Summary-based Routing for Content-based Event Distribution Networks

Yi-Min Wang, Lili Qiu, Chad Verbowski, Dimitris Achlioptas, Gautam Das, and Paul Larson
Microsoft Research, Redmond, WA, USA

Abstract— **Providing scalable distributed Web-based eventing services has been an important research topic. It is desirable to have an effective mechanism for the servers to summarize their filters for in-network preprocessing in order to optimize system performance. In this paper, we propose a summary-based routing mechanism and introduce the notion of imprecise summaries to provide a trade-off between routing overhead and event traffic. Our system uses similarity-based filter clustering to reduce overall event traffic and performs self-tuning summary precision selection to optimize throughput. We have implemented summary-based routing on top of an XML-based infrastructure that closely follows the proposed Web services standards. Measurements from the actual implementation validate our analytical and simulation results, and demonstrate the practical benefits of the proposed techniques.**

I. Introduction

Today, World Wide Web use is predominantly based on the synchronous polling model: a Web user either visits a Web page using its URL or goes through a search engine to find pages of interest. A complementary model that is becoming increasingly popular is the asynchronous publish/subscribe (pub/sub) event notification model: a user subscribes to a server by specifying events of interest, and later receives notifications when any of those events are published. The Instant Messaging buddy/contact lists, the Amazon shopping alerts, the EBay outbid notifications, and the stock quotes and news alerts from Yahoo! Alerts and MSN Mobile are all examples of this model.

In addition, new applications with subscribers being non-human entities are also emerging. User-agent applications that communicate with sensors and devices to perform automation tasks and B2B Web applications that communicate with each other to conduct day-to-day businesses will both rely heavily on the event notification model. Non-human subscribers present additional scalability challenges to the design of pub/sub systems because they can significantly increase the total number of participants, handle much more complex subscriptions, and receive and process notifications at a much higher rate.

Pub/sub systems are typically classified into two categories: topic-based and content-based. In a topic-based system, each event is associated with one of a set of predefined topics and a subscriber receives notifications for all events belonging to the subscribed topics. A content-based system supports more fine-grained selection of events by defining an event schema under a topic, which specifies the names and types of the attributes that appear in an event. *Subscription filters* are then specified as conjunctions of predicates [11] on a subset of those attributes. For example, under the topic of "stock quotes", we may have an event schema that defines three attributes: Symbol, Price, and Volume. An example event would be (Symbol=GE and Price=29.3 and Volume=30,000,000); an example subscription filter would be (Symbol=GE and Price>30.0).

A centralized, content-based pub/sub system consists of a server that receives and stores subscriptions from all subscribers, receives events from all publishers, performs matching of each event against the subscriptions, and sends notifications to all subscribers with matching subscriptions.

To provide scalability, several previous papers have proposed a distributed architecture that uses a network of servers [9] (also called proxies [31] or brokers [4]), which can be either physically co-located or distributed in geographically diverse locations, to route events between publishers and subscribers. A naive way of performing distributed pub/sub operations in such a network is to propagate every event message to every server, and have all content-based matching operations performed locally at each server. A natural optimization is to allow each server to "summarize" the entire set of local subscription filters and propagate that summary information along the reverse path of event dissemination in order for the upstream routers (or forwarders) [8] to block unnecessary event traffic at the earliest point [15], [4], [31]. The SIENA system [9] formalized this notion by arranging the subscription filters as a partially ordered set (poset), and using the set of root filters (i.e., maximal poset elements) as the summary. We refer to such a summary as a *precise summary* because it precisely captures the set of events that are needed by any downstream servers.

Two issues related to the use of summaries need to be addressed. First, if the subscription filters submitted to the same server have poor "locality" (i.e., the sets of events they match do not have much overlap), then the summaries may be too broad to allow efficient routing and effective traffic reduction.

Second, although precise summaries minimize event traffic, they may not optimize overall system throughput. Depending on the distribution of subscription filters, precise summaries may be so complex that the routers become the bottleneck, degrading system throughput. While empowering the routers with better hardware helps to alleviate the throughput degradation to some extent, it is not cost-effective to provision routers based on peak load, which can be orders of magnitudes higher than the typical load. By properly adjusting summary precision, we can dynamically optimize the overall throughput using the existing hardware under varying loads.

In this paper, we propose subscription partitioning and summary-based routing to address the above issues, and evaluate their effectiveness using analysis, simulation, and implementation. As the first step to gain insights into subscription partitioning and summary-based routing, we make the following simplifications and assumptions in this paper.

First, we focus on the case where the servers are physically co-located. As described in Section III-D, our idea can be generalized to handle widely distributed servers by incorporating network distance among servers.

Second, different from many previous work, which used reverse path forwarding to set up forwarding table (e.g., [9]), our study focuses on event notification systems that decouple the subscription paths from the notification paths. Many existing commercial systems adopt this model. For example, a user would go to a Web page to enter his or her subscriptions and specify that the notifications should be sent to his or her cell phone. Such architecture gives us the flexibility to route and cluster the subscriptions once they enter the network, and allows us to focus on optimizing the system throughput where the system excludes the notification paths (e.g., the path in a cellular network towards a subscriber's cell phone). In cases where it is desirable to have servers handle subscriptions for nearby clients, we can extend our scheme by considering subscribers' locations as additional dimensions for clustering as described in Section III-D.

Third, our work focus on the distribution topology that has one level of branching as shown in Figure 1, where an event dispatcher is connected directly to all servers. We use N_s to denote the number of servers throughout the paper. The remaining symbols in Figure 1 will be explained in Section IV and Appendix . We leave extension of our approach to handle general distribution topologies with potentially multiple levels of branching as part of our future work.

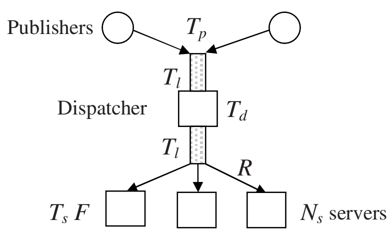


Fig. 1. System architecture evaluated in this paper.

Our key contributions and results can be summarized as follows.

First, we propose a summary-based routing framework and introduce the notion of *imprecise summaries* to provide a tradeoff between routing efficiency and event traffic: summaries with a lower precision would allow more efficient routing at the cost of a higher amount of false-positive event traffic. The framework includes previous work on precise summaries and no summaries as two extremes of the spectrum.

Second, to make summaries more compact and effective, we use similarity-based filter clustering for offline subscription partitioning and online subscription routing. We evaluate the benefit of clustering using both uniform and Zipf-like subscription and event distributions.

Third, to evaluate the impact of summary precisions on overall system throughput, we present analysis and simulation results to show that varying summary precisions can provide load balancing among system components to optimize system throughput.

