# DISTRIBUTED NETWORK QUERYING WITH BOUNDED APPROXIMATE CACHING

by

Badrish Chandramouli

Department of Computer Science
Duke University

# Contents

# List of Tables

# List of Figures

# Chapter 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 [12]. 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 [24], 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 demonstrate with our experiments. If kept unchecked, network activities caused by the queries could interfere with normal operations and lead to unintended ar-

tifacts in performance-related measurement values. These problems are exacerbated by periodic monitoring queries, 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 exactly the 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 [21]); in this work, 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 project. The first approach, described in Chapter 3, uses a recursive partitioning of the network space to place caches in a static, controlled manner. The second approach, described in Chapter 4, uses a *distributed hash table* (e.g., [26]) 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 Chapter 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 [28], a scalable Internet emulation environment, we show in Chapter 7 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 Chapter 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 Chapter 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.

# Chapter 2

# System Overview

## 2.1 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. We will discuss how to handle locality in other spaces briefly in Chapter 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 Chapter 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 are presented in Chapter 6.

## 2.2 Bounded approximate caching

As discussed in Chapter 1, the brute-force approach of contacting each owner to obtain measurement values is unnecessary, expensive (this is also shown in our experiments), 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 ID of the measurement being cached, the cached measurement value and the time at which this measurement was taken (at the owner), a bound $[-\delta^-, \delta^+]$, and 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., [23]); such techniques are outside the scope of this project and largely orthogonal to the contributions of this work. In this project, we focus on techniques for *selecting* bounded approximate caches across the network 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.

## 2.3 Selecting and locating caches

We have developed two approaches to selecting and locating caches in the network. The first approach uses a hierarchy obtained 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 Chapter 3.

The second approach, which we focus more on, 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. 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 Chapter 4, including a discussion on the cost/benefit analysis for making caching decisions. A detailed comparison between the two approaches is given in Chapter 5.

# Chapter 3

# GNP-Based Controlled Caching

## 3.1 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*) [20]. 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 [20]. We note that GNP could be replaced by any other network positioning technique such as Vivaldi [7].

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 [17]; 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 3.1 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.

## 3.2 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 3.1, an order-$i$ sibling square of $O$ is an order-$i$ square that belongs to the same

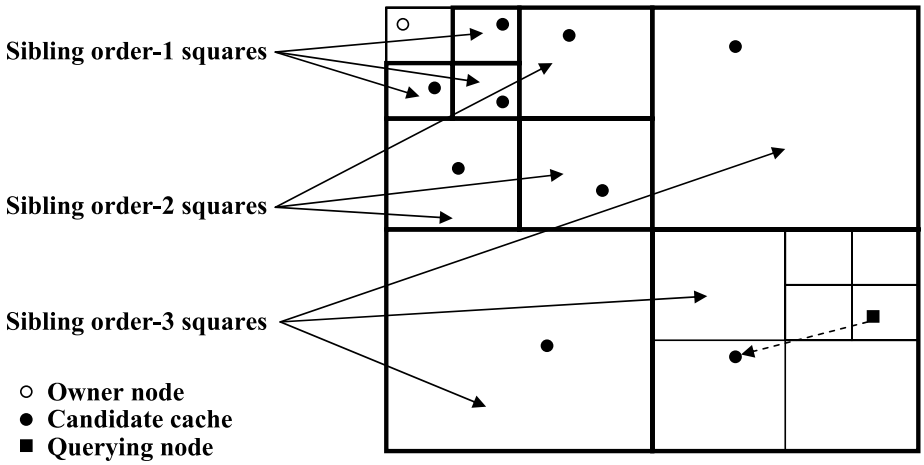


**Figure 3.1**: Recursive partitioning of the GNP space into squares.

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 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

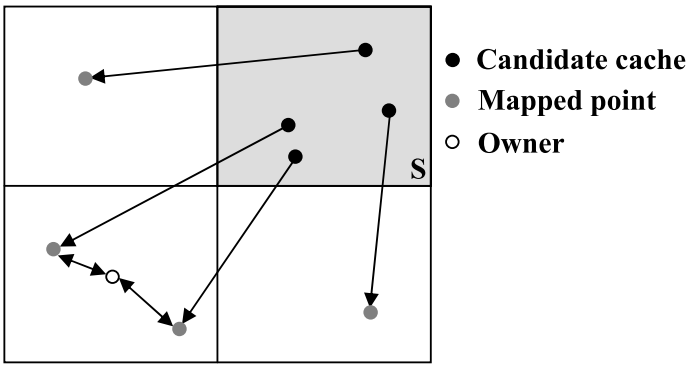


**Figure 3.2**: $k$-nearest mapped cache locator.

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 3.2. 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.

## 3.3 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 (Chapter 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 point in GNP space."

Based on this, we set up a hierarchy of GNP servers that export the following functionalities:

1. *get_region*: Given a GNP center and radius, it returns the set of nodes falling within the hypersphere defined.

2. *get_coords*: Given a node's IP address, this function returns the corresponding GNP coordinates.

3. *get_sibling_representatives*: This function takes the owner node's address as input and returns the set of representatives of that owner, at the level of that GNP server.

4. *get_closest_representative*: This function takes the querying node address and the owner node address as inputs. If the owner does not belong to any of its child squares,

it returns `null`. Otherwise, it applies the locator function to the square containing the querying node, with the owner node as input, and returns the corresponding representative node back to the caller.

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.

