Supporting Better Scalability and Richer Subscription Models in Wide-Area Publish/Subscribe

Badrish Chandramouli
Department of Computer Science
Duke University
badrish@cs.duke.edu

Abstract

With the advent of Web 2.0 and the Digital Age, we are witnessing an unprecedented increase in the amount of information collected, and in the number of users interested in different types of information. This growth means that traditional techniques, where interested users poll data sources for information they are interested in, are no longer sufficient. Polling too frequently does not scale, while polling less often may result in users missing important updates. The alternative push technology has long been the goal of publish/subscribe systems, which proactively push updates (events) to users with matching interests (expressed as subscriptions). The push model is better suited for ensuring scalability and timely delivery of updates, important in many application domains: personal (e.g., RSS feeds, online auctions), financial (e.g., portfolio monitoring), security (e.g., reporting network anomalies), etc.

Early publish/subscribe systems were based on predefined subjects (channels), and were too coarse-grained to meet the specific interests of different subscribers. The second generation of content-based publish/subscribe systems provides finer granularity and greater flexibility by supporting subscriptions defined as predicates over message contents. However, in these systems, subscriptions are stateless filters over individual messages, so they cannot express queries across different messages or over the event history. The few systems that support more powerful subscriptions do not address the problem of efficiently delivering updates to a large number of subscribers over a network. Thus, there is a need to develop next-generation publish/subscribe systems that support richer, stateful subscriptions over an event history database, with flexible notification conditions. This support needs to be complemented with robust processing and dissemination techniques that scale to high event rates and large databases, as well as to a large number of subscribers over a network.

The main contribution of our work is a collection of techniques to support efficient and scalable event processing and notification dissemination for a wide-area publish/subscribe system with a rich subscription model. We have conducted two pieces of work in this area. First, we developed a wide-area network monitoring system that intelligently places bounded approximate caches (subscriptions) across the network to exploit localities among

queries and data sources. Second, in the context of wide-area publish/subscribe systems supporting stateful subscriptions, we investigated the interface between event processing by a database server and notification delivery by a dissemination network. Previous research in publish/subscribe has largely been compartmentalized; database-centric and network-centric approaches each have their own limitations, and simply putting them together does not lead to an efficient solution. A closer examination of database/network interfaces yields a spectrum of new and interesting possibilities. In particular, we proposed message and subscription reformulation as general techniques to support stateful subscriptions over existing content-based networks, by converting them into equivalent but stateless forms. We showed how message and subscription reformulation can successfully be applied to various stateful subscriptions including range-aggregation and joins. These techniques were shown to provide orders-of-magnitude improvement over simpler techniques adopted by state-of-the-art content-based publish/subscribe systems, and were shown to scale to millions of subscriptions.

As future work in this domain, we will investigate scalable techniques for handling per-subscription notification conditions. Consider a subscriber that is only notified when the monitored value has changed by more than some prescribed threshold from the last reported value; this last reported value is an example of per-subscription state that may be unique to each subscription. Our preliminary investigations show that it is possible to support efficient event processing at a database with or without knowledge of the actual subscriptions. In addition, it is possible to efficiently disseminate notifications to a large number of subscriptions by appropriate message and subscription reformulation. Moreover, by allowing the notification requirement to be relaxed in a well-defined and disciplined manner, we can obtain commensurate improvement in processing and dissemination efficiency.

Further ahead, we plan to investigate batching techniques to handle high event rates, and tackle the problem of designing efficient dissemination techniques based on event statistics and subscription semantics. Our ultimate vision is a system with an optimization framework that intelligently chooses the appropriate interface between the database and the network to maximize efficiency. The optimization would be guided by both runtime statistics and subscription semantics. This unified view effectively treats different server processing and network dissemination techniques as indexes (with the associated costs) whose materialization and usage is driven by the optimization framework.

1 Introduction

With the advent of the Web 2.0 and the Digital Age, the necessity to capture and put together large amounts of information has brought about multiple large-scale data acquisition and integration efforts. An important next step is to disseminate data efficiently to users. Traditionally, users poll sources for information. However, polling too frequently may be inefficient, while polling less often may miss important updates. The alternative push technology has long been the goal of publish/subscribe systems, which proactively push updates (events) to users with matching interests (expressed as subscriptions). The push model is better suited for ensuring scalability and timely delivery of updates, important in many application domains: personal

(e.g., RSS feeds, online auctions), financial (e.g., portfolio monitoring), security (e.g., reporting network anomalies, distributing critical software patches), etc.

Publish/subscribe is a model of data dissemination [24], where data is *aperiodically pushed* from one or more sources (called publishers) to multiple destinations (called subscribers). Other dissemination models include periodic push (e.g., broadcast disks [2]), periodic pull (client polling systems), and aperiodic pull (classic request/response client/server systems).

A publish/subscribe system decouples data providers (sources) from data recipients (subscribers). Sources publish data to the system, while subscribers specify their preferences regarding the data they wish to be notified of, in the form of subscriptions. The underlying publish/subscribe middleware then takes the task of matching data items to the relevant subscribers, and notifying them accordingly. A typical publish/subscribe system is arranged as a set of nodes (called brokers) that cooperate to achieve the above goal. Subscribers attach themselves to brokers based on different criteria. Data sources generate updates to a database, which may be located at a server, a set of distributed servers, or distributed throughout the nodes in the network. The update could trigger a number of subscriptions. The publish/subscribe middleware is responsible for ensuring that the update is disseminated to all brokers hosting affected subscriptions.

Early publish/subscribe systems such as [35, 42, 52] are based on channels/subjects/topics. Channels are predefined, and each update is tagged with one or more channels. Subscribers, on the other hand, specify channels that they are interested in. The update is simply sent to all subscribers subscribing to that channel. These systems could take advantage of group delivery mechanisms such as multicast. However, the granularity of such systems was often found to be too coarse to fit the particular interests of individual users. This led to the second generation of *content-based* publish/subscribe systems. These systems provide finer granularity and greater flexibility by supporting subscriptions defined as filters over the message contents. A large number of such systems have been built in recent years. We briefly sample several of these systems in Section 7. In a typical content-based publish/subscribe system, events may conform to an *event schema* [38] such as `[SYMBOL: string, PRICE: float]`, with events such as `['GOOG', 240.50]`. Semantically, subscriptions are limited to being stateless filters such as `(SYMBOL='IBM' and PRICE<100)`. An incoming event can be matched to subscriptions either at a server or within the network, and the matching can be performed without knowledge of past history of events or the current subscription state. Content-based networks [10] are frequently used to disseminate events. Such systems have been extensively studied [5, 40] in both the database and networking communities.

The main contribution of our work is a collection of techniques to support efficient and scalable event processing and notification dissemination for a wide-area publish/subscribe system with a rich subscription model. We have conducted two pieces of work in this area. First, we developed a wide-area network monitoring system that intelligently places bounded approximate caches (subscriptions) across the network to exploit localities among queries and data sources. This is described in Section 3. Second, in the context of wide-area publish/subscribe

systems supporting stateful subscriptions, we investigated the interface between event processing by a database server and notification delivery by a dissemination network. Previous research in publish/subscribe has largely been compartmentalized; database-centric and network-centric approaches each have their own limitations, and simply putting them together does not lead to an efficient solution. A closer examination of database/network interfaces yields a spectrum of new and interesting possibilities. In particular, we proposed message and subscription reformulation as general techniques to support stateful subscriptions over existing content-based networks, by converting them into equivalent but stateless forms. We showed how message and subscription reformulation can successfully be applied to various stateful subscriptions including range-aggregation and joins. These techniques were shown to provide orders-of-magnitude improvement over simpler techniques adopted by state-of-the-art content-based publish/subscribe systems, and were shown to scale to millions of subscriptions. This work is described in greater detail in Section 4.

As future work in this domain, we plan investigate scalable techniques for handling a richer subscription model with support for per-subscription notification conditions. Consider a subscriber that is only notified when the monitored value has changed by more than some prescribed threshold from the last reported value; this last reported value is an example of per-subscription state that may be unique to each subscription. Our preliminary investigations show that it is possible to support efficient event processing at a database with or without knowledge of the actual subscriptions. In addition, it is possible to efficiently disseminate notifications to a large number of subscriptions by appropriate message and subscription reformulation. Moreover, by allowing relaxation of the notification requirement in a well-defined and disciplined manner, we can obtain commensurate improvement in processing and dissemination efficiency. Our preliminary observations and proposed techniques are discussed in detail in Section 5.

Further ahead, we plan to investigate batching techniques to handle high event rates. We plan to tackle the problem of designing efficient dissemination techniques based on event statistics and subscription semantics. Our ultimate vision is a scalable wide-area publish/subscribe system supporting a rich subscription model with flexible notification conditions. Such a system would incorporate an optimization framework that chooses the appropriate interface between the database and the network. This event-driven framework would be guided by both online statistics and semantic knowledge. This unified view effectively treats different server processing and network dissemination techniques as indexes (with the associated costs) whose materialization and usage is driven by the optimization framework. These ideas for future work are described in Section 6.

Finally, we present related work in Section 7 and conclude with the vision and goals in Sections 8.

2 Motivation

2.1 Powerful subscription models: challenging yet necessary

In both subject-based and content-based publish/subscribe systems, subscriptions are essentially simple stateless filters over defined over individual messages, so they cannot express queries of interest across different messages or over the event history. This limitation means that these systems cannot efficiently support modern application requirements. They make the end users deal with complex local post-processing, which requires maintaining unnecessarily large amounts of state and results in high volumes of updates.

Modern applications and users may want updates to be further transformed, correlated, and/or aggregated. They may want to receive notifications only when certain important events occur, and want to receive data that are filtered, joined, and summarized. For example, with a range-aggregate subscription, a user can track the minimum PER (price-to-earning ratio, a popular measure of stock quality) of stocks within a risk range. This subscription is stateful, because just by looking at a stock update message, the system cannot always tell whether or how the message would affect the subscription. To meet the needs of such applications, we need a publish/subscribe interface that supports a powerful subscription model that incorporates rich content definitions and flexible notification conditions.

In order to illustrate the challenges in supporting richer subscription models, consider the following example of a range-min subscription. Consider a publish/subscribe system that monitors the stock market for a large number of traders over a wide-area network. Conceptually, the system provides a database view STOCK(SYMBOL, RISK, PER, ...) that continuously tracks the up-to-date information for each stock. Suppose that a user is interested in tracking the minimum PER of stocks within a risk range $[x_1, x_2]$ that she is comfortable with. She can define a subscription over STOCK using a SQL query: `SELECT MIN(PER) FROM STOCK WHERE` x_1 `<= RISK AND RISK <=` x_2. To simplify discussion, let us focus on updates of PER, and assume that each update message has the schema $\langle$SYMBOL, RISK, PER, ...$\rangle$, where PER is the new price-to-earning ratio after the update. When such a message arrives, the system needs to notify those users whose subscription query results are affected by the PER update.

The range-min subscription is stateful. To illustrate, consider the current state of STOCK shown as a collection of points (labeled by t_i and shown in solid black) in Figure 1; the X-axis plots RISK, while the Y-axis plots PER. Each range-min subscription (labeled by s_i) is represented as a horizontal interval spanning the risk range of interest, whose height equals the minimum PER in that range.

Suppose that an update lowers t_4's PER to just below that of t_5 (indicated by a dotted line with arrow). This update should affect subscriptions s_3 and s_4, but not s_1, s_2, or s_5. For s_1 through s_4, their ranges all cover t_4's RISK. In order to determine that s_3 and s_4 are affected while s_1 and s_2 are not, the system must be able to compare t_4's new PER with the minimum PER currently maintained by each of these subscriptions; this latter information is not available

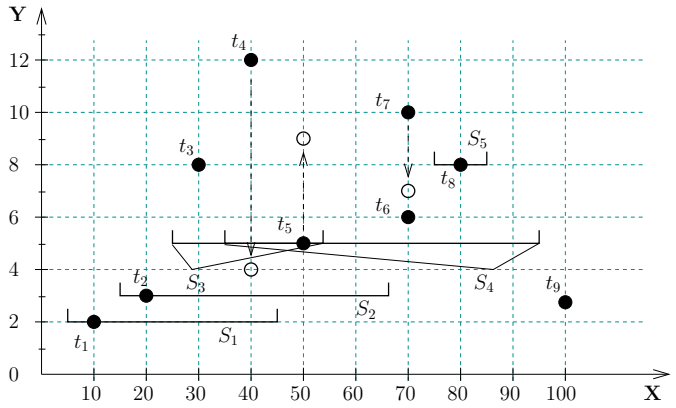

Figure 1: Example STOCK table and range-min subscriptions.

in the update message.

A more complicated situation arises when the current minimum PER shared by a group of subscriptions is updated higher, potentially exposing them to different new minima. For example, suppose that an update raises t_5's PER, as illustrated in Figure 1. As a result of this update, s_3 should be updated with t_3's PER, while s_4 should be updated with t_6's PER. Neither piece of information is available in t_5's update message. In general, the system must maintain the entire state of the STOCK table in order to handle such updates.

Support for these powerful subscription models should not impact the scalability in terms of data rates, database size, and number of subscriptions. This motivating factor is the driving force behind the work described in subsequent sections.

2.2 Need for designing the right database/network interface

At the conceptual level, the bulk of the work performed by a publish/subscribe system can be roughly divided into two components: (1) *subscription processing*, the task of matching and processing each incoming publish message with the large set of active subscriptions, and (2) *notification dissemination*, the task of notifying, over a network, those subscribers who are interested in the publish message. Previous work from the database research community has focused on efficient subscription processing; notification dissemination is rarely addressed. Most existing work assumes that a server maintains the entire database state and all subscriptions in the system, and is responsible for computing the set of subscribers affected by each incoming publish message. A straightforward way to notify this set of subscribers is to unicast a notification to each of them in turn. When many subscribers need to be notified, this approach will incur a large amount of outbound traffic from the server, and may easily overwhelm the server and its network links. As server-side subscription processing techniques (such as sharing [18] and indexing [26]) continue to mature, the dissemination bottleneck has surfaced in many systems, both research [20] and commercial [28]. Recently, the database community has made some initial efforts [20, 40] in addressing this problem (further discussed in Section 7), but much

research is still needed.

On the other hand, the networking research community has always focused on efficient notification dissemination. Notable mechanisms include *multicast* [3] and *content-based networking* [10]. With multicast, the system defines a number of *multicast groups*, each consisting of a set of subscribers, e.g., those who are interested in Google's stock. The network can efficiently disseminate the same message to all members of the group. With content-based networking, the system views each message as a tuple of attribute-value pairs; e.g., attributes in a stock update message may include `SYMBOL`, `RISK`, `PRICE`, `EARNING`, etc. Each subscription is defined as a predicate over the message tuple; e.g., `SYMBOL` $=$ `'GOOG'`, or `RISK` $\in [20, 60]$ (between moderately low to medium risk). The network is typically implemented by an *overlay* of nodes that perform application-level routing. Each overlay node maintains a summary of all subscriptions reachable from each of its outgoing overlay links, and it forwards an incoming message onto an outgoing link if the message matches the corresponding summary. Both mechanisms, however, support only *stateless* subscriptions, i.e., those that can be processed by examining the message itself. For multicast, a message's group id encodes its forwarding directions. For content-based networking, the message tuple contains all the information needed to forward the message.

To motivate the need to examine the database/network interface, let us go back to the example from Section 2.1 and examine how range-min subscriptions can be supported. Following the traditional database-centric approach, we can use a server to maintain all subscriptions and the up-to-date state of the `STOCK` table. Thus, for each incoming stock update message, this server can easily compute which subscriptions are affected and how they need to be updated. However, options for disseminating these notifications are limited. (1) Unicast is the most natural way, but can be inefficient for a large number of affected subscriptions. (2) Content-based networking is difficult to leverage because of an "impedance mismatch": A content-based network performs matching between messages and subscriptions in the network, while in this database-centric approach the server has already computed the list of affected subscriptions; converting this list back to a message for dissemination in a content-based network is not straightforward, and would be a waste of resources because of duplicate processing. (3) Multicast is a possibility, but to do a perfect job, we would need a multicast group for every possible subset of the subscriptions that could be affected by a stock update in the same way. There may be a prohibitively large number of such groups (up to 2^m if every subset from a total of m subscriptions can form a group), rendering multicast infeasible. Even if we restrict the problem to range-min subscriptions alone, it is unclear how to reduce the space of possible groups. For example, in Figure 1, although s_2's risk range contains that of s_3, not every update affecting s_3 would affect s_2, and vice versa.

An alternative is to follow a network-centric approach. Content-based networking is a natural starting point because it supports range subscriptions. We can "relax" a range-min subscription `SELECT MIN(PER) FROM STOCK WHERE` x_1 `<= RISK AND RISK <=` x_2 to a range subscription `SELECT * FROM STOCK WHERE` x_1 `<= RISK AND RISK <=` x_2. The network would then forward to each subscriber every stock update message that falls within her risk range.

Each subscriber locally maintains the content of the range subscription from which the range-min subscription can be derived. Note that all stocks in the range must be maintained (not just ones with the minimum PER) in order to handle the case when the minimum PER rises. Besides this maintenance overhead, a more serious issue is that the relaxation of a stateful subscription into a stateless one can potentially result in much more update traffic. For example, in Figure 1, any PER movement of t_4 above t_5's PER has no effect on any subscriptions, but with this approach, all updates of t_4 would still be forwarded to s_1 through s_4 simply because t_4 falls into their risk ranges.

This example shows that efficient support of stateful subscriptions is a challenging problem for wide-area publish/subscribe systems. On one hand, existing network dissemination mechanisms do not support stateful subscriptions directly. While it is possible to relax a stateful subscription into a stateless one and rely on subscribers to perform additional local post-processing, doing so requires unnecessarily large amounts of local subscription state and high volumes of notifications. On the other hand, while the database-centric approach can easily process stateful subscriptions at a server, disseminating notifications over a wide-area network remains difficult because of the inefficiency of unicasts and the difficulty in interfacing the server with advanced network dissemination mechanisms such as multicast and content-based networking.

We argue that the key to the solution lies in properly *interfacing* the database with the network, in order to combine the processing power of database servers and the dissemination power of the network effectively. In general, there is a wide spectrum of possibilities for interfacing the database with the network and for dividing up work between them. These possibilities provide an interesting set of trade-offs in terms of efficiency, scalability, and manageability of the system. To the best of our knowledge, there is no prior work that investigates this spectrum of database/network interaction models comprehensively. This unified perspective from both databases and networking enables us to identify interesting hybrid solutions that outperform approaches that are either database-centric or network-centric. This perspective is further discussed in Section 4, 5, and 6.

2.3 Need for a holistic view to wide-area publish/subscribe system design

Publish/subscribe solutions that have been proposed in both the database and networking communities have largely dealt with specific classes of scenarios. However, there has been no effort to integrate these techniques into a single framework that chooses the correct server-side indexing strategy and network-side dissemination technique based on the current scenario.

As motivation, we again look at the publish/subscribe system example from Section 2.1. Suppose that a user is interested in a simpler subscription, that of tracking the PER of a stock that she owns. However, the user wishes to be notified only when the PER changes by more than 10% from her last notified value. This is an example of a more powerful notification condition. She can define a subscription over STOCK using a SQL query: `SELECT PER FROM STOCK WHERE SYMBOL = 'GOOG' NOTIFY_WHEN RESULT_CHANGED_BY 10%`. Now, if the PER of stocks tend

to change very slowly, any update would tend to trigger very few subscriptions, and Unicast would be the preferred mechanism with very low overhead. On the other hand, if the PER of stocks tend to change rapidly and/or the notification conditions are very tight, it would be better to use a content-based network to deliver notifications. If the same large set of subscribers tend to get notified by the same events, it may be most efficient to setup a multicast group with those recipients.

From this example, we see that depending on the dynamic event and subscription characteristics, different techniques may need to be employed by the system. The interface between the network and the database is sensitive to the characteristics of the workload placed on the system. Similarly, on the database side, the use of different types of index structures is appropriate under different scenarios. This means that we need to introduce an optimization framework that chooses the appropriate interface between the database and the network. This event- driven framework would be guided by both online statistics and semantic knowledge. This unified view effectively treats different server processing and network dissemination techniques as indexes (with the associated costs) whose materialization and usage is driven by the optimization framework. We discuss this further in Section 6.