Fourth, to allow dynamic adaptation to changing system and network characteristics, we present a self-tuning algorithm that performs ongoing monitoring of throughput bottleneck and automatically adjusts summary precision to bring the system towards a new optimal operating point.

Finally, we describe an implementation of summary-based routing on an XML/SOAP messaging infrastructure conforming to a set of proposed Web Services standards [30]. With actual parameters from the implementation and the system setup, we give a detailed discussion on when imprecise summaries are useful in practice. We present experimental results from the implementation to validate our analysis and to demonstrate the capability of our self-tuning algorithm to dynamically maintain optimal throughput.

This paper is organized as follows. In Section II, we give an architecture overview of a content-based event distribution network. In Section III, we describe and evaluate summary-based subscription and event routing. In Section IV, we present an analysis and simulation results for throughput evaluation, and describe a self-tuning algorithm for adaptive throughput optimization. In Section V, we describe the architecture of our Web Services-based implementation and discuss actual experimental results. We review previous work in Section VI, and conclude the paper in Section VII.

II. System Architecture

Just as Content Distribution Networks (CDNs) have been deployed to provide scalable Web information dissemination, we propose building *Event Distribution Networks (EDNs)* to provide scalable event dissemination. An EDN will be built as a self-configuring overlay network of servers. As shown in the left-hand side of Figure 2, a geographically distributed subset of nodes, called edge servers, are deployed to provide a low-latency front-end interface to geographically distributed publishers and subscribers. Other servers reside inside the network and may host subscriptions or route traffic or both. Through the edge servers, event sources (i.e., publishers) publish events and subscribers submit subscriptions for events of interest.

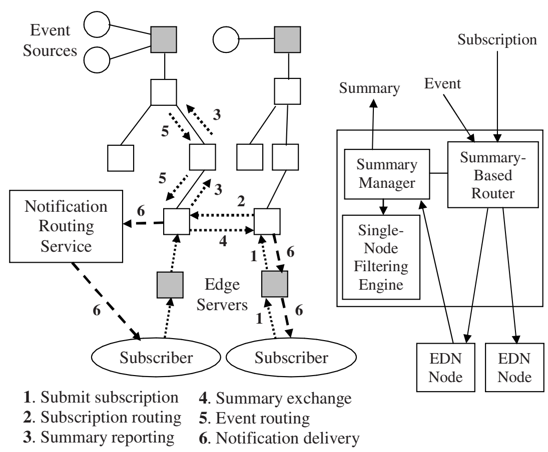


Fig. 2. EDN network architecture. (Shaded squares are edge servers; non-shaded squares are internal servers; the right-hand side is the node architecture for an internal server.)

The following definitions will be used throughout the paper: a *filter* f is any expression that defines a set of events E_f, and an event e *matches* the filter f if $e \in E_f$. A filter f' is said to *cover* another filter f if $E_f \subseteq E_{f'}$. A similar covering relationship is defined between two sets of filters. A subscription consists of a *notification address* and a *subscription filter*, which is a filter in the form of a conjunction of predicates conforming to a pre-defined event schema. When a server hosting a subscription receives a published event that matches the subscription's filter, it sends a notification to the corresponding notification address. Given a set of subscription filters F, a *summary* S of F is a set of filters that covers F. A filter $f \in F$ is *maximal* if there is no $f' \in F$ such that f' covers f. Clearly, the set of all maximal filters in F covers F.

Any subscription submitted to an edge server is forwarded to an internal server. (We will use the term "server" to mean "internal server".) In this paper, we focus on the following type of notification addresses. The subscriber specifies the address of a *notification routing service* followed by a device-independent ID such as a Yahoo! ID or a .NET Passport ID. A notification is first sent to the routing service, which resolves the ID and forwards the notification to the subscribing user via instant messaging, email, cell phone SMS, etc. [27]. When such type of notification addresses is in use, the internal servers can effectively batch notifications, which alleviates the scalability problem on the notification delivery side and allows us to concentrate on the event propagation side.

Upon receiving a subscription from an edge server, an internal server may further forward it to another server whose summary overlaps the most with (and hopefully already covers) the new filter, thus minimizing the additional event traffic handled by each server. The server makes the routing decision based on the *subscription routing summaries.* Each subscription server knows about all other subscription servers in the network, and periodically sends them a summary of its subscription set for routing subscriptions. The right-hand side of Figure 2 illustrates the architecture of an internal server. The *single-node filtering engine* can be any pub/sub engine that stores subscriptions in efficient data structures for fast matching [14], [2], [11]. The *summary manager* is responsible for maintaining summaries for the entire sub-tree rooted at the current node. [1] It intercepts all new subscriptions entering the local filtering engine, and updates the local summary accordingly. It also receives summaries from all child nodes. All these summaries are made available to the *summary-based router*, which maintains them in an efficient data structure for fast event and subscription routing. The summary manager combines all summaries into a sub-tree summary and reports it to the parent node. Upon receiving an event, a summary-based router forwards it down only those sub-trees whose summaries the event matches.

The format of the summaries can be different from that of the subscription filters, and thus the data structure used by the summary-based routers can be different from that used by the filtering engines. For example, when shopping at an online auction site, users may subscribe notifications by specifying ranges for price, seller rating, shipping cost, and time to arrive. Each subscription is then represented as a multi-dimensional rectangle. Instead of keeping rectangles for every subscription, we can summarize them using a small number of bounding rectangles to speed up routing. For equality predicates (e.g., Symbol=GE) in subscription filters, we can summarize them using a Bloom filter [6], [28]. Another way of summarizing subscriptions is to use only a selected subset of attributes in the event schema.

Let S denote a summary of the set of filters F representing all the subscriptions hosted by a single server. The summary S is *precise* if $E_F = E_S$; otherwise, $E_F \subset E_S$ and it is called an *imprecise summary*. Clearly, the set of all maximal filters of F forms a precise summary of F. The most imprecise summary covers the entire event space, which corresponds to the case of not using summaries, but forwarding every event to every server. Reducing *summary precision*, i.e., increasing the size of $E_S \setminus E_F$ (where "\" denotes set-minus), would introduce more "false-positive traffic" that invokes wasteful operations at some downstream filtering engines to produce zero match. However, it would typically reduce per-event processing time at the summary-based routers. Summary-based routing thus allows tunable in-network pre-processing of event traffic to optimize overall network performance. We note that subscription routing summaries need not be the same as those used for event routing: while the latter must guarantee correctness, the former are only used for optimization and so they can be in a less complex format and exchanged less frequently.

III. Summary-based Routing

In this section, we describe how to route incoming subscriptions and events.

A. Subscription Routing & Partitioning

In general, there are two approaches to partitioning the overall pub/sub operations among multiple servers: we either partition the event space or partition the set of subscription filters [28]. We call the former *Event Space Partitioning (ESP)*, and the latter *Filter Set Partitioning (FSP)*. As discussed in [28], the ESP approach in general needs to replicate subscriptions to multiple servers, making it difficult to support stateful update and subscription deletion. Therefore, we focus on the FSP approach in this paper to simplify subscription management.

