N can drop its cache for m, and remove the guarantee entries and shortest paths for $\mathcal{C}_m$. Finally, N sends out SPLICE_OUT_OK messages to $\mathcal{C}_m$ nodes to inform them of the change in their cache provider.

In both cases discussed above, we use a locking protocol to ensure consistency in the face of concurrent splicing requests. By design of the above operations, our system attempts to maintain the following invariant, which implies that a cache update originated from the owner would be sent over a minimal multicast tree spanning all caches if update messages were routed over $\mathcal{T}$. Again, note that false positives in subtree filters may introduce some extraneous messages, but they do not affect the overall correctness.

PROPOSITION 2. *If no subtree filters produce any false positives, then for each node N caching measurement m, its cache provider is always the first cache of m on the shortest path in $\mathcal{T}$ from N to m's owner; if there is no other cache on this path, m's owner will be N's cache provider.*

4.2.5 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. In the following, we will first describe the various components of benefit and cost, assuming that all statistics relevant to decision making are available to us. We will return to the problem of how to maintain or approximate these statistics after discussing our algorithm for making cost/benefit-based cache decisions.

We break down the benefit and cost of caching m into the following components:

- $B_{read}(m)$, 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 N.
- $B_{update}(m)$, net benefit in terms of reduction in update traffic. If N caches m, its cache provider, P_m, will be responsible for updating N. On the other hand, P_m will no longer be directly responsible for $\mathcal{C}_m$ (defined in Section 4.2.4); instead, N will forward updates to $\mathcal{C}_m$. We can approximate the cost of an update message by the number of hops it travels in $\mathcal{T}$. Thus, the benefit of using N as an intermediary to update a node in $\mathcal{C}_m$ is d_m. Let $U_{m,X}$ denote the rate of updates required to maintain a cache of m at node X (update rates can be different for caches with different bounds). Overall, $B_{update}(m) \propto d_m \times (\sum_{X \in \mathcal{C}_m} U_{m,X} - U_{m,N})$.
- $C_{update}(m)$, cost in terms of resources (processing, storage, and bandwidth) incurred by N for maintaining its child caches for m. For each child cache in $\mathcal{C}_m$, N needs to store a guarantee entry as well as the shortest path to N; N is also responsible for updating the cache when its bound is violated. Thus, $C_{update}(m)$ is linear in $\sum_{X \in \mathcal{C}_m} U_{m,X}$. N may place an upper bound on the total amount of resources devoted to maintaining child caches.
- $C_{cache}(m)$, cost incurred by N for caching m (other than $C_{update}(m)$). This cost is primarily the storage cost of m. N may have an upper bound on the total cache size.

Given a set $\mathcal{M}$ of candidate measurements to cache, the problem is to determine a subset $\mathcal{M}' \subseteq \mathcal{M}$ that maximizes

$$\sum_{m \in \mathcal{M}'} \Big(B_{read}(m) + B_{update}(m) \Big)$$

subject to the cost constraints that

$$\sum_{m \in \mathcal{M}'} C_{update}(m) \leq T_{update}, \text{ and } \sum_{m \in \mathcal{M}'} C_{cache}(m) \leq T_{cache}.$$

Here, T_{update} specifies the maximum amount of resources that the node is willing to spend on maintaining its child caches, and T_{cache} specifies the maximum size of the cache.

This problem is an instance of the *multi-constraint* 0-1 *knapsack problem*. It is expensive to obtain the optimal solution because inputs such as T_{cache} are not small integers; even the classic single-constraint 0-1 knapsack problem is NP-complete. Therefore, 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 simply decides to cache measurements with highest, non-negative pseudo-utility values. Measurements that should be cached are added with SPLICE_IN messages, and measurements that should not be cached are removed with SPLICE_OUT messages, as discussed in Section 4.2.4.