3 Distributed Network Querying

As a first step towards richer wide-area publish/subscribe, we started by looking at some popular wide-area applications and their data requirements. We looked at a scenario where a distributed set of nodes produce measurements (data) of interest, such as CPU load, average latency to other nodes, amount of free memory, etc. It turns out that approximate answers are usually sufficient as long as the approximation is quantified and the users can control the degree of inaccuracy. We addressed the problem of selecting and location bounded approximate caches in the network, by taking advantage of several optimization opportunities that exist in our target applications. The goal was to optimize overall network traffic, while adapting to different types of client workloads. This work showed that by exploiting the optimization opportunities, we can reduce network traffic to both maintain caches and answer queries approximately. A bounded approximate cache is similar to a continuous query with notification conditions. By creating such caches dynamically, placing these caches at strategic locations, and giving users a way to contact nearby caches easily, we were able to take advantage of aperiodic push while at the same time not sacrificing the ability to pull data when conditions are such that pull is more efficient.

3.1 Introduction

Consider a network of nodes, each monitoring a number of numeric measurements. These measurements may be related to performance, e.g., per-node statistics such as CPU load and the amount of free memory, or pairwise statistics such as latency and bandwidth between nodes. Measurements may also be application-specific, e.g., progress of certain tasks, rate

of requests for particular services, popularity of objects in terms of number of recent hits, etc. Such measurements are of interest to distributed monitoring systems (e.g., Ganglia [33]) as well as systems requiring support for querying distributed resources (e.g., PlanetLab [41] and the Grid [23]).

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

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

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

We have built a distributed querying infrastructure that exploits the optimization opportunities discussed above. The first opportunity can be exploited by *bounded approximate caching* [36] 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 [36]. In this work, we focus on developing efficient and scalable techniques to place, locate, and manage bounded approximate caches across a large network, so that locality, the second opportunity mentioned above, is also exploited in an effective manner.

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

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

We have developed two approaches. The first approach uses a recursive partitioning of the network space to place caches in a static, controlled manner, and is described briefly in Section 3.2. The second approach (described in Section 3.4) uses a *distributed hash table* (DHT) such as [47] to place caches in a scalable, dynamic and decentralized manner. Both approaches are designed to capture various forms of locality in queries to improve performance. We show how to make intelligent caching decisions using a cost/benefit analysis, and we show how to collect statistics necessary for making such decisions with minimum overhead. Using experiments running on ModelNet [56], a scalable Internet emulation environment, we show in Section 3.6 that our solution significantly reduces query costs while incurring low amounts of background traffic; it is also able to exploit localities in the query workload and adapt to

volatility of measurements.

Although we focus on network monitoring and distributed resource querying as motivation for our work, our techniques can be adapted for use by many other interesting applications. In [16], we briefly describe how to generalize the notion of a "query region" from one in the network space to one in a semantic space. For example, a user might create a live bookmark of top ten Internet discussion forums about country music, approximately ranked according to some popularity measure (e.g., total number of posts and/or reads during the past three hours), and have this bookmark refreshed every five minutes using a periodic query. In this case, the query region is "discussion forums about country music," and the popularity measurements of these sites are requested. Generalization would allow our system to select a few nodes to cache all data needed to compute this bookmark, and periodic queries from users with similar bookmarks will be automatically directed to these caches.

3.2 System Overview

3.2.1 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. In [16], we briefly discuss how to handle locality in other spaces.

For a query that simply requests a set of measurements from a region, the result consists of the values of these measurements. Our system allows a query to specify an *error bound* $[-\delta_q^-, \delta_q^+]$; a stale measurement value can be returned in the result as long as the system can guarantee that the "current" measurement value (taking network delay into account) lies within the specified error bound around the value returned. To be more precise, suppose that the current time is t_{curr} and the result contains a measurement value v_{t_0} taken at time t_0. The system guarantees that v_t, the value of the measurement as monitored by its owner at time t, falls within $[v_{t_0} - \delta_q^-, v_{t_0} + \delta_q^+]$ for any time $t \in [t_0, t_{curr} - lag]$, where lag is the maximum network delay from the querying node to the owner of the measurement (under the routing

scheme used by the system). More discussion on the consistency of query results in our system can be found in [16].

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

3.2.2 Bounded approximate caching.

As discussed in Section 3.1, the brute-force approach of contacting each owner to obtain measurement values is unnecessary, expensive, and can cause interference with measurements. Caching is a natural and effective solution but classic caching is unable to bound the error in stale cached values. Instead, we use *bounded approximate caching*, where bounds on cached measurement values are actively maintained by the measurement owners (directly or indirectly).

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

Whenever the measurement value at a cache provider changes, it checks to see if any of its child caches need to be updated with a new value and bound. If yes, the provider notifies the affected child caches. The cache entries at these child caches and the guarantee entry at the provider are updated accordingly. This process continues from each provider to its child caches until we have contacted all the caches that need to be updated. This update of bounded approximate caches is similar to the update dissemination techniques described in [50]. We use a timeout mechanism to handle network failures (see [16] for details).

The choice of bounds is up to the application issuing queries. Tighter bounds provide better accuracy, but may cause more update traffic. There are sophisticated techniques for setting bounds dynamically and adaptively (e.g., [37]); such techniques are largely orthogonal to our contributions. Here, we focus on techniques for *selecting* bounded approximate caches to exploit locality and the tradeoff between query and update traffic, and for *locating* these caches quickly and efficiently to answer queries. These techniques are outlined next.

3.2.3 Selecting and locating caches.

We have developed two approaches to selecting and locating caches in the network. The first is a controlled caching approach and is described in [16]. The idea is to use a coordinate space such as the one proposed by *Global Network Positioning* (*GNP*) [34] for all nodes in the

network, and perform controlled caching based on a hierarchical partitioning of the GNP space. Each owner preselects a number of nodes as its potential caches, such that nearby owners have a good probability of selecting the same node for caching, allowing queries to obtain cached values of measurements in large regions from fewer nodes. The selection scheme also ensures that no single node is responsible for caching too many measurements, and that the caches are denser near the owner and sparser farther away; therefore, queries from nearby nodes get better performance. We show in [16] that this approach does quite well compared to the naive approach of contacting the node responsible for the requested measurements. This approach, however, exploits some but not all types of locality that we would like to exploit and also restricts the amount of caching at any node by design. There is also a concern of scalability because some nodes carry potentially much higher load than other nodes. Nevertheless, because of its simplicity, the GNP-based approach is still viable for small- to medium-sized systems.

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

3.3 GNP-Based Controlled Caching

Partitioning of the GNP space. In addition to its IP address, each node can be identified by its position within the network, described as a set of coordinates in a geometric space such as the one proposed by *Global Network Positioning* (*GNP*) [34]. GNP assigns coordinates to nodes such that their geometric distances in the GNP space approximate their actual network distances.

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 [30]; 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 2 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.

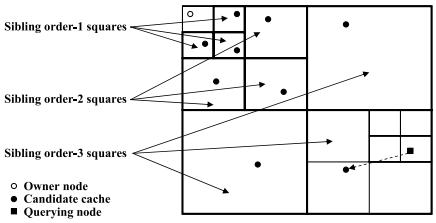

Figure 2: Recursive partition of GNP space.

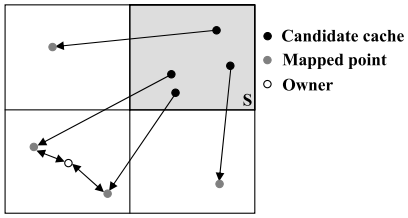

Figure 3: k-nearest mapped cache locator.

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 2, an order-i sibling square of O is an order-i square that belongs to the same order-$(i+1)$ square as O, but does not contain O itself. This scheme ensure that the candidate caches provide reasonable coverage of the entire GNP space, with better coverage closer to the owner.

To select a candidate cache in a sibling square, we use a *cache locator function*. This function takes as input a sibling square and the IP address of the owner, and returns the IP address of the owner's 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. Here, we briefly describe a cache locator function called the *k-nearest mapped cache locator*, which considers locality in query regions. Suppose that we wish to determine the representative for an owner in a sibling square S of a 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. We find the k (a small integer) points that are nearest to the owner, and order them by their distance to the owner. The candidate cache is selected to be the node corresponding to the i-th point, where i is obtained by hashing the owner IP to an integer in $[1, k]$. Since nearby owners may share many of their k-nearest points, there is a good chance that the same candidate cache will be selected.

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

reasons that led us to develop the alternative DHT-based approach (Section 3.4). 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." We omit the details for lack of space.

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

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

Each candidate cache decides on its own whether to cache a measurement and what bound to use. The decision is made using a cost/benefit analysis based on the request and update rates. We omit the details here because a similar (and more complex) analysis used by the DHT-based approach will be covered in detail in Section 3.4.2.

The owner is directly responsible for maintaining all caches of its measurement, using the procedure described in Section 3.2. As also noted in Section 3.2, we use a timeout mechanism to handle failures.

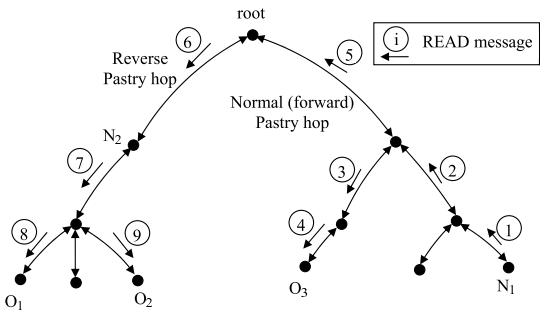

Figure 4: Two-way aggregation with Pastry.

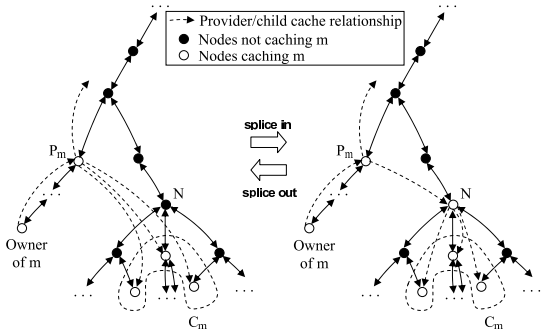

Figure 5: Splicing: add/remove a cache.

3.4 DHT-Based Adaptive Caching

3.4.1 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 network is a *Distributed Hash Table* (*DHT*). As the name implies, a DHT provides a hash table abstraction over the participating nodes. Nodes in a DHT store data items; each data item is identified by a unique key. An overlay routing scheme delivers requests for a key to the node responsible for storing the data item with that key. Routing proceeds in multiple hops and is done without any global knowledge: Each node maintains only a small set of neighbors, and routes messages to the neighbor that is in some sense "closest" to the destination.

Pastry [47] is a popular DHT that takes network proximity into account while routing messages. A number of properties of Pastry are relevant to our system. The *short-hops-first* property, a result of locality-aware routing, says that the expected distance traveled by a message during each successive routing step increases exponentially. The *short-routes* property says that the average distance traveled by a Pastry message is within a small factor of the network distance between the message's source and destination. The *route-convergence* property concerns the distance traveled by two messages sent to the same key before their routes converge. Studies [47] 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.

3.4.2 Overview of caching with Pastry.

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

Suppose that all nodes route towards a randomly selected root using Pastry. The Pastry routes naturally form a tree $\mathcal{T}$ (with bidirectional edges) exhibiting both types of aggregation, as illustrated in Figure 4. Queries first flow up the tree following normal (forward) Pastry routes, and then down to owners following reverse Pastry routes. Nodes along these routes are natural candidates for caches. Our system grows and shrinks the set of caches based on demand, according to a cost/benefit analysis using only locally maintained information. The operational details of our system are presented next. We do not discuss cache updates because the process is similar to that described in Section 3.2 (see [16] for details).

Initialization. A primary objective of the initialization phase is to build the structure $\mathcal{T}$. While Pastry itself already maintains the upward edges (forward Pastry hops), our system still needs to maintain the downward edges (reverse Pastry hops). To this end, every node in $\mathcal{T}$ maintains, for each of its child subtree in $\mathcal{T}$, a representation of the set of nodes found in that subtree, which we call a *subtree filter*. Subtree filters are used to forward messages on reverse Pastry paths, as we will discuss later in connection with querying. Nodes at lower levels can afford to maintain accurate subtree filters because the subtrees are small. Nodes at higher levels, on the other hand, maintain lossy subtree filters implemented with *Bloom filters* [8].

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

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

As a concrete example, Figure 4 shows the flow of `READ` messages when node N_1 queries measurements owned by O_1, O_2, and O_3, assuming that no caching takes place. If node N_2 happens to cache measurements owned by O_1 and O_2, then messages 7 through 9 will be saved.

It is possible to show that our system attempts to route queries towards measurement owners over $\mathcal{T}$ in an optimal manner. 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 3.4.2, 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.

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

Suppose that a node N decides to start caching a particular measurement m. Let P_m denote the first node that can be N's cache provider on the shortest path from N to the owner of m in $\mathcal{T}$. Let $\mathcal{C}_m$ denote the subset of P_m's child caches whose shortest paths to P_m go through N. An example of these nodes is shown in Figure 5. After N caches m, we would like P_m to be responsible for updating N, and N to take over the responsibility of updating $\mathcal{C}_m$, as illustrated in Figure 5 on the right. Note that at the beginning of this process, N does not know what P_m or $\mathcal{C}_m$ is. To initiate the process, N sends a `SPLICE_IN` message over $\mathcal{T}$, along the same path that a `READ` request for m would take. Forwarding of this message stops when it reaches P_m, the first node who can be a cache provider for m. We let each cache provider record the shortest incoming path from each of its child caches; thus, P_m can easily determine the subset $\mathcal{C}_m$ of its child caches by checking whether the recorded shortest paths from them to P_m go through N. Then, P_m removes the guarantee entries and shortest paths for $\mathcal{C}_m$; also, P_m adds N to its guarantee list and records the shortest path from N to P_m. Next, P_m sends back to N a `SPLICE_IN_OK` message containing the current measurement value and timestamp stored at P_m, as well as the removed guarantee entries and shortest paths for $\mathcal{C}_m$. Upon receiving this message, N caches the measurement value, adds the guarantee entries to its guarantee list, and records the shortest paths after truncating their suffixes beginning with N. Finally, N sends out a `SPLICE_IN_OK` message to each node in $\mathcal{C}_m$ to inform it of the change in cache provider. The cache removal procedure uses `SPLICE_OUT` and `SPLICE_OUT_OK` messages. It is similar to cache addition and slightly simpler (see [16] for a detailed description).

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

Caching Decisions. Periodically, the cache controller thread at N wakes up and makes caching decisions. For each measurement m that N has information about, the thread computes the benefit and cost of caching m. We break down the benefit and cost of caching m into four components: (1) $B_{read}(m)$ is the benefit in terms of reduction in read traffic. For each `READ` message received by N requesting m, if m is cached at N, we avoid the cost of forwarding the request for m, which will be picked up eventually by the node that either owns m or caches m, and is the closest such node on the shortest path from N to m's owner in $\mathcal{T}$. Let d_m denote the distance (as measured by the number of hops in $\mathcal{T}$) between N and this node. The larger the distance, the greater the benefit. Thus, $B_{read}(m) \propto d_m \times H_m$, where H_m is the request rate of m at node N. (2) $B_{upd}(m)$ is the net benefit in terms of reduction in update traffic. It's computation requires the maintenance of a large number of parameters; hence we approximate it to be proportional to the reduction in update cost from the cache provider P_m's perspective (see [16] for details). (3) $C_{upd}(m)$ is the cost in terms of resources (processing, storage, and bandwidth) incurred by N for maintaining its child caches for m. (4) $C_{cache}(m)$ is the cost incurred by N for caching m (other than $C_{upd}(m)$). We omit the details of these last three components and refer the interested reader to [16].

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

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

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

Maintaining statistics. We now turn to the problem of maintaining statistics needed for making caching decisions. For measurements currently being cached by N, we can easily maintain all necessary statistics with negligible overhead by piggybacking the statistics on various messages. A more challenging problem is how to maintain statistics for a measurement m that is

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

Fortunately, from the cost/benefit analysis, we observe that a measurement m is worth caching at N only if N sees a lot of read requests for m or there are a number of frequently updated caches that could use N as an intermediary. Hence, we focus on monitoring statistics for these measurements, over each *observation period* of a tunable duration. For example, the request rate H_m is maintained by N for each m requested during the observation period; request rates for unrequested, uncached measurements are assumed to be 0. Our techniques to estimate update rates and d_m over the observation period are more complex. More details on scalable maintenance of statistics are described in [16].

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

3.5 Discussion

Comparison of two caching schemes. The DHT-based adaptive caching approach has a number of advantages over the GNP-based controlled caching approach. First, GNP servers carry potentially much higher load than other nodes in the system. As discussed in Section 3.3, 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. Forward Pastry routing requires only $O(\log n)$ state [47]; 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 Section 3.6.

Third, the GNP-based controlled caching approach restricts the amount of caching at any node by design. While it is reasonable to avoid overloading a node with caching responsibilities, implementing this objective using a static scheme precludes opportunities for certain runtime optimization. For example, suppose that a large region of owners are being queried over and over again. If a node has enough spare capacity, we should let it cache for all owners, so that a query can be answered by contacting this node alone. With the GNP-based approach, it is impossible by design for a large region of owners to select the same cache. In contrast, with the DHT-based approach, a common ancestor of all owners in $\mathcal{T}$ can potentially cache for all of them. Experiments in Section 3.6 confirm this analysis.

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

On alternative definitions of regions. So far, we have been assuming that query regions exhibit locality in terms of network proximity. As mentioned in Section 3.2, 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. We can adapt our techniques to work with an application-defined space and distance metric. For example, in case of the DHT-based approach, we can use a second instance of Pastry to construct another tree $\mathcal{T}_{app}$ over the same of nodes using the application-defined distance metric. To process a query, we first route it upwards in the regular Pastry tree $\mathcal{T}$ constructed based on network proximity, which allows network locality among querying nodes to be exploited. After several hops, we send the query directly to one of the owners being queried. Then, we process the query over $\mathcal{T}_{app}$ as if it originated from this owner, using the exact same procedure described in Section 3.4.2 (except on $\mathcal{T}_{app}$ instead of $\mathcal{T}$), which allows locality among owners in the application-defined space to be exploited.

3.6 Experiments and Results

Implementation. We have implemented both the GNP-based and the DHT-based approaches. For the DHT-based approach, we use the MACEDON [46] 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.

3.6.1 Experimental setup.

We have implemented the GNP- and the DHT-based approaches. We conduct our experiments over ModelNet [56], a scalable and highly accurate Internet emulation environment. We emulate 20,000-node INET [17] topologies with a subset of nodes participating in measurement and querying activities. We report results for subsets with 250 nodes acting as both owners and querying nodes. These nodes are emulated by twenty 2.0GHz Intel Pentium 4 edge emulation nodes running Linux 2.4.27. All traffic passes through a 1.4GHz Pentium III core emulation node running FreeBSD-4.9.

While all results presented here use an emulated network, we have also deployed our system (with around 50 nodes) over PlanetLab [41]. Note that the number of owners and querying nodes in our experiments is not constrained by the system's scalability, but rather by the hardware resources available for deploying it over an emulated network. The advantage of deploying a full system over an emulated network is that it ensures that all costs are captured and we do not inadvertently miss out any important effects or interactions. As future work, we plan to develop a simpler simulation-based evaluation, which would allow us to demonstrate larger experiments at the expense of some realism.

3.6.2 Workloads.

We wish to subject our system to workloads with different characteristics that may represent different application scenarios. To this end, we have designed a workload generator to produce a mix of four basic types of "query groups." The four types of query groups are: (1) *Near-query-near- owner* (*NQNO*): A set of n_q nearby nodes query the same set of n_o owners that are near one another (not necessarily close to the querying nodes). This group should benefit most from caching, since there is locality among both querying nodes and queried owners. (2) *Near-query-far-owner* (*NQFO*): A set of n_q nearby nodes query the same set of n_o owners that are randomly scattered in the network. There is good locality among the querying nodes, but no locality among the queried owners. (3) *Far-query-near-owner* (*FQNO*): A set of n_q distant nodes query the same set of n_o owners that are near one another. This group exhibits good locality among the queried owners, but no locality among the querying nodes. (4) *Far-query-far-owner* (*FQFO*): A set of n_q nodes query the same set of n_o owners; both the querying nodes and the queried owners are randomly scattered. This group should benefit least from caching.