1) Range Predicates: We have investigated ESP-based subscription partitioning of equality predicates in [28]. In this paper, we study subscription partitioning of range predicates using the FSP approach. Given an event schema with d attributes, it is convenient to model each event as a point and each subscription filter as a rectangle in a d-dimensional space where each dimension corresponds to an attribute.

The objective of subscription partitioning is to minimize the total event traffic and the associated server load, while avoiding overloading any individual server. There are two versions of the problem: offline and online. In the offline version, we are given a set of existing subscription filters to be partitioned among a set of servers. Such subscriptions are usually available when a successful event service provider upgrades its pub/sub system

[1] For ease of presentation, we assume throughout the paper that the servers are statically configured into a tree. The same concept of summary-based routing can be applied to systems that dynamically construct trees through either advertisement forwarding [9] or SUBSCRIBE message forwarding [24] in a peer-to-peer routing system.

(e.g., upgrade from a single-node centralized architecture to a distributed one in order to accommodate growing demands). A service provider can also periodically re-partition the subscriptions using the offline scheme when the performance of the online version degrades to a certain threshold. In the online version, each server, upon receiving an incoming new subscription, makes a local decision based on the subscription routing summaries to route the subscription to the "best" server. To avoid server overload, only those servers with load below the pre-specified threshold are considered candidate targets for routing. Intuitively, an online algorithm would perform better if it starts with an initial good partitioning provided by an offline scheme.

Below we use similarity-based filter clustering for subscription partitioning and routing. We will evaluate the effectiveness of the clustering techniques using realistic subscription and event distributions in Section III-C.

Random routing: The random algorithm is oblivious to the similarity among different subscriptions. It randomly routes a subscription to a server. Subscriptions that have large overlap with each other may be assigned to different servers, and thus incur significant duplicate traffic and server processing load for events falling in the intersection region.

R-tree based routing: An R-tree [16] is a dynamic index structure for multi-dimensional data rectangles. It is a height-balanced tree similar to a B-tree [5], with all data rectangles residing at the leaf nodes. Typically, an R-tree algorithm tries to grow the tree in such a way that rectangles with large intersections reside at leaf nodes that are close to each other.

An offline R-tree algorithm, such as the bulk-loading algorithm described in [13], builds an R-tree in a top-down fashion. At each level, it sorts the rectangles based on their minimum, maximum, and center coordinates of each dimension, considers all cuts orthogonal to the coordinate axes that would align with the eventual partition boundaries, greedily picks the cut that minimizes a cost function, and then recursively applies the cuts to the smaller partitions until the desired number of rectangles per partition is reached. We apply the bulk-loading algorithm to the offline partitioning problem by forcing the number of children of the root node to be equal to the number of servers. Each sub-tree rooted at a child is assigned to a server. The cost function is chosen to be the sum of the volumes of the bounding rectangles of the two partitions on the two sides of the cut. Assuming that events are uniformly distributed, minimizing this sum of volumes corresponds well to minimizing the sum of event traffic.

In the online version, a new subscription is checked against each server's summary R-tree and forwarded to the server that owns a rectangle that has the maximum overlap with the subscription rectangle, and is not overloaded (i.e., current load is below a threshold).

K-Mean clustering: K-Mean [18] is a well-known clustering technique to group points into clusters based on their proximity. It starts with an arbitrary initial cluster assignment. Then it assigns each point to its closest cluster centroid, re-computes the centroids after all assignments, and iterates until the total distance between the new clusters' centroids and the old clusters' centroids is within δ (In our experiment, $\delta = 1$). By representing each subscription rectangle using its centroid, we can apply K-Mean clustering to partitioning subscriptions offline. We refer to this scheme as offline K-Mean.

Algorithm	Storage	Amortized Search Time	Total Traffic (Server Load)
Random	0	O(1)	High
Offline R-Tree	O(S)	O($log(S)$)	Lowest
Online R-Tree	O($N_s \cdot N_b$)	O($N_s \cdot N_b$)	Low
Offline K-Mean	O($S + N_s$)	O($I \cdot N_s$)	Medium
Online K-Mean	O(N_s)	O(N_s)	Medium

TABLE I

COMPARISON OF DIFFERENT PARTITION ALGORITHMS, WHERE THE NUMBER OF DIMENSIONS IN THE DATA IS A SMALL CONSTANT.

In the online version, we route an incoming subscription A to the server i whose subscriptions' centroid is closest to A's centroid, and then we update the subscriptions' centroid at the server i.

Summary: Table I compares the cost and performance of different partitioning algorithms, where N_s is the number of servers, S is the total number of subscriptions, I is the number of iterations in K-Mean, N_b is the number of summary bounding rectangles in the R-Tree for each server, and the number of dimensions of rectangles is a constant. Note that $N_s * N_b \leq S$. In Section III-C, we will evaluate the performance of these algorithms in more details.

Table I shows the trade-off between the complexity and performance of the algorithms. Random assignment is cheapest, but yields the highest amount of traffic and server load; on the other hand, R-Tree reduces the total traffic and server load at the cost of more expensive computation; and K-Mean falls in between. A nice feature of the R-Tree approach is that we can smoothly trade off the complexity for performance by adjusting the number of summary bounding rectangles N_b we use for subscription routing.

B. Event Routing

We use the R-tree algorithm to route incoming events. Given an event e, it finds all the servers which have at least one summary bounding rectangle containing e. The pseudo-code for event routing is shown below. Note that the event routing scheme is independent of the subscription routing scheme, and we use the R-tree for event routing regardless of which subscription routing scheme is used.

Online R-tree based event routing

Input: An incoming event, e
Output: The IDs of the servers to which the event is forwarded

```
count = 0;
foreach server s
    foundMatch = SearchAnyContaining(e, RTree[s]);
    if (foundMatch = TRUE)
        serverIDs[count++] = s;
    end
end
```

C. Performance Evaluation

In this subsection, we compare the performance of different subscription partitioning algorithms. Our evaluation differs from previous studies on clustering algorithms in the following ways.

First, according to our application requirement, we use event traffic as a performance metric for comparing different partitioning schemes. Specifically, we use the average per-server hit ratio as the performance metric, which is defined as the total number of forwarded events by the dispatcher divided by the total number of published events and the number of servers N_s. Given a set of subscription rectangles, a lower hit ratio is desirable as it indicates more effective traffic reduction.

Second, we examine the effectiveness of partitioning algorithms under different event and subscription distributions. In particular, we study the impact of realistic event and subscription distributions on the amount of event traffic generated.

Third, we investigate the effect of subscription summary precision on the traffic reduction.