Maintaining statistics. We now turn to the problem of maintaining statistics needed for making caching decisions. For each measurement m currently being cached by N, we can easily maintain all necessary statistics with negligible overhead. Recall from Section 4.2.4 that each cache provider records the shortest incoming path from each of its child caches. When N adds itself as a cache for m, its cache provider can calculate d_m for N based on the shortest path from N, and sends the result value back to N in the SPLICE_IN_OK message. In turn, N forwards this value to $\mathcal{C}_m$ nodes in SPLICE_IN_OK messages so these nodes can decrement their d_m by this value. If N decides to stop caching, the same value is sent to $\mathcal{C}_m$ nodes in SPLICE_OUT_OK messages so these nodes can increment their d_m accordingly. The request rate H_m can be maintained by counting the number of read requests for m received during a period. Update rates $U_{m,N}$ and $U_{m,X}$ for each $X \in \mathcal{C}_m$ can be maintained by counting the number of updates received and sent during a period. Overall, the total space devoted to these statistics is linear in the total size of the cache and the guarantee list.

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. 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. As for update rates and d_m, suppose for the moment that we route `READ_REPLY` and `CACHE_UPDATE` messages over $\mathcal{T}$ instead of direct IP. When sending out a `READ_REPLY` message for m, the owner or a cache of m can attach an estimate of the update rate for the bound requested, calculated from locally maintained update rate statistics; this estimate is available to N for any m requested at N. Also, when sending out a `CACHE_UPDATE` message for m to a child cache X, a cache provider can attach the locally maintained value of $U_{m,X}$. For each pair (m, X) of measurement and destination cache seen in `CACHE_UPDATE` messages passing through N during the observation period, N records the latest value of $U_{m,X}$ seen in such messages; $U_{m,N}$ can be estimated to be the maximum of all recorded update rates. Finally, d_m can be obtained using *hop counters* in `READ_REPLY` messages (for any m read during the observation period) or `CACHE_UPDATE` messages (for any m updated during the observation period). A hop counter is initialized to 0 and incremented by 1 for each hop traveled by the message.

In reality, our system sends most `READ_REPLY` and `CACHE_UPDATE` messages using direct IP. With small probabilities, they are routed over $\mathcal{T}$ instead of direct IP, so that downstream nodes can update their statistics according to the procedure described in the previous paragraph. These probabilities can be set in a way to ensure that, with high probability, at least one message of each type will be sent out during an observation period. For example, if we set $\max\{1, 2/(s \cdot U_{m,X})\}$ to be the probability of sending a `CACHE_UPDATE` message for m to X over $\mathcal{T}$, then we can prove that with a probability of more than 85%, a downstream node will see at least one such message during a period of s seconds (assuming independence of events). In our implementation, we have found that this approach can reduce the overhead of application-level routing with little sacrifice in the accuracy of statistics.

Overall, the total space needed to maintain the 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 length of the observation period.

5. DISCUSSION

Comparison of two caching schemes. The DHT-based adaptive caching approach has a number of advantages over the GNP-based controlled caching approach. First, GNP servers carry potentially much higher load than other nodes in the system. As discussed in Section 3, a GNP server needs to maintain precise knowledge about all nodes within its hyper-rectangle in order to locate the cache for a given owner. Thus, the amount of space required by GNP servers at higher levels is $\Theta(n)$, where n is the total number of nodes in the system. In contrast, routing and locating caches in the DHT-based approach does not depend on centralized resources like GNP servers. Forward Pastry routing requires only $O(\log n)$ state [21]; reverse Pastry routing requires subtree filters, but since false positives are tolerable, we can use Bloom filters whose sizes can be effectively controlled.

Second, the cache selection scheme used by the DHT-based approach is more dynamic and workload-aware than the GNP-based controlled caching approach. The controlled approach fails to exploit potential locality among querying nodes at runtime. It is possible for a number of close-by nodes to request the same faraway owner over and over again, yet still not find a cache nearby, because by design there will be fewer candidate caches farther from the owner, and the static cache selection scheme will not adapt to the query workload. In contrast, the DHT-based adaptive caching approach will select a cache nearby as soon as the combined requested rate from all querying nodes makes caching cost-effective. This analysis will be confirmed by experiments in Section 6.