## 3.4 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 Chapter 4.

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

# Chapter 4

# DHT-Based Adaptive Caching

While simple to implement, the controlled caching approach described in the previous section has a number of shortcomings; a detailed discussion will be presented in Chapter 5. Briefly, the problems are that GNP servers carry potentially much higher load than other nodes in the system, and that the static, hash-based cache placement scheme captures some, but not all types of locality that we would like to exploit. 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 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 [26] 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 [26].

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 [26] 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 shown in Figure 4.1. This aggregation effect is used by SCRIBE [5] in building a scalable multicast system. The same effect is exploited by our system, although for a different purpose, as we 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

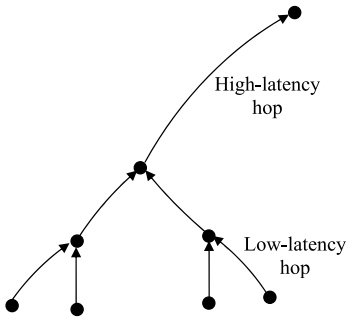


**Figure 4.1**: The message aggregation effect in Pastry.

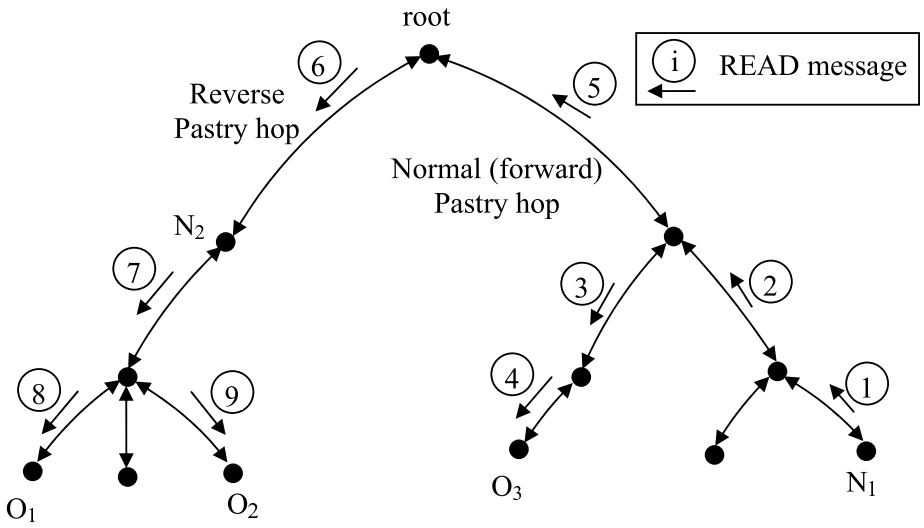


**Figure 4.2**: Two-way aggregation with Pastry.

| 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 4.1**: List of messages in the system.

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 4.2. Queries first flow up the tree following normal (forward) Pastry routes, and then down to owners following reverse Pastry routes. Nodes along these routes are natural candidates for caches. Our system grows and shrinks the set of caches organically based on demand, according to a cost/benefit analysis using only locally maintained information. The operational details of our system are presented next. Table 4.1 lists the types of messages in our system for reference.

### 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 mes-

sages 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* [2]. 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 4.2 shows the flow of `READ` messages when node $N_1$ queries measurements owned by $O_1$, $O_2$, and $O_3$, assuming that no caching takes place. If node $N_2$ happens to cache measurements owned by $O_1$ and $O_2$, then messages 7 through 9 will be saved. 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 Chapter 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 Chapter 2, we use a timeout mechanism to handle failures.

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. This tree is shown by dotted arrows in Figure 4.3, to be discussed in Section 4.2.4. 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.

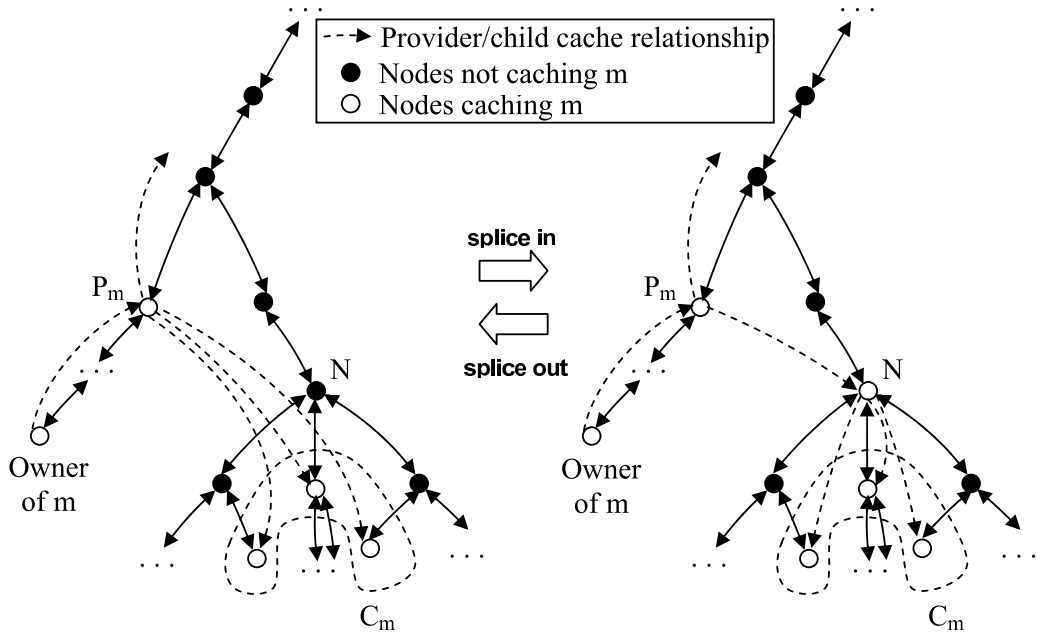


**Figure 4.3**: Splicing: adding and removing a cache.

### 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 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 4.3. 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 4.3 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 $\mathcal{C}_m$. Since updates can be sent using direct IP, the exact value of

$B_{update}(m)$ depends on the latencies between $P_m$, $\mathcal{C}_m$, and $N$. This computation is complex and requires the maintenance of a large number of parameters; hence we approximate this benefit to be proportional to the reduction in update cost from the perspective of the cache provider $P_m$. Thus, $B_{update}(m) \propto (\sum_{X \in \mathcal{C}_m} U_{m,X} - U_{m,N})$. We find this approximation to work well in our experiments.