Unless otherwise specified, we use the following settings: each server hosts 10,000 subscriptions on average, with a capacity constraint (i.e., load threshold) of 20,000 subscriptions. Events and subscriptions are both generated randomly in a 4-dimensional space with coordinate values uniformly distributed in the range of 0 to 10 in each dimension. In most experiments, the R-tree schemes use precise summaries for subscription routing. In addition, we also evaluate the effects of imprecise summaries, Zipf-like subscription and event distributions, different server load thresholds, and different numbers of dimensions.

Varying the number of servers: Figure 3 compares the performance of different partitioning algorithms by varying the number of servers. With random partitioning, the per-server hit ratio is the highest and close to 1.0. This means that almost every incoming event is forwarded to all the servers due to poor clustering of subscription rectangles. In comparison, the offline R-tree algorithm performs the best, and reduces the hit ratio by 20% to 60%. Moreover, the benefit of offline R-Tree increases with the number of servers. The offline/online R-tree curve is obtained by using 50% of the subscriptions to build an R-tree offline and then inserting the remaining subscriptions with the online R-tree algorithm. It performs very close to the offline R-tree. However the online R-tree performs noticeably worse: its hit ratio is 0.2 higher than that of the offline R-tree when the number of servers is large. This suggests a set of good initial partitions obtained from the offline algorithm is important to the performance of the R-tree. The performance of the K-Mean approach falls between random partitioning and online R-tree. The R-tree approach outperforms K-Mean because the R-tree tries to minimize the sum of rectangles volume of all server, which is a direct measure of event traffic.

Varying subscription routing summary precision: Next we evaluate the effect of summary precision on traffic reduction. We control the precision by varying the depth of the subtree of the maximal R-tree that we use for subscription routing. Shallower trees with less precise summaries would allow faster routing, but potentially at the expense of less effective clustering. Figure 4 shows the hit ratio versus the average depth of the summary R-Trees across all servers, when the number of servers is 10 and 20. The leftmost data point corresponds to every server using only the root node of the maximal R-tree; the rightmost point is for using the entire maximal R-tree as a precise summary.

We make the following observations. First, the offline/online

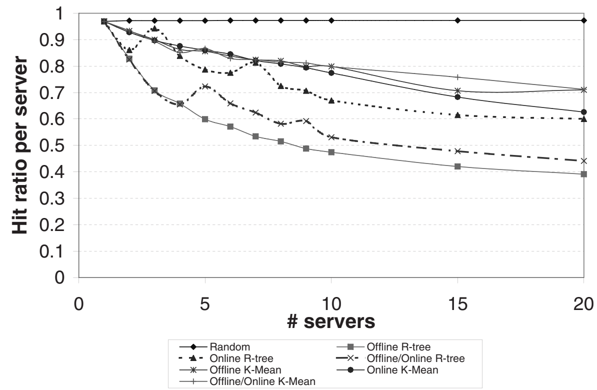


Fig. 3. Performance of different partitioning algorithms as a function of the number of servers.

R-tree outperforms the corresponding online R-tree in all cases. Second, the hit ratio tends to decrease as we use more detailed R-trees. For example, relative to the case of depth-1 R-tree, the use of precise summaries cuts down the hit ratio by 20% - 25%.

However, the decrease in hit ratio is not always monotonic. This is because the heuristic we use (i.e., route a subscription to the server that contains a leaf node that has the largest overlap with it) is sometimes sub-optimal. For example, server 1 may have two rectangles each overlapping 50% with the incoming subscription A, but the union of the two rectangle overlaps with A completely, while server 2 has one rectangle overlapping 80% with A. According to the heuristic, we would route A to server 2, but this is not as good as routing it to server 1, which incurs no extra traffic.

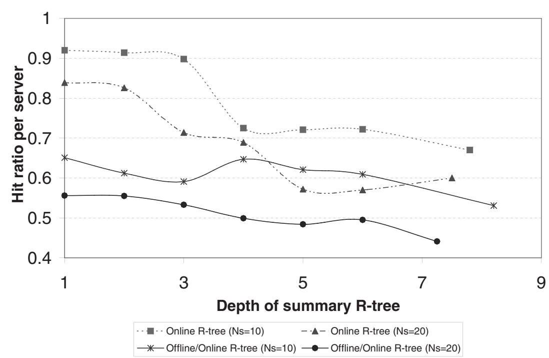


Fig. 4. The effect of subscription routing summary precision.

Zipf-like subscription/event distribution: To examine the impact of subscription distribution, below we compare the algorithms by generating subscriptions whose boundary values follow a Zipf-like distribution.

A number of studies [3], [7], [21] report that the Web requests follow a Zipf-like distribution; that is, the number of requests to the i-th popular document is proportional to $\frac{1}{i^\alpha}$, where α is a small constant. We analyzed the end-user subscriptions at a major stock-quote notification service provider, and also observed a Zipf-like distribution [28]. Based on the previous findings, it is likely that subscriptions in the form of range predicates may exhibit a similar distribution.

We use the following approach to generate subscriptions with range predicates that follow a Zipf-like distribution. Since the Zipf-like distribution is only applicable for discrete values, we first discretize values in each dimension to B bins; then for subscriptions with n attributes, we have B^n bins, where each bin is numbered as $x_1 x_2 .. x_d$, with x_i representing the value of the bin in the i-th dimension. Without loss of generality, we assign popularity ranking to these bins according to their numerical order (e.g., for a 2 dimensional space, the bins in the order of decreasing popularity are 00, 01, 02, ... , 10, 11, 12, ...). This ranking order assumes that some dimensions have greater impact on popularity than other dimensions, which is a reasonable assumption in practice. Next we assign the popularity distribution to the bins according to $\frac{1}{i^\alpha}$, where α is varied from 0.25 to 4. Then we draw the boundary values of each subscription from the above distribution. Specifically, we pick two bins, denoted as $x_1 x_2 .. x_d$ and $x'_1 x'_2 .. x'_d$. Then the new subscription's lower and upper bounds in the i-th dimension are defined by $min(x_i, x'_i)$ and $max(x_i, x'_i)$, respectively.

Figure 5 shows the results when subscriptions follow a Zipf-like distribution and events are still uniformly distributed; and Figure 6 shows the results when subscriptions and events both follow a Zipf-like distribution with the same value of α but different popularity ranking of the bins. [2]

We make the following observations. First, in both cases the event traffic decreases with increasing α for all partition algorithms. This occurs because as α increases, subscriptions are more concentrated around certain area, and it becomes more difficult for events with a different popularity distribution to find matches, thereby reducing the hit ratio. Second, when α increases, the performance of K-mean improves. This is because rectangles' lower and upper bounds are drawn from the same popularity distribution, and an increase in α makes the rectangle boundaries closer together, resulting in a small volume. This increases the effectiveness of K-mean, which uses the centriods' positions for subscription routing. Finally, the random algorithm performs much worse than the others. For example, it yields 3 - 5 times as much traffic as the offline R-tree. The clustering algorithms, such as offline R-tree, achieve a higher traffic reduction rate when subscriptions follow a Zipf-like distribution because when the rectangles are clustered together and have smaller volume, the effectiveness of clustering is improved.