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

In this document, we experiment with synthetic measurements, each generated by a random walk where each step is drawn from a normal distribution with mean 0 and standard deviation σ. If σ is large, bounds on this measurement will be violated more frequently, resulting in higher update cost. Synthetic measurements allow us to experiment with different update char-

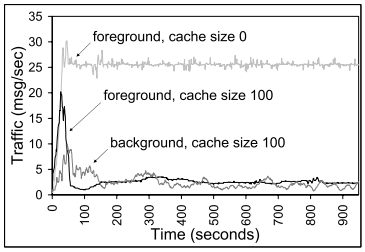

Figure 6: Traffic vs. time.

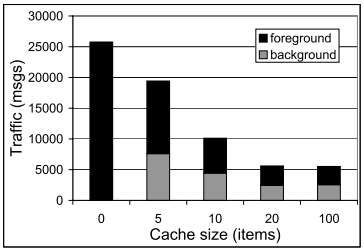

Figure 7: Traffic vs. cache size.

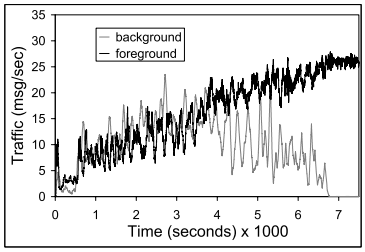

Figure 8: Adapt to volatility.

acteristics easily. Experiments with real node-to-node latency measurements demonstrate the effectiveness of bounded approximate caching, and are presented in [16].

3.7 Results for the DHT-Based Approach

3.7.1 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 experiment, we record both *foreground traffic*, consisting of `READ` and `READ_REPLY` messages, and *background traffic*, consisting of all other messages including splice messages and `CACHE_UPDATE` messages.

Figure 6 shows the behavior of our system over time, with the size of each cache capped at 100 measurements (large enough to capture the working set of W_1). We also show the behavior of the system with caching turned off. The message rate shown on the vertical axes is the average number of messages per second generated by the entire system over the last 16 seconds (same as the period of monitoring queries). From Figure 6, for cache size 100 we see that after a burst of foreground traffic when queries start running, there is an increase in the background traffic as nodes decide to cache measurements. Once caches have been established, the foreground traffic falls dramatically due to the caches. As the set of caches in system stabilizes, the background traffic also reduces to mostly `CACHE_UPDATE` messages. On the other hand, with caching turned off (cache size 0) we see that the foreground traffic remains very high at all times (there is no background traffic). The high foreground traffic outweighs the benefit of having no background traffic. In sum, caching is extremely effective in reducing the overall traffic in the system.

Figure 7 compares the performance of the system under different cache sizes (in terms of the maximum number of measurements allowed in the cache of each node). We show the total number of foreground and background messages generated by the system over the length of the entire experiment (1000 seconds). As the cache size increases, the overall traffic decreases, although the benefit diminishes once the caches have grown large enough to hold the working

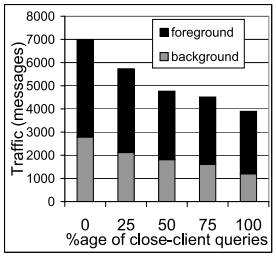

Figure 9: Traffic vs. percentage of queries from nearby nodes.

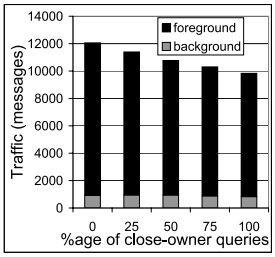

Figure 10: Traffic vs. percentage of queries to nearby owners.

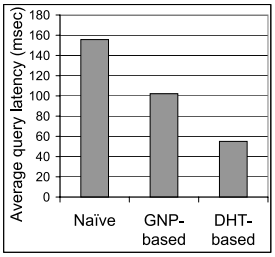

Figure 11: Comparison of average query latency.

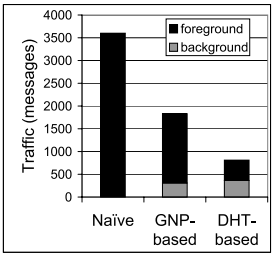

Figure 12: Comparison of total traffic.

set. Another interesting phenomenon is that for very small cache sizes, the background traffic is relatively high because of more splice operations caused by thrashing. Nevertheless, our system is able to handle this situation well.

3.7.2 Adapting to volatility in measurements.

In this experiment, we use the same workload W_1 with cache size 100. We gradually increase the volatility of measurements by increasing the standard deviation σ of the random walk steps every 500 seconds. For the requested query bound of $[-10, 10]$, we effectively increase the update rate from 0.0 to 3.0 updates per second. The result of this experiment is shown in Figure 8. Initially, with a zero update rate, there is no cost to maintaining a cache, so all frequently requested measurements are cached, resulting in low foreground and background traffic. As we increase the update rate, however, the background traffic increases. This increase in cache update cost causes nodes to start dropping cached measurements; as a result, the foreground traffic also increases. Eventually, the update rate becomes so high that it is no longer beneficial to cache any measurements. Thus, the background traffic drops to zero, while the foreground traffic increases to the level when there is no caching (cf. Figure 6). To summarize, our system only performs caching if it leads to an overall reduction in total traffic; consequently, the total traffic in the system never rises above the level without caching. This shows that our system is able to adapt its caching strategy based on the volatility of measurements.

3.7.3 Aggregation effects.

The next two sets of experiments demonstrate that our system can exploit locality in both querying nodes and queried owners. To illustrate aggregation on the querying node side, we perform a series of experiments using five workloads, $[0, 0, 2, 2]$, $[1, 0, 2, 1]$, $[2, 0, 2, 0]$, $[2, 1, 0, 1]$, and $[2, 2, 0, 0]$, where the percentage of queries issued from nearby nodes increases from 0% to 100%. We set $n_q = 5$ and $n_o = 4$ for these five workloads. From the results in Figure 9, we see that the total traffic reduces as the percentage of queries from nearby nodes increases. Figure 10 shows the second set of experiments that illustrate owner-side aggregation by using five workloads where the percentage of queries requesting nearby owners increases from 0% to 100%.

We again see that the total traffic reduces as the percentage of queries requesting nearby owners increases. These experiments show that our system derives performance benefits by exploiting locality both among querying nodes and in query regions.

3.7.4 Comparison with the naive and GNP approaches.

Figure 11 compares the average query latency (as measured by the average time it takes to obtain the requested measurement, after all caches have been created) for a simple workload that exhibits locality among querying nodes. For baseline comparison, we also measure the average query latency of a naive approach, where each querying node simply contacts the owner directly for the measurement. From the figure, we see that the DHT-based approach has the lowest query latency, while the GNP-based approach performs a little worse, but both outperform the naive approach. Figure 12 compares the total network traffic generated by the system while processing a workload in which five querying nodes repeatedly query a faraway set of 12 nearby owners over 480 seconds, using the naive, GNP-based, and DHT-based approaches. Again, the DHT-based approach outperforms the other two approaches as it exploits querying node side locality effectively.

3.8 Conclusions

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, implemented, and evaluated a DHT-based adaptive caching approach and compared it with a GNP-based controlled caching approach. Experiments over a large-scale emulated network show that our caching techniques are very effective in reducing communication costs and query latencies while maintaining the accuracy of query results at an acceptable level. The DHT-based approach is shown to adapt well to different types of workloads. In addition to temporal locality in the query workload, the approach is able to exploit spatial localities in both querying nodes and measurements accessed by region-based queries.

Although the results are promising, techniques described in this work represent only the first steps towards building a powerful distributed network querying system. As future work, we plan to investigate the hybrid approach of combining query shipping and data shipping, and consider more sophisticated caching schemes such as *semantic caching* [19].

4 The Database/Network Interface

4.1 Introduction

As mentioned in Section 2.2, the bulk of the work performed by a publish/subscribe system can be roughly divided into two components: (1) *subscription processing*, the task of matching and processing each incoming publish message with the large set of active subscriptions, and (2) *notification dissemination*, the task of notifying, over a network, those subscribers who are interested in the publish message. Previous work from the database research community has focused on efficient subscription processing; notification dissemination is rarely addressed. Most existing work assumes that a server maintains the entire database state and all subscriptions in the system, and is responsible for computing the set of subscribers affected by each incoming publish message. A straightforward way to notify this set of subscribers is to unicast a notification to each of them in turn. When many subscribers need to be notified, this approach will incur a large amount of outbound traffic from the server, and may easily overwhelm the server and its network links. On the other hand, the networking research community has always focused on efficient notification dissemination. Notable mechanisms include *multicast* [3] and *content-based networking* [10]. Both mechanisms, however, support only *stateless* subscriptions, i.e., those that can be processed by examining the message itself. For multicast, a message's group id encodes its forwarding directions. For content-based networking, the message tuple contains all the information needed to forward the message.

Traditional publish/subscribe systems have simple subscription languages that support only stateless subscriptions. In many situations, however, users may want updates to be further transformed, correlated, and/or aggregated. For example, with a range-aggregate subscription, a user can track the minimum `PER` (price-to-earning ratio, a popular measure of stock quality) of stocks within a risk range. This subscription is *stateful*, because just by looking at a stock update message, the system cannot always tell whether or how the message would affect the subscription. To meet the needs of these users, we are developing a wide-area publish/subscribe system that supports complex subscription definitions. In Section 2, we motivated the challenges in supporting such subscriptions using several examples, and previewed several possible implementation approaches.

The examples showed that efficient support of stateful subscriptions is a challenging problem for wide-area publish/subscribe systems. On one hand, existing network dissemination mechanisms do not support stateful subscriptions directly. While it is possible to relax a stateful subscription into a stateless one and rely on subscribers to perform additional local post-processing, doing so requires unnecessarily large amounts of local subscription state and high volumes of notifications. On the other hand, while the database-centric approach can easily process stateful subscriptions at a server, disseminating notifications over a wide-area network remains difficult because of the inefficiency of unicasts and the difficulty in interfacing the server with advanced network dissemination mechanisms such as multicast and content-based networking.

In this work, we argue that the key to the solution lies in properly *interfacing* the database with the network, in order to combine the processing power of database servers and the dissemination power of the network effectively. In general, there is a wide spectrum of possibilities for interfacing the database with the network and for dividing up work between them. These possibilities provide an interesting set of trade-offs in terms of efficiency, scalability, and manageability of the system. To the best of our knowledge, there is no prior work that investigates this spectrum of database/network interaction models comprehensively. This unified perspective from both databases and networking enables us to identify interesting hybrid solutions that outperform approaches that are either database-centric or network-centric. Specifically, we make the following contributions:

- We explore a number of points along the spectrum of possibilities for interfacing database processing and network dissemination, and study their trade-offs. We show that efficient support of stateless subscriptions in a wide-area publish/subscribe system calls for hybrid solutions with novel database/network interfaces. We demonstrate through experiments with synthetic and real stock datasets that our hybrid solutions offer orders-of-magnitude performance improvement over approaches that are either database-centric or network-centric.

- We formalize *message and subscription reformulation* as a general mechanism for implementing stateful subscriptions using a dissemination network that supports only stateless subscriptions. Reformulation allows us to keep a simple and clean interface between the database and the network, while at the same time providing a comparable or higher level of efficiency compared with much more complex system configurations that require application-specific extensions to routing. We have developed reformulation techniques for a number of stateful subscriptions types including range aggregation/`DISTINCT` and joins.

- For range-min subscriptions, we proposed a reformulation technique based on the concept of Mar (*Maximum Affected Range*). New data structures and group processing algorithms for this technique were developed in [15]. These techniques are also applicable in processing a group of continuous range-min/max queries, which is an interesting problem in its own right.

To recap, the combination of (1) cooperative processing and dissemination by the database and the network, (2) a clean, easy-to-implement database/network interface, and (3) efficient server-side data structures and algorithms together provide an efficient platform for supporting stateful subscriptions over a wide-area network.

The remainder of this section is organized as follows. Section 4.2 discusses various methods for interfacing the database with the network, including database-centric, network-centric, and hybrid approaches. Message/subscriptions reformulation and Mar are introduced in the context of the hybrid approach (Section 4.2.4). We concentrate on describing how to support range-min

subscriptions in this section, and briefly discuss other stateful subscriptions in Section 4.3. Section 4.4 covers additional details of system implementation and presents experimental results. Section 4.5 presents our concluding remarks.

4.2 Spectrum of Server/Network Interfaces

This section explores the spectrum of possibilities for interfacing servers with a network in order to support stateful subscriptions efficiently. We start with a brief discussion of the database-centric approach in Section 4.2.2. Then, Section 4.2.3 discusses the network-centric approach, together with some background on content-based networking and intuition behind how updates affect subscriptions, which are useful also in later discussions. Section 4.2.4 describes one of our main results—a hybrid approach that supports closer cooperation between servers and the network using message/subscription reformulation.

4.2.1 Preliminaries

The publish/subscribe system we are building offers a conceptual (and possibly virtual) *database* over which users can define *subscriptions* as SQL views. *Publish messages* are updates to the database. If a database update causes the content of a subscription view to change, we say that the database update (publish message) *affects* the subscription; in this case, the system needs to send the subscriber a *notification message* containing the change to the content of the subscription view.

To keep our discussion focused, we concentrate in this section on supporting range-min subscriptions. These subscriptions are useful in many situations where users are interesting in tracking the "best" objects in ranges of their interest, e.g., stocks with the lowest price-to-earning ratios within a risk range, or lowest-priced digital cameras with at least 4.0 megapixels. The various server/network interface approaches and the message/subscription reformulation mechanism that we are going to present later are completely general; however, the actual reformulation technique may vary for different subscription types. We discuss how to handle other subscription types in Section 4.3.

Before proceeding, we give a classification of database updates based on how they affect range-min subscriptions. To make our discussion concrete, recall from the examples in Section 2 the database table

$$\texttt{STOCK}\,(\texttt{SYMBOL}, \texttt{RISK}, \texttt{PER}, \ldots)$$

and range-min subscriptions of the form

$$\texttt{SELECT MIN(PER) FROM STOCK WHERE } x_1 \texttt{ <= RISK AND RISK <= } x_2.$$

We call RISK the *range attribute* and PER the *aggregation attribute*. To simplify discussion in this section, we further restrict ourselves to updates of the aggregation attribute (PER) only.

Insertions, deletions, and updates to other attributes require fairly straightforward extensions, details of which are presented in [15]. Let $\Delta(t : x, y_o \rightarrow y_n)$ denote an update of a stock t (with risk x) that changes PER from y_o to y_n. This update falls into one of the following categories:

- *Ignorable updates.* These are updates that, given the current state of the database, cannot possibly affect any subscriptions. In our running example, $\Delta(t : x, y_o \rightarrow y_n)$ is ignorable if there exists another stock t' with the same risk x and a PER no higher than both y_o and y_n. For example, the update of t_7 in Figure 1 is ignorable because of t_6.

- *Non-ignorable updates.* These are updates that may affect some subscription (i.e. they are not ignorable). They are further classified into two types:

  - *Bad updates.* These are non-ignorable updates whose effects on affected subscriptions cannot be determined from the content of the update itself; additional information from the database is required. In our running example, a non-ignorable update $\Delta(t : x, y_o \rightarrow y_n)$ is bad if it might "expose" a new minimum PER in some risk range. The update of t_5 in Figure 1 is an example of a bad update because it exposes both t_3 and t_6. The effect of a bad update cannot be inferred from the content of the update alone; additional information from the database is required.
  - *Good updates.* A good update is a non-ignorable update that is not bad, i.e., its effect on any affected subscription can be determined from the content of the update itself. In our running example, a non-ignorable update $\Delta(t : x, y_o \rightarrow y_n)$ is good if no other minimum PER is exposed due to that update. The update of t_4 in Figure 1 is an example of a good update.

To recap, a decreasing update can be ignorable or good, whereas an increasing update can be ignorable, bad, or occasionally good. Note that this classification scheme does not take into account what subscriptions are currently in the system. Such information can be exploited for more efficiency (e.g., if there are no subscriptions, all updates are effectively ignorable), but doing so also incurs some extra overhead; we discuss this point further in Section 4.2.4.

4.2.2 Database-Centric Approaches

In this set of approaches, we follow the traditional database-centric view of publish/subscribe—of first computing the updates to each subscription, and then disseminating these updates. We assume that a single server maintains the database state and keeps track of all subscriptions. For each publish message, we can efficiently compute all subscription updates in time sublinear in the size of the database and the number of subscriptions, using the group-processing techniques presented in [15]) The approaches below differ mainly in how subscription updates are disseminated.

S-UN: Server with Unicast Network With this approach, which we call S-UN, the server unicasts a subscription update message to each affected subscription. For our running example, the message has a constant size, and simply contains the new minimum PER for the subscription. The problem with this approach is that when many subscriptions are affected, there will be a large amount of traffic overall, and the server can easily become a bottleneck of dissemination.

If multiple affected subscriptions are hosted by the same node in the network, an additional optimization is for the server to combine multiple messages into one. This technique, which we call *message aggregation*, reduces the number of messages. However, the size of a combined massage would no longer be constant; instead, it becomes linear in the number of affected subscriptions at the node. The reason is that the message needs to list the affected subscriptions (because not all subscriptions at the node may be affected) and possibly multiple subscription updates (because different subscriptions may be affected differently by the same database update, as shown in the example in Section 2.1).

S-MN: Server with Multicast Network Multicast [3] is an efficient mechanism for disseminating messages to a group of network destinations. Ideally, we would define a multicast group for each subset of the subscriptions. After the server computes all subscription updates, it checks to see which subset of the affected subscriptions share the same update message, and sends out this message to the multicast group consisting precisely of these subscriptions. However, both IP multicast [3] and application-level multicast [11] techniques do not handle the need for large number of groups (up to 2^M for M subscriptions).

In this work we resort to *hierarchical application-level multicast*. This method builds a tree rooted at the server spanning all nodes hosting subscriptions, with a moderate fan-out c. Each non-leaf node and its children together form a multicast network with 2^c multicast groups, each supporting efficient application-level multicast from the node to a subset of its children. Thus, this method avoids the problem of having too many multicast groups by breaking down the dissemination task into a hierarchy of much smaller multicasts with group number capped at 2^c. The cost of doing so is that the update message sent out by the server must list the set of affected subscriptions; otherwise, a non-leaf node would not be able to tell which children to forward the message to.

We call the above approach S-MN. The problems with using multicast for publish/subscribe (large number of groups and large message size) have also been identified by other work [6, 38, 40]. Possible solutions are: (1) Reduce number of groups [38] by approximating group membership in which case post-processing and additional state are needed at subscribers. We have considered some of these techniques, but the details are beyond the scope of this document. (2) Use compact, "semantic" descriptions of affected subscriptions [6, 40] to avoid large update messages. This approach gives a content-based network (that still does not handle stateful subscriptions), which we consider next.

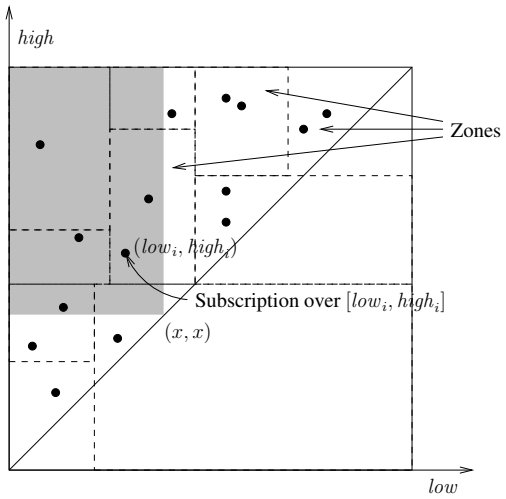

Figure 13: Space of range subscriptions.