- $C_{update}(m)$, 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 our constraints 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 above some threshold. 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 calculates $d_m$ 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 that 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 that these nodes can increment their $d_m$ accordingly. The request rate $H_m$ is 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 $X \in \mathcal{C}_m$ are 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 di-

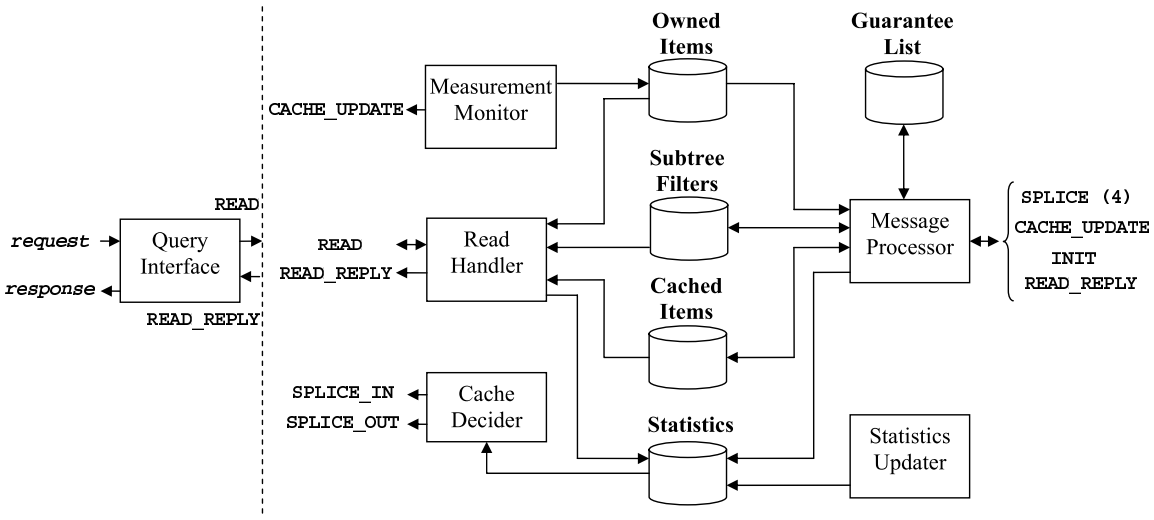


**Figure 4.4**: System architecture.

rect 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 $\min\{1, 2/(s \cdot U_{m,X})\}$ 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.

## 4.3 System Architecture

Figure 4.4 shows the detailed architecture of of our implementation. The *Request Processor* receives the measurement request, which consists of the set of owners and measurements desired, and the requested bound width. It sends a `READ` request to the *Read Handler*, which checks if the requested item is either owned locally or present in the cache at the requested bound width. If yes, the `READ_REPLY` is sent immediately. The unanswered part is forwarded as described earlier. The Read Handler at each step performs the same tasks until the request reaches either a cache or an owner. Each owner has a *Measurement Monitor* that monitors and updates measurements in the database of owned items. Whenever a guaranteed bound is violated, a `CACHE_UPDATE` is sent to the child cache, which in turn sends the same to its violated guarantees. The *Cache Decider* wakes up periodically and uses the collected statistics to make caching decisions. Based on the decisions, it send out `SPLICE_IN` and `SPLICE_OUT` messages. The *Message Processor* intercepts a number of messages, updates statistics as necessary, updates the cache if necessary, and forwards the messages or generates new messages (such as a `SPLICE_IN_OK` in response to a `SPLICE_IN` request that can be satisfied). It uses the *subtree filters* to perform reverse forwarding if the message is on the reverse route, otherwise normal Pastry routing is used in the forward direction. In addition to the statistics updates that are done by the Read Handler and the Message Processor, there are certain updates which are performed by the *Statistics Updater*. This includes operations such as calculating the local hit rate as a moving average by using a hit counter that it periodically resets.

# Chapter 5

# Discussion

## 5.1 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 Chapter 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 [26]; 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 request rate from all querying nodes makes caching cost-effective. This analysis will be confirmed by experiments in Chapter 7.

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 Chapter 7 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; the DHT-based approach has to rely on Pastry to exploit locality indirectly, which may be less effective in small systems since Pastry would have to work with a very small number of routing alternatives.

## 5.2 Alternative definitions of regions

So far, we have been assuming that query regions exhibit locality in terms of network proximity. As mentioned in Chapter 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 $k$-nearest mapped 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 Chapter 4 (except on $\mathcal{T}_{app}$ instead of $\mathcal{T}$), which allows locality among owners in the application-defined space to be exploited.

# Chapter 6

# Query Processing

In the previous three chapters, the focus has been on designing and implementing an infrastructure that can support issue and evaluation of approximate queries using ranges. In this chapter, we deal with answering queries posed by users, using the infrastructure. We first describe the design and implementation of `queryclient`, a generic query language based on relational algebra with syntax somewhat similar to SQL. We then discuss aspects concerning the use of this interface to process queries and answer set queries and top-$k$ queries.