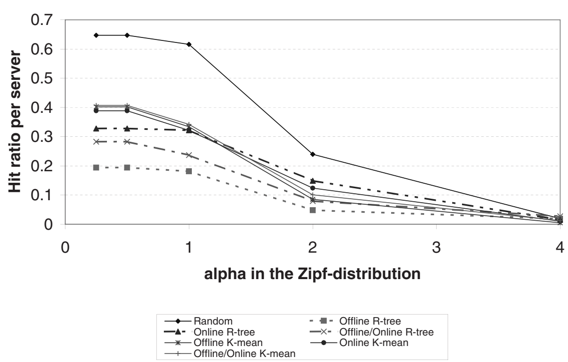


Fig. 5. The effect of Zipf-like subscription distributions, when subscriptions follow a Zipf-like distribution and events are uniformly distributed ($N_s = 10$).

Other Results: As mentioned earlier, subscription routing takes load balancing into account by routing subscriptions only

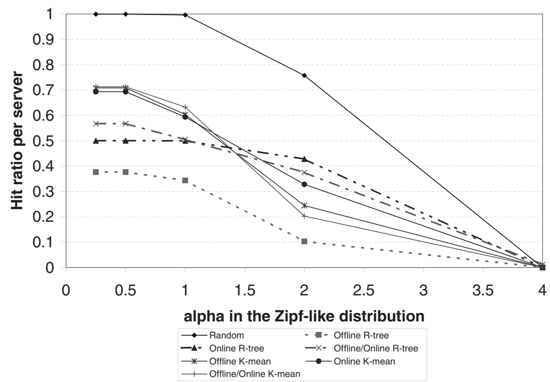


Fig. 6. The effect of Zipf-like subscription distributions when both subscriptions and events follow a Zipf-like distribution with the same value of α, but different popularity ranking of the bins ($N_s = 10$).

to those servers whose load is below a certain threshold. The threshold can be used to control the degree of load balancing, so we call it the *load balancing factor*, which is defined as the ratio of the maximum allowable server load to the average server load. We see a clear trade-off between load balancing and the hit ratio: as we relax the load balancing constraint by increasing the load balancing factor, the hit ratio drops monotonically because we are more likely to be able to route a subscription to the server closest to it in the event space rather than route it to an alternative server in order to avoid overloading the closest server.

We also study the effectiveness in traffic reduction as a function of the percentage of subscriptions used to build an R-tree offline. In general, the hit ratio decreases as we increase the percentage because more subscriptions are clustered better. For example, relative to the online R-tree data point, the hit ratio is reduced by 7%, 20% and 30% when 30%, 50%, and 100% subscriptions are used offline, respectively.

Finally, we vary the number of dimensions from 2 to 10, and observe consistent results with those from the 4-dimension case. The offline R-tree algorithm continues to reduce traffic by 30% - 50% compared to random partitioning, and online/offline R-tree performs close to offline R-tree, within 10% difference in hit ratio.

Summary: To summarize, our simulation results suggest that R-trees are most effective in reducing event traffic. So we will focus on the R-tree-based routing for the rest of the paper.

D. Extension to Widely Distributed Servers

So far, we have considered the case where servers are physically co-located. We can generalize our clustering scheme to handle widely distributed servers as follow.

First, when the paths from the dispatcher to different servers have different network distance, we can change the cost function used in the offline R-tree algorithm to be a weighted sum of traffic and network distance. That is, we choose the cut that minimizes the sum of the volumes of the bounding rectangles of the two partitions on the two sides of the cut weighted by the network distance. Similarly, in the online R-tree algorithm, a new subscription is forwarded to the server that incurs least additional traffic weighted by the network distance, where the additional traffic incurred at a server is approximated by $min_i\, |NewSubscription \setminus BoundRectangle_i|$, and i's are indices for the bounding rectangles at the server.

[2] Since in practice the popularity rankings of events and subscriptions may not match, we generate the events using the same steps as we generate subscriptions except that we assign the bins with a random popularity ranking instead of using their numerical order.

Second, for performance reasons it is sometimes desirable to have servers handle subscriptions for nearby clients. In such cases, we can estimate similarity between different subscriptions not only based on subscription filters but also based on subscribers' locations. This can be achieved in our system by considering the subscriber's location (e.g., its latency towards a set of landmarks [22]) as additional dimensions for clustering. Subscription servers can then be seeded with an affinity towards specific coordinates in a given dimension. Clustering based on both subscription filters and subscribers' location is likely to produce clusters with good locality, since subscriptions often exhibit spatial locality as shown in [1].

IV. Throughput Optimization

In addition to event traffic reduction, overall system throughput is another important performance metric to optimize. Higher system throughput would mean that the event distribution network can process incoming events at a higher rate, improving the scalability on the publisher side. Although event routing based on precise summaries minimizes event traffic, it does not necessarily optimize system throughput. Since the size and complexity of precise summaries are determined by the characteristics of the subscription filters and are not otherwise controllable, it is possible that the dispatcher may become the bottleneck and the system throughput is negatively impacted. The likelihood increases when the dispatcher serves a large number of servers, and precise summaries match only a small percentage of overall event traffic.

We propose the use of imprecise summaries as a mechanism for balancing the load between the dispatcher, the network links, and servers to optimize system throughput. The intuition is that, if event routing with precise summaries causes the dispatcher's CPU to become the bottleneck of the overall system, switching to less precise summaries would speed up event routing and allow the dispatcher to accept more events per second, enhancing system throughput. This is, however, only true when the dispatcher remains the bottleneck. Obviously, if either the dispatcher's input link or the publishers become the bottleneck, there is no incentive for the dispatcher to further reduce summary precision to speed up event routing. If either the dispatcher's outgoing link or the servers become the bottleneck, further reducing summary precision would generate more false positive traffic and cause more congestion at the bottleneck, degrading overall system throughput.

In this section, we first study system throughput using analysis and simulation, and then describe a self-tuning algorithm to dynamically adjust summary precision for throughput optimization. Note that while our simulation and implementation (in the next section) use rectangle subscriptions as an example to demonstrate the benefit of a tunable summary precision for routing, the idea is more general, and can be potentially applied to other forms of subscriptions.

A. Throughput Analysis and Simulation Study

We analyze the impact of imprecise summaries on system throughput in Appendix . In this section, we summarize the key results and equations that will be used in the simulation study.