Third, the GNP-based controlled caching approach restricts the amount of caching at any node by design. While it is reasonable to avoid overloading a node with caching responsibilities, implementing this objective using a static scheme precludes opportunities for certain runtime optimization. For example, suppose that a large region of owners are being queried over and over again. If a node has enough spare capacity, we should let it cache for all owners, so that a query can be answered by contacting this node alone. With the GNP-based approach, it is impossible by design for a large region of owners to select the same cache. In contrast, with the DHT-based approach, a common ancestor of all owners in $\mathcal{T}$ can potentially cache for all of them. Experiments in Section 6 confirm this analysis.

On the other hand, the GNP-based approach also has some advantages over the DHT-based approach. First, the GNP-based approach has simpler protocols and requires less effort to implement. Second, GNP coordinates allow better and more direct control over how locality is exploited, while the DHT-based approach has to rely on Pastry to exploit locality indirectly.

On alternative definitions of regions. So far, we have been assuming that query regions exhibit locality in terms of network proximity. As mentioned in Section 2, applications may use alternative definitions of query regions. Each node can be described by a vector of features. The distance between two nodes can be defined by the distance between their respective feature vectors in the feature vector space. A query region tends to contain nodes with similar features, i.e., those are nearby in the feature vector space. In the following, we briefly describe how to adapt our techniques to work with an application-defined space and distance metric.

For the GNP-based approach, we simply need to modify the cache locator function to exploit locality in the application-defined space instead of the GNP space. Specifically, given an owner o, in order to select a cache of o in a hyper-rectangle (in the GNP space), we map all nodes in the hyper-rectangle into points in the application-defined space. We then select one of the k such points that are closest to o in the application-defined space. The node corresponding to this selected point will be a cache of o.

In case of the DHT-based approach, we use a second instance of Pastry to construct another tree $\mathcal{T}_{app}$ over the same of nodes using the application-defined distance metric. To process a query, we first route it upwards in the regular Pastry tree $\mathcal{T}$ constructed based on network proximity, which allows network locality among querying nodes to be exploited. After several hops, we send the query directly to one of the owners being queried. Then, we process the query over $\mathcal{T}_{app}$ as if it originated from this owner, using the exact same procedure described in Section 4.2.2 (except on $\mathcal{T}_{app}$ instead of $\mathcal{T}$), which allows locality among owners in the application-defined space to be exploited.

6. EXPERIMENTS AND RESULTS

Implementation. We have implemented both approaches in Sections 3 and 4. The implementation of the GNP-based approach consists of around 3000 lines of C++ code. For the DHT-based approach, we use the MACEDON [20] implementation of Pastry. 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 approach on top of MACEDON consists of around 4500 lines of C++ code.

Experimental setup. We conduct our experiments on ModelNet [23], 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. Experiments with several large-scale distributed services have demonstrated the generality and effectiveness of the infrastructure.

For all our experiments, we use 20,000-node INET [3] topologies with a subset of 250 nodes participating in measurement and querying activities. These nodes are emulated by twenty 2GHz Intel XEON edge emulation nodes running Linux 2.4.27. All traffic passes through a 1GHz Pentium-III core emulation node running modified FreeBSD-4.9. While all results reported in this paper use ModelNet, we note that we have also run smaller experiments (with around 50 nodes) over PlanetLab [19].

Workloads. We wish to subject our system to workloads with different characteristics that may be representative of different application scenarios. To this end, we have designed a query workload generator to produce a mix of four basic types of "query groups." These four types of query groups are:

- *Near-query-near-owner* (*NQNO*): A set of n_q nearby nodes query the same set of n_o nearby owners (not necessarily close to the querying nodes). This group of queries should benefit most from caching, since they exhibit locality both among the querying nodes and among the queried owners.
- *Near-query-far-owner* (*NQFO*): A set of n_q nearby nodes query the same set of n_o owners that are randomly scattered throughout the network. These queries exhibit good locality among the querying nodes, but no locality among the queried owners.
- *Far-query-near-owner* (*FQNO*): A set of n_q distant nodes query the same set of n_o nearby owners. Each of these queries exhibits good locality among the queried owners, but there is no locality among the querying nodes.
- *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 throughout the network. This group of queries 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 (even if they have the same type); two query groups will contain two different sets of querying nodes, where each set queries a different set of owners. 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.