## 6.1 The query language

We have designed a generic query language based on relational algebra. We have developed a query client that can operate in interactive mode and batch mode. In the interactive mode, the user issues commands and the client executes these commands and generates results. In the batch mode, the user specifies the name of a query file at the command prompt, and the client executes the commands in the query file. In the following subsections, we discuss the commands supported by the client.

#### The `region` command

This command is used by the user to define a region in network coordinate space (such as the GNP space). It defines the context for user queries. For example, when users wish to select nodes from two areas, they would start off by defining two regions encompassing the areas.

Input: Center of the region and radius of the region

Output: A table containing one tuple corresponding to each node in the region

Example:

```
> r1 = region(100, 60, 150, 150)
```

This returns a table `r1`, with all nodes in the region with center $(100, 60, 150)$ and radius $150$.

**The `iplist` command**

This command is used by the user to expicitly define a region in GNP space by specifying a list of node addresses. It defines the context for user queries. For example, when users wish to select nodes from two sets of nodes, they would start off by using this command to define the two regions.

Input: List of IP addresses belonging to the region

Output: A table containing one tuple corresponding to each node in the list

Example:

```
> r1 = iplist(10.0.0.25, 10.0.1.168, 10.0.1.1, 10.0.0.129)
```

This returns a table `r1`, with the four nodes.

**The `select` command**

This command is used to select tuples from a table based on the input condition.

Input: The name of a table, and a select condition to be evaluated

Output: A new table with the result of applying the select clause

Example:

```
> r2 = select(r1, CPU_LOAD($1)<0.85)  % $
```

This returns a table containing only the nodes in `r1` which have a CPU load less than 0.85.

**The `topk` command**

This command is used to select the $k$ tuples with highest values of a specified metric from the input table.

Input: The name of a table, the metric over which the top-$k$ tuples are to be computed, and the value of $k$

Output: A new table with the result of applying the top-$k$ clause

Example:

```
> r3 = select(r2, CPU_LOAD($1), 10)  % $
```

This returns a table containing the 10 nodes from `r2` which have the highest CPU loads.

**The `join` command**

This command is used to perform a join of two tables and apply a given selection criterion.

Input: The names of the tables to be joined, and the select condition to be evaluated for the join

Output: A new table with the join result after applying the select clause

Example:

```
> r4 = join(r1, r2, LATENCY($1,$2)<100)
```

This returns a table containing only the pairs of nodes in `r1` and `r2` which have latency between them less than 100ms.

**Other commands**

`queryclient` also supports other commands such as `sleep` (to sleep for the specified number of seconds), `execute` (to execute the contents of the specified file), `bound` (to set the bound or degree of approximation), `print` (to print the contents of a table), and

`quit` (to exit the queryclient). The `print` command can also accept an `orderby` clause to optionally indicate the desired ordering of the result set of tuples.

## 6.2 The query processor

### 6.2.1 Answering set queries

We are interested in answering set queries efficiently using the infrastructure described above. Consider the model query given below:

```
select r1.node, r2.node
from   region r1, region r2
where  latency(r1.node, r2.node) < 100ms and
       cpu_load(r1.node) < 0.5 and
       cpu_load(r2.node) < 0.8
```

This type of query can be translated into our query language easily. The translated code is shown below:

```
bound 0.1
r1 = region(...)
r2 = select(r1, CPU_LOAD($1) < 0.5)
r3 = region(...)
r4 = select(r3, CPU_LOAD($1) < 0.8)
bound 10
r5 = join(r2, r4, LATENCY($1,$2) < 100)
print r5
quit
```

This query can be answered using the cached bounds on the individual tuples (as established by our network of guarantees). When a `select` or `join` clause is executed, the

client contacts the local server and issues a request for the set of measurements that are required to answer the query, along with error bounds. In the GNP-based approach, the server computes the set of closest representatives by looking up its cache or by querying the GNP servers. It then sends each closest representative the set of measurement requests that are owned by that representative, along with the error bounds. The responses are collected and returned to the query client. In case of the DHT-based approach, the querying node cannot determine the set of caches a priori. Hence, the entire request is sent along its way in the DHT. Group splitting occurs within the DHT at forward time, as explained in Chapter 4. The responses are sent directly to the querying node. The responses are collected and returned to the query client.

The result is returned by the server to the client in the form of ranges for each measurement. Using these approximate responses and the selection clause, the client splits the tuples into three sets ($T^+$, $T^-$, and $T^?$) using the terminology in [21]. $T^+$ is the set of tuples that will definitely pass the predicate (wherever they may lie within their low and high bounds). $T^?$ is the set of tuples that may or may not pass the predicate (exact value needs to be known before accepting or rejecting the tuple). $T^-$ is the set of tuples that will surely not pass the predicate. The client returns tuples in $T^+$ and also returns the number of questionable tuples and negative tuples. We can calculate the goodness of our result using the symmetric multi-set difference metric as defined in [9]. If this does not satisfy the user precision requirement, the query can be reissued with tighter requested bounds. Otherwise, we order the tuples in $T^?$ in decreasing order of their probability of migrating to $T^+$ on determining a tighter bound. We propose to process each tuple in this order, determining tighter bounds which could migrate the tuple to $T^+$. The above process is continued until the goodness of the result matches that expected by the user.