4.2.3 Network-Centric Approaches

At the other end of the spectrum, we have approaches that avoid the use of servers altogether by making the network handle as much subscription processing as possible. As discussed in Section 4.1, a natural starting point is a content-based network [10], which supports stateless subscriptions defined as predicates over the content of each message. Information about subscriptions is reflected in the distributed routing state of the network, which allows an update to be forwarded to affected subscriptions without intervention of servers. We now explore how to extend basic content-based networking to support stateful subscriptions.

Background on Content-Based Networking Content-based networks can be implemented using various techniques. Since the section focuses on range and range-min subscriptions, we discuss an implementation based on a *Content Addressable Network* (*CAN*) [44] similar to *Meghdoot* [25]. Meghdoot is designed to support only range subscriptions; we later demonstrate how to extend it in different ways to support range-min subscriptions. At the heart of Meghdoot is a CAN constructed as follows. Each attribute used in range selection (e.g., `RISK` in our running example) is mapped to two dimensions in the CAN space, one for the low end of the range and the other for the high end. A range subscription can then be represented as a point in this space. Figure 13 illustrates a 2-d CAN, where each single-attribute range subscription over $[low_i, high_i]$ is mapped to the point $(low_i, high_i)$. The space is partitioned into rectangular *zones*, each with a *zone owner*—an network node responsible for all the subscriptions in its zone. Each zone has knowledge of only its neighbors and can route messages only to them. Routing is carried out in multiple hops until the destination is reached.

Meghdoot notifies affected range subscriptions as follows. Consider an update to a tuple whose range attribute value is x. The affected range subscriptions are precisely those in the

upper-left quadrant rooted at the point (x, x) (shown as a shaded region in Figure 13). Subscriptions inside this quadrant are affected because their ranges contain x; subscriptions outside this quadrant are unaffected because their ranges do not contain x. Hence, Meghdoot first routes the update message towards the point (x, x) which is called the *event point*; the zone owner of that point then forwards this message to the neighboring zone owners in the affected quadrant, which in turn forward the message upward and leftward to their neighbors, and so on. CAN provides a convenient substrate to implement this forwarding algorithm. Further details of this algorithm can be found in [25].

CN: Serverless Content-Based Network The straightforward way to support stateful subscriptions using a content-based network, as already discussed in the example in Section 2.2, is to "relax" them into stateless subscriptions directly supported by the network. We call this approach CN. For our running example, Meghdoot can be used to support the stateless `RISK`-range subscriptions relaxed from the stateful min-`PER`-over-`RISK`-range subscriptions. Each subscriber locally maintains the content of the stateless subscription queries, and uses post-processing to derive updates to the stateful subscriptions.

The advantage of CN is its simplicity: We only need to extend the capability of the subscribers; the network substrate remains unchanged. The system does not require a central server, thereby removing a potential bottleneck. The disadvantages of CN are obvious too. All updates within the `RISK` range are sent to the subscriber, even though most of them may be ignorable in practice. Also, to cope with bad updates, each subscriber must maintain all stocks within its `RISK` range, which is rather costly.

CN$^+$: CN with Additional Routing Logic We can improve the efficiency of CN by exploiting additional information in the database state maintained for each subscription. The key observation regarding range-min subscriptions is the following (recall the notation from the beginning of Section 4.2):

> **(Subsumption property)** *If an aggregate attribute update $\Delta(t : x, y_o \rightarrow y_n)$ does not affect a range-min subscription with range $[x_1, x_2] \ni x$, then the update cannot affect any range-min subscription with range $[x'_1, x'_2] \supseteq [x_1, x_2]$.*

This observation allows us to cut off forwarding of update messages early: As soon as we hit an unaffected subscription corresponding to point (x_1, x_2) in the CAN space, we can exclude the upper-left quadrant rooted at (x_1, x_2) from further forwarding. It is easy to verify the correctness of this observation. The minimum value of the aggregate attribute in the larger range $[x'_1, x'_2]$ is the minimum among the following three quantities: (1) the minimum in $[x'_1, x_1)$, (2) the minimum in $[x_1, x_2]$, and (3) the minimum in $(x_2, x'_2]$. Quantities (1) and (2) cannot change because the update falls in $[x_1, x_2]$. Thus, if the minimum in $[x_1, x_2]$ is not affected, the minimum in the larger range cannot be affected either.

We can stop forwarding even more effectively with the following observation, which is stronger and slightly more subtle:

> **(Cutoff property)** *Suppose that update* $\Delta(t : x, y_o \rightarrow y_n)$ *does not affect a range-min subscription with range* $[x_1, x_2] \ni x$ *because of another tuple* t' *with range attribute value* $x' \in [x_1, x_2]$ *and aggregate attribute* $y' \leq \min(y_o, y_n)$. *Then the update cannot affect any range-min subscription with range* $[x'_1, x'_2] \supseteq [\min(x, x'), \max(x, x')]$.

For intuition, consider the example in Figure 1. Update of t_4 does not affect subscription s_2 because of stock t_2. The presence of t_2 "protects" any range-min subscription whose range includes both t_4 and t_2 (e.g., s_1) from being affected by the update of t_4. This property allows us to cut off forwarding of update messages early, instead of forwarding them all the way towards the upper-left corner of the CAN space as is done in CN.

We are now ready to present the CN$^+$ approach using our running example. For each range-min subscription, CN$^+$ maintains a stock tuple with the minimum PER in subscription's RISK range. PER updates (assuming for now that they are ignorable or decreasing) are handled as follows. As in Meghdoot, a PER update to stock with RISK x is first routed to point (x, x) in the 2-d CAN space. The zone owner of this point tags the message with a *cutoff point* (c_1, c_2), initially set to $(-\infty, \infty)$. This message is routed upward and leftward to the zone owners in the upper-left quadrant rooted at (x, x), just as in Meghdoot, but with several differences. First, the message is not forwarded outside the rectangle spanned by (x, x) and the cutoff point. A zone owner does not need to process subscriptions outside the same rectangle because they cannot be affected. For each subscription inside the rectangle, we check its currently maintained minimum PER to see if it is affected. If yes, it is updated. Otherwise, we refine the cutoff point in the message based on the RISK value x' of the stock with the minimum PER: If $x' < x$, we raise c_1 to x'; if $x' > x$, we lower c_2 to x'. The cutoff property discussed earlier is the basis for this refinement.

It is possible for CN$^+$ to handle non-ignorable increasing PER updates with reasonable efficiency, but the details are very messy and we omit them here. On a high level, we distribute the entire database state along the diagonal of the 2-d CAN space, which supports computation of any new minima exposed by bad updates. We have also developed optimizations for sharing this computation among multiple affected subscriptions.

CN$^+$ has a big performance advantage over CN. An ignorable update is detected and stopped very early, as it will not be forwarded beyond the subscription with the smallest range containing it. CN$^+$ also attempts to cut off forwarding of non-ignorable updates as early as possible. The disadvantage of CN$^+$ is its complexity. CN$^+$ pushes a significant amount of application-specific routing logic into the content-based network layer, and the specialized routing algorithms require access to additional state including the contents of subscriptions and the database. The resulting system is difficult to implement and maintain because of the lack of a clean interface separating the network from the database.

4.2.4 Hybrid Approaches

Our goal in this section is to develop techniques that offer the same or higher level of efficiency as CN$^+$, but without complicating the network substrate with application-specific routing algorithms. To achieve this goal, we need to rethink the traditional responsibilities of servers in a publish/subscribe system, and divide the work carefully between servers and the network. We seek to maximally exploit the capability of a content-based network within the confines of its standard interface. Recall that a content-based network supports subscriptions defined as predicates over the content of each message. In this section, we show that with our message/subscription reformulation techniques, we can support stateful range-min subscriptions efficiently using stateless subscriptions of the form "the data rectangle in the message contains the point of interest." Such subscriptions are a standard feature in most content-based networks, e.g., [10]; in particular, we show that Meghdoot can handle these subscriptions very efficiently with minimal extension. While this section focuses on range-min subscriptions, we note that message/subscription reformulation is a general mechanism; reformulation techniques for other types of subscriptions will be discussed in Section 4.3.

S-CN: Server with Content-Based Network Under this approach, which we term S-CN, a central server maintains the database state and is responsible for generating notification messages and injecting them into a content-based network for dissemination. Interestingly, the server does not need to know the set of subscriptions, which makes S-CN particularly attractive when subscriber anonymity is desired, or when it is expensive for a server to maintain a large, dynamic set of subscribers.

Message/Subscription Reformulation The key idea is for the server to reformulate each publish message into zero or more notification messages whose contents carry additional information derived from the current database state. This additional information effectively removes the dependency of stateful subscriptions on the database state. When stateful subscriptions register with the content-based network, they are first reformulated into stateless subscriptions (without any knowledge of the database state) to work with the reformulated notification message format.

To illustrate the general reformulation mechanism, let us consider a very naive reformulation technique as a warm-up exercise. The server can simply embed the entire database state into each notification message. Doing so obviously makes all stateful subscriptions stateless, but it incurs too much overhead, and may exceed the capability of most content-based networks as they may not support full SQL queries over the message content. How to do better than this naive technique requires non-trivial understanding of different subscription types.

Mar-Based Reformulation It turns out that for range-min subscriptions, there exists an efficient and effective reformulation based on Mar (for *Maximum Affected Range*), which intu-

itively captures an update's "extent of influence" on range-min subscriptions. Informally, using our running example, the Mar of a stock t is the maximum RISK range in which t has the minimum PER and is the only stock with this PER. We formally define Mar below, where a point (x, y) represents a tuple with range attribute value x and aggregate attribute value y:

Definition 1 (Maximum affected range) $\mathsf{Mar}(x_0, y_0)$*, the* Mar *of point* (x_0, y_0) *with respected to a set of distinct points* P*, is the maximum range* $(x_l, x_r) \ni x_0$ *for which there exists no point* $(x, y) \in P$ *such that* $x_l < x < x_r$ *and* $y \leq y_0$*. Let* $\mathsf{Mar}(x_0, y_0) = \varnothing$ *if no such range exists; i.e.,* $\exists (x_0, y) \in P : y \leq y_0$.

The Mar *of an update* $\delta = \Delta(t : x, y_o \rightarrow y_n)$*, denoted* $\mathsf{Mar}(\delta)$*, is the union of* $\mathsf{Mar}(x, y_o)$ *and* $\mathsf{Mar}(x, y_n)$*, both of which are defined with respect to the set of points representing all tuples in the relation other than* t.

For example, Figure 14 shows $\mathsf{Mar}(x_0, y_0)$ with respect to a set of points (shown as solid black dots). Basically, $\mathsf{Mar}(x_0, y_0)$ is an open interval between two points: the first one to the left of x_0 and the first one to right of x_0, both with height less than or equal to y_0. As another example, in Figure 1, the Mar of the t_5 update is the range $(20, 100)$. We show in [15] how to compute Mar efficiently (in time logarithmic in the size of the database). The following results establish the utility of Mar in range-min subscription processing:

Theorem 1 *A range-min subscription with range* $[x_1, x_2]$ *is affected by an update* δ *if and only if* $x \in [x_1, x_2] \subseteq \mathsf{Mar}(\delta)$.

Corollary 1 (Update classification) *Consider an update* $\Delta(t : x, y_o \rightarrow y_n)$*. (1) If* $\mathsf{Mar}(t) = \varnothing$*, the update is ignorable. (2) If* $\mathsf{Mar}(t) \neq \varnothing$*, and* $\mathsf{Mar}(x, y_o) \subseteq \mathsf{Mar}(x, y_n)$ *(with respect to the set of points representing all tuples other than* t*), then the update is good, and the new minimum for any affected range-min subscription is* y_n*. (3) Otherwise, the update is bad.*

Corollary 1 provides the tests for the server in S-CN to run in order to classify each incoming database update. Furthermore, this corollary leads immediately to the following reformulation techniques:

- **(Message format)** Each database update is reformulated into zero or more notification messages of the form
$$\langle \texttt{NEW\_MIN}, \texttt{INNER\_L}, \texttt{INNER\_R}, \texttt{OUTER\_L}, \texttt{OUTER\_R} \rangle$$
and injected into the network.

- **(Subscriptions)** Each range-min subscription over range $[x_1, x_2]$ is reformulated into a predicate
$$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R})$$
over the notification message. Upon receiving a message matching the reformulated predicate, a subscriber simply updates the minimum to NEW_MIN.

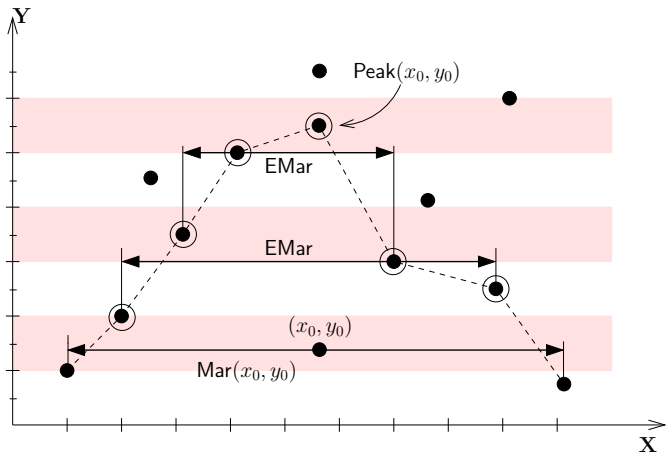

Figure 14: Mar and Hull.

- **(Ignorable updates)** They are simply discarded by the server.
- **(Good updates)** Each good update $\Delta(t : x, y_o \rightarrow y_n)$ is reformulated as $\langle y_n, x, x, x_1, x_2 \rangle$, where $(x_1, x_2) = \mathsf{Mar}(x, y_n)$, computed with respect to the set of points representing all tuples other than t. For example, the good update $\Delta(t_4 : 40, 12 \rightarrow 4)$ in Figure 1 is reformulated as $\langle 4, 40, 40, 20, 100 \rangle$.

Reformulating Bad Updates As we have seen in the example from Section 2.1, a bad update (such as the rise of t_5's PER in Figure 1) is tough to handle because it "exposes" different new minima for different subscriptions. Interestingly, with the help of Mar and the concept of *upper hull* introduced below, we can capture all effects of a bad update on affected subscriptions succinctly and precisely, and in a way that allows the server in S-CN to encode them in the same format as the reformulated notification messages for good updates.

Definition 2 (Upper hull) *Consider point (x_0, y_0) and a set P of points. Suppose $[x_l, x_r] = \mathsf{Mar}(x_0, y_0) \neq \varnothing$. $\mathsf{Hull}(x_0, y_0)$, the* upper hull *of point (x_0, y_0) with respect to P, is the set of points consisting of the following:*

- *The* peak*, denoted $\mathsf{Peak}(x_0, y_0)$, is the point $(x_0, y) \in P$ where y is the smallest possible. Let the peak be (x_0, ∞) if no such point exists, i.e., P has no point with X-coordinate of x_0.*
- *The* left upper hull*, denoted $\mathsf{LHull}(x_0, y_0)$, is the set of all points $(x', y') \in P$ where $x_l < x' < x_0$, and there exists no other point $(x, y) \in P$ such that $(x' \leq x < x_0) \wedge (y \leq y')$.*
- *The* right upper hull*, denoted $\mathsf{RHull}(x_0, y_0)$, is the set of all points $(x', y') \in P$ where $x_0 < x' < x_r$, and there exists no other point $(x, y) \in P$ such that $(x_0 < x \leq x') \wedge (y \leq y')$.*

For example, Figure 14 circles the points in $\mathsf{Hull}(x_0, y_0)$. Basically, $\mathsf{Hull}(x_0, y_0)$ consists of the two "skylines" [39] that we observe by looking towards left and right from (x_0, y_0). As it turns out, each point $(x', y') \in \mathsf{Hull}(x_0, y_0)$ corresponds to a new minimum that would be exposed by the removal of (x_0, y_0). Intuitively, using $\mathsf{Mar}(x_0, y_0)$, we can capture the set of subscriptions that will have y' as their new minimum. This observation is formalized by the following theorem:

Theorem 2 *Consider a bad update $\Delta(t : x, y_o \rightarrow y_n)$. Let P be the set of points representing the set of tuples after the update has been applied. A range-min subscription with range $[x_1, x_2]$ is affected by the update if and only if there exists a point $(x', y') \in \mathsf{Hull}(x, y_o)$ (with respect to P) such that $[\min(x, x'), \max(x, x')] \subseteq [x_1, x_2] \subseteq \mathsf{EMar}(x', y')$, where $\mathsf{EMar}(x', y')$ is the* Exposed Maximum Affected Range *of (x', y') with respect to $P - \{(x', y')\}$. Furthermore, the new minimum for this affected subscription is y'.*

Note that we have used EMar instead of Mar in the above theorem. The two concept are identical except the special case where two points in P have the same Y-coordinate. The difference between EMar and Mar is rather minor and does not affect the exposition in this section. Theorem 2 provides the basis for the following technique for reformulating bad updates:

- **(Bad updates)** Given a bad update $\Delta(t : x, y_o \rightarrow y_n)$, for each point $(x', y') \in \mathsf{Hull}(x, y_o)$, the server generates a notification message $\langle y', \min(x, x'), \max(x, x'), x_1, x_2\rangle$, where $(x_1, x_2) = \mathsf{EMar}(x', y')$ (see Theorem 2 for what point sets Hull and EMar are computed with respect to).

Both the notification message format and the behavior of reformulated subscriptions are consistent with those for good updates. The only difference is that the server generates more than one notification message per bad update. As an example, the bad update $\Delta(t_5 : 50, 5 \rightarrow 9)$ in Figure 1 is reformulated as 3 notification messages: $\langle 9, 50, 50, 30, 70\rangle$, $\langle 8, 30, 50, 20, 70\rangle$, and $\langle 6, 50, 70, 20, 100\rangle$.

Disseminating Reformulated Messages Recall that S-CN reformulate a range-min subscription over range $[x_1, x_2]$ into the following predicate over reformulated notification messages:

$$(\texttt{OUTER\_L} < x_1 \leq \texttt{INNER\_L}) \wedge (\texttt{INNER\_R} \leq x_2 < \texttt{OUTER\_R}).$$

We now illustrate how S-CN can disseminate messages to such subscriptions efficiently using Meghdoot with minimal extension. As in Section 4.2.3, we can picture each subscription as a point (x_1, x_2) in a 2-d CAN space. Each reformulated notification message can be seen as specifying two opposing corners $(\texttt{INNER\_L}, \texttt{INNER\_R})$ and $(\texttt{OUTER\_L}, \texttt{OUTER\_R})$ of a rectangle in this space (as shown in Figure 15). This message matches precisely those subscriptions that fall within the rectangle. Meghdoot already knows how to disseminate a message from a point along the diagonal of the CAN space to its upper-left quadrant. To support dissemination to a rectangular region, we simply need to (1) start the Meghdoot forwarding algorithm from the

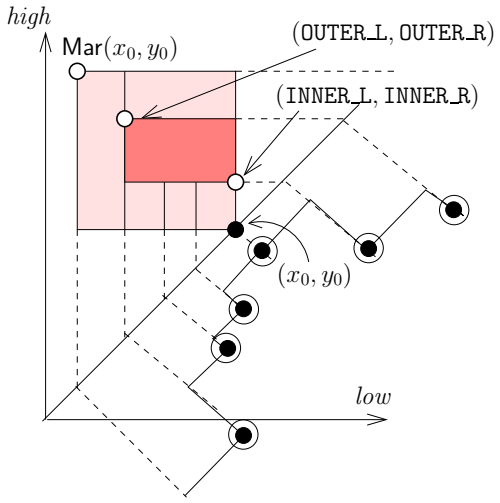

Figure 15: S-CN routing.

lower-right corner (INNER_L, INNER_R), and (2) stop forwarding once the message goes beyond either OUTER_L or OUTER_R. Note that this extension to Meghdoot is needed only because Meghdoot is a very specialized content-based network designed to support only range subscriptions. Many content-based networks (e.g., [10]) allow subscriptions to be general predicates over message content, and therefore should work with S-CN without additional extension.

It is interesting to visualize the messages reformulated from good and bad updates as rectangles in the CAN space. A good update is reformulated into a single rectangle, with its lower-right corner corresponding to an 0-length range containing just the update position, and its upper-left corner corresponding to the Mar of the update. A bad update is reformulated into a collection of non-overlapping rectangles, whose union is a big rectangle spanning the position and Mar of the update (Figure 15).