We experiment with both "real" (as emulated by ModelNet) and synthetic measurements. The real measurements are based on the actual network latencies observed between the nodes in our system. Each synthetic measurement is 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. The synthetic measurements allow us to experiment with different update characteristics easily.

6.1 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 6 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). Figure 7 shows 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 6, we see that there is a burst of foreground traffic when queries start running. This initial burst is followed by an increase in the background traffic consisting mostly of splice messages, as nodes decide to cache measurements. Once caches have been established, the foreground traffic falls dramatically because many reads can now be satisfied by caches. As the set of caches in system stabilizes, the background traffic also reduces to mostly `CACHE_UPDATE` messages. On the other hand, in Figure 7, we see that without any caching, the foreground traffic remains very high at all times, which far outweighs the benefit of having no background traffic. In sum, caching is extremely effective in reducing the overall traffic in the system.

Figure 8 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 eventually diminishes once the caches have grown large enough to hold the working set of the workload. Another interesting phenomenon is that for very small cache sizes, the background traffic is relatively high because of a large number of slice operations caused by trashing. Nevertheless, our system is able to handle this situation reasonably well; the overall traffic is still much lower than if no caching is used.

Adapting to volatility in measurements. In this experiment, we use the same workload W_1 and fix the cache size at 100. During the course of 1000 seconds, we gradually increase the volatility of measurements by increasing the standard deviation σ of the random walk steps. 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 9. Initially, with a zero update rate, there is no cost to maintaining a

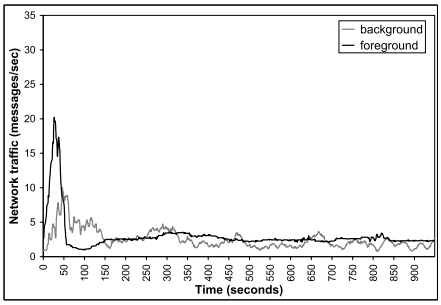

Figure 6: Traffic vs. time; cache size 100**.**

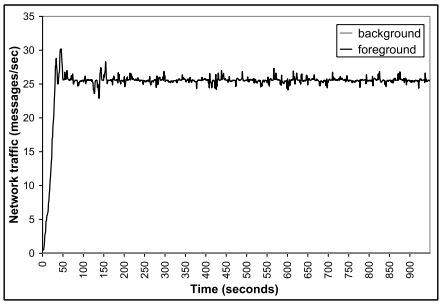

Figure 7: Traffic vs. time; cache size 0**.**

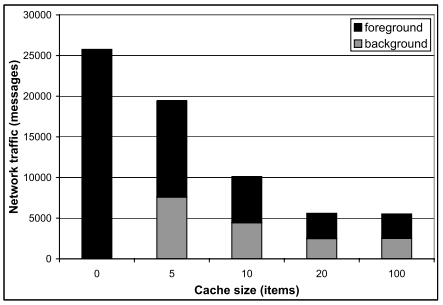

Figure 8: Total traffic; various cache sizes.

cache, so all frequently requested measurements are cached, resulting in low foreground and background traffic. As we increase the volatility of the measurements, 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 back to zero, while the foreground traffic increases to the same level when there is no caching (cf. Figure 7). To summarize, our system only performs caching if it leads to an overall reduction in total traffic; consequently, the total amount of traffic in the system never rises above the level without caching. This experiment shows that our system is able to adapt its caching strategy according to the volatility of measurements.

Aggregation effects. The next two sets of experiments are designed to demonstrate that our system can exploit locality in both querying nodes and queried owners, by taking advantage of the two-way aggregation effects in Pastry. 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 100%. We set $n_q = 5$ and $n_o = 4$ for these five workloads; all other parameters remain unchanged from previous experiments. Figure 10 shows the total foreground and background traffic generated by the system for all five workloads. We see that the total traffic reduces as the percentage of queries from nearby nodes increases, meaning that our system is able to exploit the locality among querying nodes to improve performance.