Clearly, the above description assumes that a particular tuple is returned only if it is absolutely certain to be present in the result. We could also consider the situation where

a tuple is known to be present in the result with a particular degree of confidence, defined by the probability of it satisfying the predicate of the query. If the degree of confidence in that tuple is acceptable to the user, we can speculatively return the tuple (without incurring the additional expense of determining for sure that the tuple is in the output set) along with the degree of confidence. To measure the degree of confidence of satisfying the query predicate, we can use the past history of the measurement, i.e. the distribution function of the measurement variable, to predict the confidence of that tuple satisfying the predicate. The owner of a measurement variable (e.g., the owner of the latency from node $N_b$ to node $N_c$) knows the exact values that the variable took over its past $k$ minutes. It could then assume a model distribution (e.g., normal) and fit the data into the distribution. Once the distribution is approximated, the owner could propagate this information to its representative nodes along with the measurements. This information is finally propagated to the requesting node when it requests the data item. Using this distribution, the guaranteed bounds, and the latest measurement along with timestamp, the requesting node can approximate the probability of the bounded measurement satisfying the query predicate.

### 6.2.2 Answering top-*k* queries

Though the focus of this work has been on building the infrastructure and answering set queries using the infrastructure, we discuss here some aspects and issues to be dealt with in order to extend the system to handle top-$k$ queries. Top-$k$ queries are an important class of queries, where the user is interested in only the set of result tuples with the highest $k$ values of some metric. The value of $k$ and the metric can be specified using the `topk` clause described earlier. If the user wishes to retrieve an ordered list of results, they can specify the same using the `orderby` clause of the query language discussed earlier.

There are several issues that need to be addressed when we wish to answer top-$k$ set queries approximately. It might be an implicit user requirement in an ordered top-$k$ query

| Tuple | From | To | Lower bound | Upper bound | Last value |
|---|---|---|---|---|---|
| $T_1$ | $N_2$ | $N_5$ | 40ms | 49ms | 48ms |
| $T_2$ | $N_3$ | $N_6$ | 10ms | 55ms | 11ms |
| $T_3$ | $N_4$ | $N_7$ | 1ms | 8ms | 4ms |
| $T_4$ | $N_3$ | $N_8$ | 60ms | 80ms | 62ms |
| $T_5$ | $N_2$ | $N_8$ | 50ms | 59ms | 51ms |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

**Table 6.1**: Example list of measurements in the local database of a node

that the top result returned be the best i.e. there should be no tuple returned later, that would be ranked higher than the results already returned. We could relax this requirement by associating each tuple returned with a confidence metric indicating the degree to which we are sure of its position in the ranked result. The technique put forth in section 6.2.1 will need to be modified to handle top-$k$ queries. We could extend our system and implement techniques of answering top-$k$ set queries approximately, and with low communication cost.

As a motivating example, suppose a node $N_1$ has latency tuples in its local database as shown in table 6.1. The technique in section 6.2.1 for answering the model query will select tuple $T_1$ over $T_2$ because it is sure of being in the result set. However, from a top-$k$ set query point of view, we might prefer returning the second tuple since it is likely to be ranked higher than the first tuple.

In general, consider a set of $measurement$ tuples (as shown in table 6.1). Given this set, we wish to rank the tuples in decreasing order of the true value of the measurements. We can establish a $partial\ order$ on the set of tuples based on the known bounds for each of them. The Hasse diagram for the ordering relation in the above example is shown in figure 6.1. In this diagram, an edge pointing upwards from tuple A to tuple B indicates that $A \leq B$ according to our ranking criterion. We could return this partial order to the user. We might instead want to choose a topological sorting of this partial order such that the

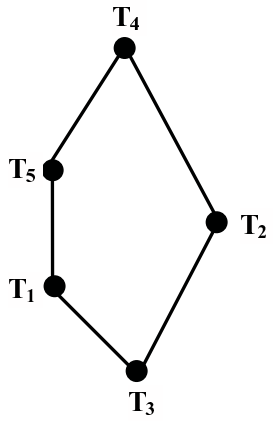


**Figure 6.1**: Hasse diagram for the ordering of tuples in table 6.1.

error in the top-$k$ result is minimized. [11] is a comprehensive treatment of the subject of comparing top-$k$ lists. We could use a distance metric such as Kendall's tau to quantify the difference between two top-$k$ lists (in our case, between the approximate top-$k$ list and the exact top-$k$ list). This could serve as the error metric for which we minimize.

A more general question relates to how top-$k$ is specified. It could be in the form of a number of **order by** clauses but this forces a strict priority. We could have a scoring function to map each tuple to a single score, and then rank by the score. But, devising such a metric is a hard problem in general. Another solution would be to use the $skyline$ clause [3]. The skyline clause is syntactically similar to an order-by clause and is based on the idea that a tuple is kept only if there is no better tuple with respect to each of the preference criteria.

# Chapter 7

# Experiments And Results

## 7.1 Implementation

We have implemented both approaches in Chapters 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 [25] 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.

## 7.2 Experimental setup

We conduct our experiments on ModelNet [28], 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.