Referring to Figure 1, let TP_D, TP_{OL}, and TP_S denote the individual throughput components of the dispatcher CPU, the outgoing link, and the servers, respectively. These components and the overall system throughput TP can be calculated as:

$$TP_D = 1/(T_d \cdot N_s)$$
$$TP_{OL} = 1/(R \cdot T_l \cdot N_s)$$
$$TP_S = 1/(R \cdot T_s \cdot F)$$
$$TP = min(TP_D, TP_{OL}, TP_S)$$

where T_d is dispatcher's average per-server per-event routing time, N_s is the total number of servers receiving event messages from the dispatcher, R is the average hit ratio per server, T_l is the average per-message transmission time by dispatcher's outgoing link, T_s is the average per-event processing time on a single-node filtering engine, and F is the server's load expansion factor as defined in the Appendix.

Let $TP' = min(TP_{OL}, TP_S)$; that is, TP' is the throughput bottleneck downstream from the dispatcher CPU. If $TP' < TP_D$ for all summary precisions, then precise summary offers the optimal throughput. If $TP' > TP_D$ for all summary precisions, then the no-summary operating point is the optimal one. In the remaining case, the TP' and TP_D curves intersect and the summary precision corresponding to the intersection provides the optimal throughput. If $TP' = TP_{OL}$, the optimal *Relative ThroughPut (RTP)* equations with respect to the precise-summary and no-summary operating points are

$$RTP_p^o = \frac{T_{dp}}{R_o \cdot T_l},\ and \tag{1}$$

$$RTP_n^o = \frac{1/(R_o \cdot T_l \cdot N_s)}{1/(T_l \cdot N_s)} = \frac{1}{R_o}, \tag{2}$$

respectively, where R_o is the average hit ratio per server at the optimal operating point.

We use simulations to study the effect of the number of subscriptions, the number of dimensions, and subscription partitioning algorithms on the optimal relative throughput achievable by imprecise summaries. The impact of actual systems and network parameters on the enhancement of absolute throughput will be discussed in the implementation section. Here we assume that all communication links have a bandwidth of 100Mbps or roughly 10MBps, and that the average size of each event message is 1KB. This translates into $T_l = \frac{1K}{10M}$ = 100 microseconds per message. We present simulation results only for the case of $TP' = TP_{OL}$; results for the case of $TP' = TP_S$ are similar. The main performance metrics are therefore the two optimal relative throughput RTP_p^o and RTP_n^o defined in Eq. 1 and Eq. 2, respectively. In our experiments, all events and subscriptions have 8 dimensions, and the values on each dimension are uniformly distributed between 0 and 10, unless otherwise specified.

Varying the number of rectangles: Figure 7 shows the relative throughput as a function of *summary precision*, which is represented as the ratio between the number of bounding rectangles and the total number of subscription rectangles. Each curve corresponds to a different number of subscription rectangles and is normalized by its own precise-summary throughput, which is at the rightmost of the curve. As it shows, initially when the system's bottleneck is at the outgoing link, increasing the summary precision reduces the amount of out-going traffic and alleviates the bottleneck, thereby improving the overall throughput. On the other hand, an increase in the summary precision also puts a higher load on the CPU. At some point when the CPU becomes a bottleneck, a further increase in summary precision aggravates the bottleneck, and decreases the overall throughput. The peak of each curve marks the intersection of the TP_{OL} and TP_D curves (i.e., when the throughput of dispatcher CPU equals to that of outgoing link), and thus the optimal operating point, and its Y-axis value gives the RTP_p^o measure. The RTP_n^o measure can be calculated based on the hit ratio R_o shown in the legend. (R_p is the average hit ratio per server when precise summaries are used.)

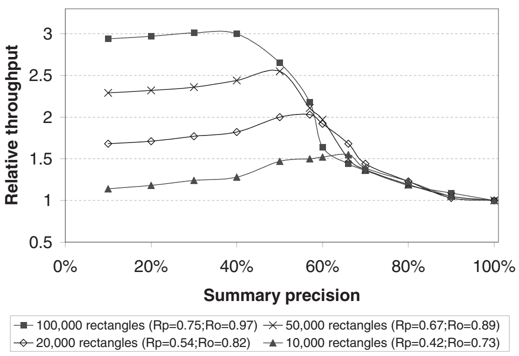


Fig. 7. System throughput for different number of rectangles.

As the number of rectangles increases, both the number of maximal rectangles and the hit ratio R increase, which together cause the average precise-summary routing time T_{dp} to increase. As a result, RTP_p^o increases from 1.55 for 10,000 rectangles to 3.01 for 100,000 rectangles, a 200% increase in system throughput over precise summaries. This demonstrates that, compared to precise summaries, imprecise summaries are especially useful when the number of subscriptions is large.

On the other hand, the optimal relative throughput with respect to the no-summary operating point is $RTP_n^o = \frac{1}{R_o}$ and the value drops from 1.37 to 1.03 as the number of rectangles increases. Fortunately, Figure 8 shows that this negligible 3% throughput gain (for 100,000 rectangles) is not inherent as we scale up to a large number of subscription rectangles. When we limit the volume of each rectangle and thus reduce the hit ratio, the use of imprecise summaries can still increase system throughput by 67% ($\frac{1}{0.60} - 1$) compared to no summaries, as shown by the lower curve in Figure 8. The shape of the curve in fact resembles the lower curves in Figure 7.

Varying the number of dimensions: Figure 9 studies the effect of the number of dimensions on relative throughput while keeping a constant number of 10,000 subscription rectangles. As the number of dimensions increases, the hit ratio R_p decreases because the volume of the event space increases exponentially, while the routing time T_{dp} increases due to a higher overhead incurred in checking if a point belongs to a rectangle. As a result, the dispatcher's CPU is more likely to be the bottleneck, and RTP_p^o tends to increase as the number of dimensions increases, as shown by the two upper curves. In con-

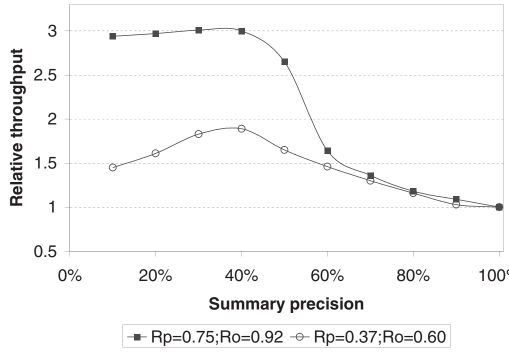


Fig. 8. Effect of hit ratios on optimal throughput gain (100,000 rectangles).

trast, the two lower curves illustrate the cases where imprecise summaries are not useful: the low route time and high hit ratio result in $TP_{OL} < TP_D$ for all summary precisions, and so precise summaries provide the optimal throughput.