To illustrate aggregation on the owner side, we use five workloads, $[0, 0, 0, 4]$, $[0, 0, 1, 3]$, $[0, 0, 2, 2]$, $[0, 0, 3, 1]$, and $[0, 0, 4, 1]$, where the percentage of queries requesting nearby nodes increases from 0% to 100%. Since our focus is on showing the effect of owner-side aggregation, we discourage caching on the querying node side by avoiding NQNO and NQFO query groups in the workloads, and by limiting the size of the cache to 8. The results are shown in Figure 10. We see that the total traffic reduces as the percentage of queries requesting nearby owners increases, showing that our system derive performance improvements by exploiting locality in query regions.

Effect of bound width on update traffic. In this experiment, we test our hypothesis that bounded approximate caching is an effective way of trading off accuracy for lower update traffic. For this experiment, synthetic measurements are not meaningful; instead, we use actual latencies observed under ModelNet emulation. Each node monitors its latency to another node over the network with periodic ping messages; these latency values serve as measurements to be queried and cached. Figure 12 shows the rate of `CACHE_UPDATE` messages in the system as we vary the cache bound width from 0.01 msec to 0.41 msec. As we can see from the figure, the update rate drops significantly as we increase the bound width. The reason is that under normal circumstances, real latency measurements tend not to fluctuate wildly, particularly when they are measured as running averages. Our system would provide the maximum benefit for relatively stable measurements; should they begin to fluctuate wildly, our system will be able to handle them gracefully, as shown by the earlier experiment on adapting to volatility in measurements.

6.2 Results Comparing the GNP- and DHT-Based Approaches

Query latency. As discussed in Section 5, the GNP-based approach selects candidate caches statically and therefore often fails to exploit locality that arises at runtime among querying nodes. On the other hand, the DHT-based approach can dynamically detect such locality and elect a cache that can reduce query latency for all nearby querying nodes. To confirm our analysis, we have designed a simple workload as follows. We select four querying nodes that are close to each other in both GNP space and Pastry tree. These nodes run the same monitoring query, periodically requesting the same measurement from a node faraway. The update rate of the measurement is just high enough so that each querying node will not start caching the measurement locally.

Figure 13 compares the average query latency for this workload (as measured by the average time it takes to obtain the requested measurement, after all caches have been created) using the GNP- and DHT-based approaches. 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. Looking at the execution traces, we find that with the GNP-based approach, the four querying nodes are able to obtain the measurement from the same cache located in a large sibling hyper-rectangle of the owner. Compared with the owner, this cache is indeed closer to the querying nodes, but it still turns out to be quite faraway. Since the set of candidate caches are determined statically, the four querying nodes are stuck with this cache in the GNP-based approach. The DHT approach is able to elect the lowest common ancestor of the four querying nodes as a cache, which is very close to them.

Total traffic. As we have also discussed in Section 5, the GNP-based approach tries to limit the amount of caching at each node by design, even if the node has enough spare capacity at runtime to cache a large query region. On the other hand, the DHT-based approach will allow a single ancestor node with enough capacity to cache a entire region, which can dramatically reduce the number of

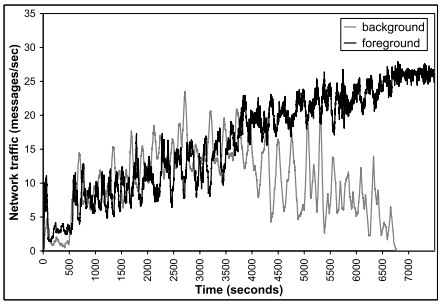

Figure 9: Traffic vs. time as update rate increases; cache size 100**.**

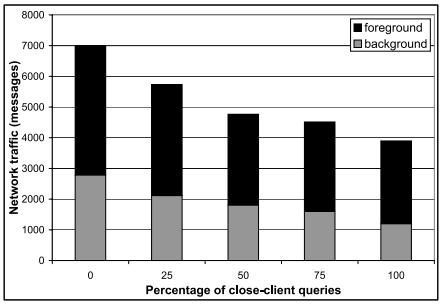

Figure 10: Total traffic as the percentage of queries from nearby nodes increases; cache size 100**.**