Figure 7.1 shows the overall architecture of our evaluation methodology. For all our experiments, we use $20{,}000$-node INET [6] topologies with a subset of $250$ nodes participating in measurement and querying activities. These nodes are emulated by twenty

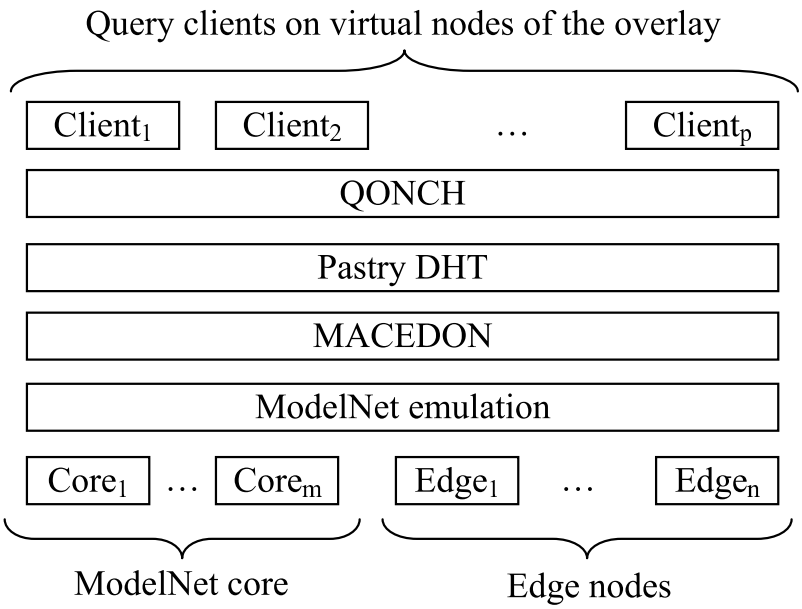


**Figure 7.1**: Evaluation architecture.

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 modified FreeBSD-4.9. We deploy ModelNet on these nodes. For the DHT-based approach, we run MACEDON on the emulated overlay nodes. The Pastry DHT is deployed using MACEDON. Finally, the system runs on top of the API exposed by the Pastry implementation of MACEDON.

Our system exposes an interface for issuing queries; this interface can be accessed via a local socket connection. Querying clients run on the virtual nodes in the overlay, and accept queries in the query language described earlier (either from the command line or via a batch interface). The requests for measurements of specified bound width are sent to our system via the socket interface.

While all results reported in this work use ModelNet, we have also run smaller experiments (with around 50 nodes) over PlanetLab [24]. We note that the number of owners and querying nodes in our experiments is not constrained by the scalability of our system, but rather by the hardware resources available for emulating or deploying it; as future work, we plan to use a simulation-based evaluation approach, which would allow us to perform

larger experiments with some sacrifice in realism.

## 7.3 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 $n_o$ owners that are near one another (not necessarily close to the querying nodes). This group of queries should benefit most from caching, since they exhibit locality among both querying nodes and 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 in 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 $n_o$ owners that are near one another. 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

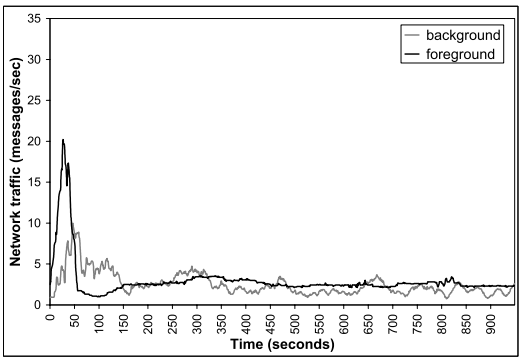


**Figure 7.2**: Traffic vs. time; cache size $100$.

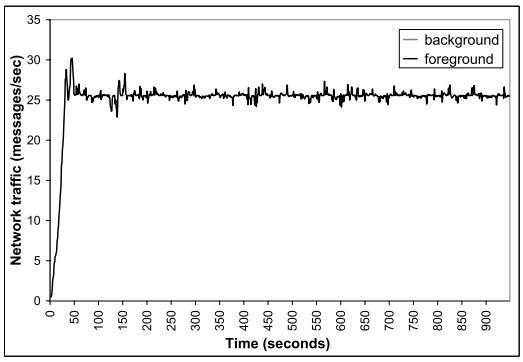


**Figure 7.3**: Traffic vs. time; cache size $0$.

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 $\sigma$. If $\sigma$ 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.

## 7.4 Results for the DHT-Based Approach

### 7.4.1 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 exper-

iment, 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 7.2 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 7.3 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 7.2, 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 7.3, 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 7.4 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 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 splice operations caused by thrashing. Nevertheless, our system is able to handle this situation reasonably well; the overall traffic is still much lower than if no

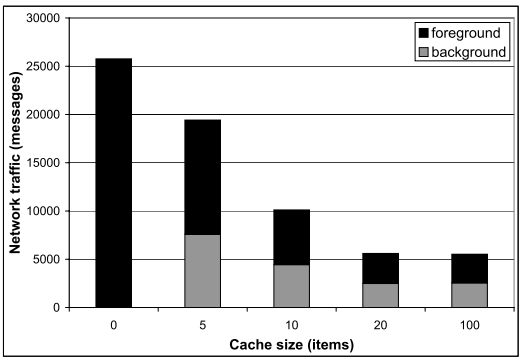


**Figure 7.4**: Total traffic; various cache sizes.

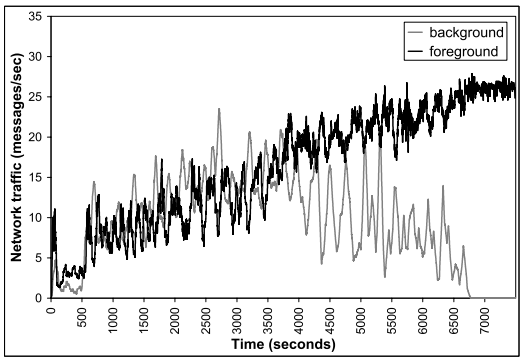


**Figure 7.5**: Traffic vs. time as update rate increases; cache size $100$.

caching is used.