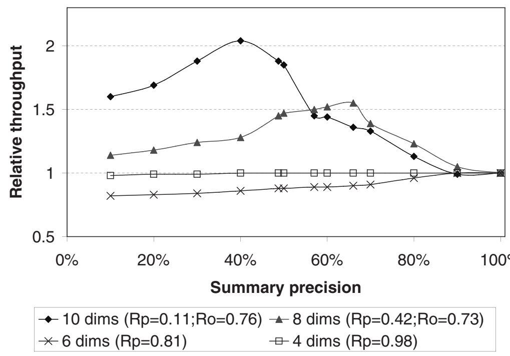


Fig. 9. System throughput for different numbers of dimensions.

Subscription locality and partitioning: The results presented so far have assumes that randomly generated subscription rectangles are randomly partitioned among the servers. We next evaluate the effect of subscription locality and partitioning on system throughput. We simulate locality by generating the same random subscriptions within a subspace of the event space. The results (not shown here) are very close to the following analytical results. Suppose the ratio of the subspace volume to the entire event space volume is x. Clearly, the hit ratio R will drop by that ratio. The routing time T_d also drops by approximately the same ratio because events within the subspace are routed using the same R-tree (with all rectangles proportionally smaller) and those outside the subspace can be filtered out very quickly by the top-level bounding rectangle. The absolute throughput thus increases by a ratio of $\frac{1}{x}$, but the optimal operating point stays the same and so RTP_p^o remains unchanged. The relative throughput RTP_n^o increases by a ratio of $\frac{1}{x}$.

A very similar behavior is observed when random subscriptions are partitioned offline using a bulk-loading R-tree algorithm. Figure 10 shows the relative throughput for 100,000 rectangles on 10 servers with and without offline partitioning. The top two curves (with square points) are normalized against the same precise-summary throughput value to illustrate the benefit of offline partitioning in terms of increased system throughput. In this case, offline partitioning reduces the routing time T_d by approximately 50% and hence doubles the throughput. The bottom curve is constructed by normalizing the top curve against its own precise-summary throughput. Since offline partitioning also reduces the hit ratio R by approximately 50%, this curve matches the without-partitioning curve very well, which indicates that imprecise summaries remain effective in further enhancing the already increased system throughput.

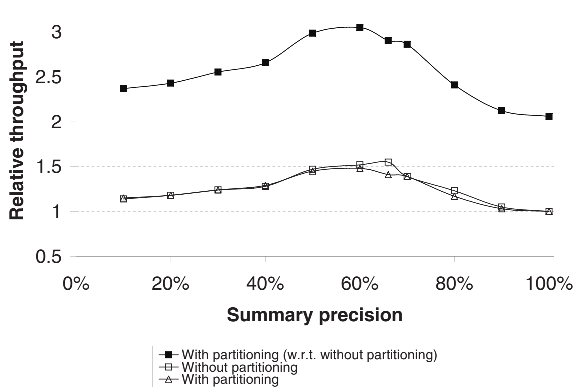


Fig. 10. Effect of subscription partitioning on system throughput.

B. Self-tuning

In addition to demonstrating the benefits of imprecise summaries for throughput optimization, the analysis and simulation results presented so far can also be used to perform offline selection of optimal summary precision, if a sufficient number of representative subscriptions are available. However, if those are not available or if the optimal operating point shifts due to changes in the event or subscription distributions or in network conditions, the system must be able to automatically determine the (new) optimal operating point and self-adapt using the optimal summary precision for routing. We describe such a self-tuning algorithm in this subsection.

The main idea behind the algorithm is based on the following observation: if the dispatcher's CPU is the bottleneck, we can move towards the optimal operating point, OPT, by reducing summary precision; otherwise, either the dispatcher's outgoing link or the servers are the bottleneck and we can move towards OPT by increasing summary precision. Instead of measuring the various parameters used in the throughput equations, our self-tuning algorithm simply relies on the dispatcher to monitor its own CPU load and declare itself the bottleneck if and only if it is fully loaded. In practice, the dispatcher periodically samples its CPU usage and compares the average CPU load during the last time interval (e.g., 1 minute) with a threshold (e.g., 96% usage). A threshold lower than 100% is necessary to account for measurement errors and small fluctuations in CPU load.

The following pseudo-code describes our self-tuning algorithm. When either the (variable) timeout t occurs or a significant change in system throughput is detected, the dispatcher increments the self-tuning step index p, changes the summary precision by x in the direction discussed previously, and invokes the $update\_t()$ function. Depending on the supported granularity of x (which may be determined based on implementation efficiency and complexity [3]), the limited number of operating points may force the algorithm to oscillate around the optimal point. The $update\_t()$ function tries to capture the fact that the best achievable operating point has been reached by detecting the number of repeated oscillations exceeding a threshold (e.g, 3 in the pseudo-code). When that happens, the timeout value t is increased to $n * t$, $n \geq 1$, (subject to an upper bound of $maxT$) to dampen the oscillation. The factor n provides a trade-off between stability and adaptivity: a smaller n would allow the system to detect changes in optimal operating point faster, achieving greater adaptivity; this would however cause the oscillation to happen at a higher frequency, and thus less stability.

[3]To improve adaptivity, we can use a larger step size when we are farther away from the optimal point, and use a smaller step size when we get closer to the optimal.

```
Self-tuning algorithm
wait for timeout t or throughputChangeTrigger
p++;
if (dispatcher CPU-bound)
    decrease summary precision by x
else
    increase summary precision by x
end
update_t();
```

```
function update_t()
if (p < 3)
    t = 1;
else if (precision[p] = precision[p − 2] and
    precision[p − 1] = precision[p − 3])
    if (precision[p] > precision[p − 1])
        numOscillations++;
        if (numOscillations >= 3 and n ∗ t ≤ maxT)
            t = n ∗ t;
            numOscillations = 0;
        end
    end
else
    numOscillations = 0;
    t = 1;
end
```

V. An XML-based Implementation

We next describe an implementation of summary-based subscription and event routing in an XML-based Web Services framework. The framework implements a set of SOAP-based protocols that are being proposed to W3C as standards [30].

A. Design & Implementation

To allow experimentations with different algorithms, we first modified the framework to support extensibility at various points of the system, and then plugged in our routing and matching modules through the extensibility mechanism.

Figure 11 illustrates the high-level software architecture of the extensible framework as well as EDN-specific components (in shaded boxes). The Messaging Layer provides the infrastructure for sending and receiving XML messages between Web Services endpoints. The Namespace Binding Layer maintains a hierarchical namespace (for event topics or routing table entries) and associates each name entry with a matcher class that would be instantiated to store the filters contained in messages sent to that name. When a new route or topic entry is created in the namespace, the creation message has the option of specifying a URI that identifies the matching engine to use for handling any filter operations associated with the topic/route. The default matcher class in the framework is the standard XPath filter matcher, which is also used by the Messaging Layer for message dispatching.