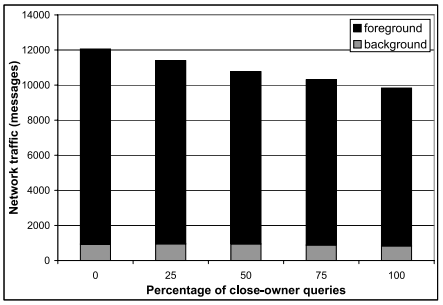

Figure 11: Total traffic as the percentage of queries to nearby owners increases; cache size 8**.**

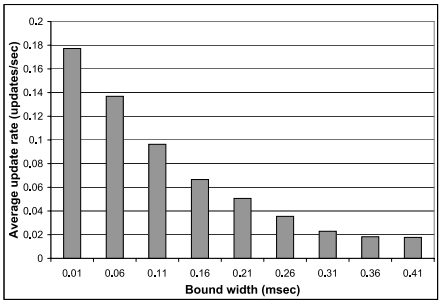

Figure 12: Update rate as bound width increases; real latency measurements.

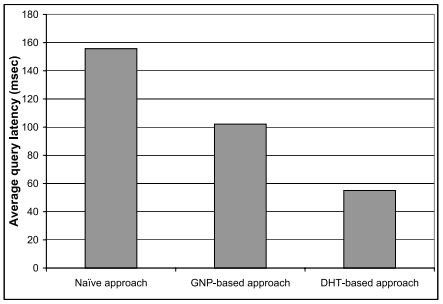

Figure 13: Average query latency comparison for three approaches.

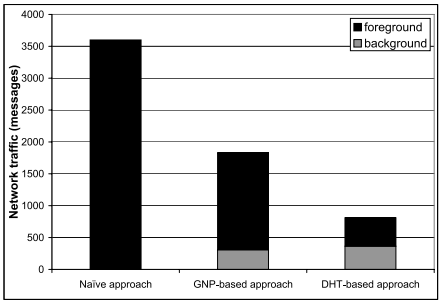

Figure 14: Total traffic comparison for three approaches.

messages. To confirm this advantage of the DHT-based approach, we use a simple workload in which five querying nodes repeatedly query a faraway set of 12 nearby owners.

Figure 14 compares the total network traffic generated by the system while processing this workload over 480 seconds, using the native, GNP-based, and DHT-based approaches. As the execution traces reveal, the DHT-based approach is able to choose one node to cache for all 12 owners, which saves a huge amount of the foreground traffic and results in the lowest total traffic among the three approaches. The GNP-based approach is able to cache 12 owners with a small number of nodes (5 in this experiment), which leads to a moderate saving in the foreground traffic. For the naive approach, each query must always contact all 12 owners, which, not surprisingly, results in the highest total traffic. Finally, we note that the larger the size of the query region, the larger the performance gain of the DHT-based approach over the other two approaches, though eventually this gain will constrained by the capacity of caches.

7. RELATED WORK

Network monitoring. Astrolabe [24] is a distributed information management system which collects large-scale system state, permitting rapid updates and providing online attribute aggregation. This makes it possible to solve a wide variety of management and self-configuration problems. In contrast, our work is more specifically tied to the issue of low-cost network monitoring. We focus on providing answers to set queries so as to reduce costs without sacrificing too much accuracy. Ganglia [14] is a scalable distributed monitoring system for high performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. We also place our system as a scalable distributed monitoring system, but its focus (answering queries approximately at reduced cost) and methodology (controlled or adaptive caching of bounded measurements) are both different from Ganglia. Any of these systems could be enhanced to use bounded measurements and provide approximate answers at a reduced cost.

Data processing on overlay networks. PIER [9] is a massively distributed query engine based on overlay networks, which is intended to bring database query processing facilities to new, widely distributed environments. The authors of that paper motivate the need for massively distributed queries, and argue for a relaxation of certain traditional database research goals in the pursuit of scalability and widespread adoption. Another paper [22] proposes a technique for evaluating range queries efficiently over a peer-to-peer network by caching answers at the nodes. Aggregation of data using a DHT is done in systems such as SCRIBE [2]; the SDIMS [25] paper uses DHTs to create scalable aggregation trees for information management. On the other hand, we use locality-sensitive DHT aggregation trees to cache bounded measurements and act as a nearby cache on the querying node side. We also perform aggregation on the owner side by using reverse paths.