As noted at the beginning of Section 4.2, we detect ignorable updates without the knowledge of the active subscriptions in the system. Thus, some non-ignorable updates may turn out to be effectively ignorable because certain ranges may not be covered by subscriptions. As a simple optimization, the server in S-CN can maintain the ranges of active subscriptions in the system, and perform a check before injecting a notification message into the network. Doing so would incur extra maintenance overhead; on the other hand, S-CN can still provide some protection of subscriber anonymity, because the server only needs to know the subscription definitions, but not who or where the subscribers are.

DS-CN: Distributing the Server in S-CN We can replace the central server in S-CN with multiple servers that together maintain the database in a distributed manner, resulting in an approach we call DS-CN. The idea is to leverage the network substrate not only for disseminating notifications, but also for distributing the database state. Consider again our running example. Using CAN/Meghdoot as the network substrate, DS-CN maps a stock with RISK x

to a point (x, x) on the diagonal of the CAN space. This stock would be maintained by the zone owner responsible for the corresponding point in the CAN space. In addition, each zone owner along the diagonal maintains pointers to its two immediate neighboring zone owners (left and right) along the diagonal. When a PER update $\delta = \Delta(t : x, y_o \rightarrow y_n)$ enters the system, DS-CN routes it to the zone owner responsible for (x, x). The zone owner then initiates two linear, distributed traversals starting from x: One traversal follows the left zone-owner pointers, scanning stocks in decreasing order of RISK until one (other than t) with PER no greater than $\min(y_o, y_n)$ is reached; the other traversal follows the right zone-owner pointers, scanning stocks in increasing order of RISK using the same stopping condition. When both traversals stop, DS-CN will have examined all stocks in $\mathsf{Mar}(\delta)$, which provide enough information to reformulate the update (detailed omitted). The rest proceeds in the exact same way as S-CN.

The advantage of DS-CN over S-CN is that there is no bottleneck of a central server. Publish messages no longer need to rendezvous at the same server, and reformulated notification messages are now sent out from different servers. Furthermore, Meghdoot's zone-splitting algorithm [25] can be easily adapted to split diagonal zones that hold too many database state, thereby proving effective load balancing. Note that the two uses of the network substrate by DS-CN—for disseminating notifications and for distributing database state—are completely orthogonal and do not need to interfere with each other; in fact, we can use different overlay networks for the two purposes.

4.3 Other Subscription Types

In this section, we briefly discuss how to handle other subscriptions with a hybrid approach such as S-CN, using our general message/subscription reformulation mechanism.

Range-max subscriptions can be handled by the same techniques as range-min. Range-count/sum/average subscriptions are easier to handle: We simply reformulate them into range subscriptions without aggregation; publish messages do not need to be reformulated (though obviously irrelevant updates can be ignored, e.g., those updating neither range nor aggregation attributes). Unlike range-min/max, relaxing these range-aggregation subscriptions would not result in excessive traffic, because relevant updates that fall within a subscription range generally do affect the subscription.

A range-DISTINCT subscription tracks the set of distinct values of an attribute Y for tuples whose range attribute X fall within some range. Simply relaxing this subscription into a range subscription may generate a lot of unnecessary traffic if there are many duplicates. In this case, we can extend the concept of Mar as follows: Mar of an insertion (or deletion) is the maximum X range that contains the insertion (or deletion) point and no other tuples with the same Y value. An insertion (or deletion) is reformulated into a message containing X and Y values and the Mar, if it is not empty. A range-DISTINCT subscription is reformulated into a stateless selection subscription that checks if the subscription range contains the X value and is also contained by the Mar.

We have also extended techniques for 1-d range-aggregation and range-DISTINCT subscriptions to multiple dimensions, where each subscription can specify orthogonal range conditions for multiple attributes. In higher dimensions, Mar is the union of a collection of hypercubes cornered at the update point. Both Mar and Hull become complex shapes in higher dimensions, and reformulated messages no longer have constant size even for good updates (see [14] for details). One way of coping with this complexity is to relax Mar into a simpler bounding region, and use subscriber state to filter false positives.

Finally, select-join subscriptions are stateful as well. Given a publish message that applies an update δR to table R, its effect on subscription $\sigma_p(\sigma_{p_R}R \bowtie \sigma_{p_S}S)$ is $\sigma_p(\sigma_{p_R}\{\delta R\} \bowtie \sigma_{p_S}S)$, which requires accessing state (content of table S) not in the original update message. Following the message/subscription reformulation approach, a server maintaining the database state can reformulate each δR into a series of notification messages, each containing a result tuple in $\{\delta R\} \bowtie S$. Meanwhile, the select-join subscription $\sigma_p(\sigma_{p_R}R \bowtie \sigma_{p_S}S)$ is reformulated into a stateless subscription that checks condition $p \wedge p_R \wedge p_S$ over reformulated messages.

In a publish/subscribe system that uses a single network substrate to support more than one types of subscriptions, we use an additional TYPE field in reformulated messages to distinguish those intended for different types of subscriptions. A reformulated subscription would also include an extra condition that selects notification messages with the appropriate TYPE value.

4.4 Evaluation

4.4.1 Implementation Details

The server module supports well-defined network interfaces to a regular network for unicast and multicast, and CAN for S-CN. On the network side, we have implemented a network simulator for a large-scale publish/subscribe system. The first phase of network simulation generates application-level routing details that are used by a second phase which can accept any topology generated using INET [17], an Internet-like network topology generator. This phase performs a link-level simulation (timing is not simulated) of the network topology. We support a number of dissemination styles, as discussed next.

Unicast from a centralized server. First, the server determines the set of subscribers affected by an update. The network simulator sends a single hop unicast message to each subscriber, carrying the new answer to that subscription. The route follows the shortest path over the underlying IP substrate from the server to the destination.

Multicast from a centralized server. For each exposed answer, the server determines the set of destinations that need to receive a multicast message with that answer.

A message containing this new answer tuple has to be multicast to the set of destinations. The network simulator uses this application-level data to perform multicast as described next. Given a set of N possible destinations, it would take 2^N groups to be able to directly multicast to any subset of recipients. We use application-level multicast in our simulator, as IP multicast is not widely supported and has severe limitations in terms of number of groups. We implement

an efficient hierarchical multicast approach that limits the number of groups that any single node needs to be aware of.

The hierarchical multicast that we have designed and implemented is inspired by NICE [7] and works as follows. Consider a network with N nodes. 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*) [34]. GNP assigns coordinates to nodes such that their geometric distances in the GNP space approximate their actual network distances. We then use a geographic clustering algorithm such as k-means to form a tree of nodes as follows. We define a limit c on the number of children at any node in the tree. Clustering is performed on a set of n nodes to give $min(c, n/c)$ sub-clusters of nodes. Each sub-cluster has a leader, and the leader of the original cluster is the parent of these sub-cluster leaders in the tree. The tree is rooted at the server which acts as the leader of the highest level cluster consisting of all nodes in the network. Each cluster C with leader L_C is further clustered in the same manner and L_C is the parent of the leaders of each of these sub-clusters. Clustering is no longer performed on a cluster if it has no more than c nodes. This gives a tree rooted at the server, with each non-leaf node having at most c children, and the tree has a total of N leaves. Each leader L with k children forms a multicast group for each of the 2^k possible subsets of the k children. It maintains a routing table that provides, for each multicast group, the application-level multicast tree that is constructed rooted at L and reaching all the members of that multicast group. The application-level multicast tree can be constructed in a number of ways - we use a greedy offline algorithm to construct a bottleneck bandwidth tree; this was used in Bullet [29] as a good offline technique to compare with dynamic application-level multicast techniques. Note that the number of group IDs at any node is bounded by 2^c and hence, by limiting c we can ensure that no node needs to be aware of a large number of multicast groups.

When a node L receives a message along with a set S of recipient nodes for that message, it can compute the subset of its children that need to receive the message. All it needs to compute this is a bitmap of all the nodes, formed by a left-to-right ordering of the nodes in the tree under node L. At every level, the nodes in each sub-cluster form a contiguous chunk in the bitmap. By looking at the bit positions that need to be reached in its chunk of the bitmap, a node can determine the subset of its children to which the message needs to be forwarded. It can then look up the multicast group ID for that subset and forward the message along the application-level multicast tree described earlier. This process is repeated until the message reaches all its intended recipients. Note that the message reaching a node N needs to encode the exact set of destination subscribers located below N as this is needed to determine the set of affected children at that node.

CAN routing with range predicates, with Mar**, and with additional state (no server).** We use the Meghdoot simulator [25] in order to evaluate the basic CNapproach with just range predicates (described in Section 4.2.3), without specialized techniques to handle more complicated queries. The simulator is augmented with our link-level simulator for more accurate routing statistics. In order to support the sophisticated techniques used in S-CN (such as Mar)

and DS-CN (such as diagonal two-way routing), we implemented appropriate extensions to the simulator.

4.4.2 Evaluation Metrics

We use both server- and network-side metrics for evaluation. On the server-side, we track processing time, which is measured as the period between the time at which an update arrives at the server and the time at which the server completes generation of all outgoing messages for dissemination. On the network-side, we track, for each event: (1) *Number of overlay message hops*: This is the total number of messages sent between overlay nodes, in order to process and disseminate that event. (2) *Number of IP message hops*: This the number of hops over IP-level links, during dissemination of an event. An overlay hop may traverse a number of IP-level links on its path. (3) *Network traffic*: We define network traffic as the total number of bytes that need to be transferred between overlay nodes during dissemination. (4) *Maximum node stress (MNS)*: We define node stress as the number of overlay messages originating from a node. MNS for an event is the highest node stress among all nodes while processing that event.

4.4.3 Workload

We use normal distributions to derive 1-d range-min subscriptions. To model a *hot* range which interests more subscribers, the position of the subscription interval is normally distributed around center of domain of local selection attribute. The interval length also follows a normal distribution. This subscription workload is used in association with both synthetic and real update workloads.

Our synthetic update workload consists initially of a database that contains between $10,000$ and $100,000$ tuples uniformly distributed in the domain. We generate $200,000$ events (long enough to reach stable measurements), each being an update of the aggregate attribute PER, and collect the measurements of each update. All experimental parameters are summarized in Table 1. We vary one or more of these parameters to perform experiments with varying database size, number of subscriptions, percentage of ignorable updates, and the average number of subscriptions affected.

We update a tuple by increasing or decreasing its output attribute using a random walk model. The step depends on the current value and updates are independent of each other. Each random variable (tuple) in this model is actually an irreducible, finite, and aperiodic Markov chain; hence there exists a stationary distribution for the value of each tuple. Consequently, update statistics such as the number of subscriptions affected by an update will be stationary over time. Our simulation is long enough to ensure convergence. Detailed proof of convergence of our update model is omitted for brevity. To make the workload more realistic, we also introduce a small percentage of *spikes*. A spike is an update where the PER drops suddenly (affecting a large number of subscriptions) and then bounces back to its old value. We also

parameter	value
number of overlay nodes	1000
number of physical nodes	$20,000$
domain of range select attribute	$[0, 10, 000]$
domain of aggregate attribute	$[0, 10, 000]$
number of subscriptions	$100k - 1M$
midpoint of selection range	$N(5000, 1500)$
length of selection range	$N(1000, 1000)$
number of tuples in initial DB	$10k - 100k$
number of simulation events	$200,000$
percentage of spiked events	0.5%

Table 1: Summary of all parameters used in experiments.

use a real update workload based on stock data from Yahoo! Finance [22]. More details are provided in Section 4.4.5.

4.4.4 Experimental setup

We perform detailed link-level simulation of a $20,000$-node INET topology. Of these, 1000 nodes are chosen as the end nodes participating in an overlay network. Events are generated by publishers assumed to be randomly scattered throughout the network.

We assume that publishers are distributed throughout the network. We do not model the message hop from the publisher to the server/event point, because this cost would be incurred in all systems. If the publisher is not a part of the CAN, it could contact some peer as an entry point into the CAN. Similarly, subscriptions could be distributed throughout the network, but each subscription chooses a peer in the overlay network as its gateway to the CAN. We use the following technique to choose a gateway in our experiments. In CAN-based techniques, the zone owner is chosen as the gateway for all subscriptions that map to that zone. The same mapping is used in all the compared approaches. We assume that subscriptions reside at the gateway and do not model the propagation of messages from the gateway to the end subscriber. In case of multicast, the network-side metrics for each group ID at each node in the hierarchy are precomputed to speed up the simulation. In addition, c is fixed at 10 so that no more than 1024 multicast trees need to be stored at any node in the system.

4.4.5 Experiments and results

Demonstration of scalability In this set of experiments, we compare the techniques in terms of their ability to scale to large numbers of tuples and subscriptions. On the server side, we compare the average processing time per update for S-CN, unicast and multicast (recall that CN and DS-CN do not have a central server). On the network side, we compare the average network traffic (bytes) generated per event. All updates are non-ignorable and the percentage of spikes is 0.5%.

db size	$20K$	$40K$	$60K$	$80K$	$100K$
S-CN	2.10	2.15	2.18	2.20	2.22
Unicast	726	671	678	655	654

Table 2: Number of outgoing messages from server, varying database size

# subs.	$200K$	$400K$	$600K$	$800K$	$1M$
S-CN	2.08	2.08	2.08	2.08	2.08
Unicast	303	616	909	1218	1441

Table 3: Number of outgoing messages from server, varying num. of subscriptions

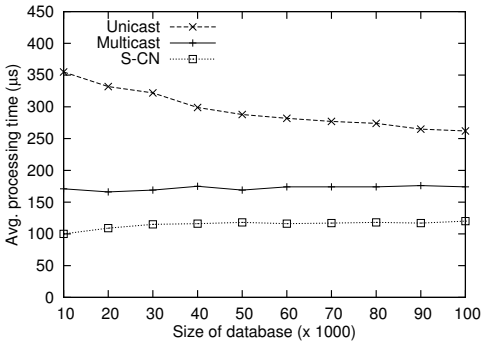

Figure 16: Avg. processing time; increasing database size.

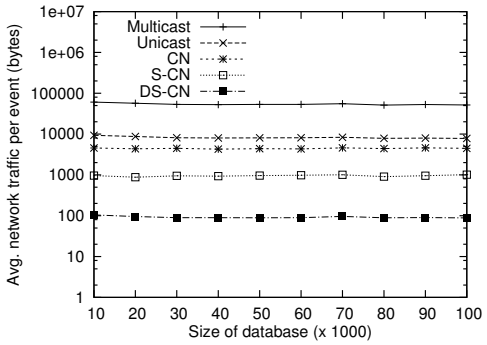

Figure 17: Avg. network traffic; increasing database size.

We first vary the database size from $10,000$ to $100,000$ tuples, for $500,000$ subscriptions. Figures 16 and 17 show the server- and network-side costs respectively. On the network side, CN, S-CN, and DS-CN perform much better than unicast and multicast, and the average network traffic is independent of database size. Note that we have shown network traffic in log scale on the y-axis because of the order of magnitude difference between the various approaches. Among these three approaches, DS-CN performs the best in terms of traffic as the diagonal traversal usually takes very few hops. S-CN is next, the main factor for performing worse than DS-CN is the cost of routing from the server to the event point. This is followed by CN. Among central server approaches, S-CN achieves the lowest processing time, and its processing time increases only by 20% when the database size increases by a factor of 10. The processing time of multicast is roughly 50-70% higher than S-CN due to the cost of identifying all affected

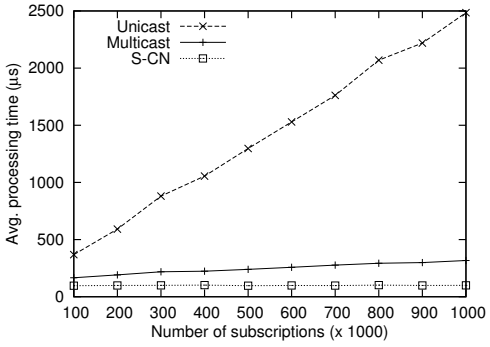

Figure 18: Avg. processing time; increasing number of subscriptions.

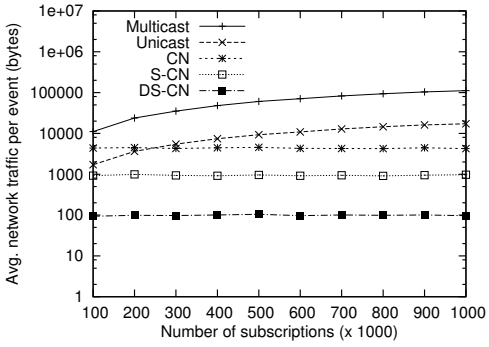

Figure 19: Avg. network traffic; increasing number of subscriptions.

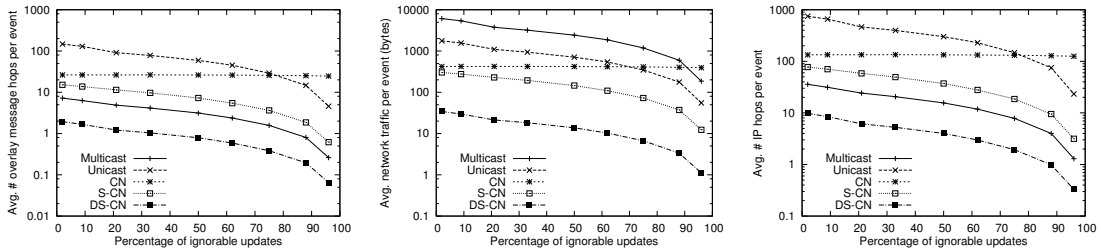

Figure 20: Performance of various approaches, varying the percentage of ignorable updates.

subscriptions. The trend for multicast is flat because the slight increase in processing time over increasing database size is compensated by a decreasing number of affected subscriptions: a larger database gives a subscription more resistance against update. However, the network-side performance of multicast is the worst because the set of affected subscriptions needs to be encoded in each message, even though the number of multicast messages may be small. Unicast performs badly on both server- and network-side due to the high cost of assembling and disseminating outgoing messages for each affected subscription. The processing time of unicast also benefits from a larger database.

Next, we keep the database size at $10,000$ tuples and vary the number of subscriptions from $100,000$ to 1 million. Figures 18 and 19 show scalability on the server and network side respectively. S-CN is completely independent of subscriptions; thus, both its processing time and network traffic is approximately constant. The three approaches of CN, S-CN, and DS-CN perform well with traffic never rising above $1kB$. All these approaches are independent of the number of subscriptions, which makes them scalable. The processing time of multicast increases over number of subscriptions. Unicast also increases but it performs much worse due to the overhead of memory allocation for constructing objects corresponding to outgoing messages (there are many outgoing messages for unicast). Even if we can optimize it using bulk memory allocation, it cannot beat multicast, which is worse than S-CN. On the network side, multicast is good in terms of number of overlay hops (not shown) but it performs badly in terms of total traffic generated. Tables 2 and 3 compare the number of outgoing messages (from server) of S-CN and unicast for the two sets of experiments. In all cases, S-CN saves more than 99% of outgoing messages. We also evaluated CN$^+$ for the portion of workload with decreasing updates. We found that early stopping is effective and results in traffic reduction of more than 98% compared to CN. Early stopping in CN$^+$ (for decreasing updates) was around 2% less effective in terms of traffic than the Mar-based stop conditions in S-CN. However, handling increasing updates in CN$^+$ is complex (as discussed earlier) and would cause more traffic. We do not advocate CN$^+$ because it pushes a significant amount of complex application-specific routing logic into the network layer.

Varying percentage of ignorable updates We next demonstrate the effect of increasing the percentage of ignorable updates I. To better control the parameter, we use a subscription distribution where the midpoint of the selection range is taken from a normal distribution

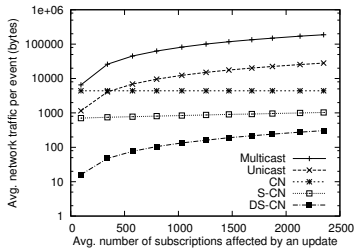

Figure 21: Traffic vs. # affected subs.

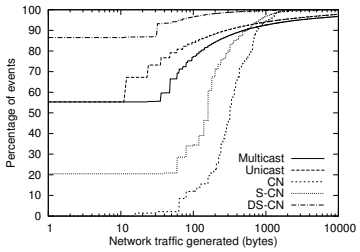

Figure 22: CDF of network traffic.