### 7.4.2 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 $\sigma$ 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 7.5. 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 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 7.3). 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

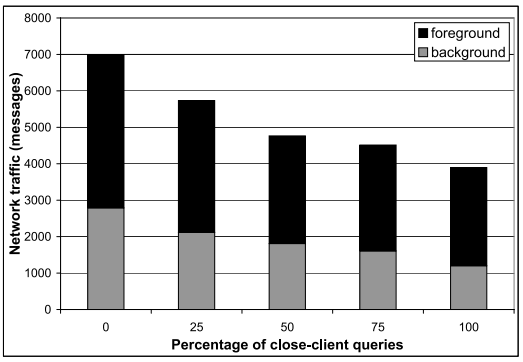


**Figure 7.6**: Total traffic as the percentage of queries from nearby nodes increases; cache size $100$.

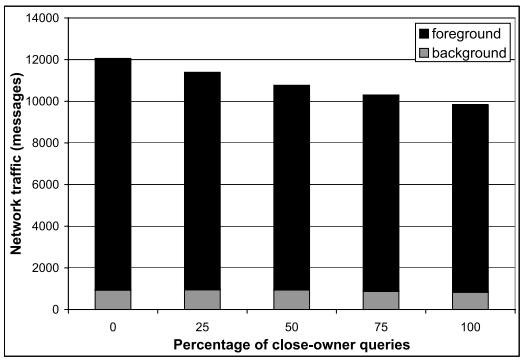


**Figure 7.7**: Total traffic as the percentage of queries to nearby owners increases; cache size $8$.

strategy according to the volatility of measurements.

### 7.4.3 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 7.6 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 $[0, 0, 4, 0]$, where the percentage of queries requesting nearby nodes increases from $0\%$ to $100\%$. Since we want to show 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 $8$. From the results in Figure 7.7, we see that the total traffic reduces as the percentage of queries requesting nearby owners increases. This shows that our system derives performance benefits by exploiting locality in query regions.

### 7.4.4 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 7.8 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. We see that 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.

## 7.5 Results Comparing the GNP- and DHT-Based Approaches

### 7.5.1 Query latency

As discussed in Chapter 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

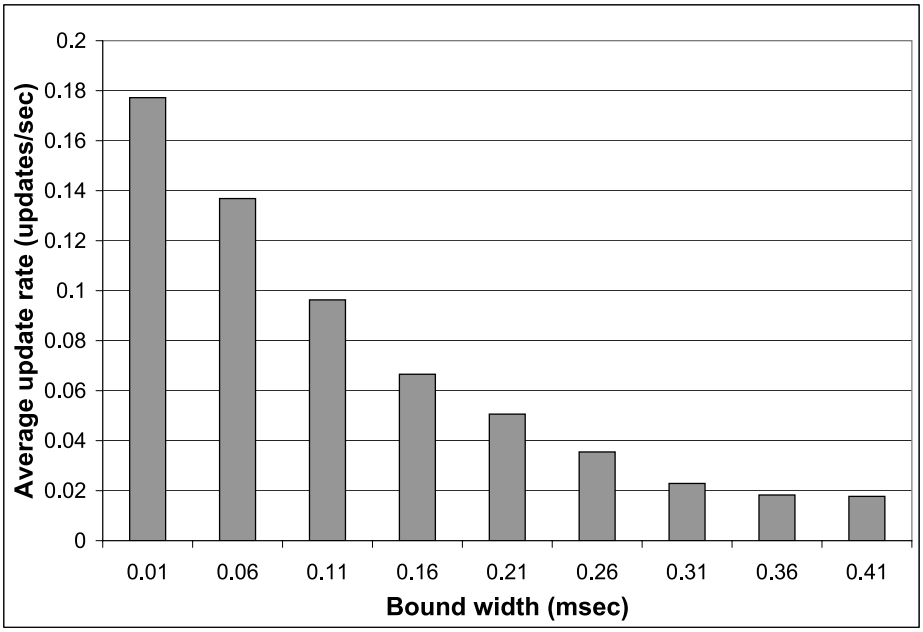


**Figure 7.8**: Update rate as bound width increases; real latency measurements.

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 that is far away. The update rate of the measurement is just high enough so that each querying node will not start caching the measurement locally.

Figure 7.9 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

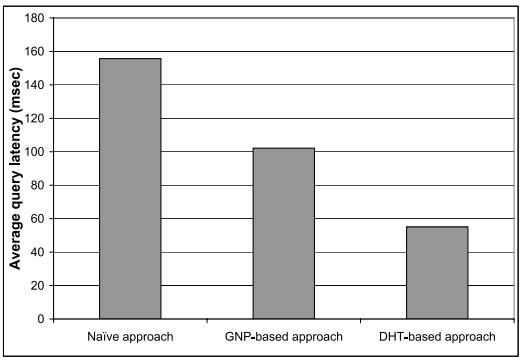


**Figure 7.9**: Average query latency comparison for three approaches.

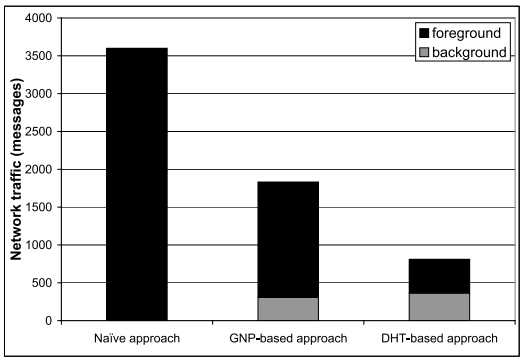


**Figure 7.10**: Total traffic comparison for three approaches.

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. Static cache selection causes the four querying nodes to be stuck with this cache in the GNP 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.

### 7.5.2 Total traffic

As we have also discussed in Chapter 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 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 7.10 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.