Approximate query processing. The idea of establishing and maintaining bounds for data was explored in Chris Olston's work [18] in detail. This work focuses on maximizing precision for a given communication cost, by selecting which tuples to refresh. It also gives a technique of answering continuous queries by installing filters at the data sources, so that user-defined precision requirements can be met. It suggests using bounds to answer aggregation type one-time queries approximately, with optional selection predicates. This is different from our work in that we are basically focussed on answering set queries using bounds on individual data items. Also, we extend the replication scheme by allowing guarantees to be provided not only by the owner, but also by any other node with the necessary information. The bounds that our infrastructure guarantees to query nodes is used by the set query engine to answer user

queries approximately.

Another related paper [10] suggests techniques related to approximate aggregation for sensor databases within the network using small sketches. One paper [7] proposes integrating the database system with a centralized probabilistic model that identifies correlations in sensor readings to answer queries. Another paper [11] uses a distributed localized algorithm to elect representative nodes that that can be used to provide approximate answers quickly. The problem of evaluating selection queries over imprecisely represented objects is examined in [12]. Although our paper places emphasis on techniques for approximate caching and does not consider query processing issues in detail, the ideas in [12] can be used for answering queries using these approximate cached items.

8. CONCLUSIONS

In this paper, we tackle the problem of querying distributed network measurements. The focus is on processing set-valued queries using bounded approximate caching of individual measurements. We have proposed and evaluated two approaches to selecting and locating the caches in a wide-area network. Experiments using large-scale simulations show that bounded approximate caching has a high impact 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 to different types of workloads successfully. In addition to temporal locality in the query workload, the two-way caching method built on Pastry is also able to exploit spatial localities in querying nodes as well as measurements accessed by region-based queries. We have found the GNP-based approach to be very effective in getting large queries answered by a small number of representatives. However, it is less adaptive than the DHT-based approach, and the latency of queries is higher.

Although the results are promising, techniques described in this paper represent only the first steps towards building a scalable and powerful distributed network querying system. We are investigating the hybrid approach of combining query shipping and data shipping. Finally, we plan to consider more sophisticated caching schemes such as semantic caching [6].

9. REFERENCES

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

[2] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[3] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[4] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical internet coordinates for distance estimation. In *International Conference on Distributed Systems*, Tokyo, Japan, March 2004.

[5] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[6] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341, Bombay, India, September 1996.

[7] Amol Deshpande, Carlos Guestrin, Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *Proceedings of 30th International Conference on Very Large Databases (VLDB)*, 2004.

[8] Ian Foster and Carl Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., 1999.

[9] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, September 2003.

[10] George Kollios, Jeffrey Considine, Feifei Li, and John Byers. Approximate aggregation techniques for sensor databases. In *IEEE Conf. on Data Engineering*, 2004.

[11] Yannis Kotidis. Snapshot queries: Towards data-centric sensor networks. In *IEEE Conf. on Data Engineering*, 2005.

[12] Iosif Lazaridis and Sharad Mehrotra. Approximate selection queries over imprecise data. In *IEEE Conf. on Data Engineering*, pages 140–152, 2004.

[13] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 120–130. ACM Press, 2000.

[14] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, April 2004.

[15] T. S. Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. *Proceedings of IEEE Infocom*, 2002.

[16] T. S. Eugene Ng and Hui Zhang. A network positioning system for the internet. In *USENIX Annual Technical Conference 2004*, Boston, MA, June 2004.

[17] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 355–366, Santa Barbara, California, May 2001.

[18] Chris Olston. *Approximate Replication*. PhD thesis, Stanford University, 2003.

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

[20] Adolfo Rodriguez, Charles Killian, Dejan Kostić, Sooraj Bhat, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

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

[22] Ozgur Sahin, Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. A peer-to-peer framework for caching range queries. In *IEEE Conf. on Data Engineering*, 2004.

[23] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[24] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, 2003.

[25] Praveen Yalagandula and Mike Dahlin. A scalable distributed information management system. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 379–390. ACM Press, 2004.