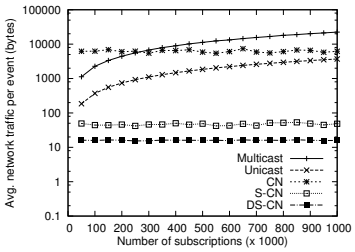

Figure 23: Result of real workload.

$N(5000, 10000)$ and the length is taken from a normal distribution $N(50, 10)$. There are no spikes introduced. Figure 20 shows results for three network metrics. The left figure shows the average number of overlay hops per event, while the middle figure shows the average network traffic (in bytes) per event, for all approaches. The serverless CN approach is independent of I because we cannot truncate ignorable updates. As I is increased, the performance of all the approaches except CN improves and, for high values of I, CN becomes the worst approach. For high values of I, serverless approaches are less desirable as they are unable to detect and truncate ignorable events. However, DS-CN still performs well because in most cases where an update is ignorable, it can be detected at the zone owner containing the event point itself. Overall, the best performance is achieved by DS-CN. Multicast performs quite well in terms of number of overlay messages, with S-CN following up in third position. However, as the middle figure shows, multicast is the worst in terms of total network traffic (due to message size). The right figure shows the average number of IP-level hops per event, for each of the various approaches. The trends are similar to those of overlay hops. In further experiments, we show only the average network traffic (in bytes) for various approaches; the other metrics were found to follow similar trends as described here, in all our experiments. In summary, the single server-based S-CN and the distributed DS-CN perform the best as the messages encode the affected subscriptions very compactly and messages are sent only to those CAN regions which could be affected by the event.

Varying number of affected subscriptions We increase the average number of subscriptions affected by a non-ignorable update by controlling the percentage of spikes which affect a large number of subscriptions. All updates are non-ignorable. The number of subscriptions is $500,000$ and the database size is $50,000$ tuples. Figure 21 shows the performance of each of the approaches in terms of average network traffic (in bytes). From the figure, we see that unicast and multicast worsen in performance linearly with increase in average number of affected subscriptions. CN is independent of this parameter and hence shows a flat line. The network traffic generated by S-CN increases very slightly across the workloads as seen from the figure. This is due to larger average Mar as a result of increasing percentage of spikes. Finally, DS-CN shows an increasing trend; the reason is that since spikes have a large Mar, the diagonal traversal generates more traffic. In real workloads, such updates with large Mar are extremely

Approach	Max. node stress
Unicast	$10,282$
Multicast	3
S-CN	23
DS-CN	25
CN	8

Table 4: Maximum node stress

rare.

Maximum node stress for various approaches We compared the maximum node stress (MNS) for a workload with 21% ignorable updates, $500,000$ subscriptions, and $10,000$ database tuples. There were no spiked events. The highest MNS (across events) for unicast was found to be extremely high $(10,282)$ at the server. From the CDF of MNS over events (not shown), we found that although more than half the events generate no message, a fair percentage of events generate very high node stress at the server. This demonstrates the main disadvantage of unicast: lack of scalability for large-scale subscriptions. The highest MNS of S-CN and DS-CN were found to be just 23 and 25 respectively. In S-CN, 21% of events had 0 MNS, while in DS-CN, nearly 87% of events had 0 MNS. In both approaches, over 99% of events had a MNS less than 10. Multicast had a highest MNS of just 3 because the multicast trees are extremely skinny. Finally, CN had a highest MNS of 8. Although nearly every event has a non-zero MNS in CN, most events have a small MNS, for example, over 94% of events have a MNS of no more than 3.

Distribution of network traffic We plot the CDF (over events) of network traffic for all approaches, in Figure 22. There are $500,000$ subscriptions and $10,000$ database tuples. The percentage of ignorable updates is 21%. From the CDF, we see that S-CN, CN, and DS-CN perform well with average network traffic generated by an event less than $1kB$ for nearly all events. DS-CN is seen to be the most efficient as nearly 87% of events do not generate any traffic. S-CN also performs well but only the ignorable updates generate no traffic (other events need to be routed to the event point). However, nearly 97% of events generate traffic of less than $1kB$. The performance of CN is worse. Around 94% of events generate less than $1kB$ traffic, but more than half the events generate 320 bytes or more traffic. Unicast and multicast perform poorly overall as expected. However, these approaches perform well for events that affect few or no subscriptions: as a result, more than 77% of messages cause network traffic of fewer than 100 bytes. But, the performance degrades rather rapidly for the rest of the events, making these approaches very bad overall. Traffic of upto $680kB$ is seen for some events that affect a large number of subscriptions.

Experiments on a real workload In order to evaluate our techniques on a real trace, we obtained information for 3053 stocks from Yahoo! Finance [22]. We gathered data for earnings per stock (EPS) for each of the stocks. In addition, computed the average recommendation over the past month (RECO) for each stock. RECO varies from 1.0 (strong buy) to 5.0 (strong sell). We collected open and close price data over the course of 60 days, and used EPS to compute PERfor each price. We thus obtained a trace of events, each being an update of PER with RECO constant. The trace had $338,415$ events. 11.7% of the events were non-ignorable events. Note that although this trace has only two events per day, real-time stock prices over the course of the day would follow a similar update trend, with several thousand updates generated every few seconds. This would need efficient database/network coordination to scale to large subscription sets.

We generated $500,000$ subscriptions; each subscription requests the minimum PER over a specified RECO range. This is a meaningful query because stocks with lower PER are intuitively better. Moreover, people may desire stocks rated at different ranges of RECO.

Figure 23 shows the average network traffic per event as we increase the number of subscriptions. We see that S-CN and DS-CN generate orders of magnitude lesser traffic than CN, unicast, and multicast. Both unicast and multicast do not scale well; their performances degrade linearly with increase in number of subscriptions. CN shows constant but bad performance. S-CN and DS-CN perform very well and are both independent of number of subscriptions. They generate less than 100 bytes of network traffic per event on the average, with maximum node stress never rising above 10.

4.5 Conclusions

We approach the construction of large-scale publish/subscribe systems by viewing the problem from the perspective of the interface between the database and the network. Different techniques vary in the degree of database/network cooperation; some are more suitable than others for certain types of queries and/or workloads. The tradeoffs are illustrated by the following table, which compares techniques based on how they handle stateful subscriptions.

Technique	Network side			Server side		Implementation cost
	Traffic	MNS	State (subs)	Processing	State	
S-UN	Very high	High	None	Medium	High	Low
S-MN	Very high	Low	None	Medium	High	Low
CN	High	Low	High	None	None	Medium
CN$^+$	Medium	Low	Medium	None	None	High
S-CN	Low	Low	Low	Low	Low	Medium
DS-CN	Low	Low	Low	None	None	Medium

It is clear that each technique has its strengths and weaknesses. For example, although unicast does not require state at subscriptions, the update traffic is very high. CN$^+$ introduces

application- specific logic into the network and needs per-subscription state. S-CN and DS-CN perform well overall, with dramatic reduction in traffic at low server-side processing cost. We showed that simply converting a stateful subscription to a stateless one does not yield a scalable solution. Our message and subscription reformulation mechanisms are better because they can efficiently embed state information into messages. It is possible for a normal content-based network (which can handle only stateless subscriptions) to handle several classes of stateful subscriptions efficiently: the key is to transform events into a semantic description of affected subscriptions, and subscriptions into a predicate over the semantic description. We demonstrated this using several query classes including range aggregation, distinct, and join. We experimentally validated our techniques for range aggregation, and showed that it is possible to achieve orders-of-magnitude improvement over a naive transformation of the stateful subscription to a stateless one.

5 Rich Notification Conditions

5.1 Introduction

Subscribers in a publish/subscribe system specify what data they are interested in, using subscription queries. However, oftentimes subscribers are interested in receiving notifications only when certain conditions are triggered. Such conditions are referred to as *notification conditions*. Notification conditions are per-subscription state that need to be processed intelligently in order to allow the system to scale to large numbers of subscriptions. Our ongoing work strives to address this problem from the perspective of the interface between the database and the network.

We first briefly describe our proposed notification semantics for publish/subscribe systems in Section 5.2. Then, in Section 5.3, we describe a spectrum of techniques to handle value-based notification conditions.

5.2 Notification Semantics

We augment the subscription language to allow specification of notification conditions, by introducing a `NOTIFY_WHEN` clause. The notification condition is expressed as part of this clause.

Notification conditions can be of various types. Value-based notification conditions are those that request notification when the value of the true subscription result differs by some amount from the result value that was last notified to the subscription. These can be expressed as `RESULT_CHANGED_BY` δ where δ specifies the acceptable difference. For strict semantics, we have to ensure that a subscription receives a notification condition with the current value if and only if the current value differs from the last notified value by at least x. To make this clear, assume that there exists a subscription S and that it was last notified with a result value of x. This means that whenever there is an update U that changes the result to x' such that $x' \geq x + \delta$ or $x' \leq x - \delta$, subscription S needs to be updated with the new result x'. Any other

update should not cause S to be notified. In Section 5.4, we will investigate how to relax these semantics in a disciplined manner in order to achieve higher scalability.

Notification conditions can also be time-based. For example, a subscription may wish to be notified of updates to its result every 30 seconds. We can express such notification conditions as `TIME_CHANGED_BY` δ. Similar techniques apply to such subscriptions, but we do not cover this class of notifications in this document.

5.3 Value-based Notification Techniques

We now discuss a series of techniques to support value-based per-subscription notification conditions. We address the case where all subscriptions have the same filtering condition, but each subscription could potentially have a different notification condition. Let there be N subscriptions in the system, where the i^{th} subscription is defined as $S_i =$ `SELECT PRICE FROM STOCK WHERE` *filter* `NOTIFY_WHEN RESULT_CHANGED_BY` δ_i. Note that each subscription i has the same *filter*, whereas it could potentially have a different notification condition δ_i. We use the term *bound* to refer to the notification condition. Let the maximum possible allowed bound be δ_{limit}. The last notified value for subscription i is denoted by L_i. At the initial state, assume for convenience that all subscriptions have the latest notification v_0. Addition of new subscriptions to the system is investigated along with each approach described below.

5.3.1 Unicast with naive server-based processing

The naive method of processing the subscriptions is to store all the subscriptions at the server, along with L_i and δ_i for each subscription i. When a new update comes in, we scan through the list of subscriptions, and if the new update falls outside $(L_i - \delta_i, L_i + \delta_i)$ for any subscription i, it is added to the list of affected subscriptions, and its last notified value is updated. The server then sends out the new update to all the affected subscriptions via unicast. This technique has a time complexity of $O(N)$.

We can improve the efficiency by introducing an index. Every subscription S_i is indexed into a B-tree twice, first with an index key of $L_i - \delta_i$ and second with an index key of $L_i + \delta_i$. Thus, we index the left and right edges of the value boundaries of every subscriptions. When an update falls outside these boundaries, it needs to be notified. Assume that the last update is v. When a new update v' comes in, we look up v in the index, and traverse the leaves in the direction of v' until we reach v'. All the subscriptions we meet are the ones that are affected and need to be updated with v'. This will cause their left and right boundaries to change, hence they need to be deleted from the index (along with their other edge) and reinserted twice as before. If M is the number of subscriptions affected by an update, the operation has a total time complexity of $O(M \lg N)$ and this work needs to be performed after every update.

5.3.2 Unicast with smart server-based processing with periodic reorganization

We can improve upon the naive processing technique by indexing subscriptions in a different manner. For this purpose, we define two types of trees, clean trees (materialized and virtual) and dirty trees.

We define a *materialized clean tree* T_i as a B-tree in which every indexed subscription has the same last notified value, as of the time of creation. The term materialization refers to the fact that this index structure is physically created by the system. For tree T_i we denote this value by LNV_i. We index a subscription i into the materialized clean tree using δ_i as the key.

In addition, we define one or more *virtual clean trees* ($T_{i,1}, T_{i,2}, \ldots$) which can be regarded as views over the materialized clean tree i.e. they are not physically created. At the time when a new materialized clean tree T_i is created, it is associated with just one default view $T_{i,1}$. A virtual clean tree $T_{i,j}$ is associated with two attributes: (1) $LNV_{i,j}$, which is the last notified value for all subscriptions in that virtual clean tree. (2) $\delta_{i,j}^{min}$ and $\delta_{i,j}^{max}$, which are respectively the smallest and largest bounds of subscriptions indexed within that virtual tree. If an update value falls within the interval $[LNV_{i,j} - \delta_{i,j}^{min}, LNV_{i,j} + \delta_{i,j}^{min}]$, it will not affect any subscriptions indexed in that tree. $\delta_{i,j}^{min}$ and $\delta_{i,j}^{max}$ are maintained only for convenience, for example in determining if a virtual tree is affected by an update, as we shall see later.

A virtual clean tree can be regarded as a tuple, whose attributes help perform three functions: (1) On an incoming update, $\delta_{i,j}^{min}$ helps identify whether the virtual clean tree is *affected* by the update, i.e., whether the corresponding clean materialized tree needs to be looked up (one materialized clean tree may need to be looked up more than once if several corresponding virtual trees are affected by an update. (2) $LNV_{i,j}$ specifies how the tree needs to be looked up. The same materialized virtual tree needs to be looked up differently depending on the attributed of the affected virtual clean tree which initiated the lookup. (3) Once a virtual clean tree is identified and affected subscriptions are looked up, $\delta_{i,j}^{max}$ helps determine if the virtual clean tree needs to be divided to create a new virtual clean tree. These functions are described in greater detail below.

An an example, in the initial condition, we have one materialized clean tree T_1 in the system. Assuming that the latest notification is v and the smallest and largest subscription bound widths currently in the system are respectively δ_{min} and δ_{max}, this materialized clean tree has $LNV_1 = v$. It is associated with only one virtual clean tree $T_{1,1}$ whose attributes are: $LNV_{1,1} = v, \delta_{1,1}^{min} = \delta_{min}, \delta_{1,1}^{max} = \delta_{max}$.

Handling incoming updates. When an update with value v' comes in, we follow a two-phase process of determining the set of affected subscriptions. First, we determine the set of affected virtual clean trees. To determine this, we use the condition that a virtual tree $T_{i,j}$ is affected only if the update falls outside the range $[LNV_{i,j} - \delta_{i,j}^{min}, LNV_{i,j} + \delta_{i,j}^{min}]$. This can be performed efficiently using another index, as discussed later. Second, if a virtual clean tree is determined to be affected, we perform an index lookup for the value $|v' - LNV_{i,j}|$ on

the corresponding materialized clean tree. Traversing the leaves of the materialized clean tree towards the left until we reach $\delta_{i,j}^{min}$ will give us the exact set of subscriptions affected by this update and belonging to virtual tree $T_{i,j}$.

Once the affected subscriptions in a virtual clean tree are identified, we need to update the data structures. Defining virtual clean trees has an advantage, because in the simpler approach of maintaining just the materialized clean tree, we would have to divide the tree into two trees by deleting all the affected subscriptions from the old materialized clean tree and inserting them into a new materialized clean tree. However, in our case, all we need to do is to define a new virtual clean tree and update the attributes of the affected virtual clean tree, which takes constant time. We first determine if the affected virtual clean tree needs to be divided. A virtual clean tree will need to be divided if $|v' - LNV_{i,j}| < \delta_{i,j}^{max}$. If the virtual clean tree needs to be divided, we define a new virtual clean tree $T_{i,j'}$ with attributes $LNV_{i,j'} = LNV_{i,j}$ and $\delta_{i,j'}^{max} = \delta_{i,j}^{max}$. Further, $\delta_{i,j'}^{min}$ is set to the smallest subscription in $T_{i,j'}$ (this is simply the subscription to the right of the first affected subscription encountered during the lookup of $|v' - LNV_{i,j}|$ which we performed earlier). Finally, whether or not a new virtual tree needed to be defined, we update the attribute $LNV_{i,j}$ of the existing virtual clean tree $T_{i,j}$ to v' and $\delta_{i,j}^{max}$ to the bound width of the first affected subscription we met during the traversal to determine affected subscriptions.

Coming back to the example of the initial condition with one virtual clean tree $T_{1,1}$, assume an update v' comes in such that $\delta_{1,1}^{min} \leq |v' - v| < \delta_{1,1}^{max}$. We would define a new virtual clean tree $T_{1,2}$ with attribute $LNV_{1,2} = v$, $\delta_{1,2}^{max} = \delta_{1,1}^{max}$. $\delta_{1,2}^{min}$ is updated appropriately. We would finally update the old virtual clean tree $T_{1,1}$ with $LNV_{1,1} = v'$ ($\delta_{1,1}^{max}$ would also be updated appropriately).

Indexing virtual clean trees. We can quickly determine the set of affected virtual clean trees by indexing their attributes into a B-tree. We use a B-tree to index intervals as follows. A virtual clean tree $T_{i,j}$ is indexed twice with keys $LNV_{i,j} - \delta_{i,j}^{min}$ and $LNV_{i,j} + \delta_{i,j}^{min}$ respectively. When an update v' comes in (assuming the last update was v), we look up v in the B-tree and traverse the leaf nodes in the direction of v' until we reach v'. This will allow us to encounter exactly the set of affected virtual clean trees. An affected virtual clean tree will need to be deleted and reindexed into this B-tree using its new attributes. Note that this index structure is typically expected to be much smaller than the original B-tree containing all subscriptions.

Merging clean trees into a dirty tree. The basic problem with the scheme just outlined is that after k updates, there could be (in the worst case), $O(k^2)$ virtual clean trees in the system. An incoming message could potentially affect all of them, which means that $O(k^2)$ lookups of the materialized clean tree may need to be done (although it can be shown that only $O(k)$, of the affected virtual clean trees, in the worst case, would actually require division). Hence, we introduce a periodic *merge* operation, which takes as input the entire set of virtual clean trees, and merges them to produce a single materialized B-tree, which is called a *dirty tree*.

The distinguishing factor of a consolidated *dirty tree* is that it can contain subscriptions with different last notified values. Hence, we cannot index the subscriptions simply by their δ values. We index a subscription i into a dirty tree twice, with keys $L_i - \delta_i$ and $L_i + \delta_i$. Each key maintains a pointer to its counterpart. There can be at most one dirty tree in the system, as will be explained later.

A dirty tree T_d is associated with the following attributes: (1) LUV_d, which is the last update received by the system. (2) An interval $I_d = [V_{low}, V_{high}]$; if an update falls within this interval, it will not affect any subscriptions indexed in that tree. Suppose that we merge the clean trees after an update v; the resulting dirty tree will have the attribute $LUV = v$, V_{low} would be the minimum $L_i + \delta_i$ over all subscriptions i, V_{high} would be the maximum $L_i - \delta_i$ over all subscriptions i.

Handling updates using a dirty tree. Suppose an update v' comes in which could affect the dirty tree i.e., the update falls outside the interval I_d. We look up v' in the dirty tree, and traverse the leaves towards LUV_d, and we will encounter all the affected subscriptions. Now, instead of reinserting these subscriptions into the dirty tree, we simply delete them and create a new materialized clean tree T_i (this is possible because all these affected subscriptions will now have the same last notified value). The default single virtual tree $T_{i,1}$ over the new materialized clean tree will have attribute $LNV_{i,1} = v'$, while $\delta_{i,1}^{min}$ and $\delta_{i,1}^{max}$ depend on the actual bounds of the affected subscriptions. The dirty tree's attributes are updated as follows: First, I_d is updated based on the set of affected subscriptions. Then, LUV_d is updated to v'. Since the split of a dirty tree does not produce any more dirty trees, there can be at most one dirty tree in the system. Subsequent updates are handled in a similar fashion. The next merge operation would involve all the clean trees as well as the remainder of the dirty tree (the size of the dirty tree monotonically decreases with incoming updates). We propose to develop and use a cost model to determine the appropriate merging frequency.

The advantage of inserting the affected subscriptions into a new materialized clean tree instead of reinserting them into the dirty tree is that handling subsequent updates is very efficient on clean trees. This is because in a clean tree, when subscriptions are affected, they do not need to be deleted and reinserted as explained previously. We just need to define a new virtual clean tree which is a constant time operation. New subscriptions are handled by inserting them into the dirty tree. Alternatively, if there is a newly created materialized clean tree with the last update, we could insert the subscription into that clean tree. Note also that when there are multiple virtual clean trees, there could be more than one virtual clean tree with the same last notified value. These virtual clean trees can be removed from their respective materialized clean trees, and merged into a single materialized clean tree to reduce the number of trees in the system.