# Chapter 8

# Related Work

*Network monitoring.* A large number of network monitoring systems have been developed by both the research community and commercial vendors; we discuss two representative systems here. Astrolabe [29] is a system that continuously monitors the state of a collection of distributed resources and reports summarized information to the its users; it supports scalable aggregation through a hierarchy of "zones" and efficient propagation of updates. Ganglia [18] is a system for monitoring a federation of clusters: Within a single cluster, each node pushes its updates to all other nodes using a multicast protocol; federation is achieved using an aggregation tree of representative cluster nodes where each parent periodically polls its children for aggregated information. While our work consider the same general problem of network monitoring, 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 existing network monitoring systems in terms of both functionality and performance.

*Data streaming.* A lot of attention has recently been focused on the general field of data streaming techniques, e.g. STREAM [19] and Aurora [1]. Stardust [4] is a more recent proposal that offers a unified solution to the problem of summarizing and indexing multiple data streams in real time. While we answer queries over distributed parameters, we do not use streaming techniques directly in our system because of the high network cost of streaming values from all the nodes to the query node. Even if optimizations such

as installing filters at the source [22] are used, the network costs are unlikely to scale to a very large number of nodes.

*Data processing on overlay networks.* PIER [13] is a DHT-based massively distributed query engine that brings database query processing facilities to new, widely distributed environments. In pursuit of scalability and widespread adoption, PIER relaxes certain traditional database assumptions, e.g., ACID transactions and standard database schema. 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.

Şahin et al. [27] propose a DHT-based technique for evaluating range queries, by efficiently caching an answer (or pointers to it) at a "target node" responsible for the range; in effect, nodes in the DHT cooperatively store range partitions of a database. Their technique is similar in spirit to our controlled caching approach. Our GNP-based controlled caching approach does not guarantee that all measurements in a query region will be cached at the same node, but it is better at capping cache update costs. Furthermore, the GNP-based approach is able to ensure a good spread of candidate caches over the network for each data item.

Locality-aware DHTs have been used to build SCRIBE [5], a scalable multicast system, and SDIMS [31], 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 [21], along with techniques such as adaptive bound setting, source cooperation in cache synchronization, etc. We apply bounded ap-

proximate 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.

Kollios et al. [14] use sketch-based techniques to support approximate aggregation for sensor databases. Lazaridis et al. [16] consider the problem of evaluating selection queries over imprecise data. Although our paper focuses on caching and does not consider query processing issues in detail, the ideas in [14, 16] can be applied in our system.

Deshpande et al. [10] propose integrating the database system with a probabilistic model that captures correlations in sensor readings, so that one reading can be derived approximately from others. Kotidis [15] uses a distributed localized algorithm to elect representative sensors to approximate correlated readings from neighboring sensors. Our bounded approximate caching techniques can be used to support continuous queries that monitor the validity of models or maintain model parameters at remote querying nodes.

*Web caching and web replication.* Web caching is often done by ISPs using web proxy servers. Cooperative proxy caching is often employed in the World Wide Web; this is studied in detail by Wolman et al. [30]. Web caches perform caching of relatively static data close to the requestor to provide low latency access without going to the source. They do not consider the dynamics of the data itself because there are no bounded approximation guarantees; when content changes, either the cached data is updated or the item is purged from the cache. Web replication refers to data sources spreading their content across the network, primarily for load balancing. Content distribution networks (CDNs) such as Akamai perform replication across the Internet by establishing mirrors and redirecting users to nearby servers. Here too, the cache content is stored exactly and most often relatively stable content (e.g. images) is replicated. They do not deal with the problem of rapidly

updating data; this means that they can afford to establish a large number of 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 dynamicaly chosen nodes in the network. In constrast, traditional CDNs and web proxies have relatively static replica selection.

# Chapter 9

# Conclusions And Future Work

In this work, 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 and evaluated a GNP-based controlled caching approach and a DHT-based adaptive caching approach. Experiments using large-scale emulation 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 to different types of workloads successfully. 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. The GNP-based approach is more centralized and controlled, and less effective than the DHT-based approach in exploiting certain types of localities. Nevertheless, the GNP-based approach is simpler, and provides more direct control over locality for smaller systems.

Although the results are promising, techniques described in this work represent only the first steps towards building a powerful distributed network querying system. We are currently working on improving failure handling in our system. The DHT-based approach provides a good starting point because of the resiliency of DHTs. With minor tweaks, our protocols can ensure correctness in case of failures, but we still need a closer look at the impact of failures on performance. We also plan to investigate the hybrid approach of combining query shipping and data shipping, and consider more sophisticated caching

schemes such as *semantic caching* [8].

# Bibliography


[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *SIGMOD*, 2003.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[3] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[4] A. Bulut and A. K. Singh. A unified framework for monitoring data streams in real time. In *ICDE*, 2005.

[5] 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.

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

[7] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, 2004.

[8] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[9] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003.

[10] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[11] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003.

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

[13] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *VLDB*, 2003.

[14] G. Kollios, J. Considine, F. Li, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.

[15] Y. Kotidis. Snapshot queries: Towards data-centric sensor networks. In *ICDE*, 2005.

[16] I. Lazaridis and S. Mehrotra. Approximate selection queries over imprecise data. In *ICDE*, 2004.

[17] J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, 2000.

[18] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 2004.

[19] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[20] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *IEEE INFOCOM*, 2002.

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

[22] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

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

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

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

[26] 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.

[27] O. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, 2004.

[28] 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.

[29] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 2003.

[30] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *SOSP*, 1999.

[31] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, 2004.