5.3.3 S-CN with passive subscriptions in network

Consider how we can support the dissemination of per subscription notification conditions in a stateless content-based network. An update could potentially affect any large subset of subscriptions, so we desire a semantic description of affected subscriptions to avoid sending a unicast message for each affected subscription.

In this section, we make the assumption that subscriptions are not active, i.e. they cannot migrate to different brokers in the content- based network based on their per-subscription state. It would appear that in such a network, it would not be possible for the network to statelessly support the kind of notifications that we have been considering. However, it turns out that we can support such a dissemination as described next.

Assume that we have to index the subscriptions in the content-based network in a static manner. This is done by the following reformulation techniques:

- **(Message format)** Each database update is reformulated into zero or more notification messages of the form
$$\langle \texttt{INTERVAL\_LOW}, \texttt{INTERVAL\_HIGH}, \texttt{UPDATE} \rangle$$
and injected into the network.

- **(Subscriptions)** Each subscription S_i with bound δ_i is reformulated into a predicate
$$(\texttt{INTERVAL\_LOW} < \delta_i \leq \texttt{INTERVAL\_HIGH})$$
over the notification message. Upon receiving a message matching the reformulated predicate, a subscriber simply updates its last notified value (L_i) to UPDATE.

- **(Ignorable updates)** They are updates which have the same value as the previous update. They are simply discarded by the server.

Reformulating other updates. The basic idea is for the server to maintain a set of *intervals*. Intuitively, an interval is a contiguous range of bounds for a particular last notified value (one last notified value could have more than one such contiguous range). An interval I_i is associated with three attributes, (1) the last notified value (LNV_i), (2) the lower edge of the bound (B_{low}), and (3) the upper edge of the bound (B_{high}). We represent an interval by the notation $[LNV_i, B_{low}, B_{high}]$. At the beginning, there is just one interval $[v_0, 0, \delta_{limit}]$ in the set.

Assume that the last update was v. When a new update v' arrives, an interval I_i is affected if v' falls outside the range $[LNV_i - B_{low}, LNV_i + B_{low}]$. For such an affected interval, if v' lies outside the range $[LNV_i - B_{high}, LNV_i + B_{high}]$, it implies that the entire interval is affected and hence this interval's attributes are updated to $[v', B_{low}, B_{high}]$. In addition, a reformulated message $\langle B_{low}, B_{high}, v' \rangle$ is sent out.

On the other hand, if v' lies within the range $[LNV_i - B_{high}, LNV_i + B_{high}]$, that affected interval is actually only partly affected. Hence, it is split into two intervals as follows: the interval from B_{low} to $|v' - LNV_i|$ is affected and is split into a new interval with attributes $[v', B_{low}, |v' - LNV_i|]$. The remaining interval is updated with the new range, and is therefore

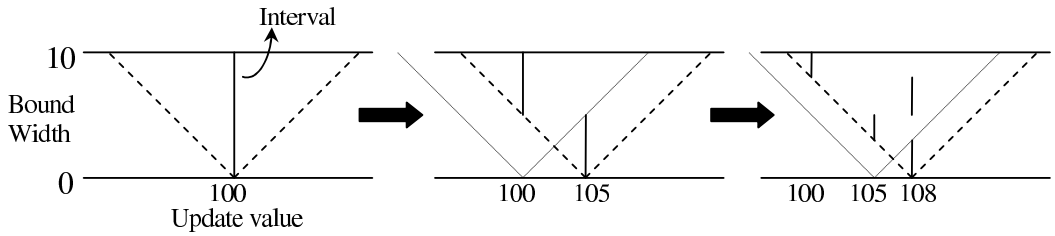

Figure 24: Example interval splits for the sequence of updates $100 \rightarrow 105 \rightarrow 108$.

set with the attributes $[LNV_i, |v' - LNV_i|, B_{high}]$. One reformulated message is sent out: $\langle B_{low}, |v' - LNV_i|, v' \rangle$.

As a concrete example, refer to Figure 24. The x-axis denotes update value, while the y-axis denotes the bounds. The maximum bound is 10 in this example. Initially, the last update is 100 and there is just one interval $[100, 0, 10]$. An update of 105 is shown in the second figure. As seen, the original interval splits into two intervals with attributes $[100, 5, 10]$ and $[105, 0, 5]$ respectively. One reformulated message is sent out: $\langle 0, 5, 105 \rangle$.

A second update of 108 now comes in, and this update happens to split both of the current intervals. This gives rise to a total of four intervals: $[100, 8, 10]$, $[105, 3, 5]$, $[108, 0, 3]$, and $[108, 5, 8]$. Two reformulated messages are sent out: $\langle 0, 3, 108 \rangle$ and $\langle 5, 8, 108 \rangle$.

An update can also cause some intervals to merge. If two intervals have contiguous bound ranges, but are separate because they have different last notified values, and an incoming update affects both of them, they can be merged into a single range after the update. As an extreme case, we define a large update as an update v' such that $|v' - v| \geq \delta_{limit}$, where v is the last update before this one. Such a large update would merge all intervals back into just one, $[0, \delta_{limit}]$.

An interesting feature of this technique is that the server does not need knowledge of subscriptions to maintain the set of intervals, although this knowledge if present could be used to avoid maintaining intervals that contain no subscriptions. Thus, we see that an update can be formulated into a group of disjoint intervals, which can be processed by a stateless content-based network with passive subscriptions. The network could also be implemented specifically for this formulation using Chord-like distributed hash tables or even a content-addressable network (CAN) [44]. Finally, note that insertion of new subscriptions is trivial because we do not depend on subscriptions for defining intervals.

Maintaining the intervals. If we have a large number of intervals, we need to be able to quickly identify affected intervals. We use a B-tree to index intervals as follows. An interval $[LNV_i, B_{low}, B_{high}]$ is indexed twice with keys $LNV_i - B_{low}$ and $LNV_i + B_{low}$ respectively. When an update v' comes in (assuming the last update was v), we look up v in the B-tree and traverse the leaf nodes in the direction of v' until we reach v'. This will allow us to encounter exactly the set of affected intervals. An affected interval will need to be deleted and reindexed into the B-tree using its new attributes.

Managing the number of intervals. The main limitation of this technique is that the number of intervals that need to be sent out grows with the number of updates, and is $O(k^2)$ is the worst case (although we expect it to be closer to linear in practice, with some updates causing a merge of adjacent intervals and large updates shrinking the number back to 1 as described earlier). In the absence of large updates, there are other techniques we can employ to keep the number of intervals bounded. One technique involves relaxing the subscription semantics in a disciplined manner using the concept of tolerances, and is detailed in Section 5.4. Another technique can be applied if subscriptions do not mind rarely receiving unnecessary notifications. In this case, the system can merge small intervals that are adjacent, into a single larger interval, by simply sending out a single reformulated message with the latest notification value to the larger interval. The selection of intervals to merge could be based on statistics to ensure that minimum number of subscriptions receive such unnecessary notifications.

A network with passive subscriptions following the strict notification semantics cannot merge intervals because that would need a knowledge of the last notified value at the subscriptions. As a result, the performance could continue to degrade with updates, in the absence of merging opportunities. The desire for a smaller number of messages to be sent out by the server (per incoming update) brings us to a content-based network with active subscriptions, which we describe next.

5.3.4 S-CN with active subscriptions in network

If we allow subscriptions to be active i.e. they can move between brokers to aid in dissemination, then we can have a simpler interface between the server and the network with just one reformulated message per update. This is however at the expense of potential reorganization traffic for the active subscriptions.

We can organize a content-based network using a content-addressable network (CAN) [44] which is constructed for this purpose. CAN-based dissemination techniques have previously been used for both stateless [25] and stateful [15] subscriptions; here we extend their use to handle per-subscription notification conditions.

We define a two-dimensional CAN space, where a subscription S_i is mapped to the point $(L_i - \delta_i, L_i + \delta_i)$. A point (v, v) along the diagonal corresponds to an *update point* with value v. If the last update is v', it is clear that all subscriptions would lie in a CAN shaped as a right triangle with bottom-right corner (v', v') and the two other corners being $(v' - 2\delta_{limit}, v')$ and $(v', v' + 2\delta_{limit})$. We refer to this triangular region as the *subscription triangle*. Examples of subscription triangles are seen in Figure 25 (the figure is explained in a later paragraph).

Handling updates. The system has knowledge of the last update v, i.e. the bottom-right corner of the subscription triangle. When a new update v' enters the system, it effectively slides the subscription triangle to now rest on the point (v', v'). The subscriptions that lie outside the new subscription triangle are the ones that are affected by this update. The set of affected

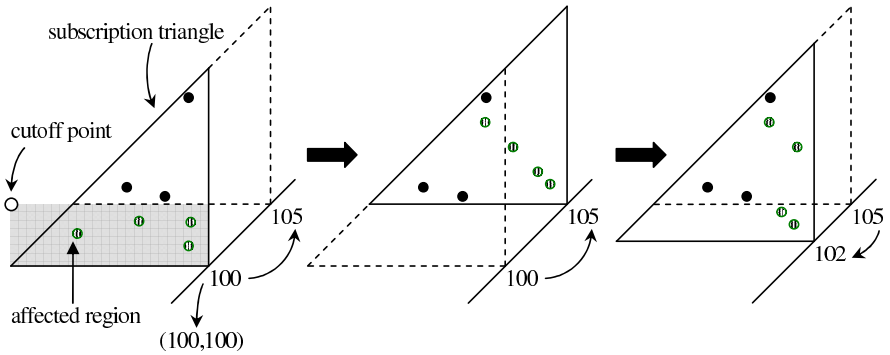

Figure 25: Example subscription movement on notification $100 \rightarrow 105 \rightarrow 102$.

subscriptions on an update can succinctly be described by the server using a single message with the update and tagged with a cutoff point [15] $(v - 2\delta_{limit}, v')$. The rectangle spanning (v, v) and the cutoff point contains all the affected subscriptions. Hence, we adopt the following simple routing scheme to reach all affected subscriptions. First, the update is routed to the point (v, v). Secondly, the update is propagated to the upper- left of this point in CAN space, with the routing being terminated beyond the cutoff point. Publish/subscribe systems such as Meghdoot [25] can easily be extended to support cutoff points. Note that the rectangular routing region defined by the cutoff point is actually slightly larger and simpler than the actual affected region (see Figure 25), but the extra space covered will not have any subscriptions. All subscriptions mapping within the routing region are notified with the update v' and these subscriptions are moved to their new locations, which is simply $(v' - \delta_i, v' + \delta_i)$ for subscription S_i. All these affected subscriptions move to the altitude (which passes through (v', v')) of the new subscription triangle (because their bounds are now recentered around their new updated value).

As a concrete example, refer to Figure 25. On the left, we show a sample CAN with the latest update being of value 100, and $\delta_{limit} = 8$. Zone divisions and axes are not shown for clarity. Subscriptions are indicated by dots in the figure. When an update of 105 comes in, all subscriptions in the shaded region are affected (the affected subscriptions are patterned). The 4 affected subscriptions move to the altitude of the new subscription triangle, as seen from the center of the figure. Now, suppose an update 102 comes in. Only 2 subscriptions are affected, and these are moved to their new location in the new triangle as seen in the figure to the right.

The above description is analogous to the server side technique of maintaining a dirty tree such that every time an update comes in, all the affected subscriptions are reinserted into the tree immediately. Similarly, every time an update comes in, although the affected subscriptions can easily be located by a simple reformulated message, they all have to be moved to their new zone owners.

We can improve performance by leveraging the following observation. All the subscriptions affected by an update have the same last notified value (after the update). Let us examine a scheme where we insert them into a new CAN (called C_{new}) instead of the main CAN. C_{new} has

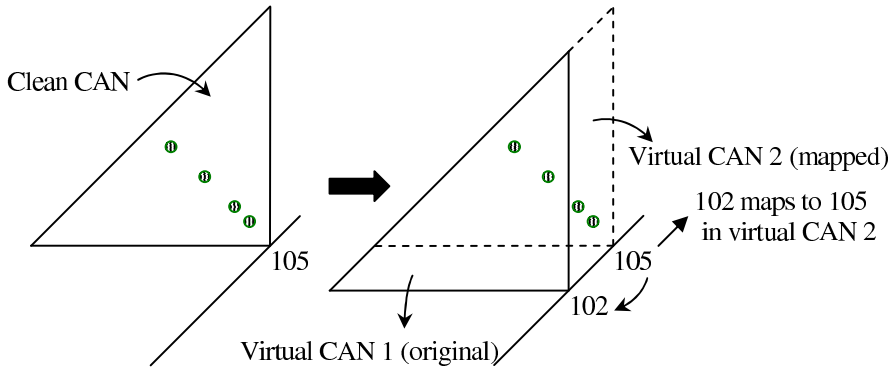

Figure 26: Clean CAN splits into 2 virtual CANs (no subscription movement) on notification $105 \rightarrow 102$.

all its subscriptions along the altitude of the subscription triangle. This characteristic is similar to that of a clean tree. When a new update comes in, we do not need to perform subscription movement for this CAN. All we need to do is define two *virtual* CANs over C_{new}. For the first virtual CAN, is just the original one itself. The second one, however, is addressed on the same set of zone owners, except that a translation is performed on coordinates before routing. In effect, instead of sliding the affected subscriptions towards their new position in the CAN, we slide the CAN coordinates in the opposite direction as this does not involve physical movement of subscriptions. We are able to do this only because of the special property of this CAN that it started off with all subscriptions having the same last notified value. The result of this operation is that there is one original CAN, and one clean CAN which defines two virtual CANs in it. Clean CANs can periodically be merged back to the original CAN just as in the merging operation of clean trees into the dirty tree. We propose to investigate the tradeoffs involved in these alternatives in greater detail.

To make the above description more concrete, let us go back to the example described previously, immediately after the first update of $100 \rightarrow 105$. As noted previously, 4 subscriptions were affected by the update. Under the new approach, we would insert these four subscriptions into a new CAN which is shown in Figure 26. This is a clean CAN because all the subscriptions have the same last notified value, with all subscription centered around their bounds. Now, when the next update $105 \rightarrow 102$ comes in, we can reach the affected subscriptions using the usual cutoff point. However, the affected subscriptions in the clean CAN are not moved. Instead, the CAN is split into two virtual CANs. The first of these is the original clean CAN and contains only those subscriptions unaffected by this update. The second virtual CAN contains the affected subscriptions, and the following mapping is performed on the CAN coordinates in this virtual CAN: a request for coordinate (a, b) is translated to a request for $(a + 3, b + 3)$. In other words, instead of the affected subscriptions sliding to the lower-left, the CAN is effectively slid to the upper-right. We can maintain, for each virtual CAN, the mapping and the range of subscriptions in that virtual CAN. Using this, when a new update comes in, we would simply check if a virtual CAN is affected. If yes, the message would be sent to that virtual CAN

using the mapped coordinates.

The downside of this technique is the increase in number of virtual CANs with updates. However, the advantage is that subscriptions do not need to be relocated each time they are affected. A subscription may get relocated from the original CAN to a clean CAN the first time, and once it is in a clean CAN, we would no longer need to perform relocations because we would then split the clean CAN into virtual CANs instead of relocating subscriptions. However, a larger number of virtual CANs means that we would need to perform matching of an incoming message to check which virtual CANs may be affected, and for each affected CAN we would need to send out a message into that CAN. The growth in number of virtual CANs is similar to the growth of intervals in S-CN with passive subscriptions, i.e. after k updates there may be $O(k)$ clean real CANs (because in the worst case, each update may break off a chunk of the original CAN into a new clean CAN). There may be a worst case of $O(k^2)$ total virtual CANs within these clean CANs. Hence, when the number of virtual CANs is large, we can perform a merge operation where all the clean CANs are merged back into the original CAN and the process repeats. Finally, note that new subscriptions can be supported by inserting them into the original CAN.

5.4 Value-based notifications with tolerance

We can also allow more relaxed semantics for notifications, by introducing a tolerance factor. A subscription i can specify, in addition to a bound δ_i, a *tolerance* t_i. The tolerance specifies how strictly the user wishes to control the receipt of unsolicited notifications, and is expressed as `TOLERANCE` t_i.

A subscription using the additional clause would have the form $S_i =$ `SELECT PRICE FROM STOCK WHERE` *filter* `NOTIFY_WHEN RESULT_CHANGED_BY` δ_i `TOLERANCE` t_i. The meaning of this subscription, assuming that the last notified value is L_i, is: (1) S_i **must** receive any update that falls outside the interval $(L_i - \delta_i, L_i + \delta_i)$, and (2) S_i **may** receive any update that falls outside the interval $(L_i - \delta_i + t_i, L_i + \delta_i - t_i)$.

Tolerance can take any value from 0 (where it is equivalent to the strict notification semantics) to δ_i (which is equivalent to the subscription being ready to accept any notification regardless of its bound).

To see how tolerances can reduce the notification costs, consider a simple case in which every subscription specifies the same tolerance t. We see how this can reduce the maximum number of bound intervals that may need to be sent out by S-CN with passive subscriptions. Consider a simple partitioning of the interval $[0, \delta_{limit}]$ into sub-intervals of size t. Whenever we need to break an interval into two sub-intervals, we simply constrain the interval to break only along the immediately lower partition boundary. This means that in the worst case, there would only be $\lceil \delta_{limit}/t \rceil$ intervals, instead of $min(N, k^2)$ intervals in the case with no tolerances. It is easy to see that in this case, although the maximum tolerance is t, the average tolerance (assuming a uniform spread of subscriptions over the complete bound interval), is actually $t/2$.

Tolerances may also be used to perform periodic interval merging to reduce the number of intervals. For example, in case of unlimited tolerances, we can simply perform a broadcast notification over the entire bound interval, and merge all the interval fragments into a single one. Note that such a notification effectively allows the publish/subscribe middleware to *forget* all the previous disseminated updates that would otherwise have been necessary to retain for precise notification semantics. This operation can be done periodically or when the number of fragments exceeds a given threshold.

As part of our work, we will further investigate the opportunities for optimizations that are presented by the proposed relaxed semantics.

6 The Road Ahead

6.1 Statistics-based wide-area publish/subscribe

As an area of future work, we plan to look closely at runtime optimizations in a publish/subscribe system. In the past, publish/subscribe design has been largely based on specific types of scenarios, for example, range queries. The optimization techniques take advantage of the properties of these scenarios. However, we can also leverage the runtime characteristics of the publish/subscribe system to improve performance. Semcast [40] is an example of a preliminary effort at leveraging such runtime characteristics, but it is limited to channel-based allocations. The basic idea behind SemCast is that the system creates one or more *channels*, where a channel is a tree-based dissemination structure with a single root. Each channel has a channel description which identifies the events that need to be assigned to the channel. When an event comes in, it is assigned to one or more channels. Every event is propagated to all the nodes within a channel. Although channels allow for faster matching, every event assigned to a channel will reach all the nodes in the channel regardless of the subscriptions. A content-based network, on the other hand, performs matching at every node on the path from the source. This means that messages that do not need to reach nodes in a subtree can be truncated. SemCast in this case would rely on assigning such subtrees to a separate channel. Since the number of channels that the system can support is limited, the approach would not be able to perform in-network filtering that is as effective. In addition, SemCast does not consider the assignment of clients to gateway brokers based on dynamic statistics and semantic information. For example, many clients having very similar interests may be assigned to different gateway brokers. Finally, even among the gateway brokers, although SemCast forms channels based on semantic overlap and overlap based on dynamic heuristics, there is no guarantee that such a technique will lead to a minimum cost channel assignment.

We plan to take a broader view of using semantic information and dynamic statistics in optimizing traditional forms of stateful and stateless publish/subscribe systems such as content-based publish/subscribe systems.

6.1.1 Intelligent network design in wide-area publish/subscribe

We first describe an often overlooked issue in designing networks that are suitable for wide-area publish subscribe. Recall that such a system consists of an overlay network of brokers that cooperate to disseminate data. A data source registers itself with some broker, which then acts as the source of data. An interested subscriber registers its interest with some broker. A content-based routing tree is rooted at the source broker and spans all the interested brokers, and events are thus sent to the brokers which need the data. The broker themselves unicast the data to all subscribers which are registered with them and are affected by the event.

Prior work has assumed either that subscribers attach themselves to the closest broker [4], or use simple semantic considerations (like the simple exclusive partitions [20]) to decide which broker to attach to. Neither alternative is attractive: the former may cause every event to be delivered to nearly every broker, and the latter may require many subscribers to attach themselves to brokers that are located very far away.

To clarify the problem, assume that we have a content-based publish/subscribe system with attributes from a set A, in the event schema. Further, let there be k brokers and N subscribers. The *event space* is a $|A|$-dimensional space over all possible events, where any event can be mapped into a point in the event space. A subscription provides values (or ranges) for some non-empty subset A' of the attributes, and can be mapped to a shape in the event space. For example, if $|A| = 2$, a subscription can be a point, horizontal or vertical line, or a rectangle in the two-dimensional event space. If we assume that subscriptions are all range-subscriptions, every subscription would be a rectangle in the event space. We assume knowledge of a window of the E most recent events.

The general assignment problem consists of three phases:

(1) **Partitioning**: Create a *partitioning* P of the event space, which consists of k partitions of the event space. The partitions need not be disjoint, but need to cover every possible subscription. For any partitioning P, every event in the event history stabs one or more partitions. That event needs to reach all the stabbed partitions. We define the filtering power [20] of a partitioning to be the average number of partitions stabbed by an event from the event history. Thus, we have an infinite number of possible partitionings, each with a potentially different filtering power. In general, a partition can be any shape, and it does not have to be a single contiguous chunk. However, as a simplification, we can impose some constraint on the shape. For example, we may require that partitions be rectangular or a union of two rectangles, and so on.

(2) **Broker assignment**: Given a partitioning P_i, we have to assign the k partitions in this partitioning to the k brokers (one partition per broker). There are $k!$ possible broker assignments for every partitioning,

(3) **Subscriber assignment**: Given a partitioning and an associated broker assignment, the next step is to assign subscriptions to brokers. A subset of partitions whose union covers a subscription is called a covering subset of the subscription. A subscription could have many possible covering subsets (the entire set of partitions is trivially one such covering subset). We

can focus on only the *minimal* covering subsets (there could be more than one unique minimal covering subset). Minimal covering subsets are simply those covering subsets that do not contain a subset which is itself a covering subset. For each minimal covering subset (for a particular partitioning and broker assignment), we define a per-subscriber assignment cost which is a function of the latencies from the subscriber to all the brokers that are assigned the members of that covering subset. The subscriber assignment cost is the total cost over all subscribers.

For any given partitioning, we can identify the lowest cost subscriber assignment for each of the $k!$ broker assignments. The overall cost of the partitioning and broker assignment, then, is a function of the filtering power and the lowest-cost subscriber assignment for that partitioning and broker assignment. The optimal solution is the partitioning, broker assignment, and subscriber assignment that yields the lowest overall cost.

The general solution of this problem may be intractable, and so we propose to develop heuristics that do well in practice. The heuristics would involve the use of collected statistics. For example, the broker assignment and subscriber assignment phases use statistics on latencies between subscribers and all the brokers in the system. This could be approximated using techniques such as Global Network Positioning to map nodes to points in a network space. We will also need to maintain event statistics. Such statistics may be maintained as a histogram over the event window. These statistics would help in the partitioning phase. For instance, if there are a large number of events in a certain region of the event space, we may want only one partition to cover that region (assuming that the partition has the resources to handle the event rate), so that these events do not have to be sent to more brokers. The goal is to reduce the filtering power of a partitioning, and this is computed based on the available dynamic event statistics.

Subscription/event biclustering. Another form of dynamic dissemination organization uses biclustering over events and subscriptions. This is useful when subscriptions are blackboxed. The basic idea behind biclustering is that we can model events and subscriptions as a bipartite graph. An edge between an event and a subscription indicates that the event affects that subscription. We wish to find a partitioning of this bipartite graph such that the total cost of edges between partitions in minimized. Each partition could be assigned to a broker. When a new event comes in and has been seen before, we first locate the partition associated with that event. The partitioning strategy assures us that most affected subscriptions are present in this partition, and hence they would receive the events at low latency. We can build a dissemination tree rooted at this broker, and spanning all other brokers that may contain subscriptions affected by this event. This tree can be used to propagate the event to all the other interested subscribers. Disseminating new incoming events in such a scenario is a more challenging problem because we do not have a priori knowledge to guide us to the correct partition with most affected subscriptions. We could use a learning algorithm that uses history to estimate the best partition that an event needs to be assigned to. We will address these challenges as part of future work in this domain.

Publish/subscribe system	Database system
Event	Query
Dissemination plan	Execution plan
Overlay maintenance	Index update
Event history statistics	DB statistics (e.g., histograms)

Table 5: Parallels between publish/subscribe systems and database management systems.

6.1.2 Dynamic optimization framework

Depending on the dynamic event and subscription characteristics, different techniques may need to be employed by a publish/subscribe system. The interface between the network and the database is sensitive to the characteristics of the workload placed on the system. Similarly, on the database side, the use of different types of index structures is appropriate under different scenarios. This means that we need to introduce an optimization framework that chooses the appropriate interface between the database and the network. This optimization would be guided by both online statistics and the semantics of subscriptions. This unified view effectively treats different server processing and network dissemination techniques as indexes (with the associated costs) whose materialization and usage is driven by the optimization framework. We propose to develop cost models for online optimizer decisions.

We use Table 5 to further illustrate the parallels between query optimization in a database system, and technique selection in a wide-area publish/subscribe system.

As a concrete example of the importance of such an optimization framework, first consider the need for subscription-based choices. If most subscriptions require every published event (for example, in case of `SELECT *` type of subscriptions or subscriptions with wide selection ranges), the best strategy may be to use a broadcast mechanism (with no database or subscription state at the server) and rely on filtering at subscribers to eliminate false positives. On the other hand, if different subscriptions request different ranges of attributes over a defined event schema, a content- based approach may be preferable. In this case, we could either eliminate the server in case of stateless subscriptions, or allow the server to maintain limited state and index structures in case of stateful subscriptions. In case all subscriptions are stateful, and most are rarely affected by incoming events, the ideal strategy may be to maintain more state (with simple index structures) at a server and use unicast to reach the exact set of affected subscribers.

Secondly, if available, it is also beneficial to take event statistics into account while making such optimization decisions. For example, assume that most events fall in the region of event space where they affect very few subscriptions. In such a case, even if all subscriptions have wide selection ranges, we would still prefer to use unicast as the dissemination mechanism. If many events are found to affect the same approximate set of subscriptions, we could build a multicast group for that set of subscriptions to enable efficient dissemination of similar future events. The subscription and event statistics can also guide the building of a dissemination technique in addition to the choice of strategy, for example using the techniques proposed in Section 6.1.1.

6.2 Batch event processing and dissemination

Further ahead, we plan to examine batch event processing and dissemination. Intuitively, it is useful to batch incoming events as long as the consistency requirements are met. This is because processing a batch of updates is typically more efficient than processing the events individually. Further, dissemination of a batch of events gives more room for optimized dissemination as compared to the dissemination of each event by itself. We will examine the tradeoffs involved in batching and will develop robust techniques that take advantage of situations where batching can give us improvement in system cost while meeting the user-defined fidelity or performance goals.

7 Related Work

The increasingly widespread use of publish/subscribe has attracted much attention from the research community, as evidenced by the large number of publications in the annual International Workshop on Distributed Event-Based Systems (since 2001). We first describe the work related to our preliminary research efforts on distributed network querying. We then focus on the main pieces of work that are related to our current and future areas of research.

7.1 Distributed network querying

7.1.1 Network monitoring.

A large number of network monitoring systems have been developed by both the research community and commercial vendors. Astrolabe [57] is a system that continuously monitors the state of a collection of distributed resources and reports summarized information to the its users. Ganglia [33] is a system for monitoring a federation of clusters. While our work also considers the network monitoring problem, we focus on supporting set-valued queries approximately rather than aggregation queries. Our approach of bounded approximate caching and methods for locality-aware, cost-based cache management offer better flexibility and adaptability than these systems, which are preset to either push or pull each piece of information. Our techniques can be used to enhance these and other existing network monitoring systems.

7.1.2 Data processing on overlay networks.

PIER [27] is a DHT-based massively distributed query engine that brings database query processing facilities to new, widely distributed environments. For network monitoring, also one of PIER's target applications, we believe that bounded approximate caching meshes well with PIER's relaxed consistency requirement, and our DHT-based caching techniques can also be applied to PIER. Locality-aware DHTs have been used to build SCRIBE [12], a scalable multicast system, and SDIMS [58], 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.

7.1.3 Approximate query processing for networked data.

The idea of bounded approximate caching has been explored in detail by Olston [36], along with techniques such as adaptive bound setting, source cooperation in cache synchronization, etc. We apply bounded approximate caching in this paper, but we focus on how to select caches across the network to exploit locality, and how to locate these caches quickly and efficiently to answer queries. We also extend the approximate replication scheme by allowing guarantees to be provided not only by the owner, but also by any other cache with a tighter bound.

7.1.4 Web caching and web replication.

Web caching [43] is often done by ISPs using web proxy servers. Web replication [43] refers to data sources spreading their content across the network, primarily for load balancing. In both cases, the cache content is stored exactly and most often relatively stable content (e.g. images) is replicated at static locations. They do not deal with the problem of rapidly updating data; this means that they can afford to establish a large number of caches/replicas. Our system deals with replication of dynamic measurements and therefore update costs are high. We reduce update costs by caching bounded measurements, and balance update and query costs by caching at dynamically chosen nodes in the network.

7.2 Publish/subscribe systems

A survey of data-dissemination systems can be found in [24]. We adopt the publish/subscribe model of dissemination, which is based on aperiodic push. Other models include periodic push (e.g., broadcast disks [2]), periodic pull (client polling systems), and aperiodic pull (classic request/response client/server systems). Early publish/subscribe systems are based on channels or subjects, see, for example, [35, 42]; the publish/subscribe feature of Java Message Service (JMS) [52] also falls into this category. Channels are predefined and their granularity is often too coarse to fit the particular interests of individual users.

Recently, research efforts have been focused on content-based publish subscribe systems which provide fine granularity and flexibility. These can be broadly characterized as systems where publishers publish events following a particular predefined schema, and subscribers express their interests as *profiles* [20] which are predicates over the schema. A large number of such systems have been built in recent years, e.g., SIFT [59] (for text documents), ONYX [20] (for filtering and transformation of XML messages), and the wide-area event notification service [9]. In all these systems, subscriptions are stateless filters defined over individual messages, so they cannot express queries of interest across different messages or over the event

history. Profiles are not powerful enough to accommodate stateful SQL-style subscription requirements. ONYX supports on-the-fly transformation of an XML message according to a subset of XQuery; but filtering and transformations are still limited to individual messages. A few continuous query systems also support rich query languages. Unfortunately, these do not address the problem of efficiently delivering updates over a network. ONYX has begun addressing this problem; however, the focus of ONYX on supporting transformation of XML messages is different from our goal of supporting more general stateful SQL subscriptions that cannot be processed on individual messages.

7.3 Database-side processing

Continuous query systems [31, 18, 54] and stream processing systems [1] can be regarded as a form of publish/subscribe system where continuous queries over streams correspond to our subscriptions. These systems provide automatic notification whenever a continuous query result changes. OpenCQ [31] supports powerful notification conditions that refer to current and previous database states; however, conditions cannot refer to the update history. NiagraCQ [18] supports timer-based notification conditions. In other words, NiagraCQ allows control over the staleness of subscriptions, but not over their accuracy, in terms of user-defined metrics.

The idea of group processing has been identified and used in trigger processing and continuous query processing systems [26, 18]. Work on scalable database trigger processing [26] focuses on exploiting common patterns in triggering conditions (like our notification conditions). Work on scalable continuous query processing (e.g., [18, 32]) focuses on exploiting common patterns in continuous queries. In particular, predicate and query indexing techniques have been developed in [26, 18, 21] to speed up group processing. The upper hull computed for dissemination is similar to computing dynamic skyline in [39] which uses a regular R-tree.

7.4 Network dissemination

A server needs to deliver notifications to affected subscribers over a network. The problem of efficient message delivery has long been tackled in networking and distributed systems research. The traditional delivery mechanism is based on client polling. The next generation of delivery mechanism uses real push techniques based on group-based multicast protocols, e.g., IP multicast. Multicast provides a perfect interface for channel-based subscription services. IP multicast has also been exploited in building publish/subscribe systems that support more general filter-style subscriptions [38]. Because of slow adoption of IP multicast, there have been proposals for supporting application-level multicast using an overlay network (e.g., [45, 61, 11]). Oftentimes, they use an overlay network called distributed hash tables, which provide a convenient hash table abstraction over the participating overlay nodes (e.g., [48, 44, 51, 60]). Bullet [29], on the other hand, creates a mesh over the multicast dissemination tree in order to improve throughput. The problems with multicast were discussed earlier. Although we have included multicast as a comparison in some of our work, the problem with most multicast techniques is

that of number of groups. In order to implement publish/ subscribe, we need a large number of groups (one for each possible subscriber set) and these techniques do not scale to large number of groups. Group reduction techniques [38] require the end subscriber to have the ability to filter out unwanted messages. The group size problem is seen in some of our experimental results with our implementation of multicast, where we avoided the need for large number of groups by encoding the set of subscriptions as part of the message.

An alternative dissemination interface is content-based networking [10, 5, 13]. A content-based network can be used to implement a publish/subscribe system supporting filter subscriptions. A number of such systems have been developed (e.g., [55, 53, 25]). SemCast [40] proposes a number of techniques for efficient dissemination including the use of dynamic statistics. However, subscriptions in all these systems are limited to stateless filters. Nevertheless, they can still be used by our system as the messaging layer for delivering notifications once they are computed. We use message and subscription reformulation to enable traditional content-based networks to handle stateful queries.

7.5 Notification conditions and bounds

In Section 5, we introduced bounds as a means of specifying notification conditions. Bounds have been studied extensively in database literature. Chris Olston [36] proposes bounds in the context of approximate caching, where caches maintain bounds on the source data and sources update the caches whenever bounds are violated. He does not consider efficient indexing or dissemimnation structures for these bounded approximate caches, and our work can be though of as such an extension although in a completely different setting and scale. In [37], the focus is on how these bounds are set adaptively based on query workloads. However, we assume that subscription notification conditions are specified in advance and cannot change with time.

In the context of dynamic data dissemination, the work by Ramamritham et al [50, 49] is relevant in that they are also targeting bounds over values. However, they assume a different notification semantics which is weaker than ours; they assume that subscriptions (repositories in their applications) are ready to receive any notification even if it does not violate the bounds. However, we focus on stricter semantics with disciplined relaxation using tolerances. In addition, they do not consider server-side approaches, however we have developed efficient indexing structures with periodic maintenance to make the server-side approach feasible. In addition, we have introduced several network-dissemination strategies for our strict notification semantics using content-based networks.

8 Conclusions

As the Digital Age has matured, we see a clear trend towards applications with more and more sophisticated data needs. Publish/subscribe systems, with their push-based technology, have proven themselves to be very scalable and more adaptable to modern applications than tradi-

tional dissemination technologies such as periodic pull. However, the current generation of wide-area publish/subscribe systems, which are mostly stateless and content-based, lack a rich subscription model to capture modern data needs. Continuous query systems that provide richer subscription models do not consider the dissemination bottleneck which is core in a wide-area publish/subscribe system.

Our research approaches to bridge this problem by approaching it from a number of different angles: (1) We advocate a powerful subscription model that scalably supports the application needs, data sizes, and data rates of tomorrow. (2) We believe that there is a need to design the right interface between the database and the network in order to maximize performance while using richer subscription models. (3) We propose a holistic view of publish/subscribe systems where processing and dissemination is guided by an optimization framework that takes runtime as well as semantic statistics into account.

Towards these goals, we first examined the data needs of some typical wide-area applications (network monitoring and resource querying) and developed techniques in Section 3 to efficiently support them using approximate caches to which updates are pushed by publishers. We showed in Section 4 that rich stateful subscriptions such as range- MIN can be scalably supported in publish/subscribe systems by designing the right interface, and by performing smart message and subscription reformulation. We have begun looking at scalable support for per- subscription notification conditions in a wide-area publish/subscribe system, by again approaching the problem from the interface perspective. Our initial results were discussed in Section 5. Finally, we advocate the use of runtime system statistics to improve the system along a number of vectors, including good subscription and broker assignments and helping an online optimizer to choose the correct indexes and dissemination techniques based on these statistics. These ideas form the basis for our future work, and are briefly described in Section 6.

References

[1] Special issue on data stream processing. *IEEE Data Engineering Bulletin*, 2003.

[2] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *SIGMOD*, 1995.

[3] L. Aguilar. Datagram routing for internet multicasting. In *SIGCOMM*, 1984.

[4] M. K. Aguilera, R. Strom, D. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *1999 ACM Symp. on Principles of Distributed Computing*, Atlanta, Georgia, USA, May 1999.

[5] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.

[6] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish- subscribe systems. In *ICDCS*, 1999.

[7] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *SIGCOMM*, pages 205–217, 2002.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 1970.

[9] A. Carzaniga, D. S. Rosenblum, , and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, 2001.

[10] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, 2001.

[11] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Journal on Selected Areas in Communication*, 2002.

[12] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 2002.

[13] R. Chand and P. Felber. A scalable protocol for content-based routing in overlay networks. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, 2003.

[14] B. Chandramouli, J. Xie, and J. Yang. On the Database/Network Interface in Large-Scale Publish/Subscribe Systems. Technical report, Department of Computer Science, Duke University, November 2005.

[15] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *2006 SIGMOD*, Chicago, Illinois, USA, 2006.

[16] B. Chandramouli, J. Yang, and A. Vahdat. Distributed network querying with bounded approximate caching. Technical report, Department of Computer Science, Duke University, June 2004.

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

[18] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

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

[20] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.

[21] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/ subscribe. In *SIGMOD*, 2001.

[22] Yahoo! Finance. http://finance.yahoo.com.

[23] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[24] M. Franklin and S. Zdonik. A framework for scalable dissemination-based systems. In *OOPSLA*, 1997.

[25] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *Middleware*, 2004.

[26] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *ICDE*, 1999.

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

[28] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR DB system. In *ICDE*, 2003.

[29] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[30] 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.

[31] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering*, 1999.

[32] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

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

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

[35] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: An Architecture for Extensible Distributed Systems. *ACM Operating Systems Review*, 27(5):58–68, 1993.

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

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

[38] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, 2000.

[39] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM TODS*, 2005.

[40] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.

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

[42] D. Powell. Group communication. *Communications of the ACM*, 39(4):50–53, 1996.

[43] M. Rabinovich and O. Spatschek. *Web caching and replication*. Addison-Wesley, 2002.

[44] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.

[45] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *2001 Intl. Workshop on Networked Group Communication*, 2001.

[46] A. Rodriguez, C. Killian, D. Kostić, S. Bhat, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, March 2004.

[47] 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.

[48] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *2001 Middleware*, November 2001.

[49] S. Shah, S. Dharmarajan, and K. Ramamritham. An efficient and resilient approach to filtering and disseminating streaming data. In *VLDB*, 2003.

[50] S. Shah, K. Ramamritham, and P. J. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *VLDB*, 2002.

[51] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *2001 SIGCOMM*, 2001.

[52] Sun Microsystems. Java Message Service. `http://java.sun.com/products/jms/`.

[53] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.

[54] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 1992.

[55] P. Triantafillou and I. Aekaterinidis. Content-based publish/subscribe systems over structured P2P networks. In *DEBS*, 2004.

[56] 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.

[57] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 2003.

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

[59] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM TODS*, 1999.

[60] Y. B. Zhao, J. Kubiatowitcz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, Department of Computer Science, Univ. of California at Berkley, 2001. Tech. Rep. UCB/CSD-01-1141.

[61] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide- area data dissemination. In *2001 Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, 2001.