The Base Route Manager registers a handler with the Messaging Layer to receive all incoming messages, and uses a namespace layer instance and the matcher instances associated with its name entries to make routing decisions. It also registers another handler to receive route administration messages for creating, deleting, and enumerating route information. The Base Subscription Manager registers a handler to receive all messages related to topic management, subscriptions, and event publications. In addition to supporting pluggable matcher and namespace implementations, the extensibility also allows applications to extend the base classes in order to add custom XML elements to base XML administrative messages and to override or include additional logic in route or topic management.

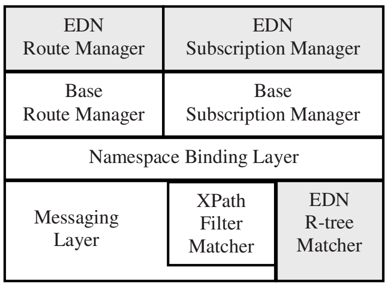


Fig. 11. Software architecture of extensible Web Services framework.

In an R-tree-based EDN system, the dispatcher runs an extended Route Manager with namespace entries associated with the *RTreeRouteSet* class (instead of the default XPath filter matcher class); an *RTreeRouteSet* instance holds the set of routing R-trees, one from each server, matches an incoming event against all the trees, and returns a list of servers that have a match. Each EDN server runs an extended Route Manager for random, R-tree, or K-Mean-based subscription routing; it also runs an extended Subscription Manager with namespace entries associated with the *RTreeMatchingEngine* class. An *RTreeMatchingEngine* instance uses an R-tree matcher as the single-node filtering engine and another R-tree matcher as the summary manager that maintains the maximal R-tree. When the server's routing R-tree is changed due to the insertion of new subscriptions, the updated R-tree is sent to the dispatcher's Route Manager as a route update message. Alternatively, an *RTreeMatchingEngine* instance can use the XPath filter matcher as the filtering engine and extract the "most distinguishing" range predicates from each new subscription for use in the R-tree summary manager; that is, it allows the use of different formats for filtering and routing.

We have set up a network of 10 servers for running experiments. All of them are 1.7GHz Pentium 4 PCs with 1GB of memory. The results on subscription partitioning with offline R-trees matched the simulation results. The hit ratios from the online R-tree and offline/online R-tree experiments differed slightly from the simulation results due to the nondeterministic relative timing between subscription arrivals and summary updates. We will focus on the system throughput issues in the remainder of this section.

B. When is Imprecise Summary Useful?

A key question to ask is: *"in practice, when is imprecise summary useful, and how to choose the precision to optimize system throughput?"* We answer the question in three steps. First, we extend the dispatcher CPU-bound throughput equation to accommodate various system parameters from actual implementations. Next, we show that imprecise summaries indeed provide throughput gains for our Web Services-based implementation, and the actual throughput measurements closely match the calculated curves from the equations. We then use the calculated curves to explore the *"what if"* scenarios as we vary system parameters such as messaging overhead, network bandwidth, and CPU speed.

The throughput equation $TP_D = \frac{1}{T_d \cdot N_s}$ assumes that messaging overhead is negligible with respect to routing overhead. This may not be true in XML-based implementations, which try to provide self-describing messages and interoperability at the cost of increased message size and message processing overhead. We extend the equation as follows:

$$TP_D = \frac{1}{T_f + (T_d \cdot N_s) + T_c \cdot (R \cdot N_s)} \quad (3)$$

where T_f is the per-event fixed overhead for the receiving and sending operations, and T_c is the per-copy sending overhead. The network-bound throughput equation is re-expressed in terms of the network bandwidth B and average message size S_m as follows.

$$TP_{OL} = \frac{B}{R \cdot S_m \cdot N_s} \quad (4)$$

In our base-line implementation, we have $T_f = 1.6ms$, $T_c = 0$ (with an efficient multicast repeater), and $S_m = 3KB$. In our experimental setup, we have $N_s = 10$, CPU speed of 1.7GHz, and achievable network bandwidth $B = 80Mbps$ on a 100Mbps link. The route time T_d and hit ratio R are obtained from offline simulations. (Note that T_d and R can only be obtained when a set of existing subscriptions are available. In the absence of such information, we will rely on the self-tuning algorithm to automatically determine the optimal summary precision, as will be demonstrated in the next subsection. Also note that we will focus on the above two throughput equations in the remainder of this section; server CPU-bound scenarios that involve TP_S have similar behaviors.)

The "TP-Measured" curve in Figure 12 illustrates the actual throughput gain measured from the system. The optimal summary precision is around 50%, and the optimal throughput is 438 messages/sec, which is 31% and 27% higher than the precise-summary and no-summary throughput, respectively. It is clear from the figure that the "TP-Measured" curve closely matches the calculated curve, which is the minimum of the "TP_OL" curve (i.e., Eq. (4) with $B = 80Mbps$) and the "TP_D-1.6ms" curve (i.e., Eq. (3) with $T_f = 1.6ms$).

The other two curves in Figure 12 study the throughput behavior as the per-event fixed overhead T_f changes and the "TP_OL" curve remains the same. The "TP_D-0.2ms" curve represents, for example, the case when a binary wire format becomes a standard and significantly reduces the messaging overhead. This would push the CPU-bound throughput curve upwards and move the optimal operating point towards precise summary. In contrast, as T_f increases due to, for example, message authentication and integrity checking overhead, the optimal operating point would move towards no summary. In the extreme case where T_f is so high that the CPU-bound curve no longer intersects the network-bound curve, imprecise summaries would no longer be useful and the dispatcher should multicast every event to all servers without performing any summary-based routing.

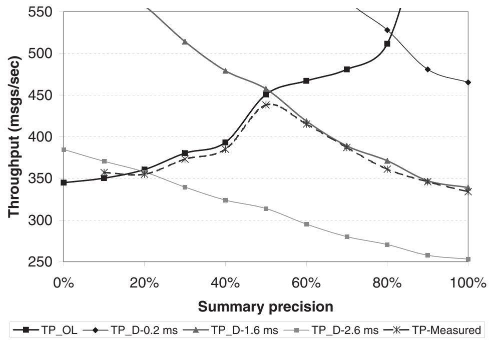


Fig. 12. System throughput as per-event overhead at the dispatcher changes.

Figure 13 examines the effect of available network bandwidth on system throughput, while keeping the CPU-bound curve unchanged (which is the same as the "TP_D-1.6ms" curve in Figure 12). The figure shows that, as network bandwidth decreases from 100Mbps, 80Mbps, 40Mbps, to 10Mbps, the optimal operating point moves towards the right. The "TP_OL-10Mbps" curve no longer intersects the "TP_D" curve, making the precise summary the optimal operating point. Figure 13 demonstrates the importance of self-tuning: even when the distributions of events and subscriptions remain stable, changes in network conditions may affect the available bandwidth and cause the optimal operating point to drift. The system must be able to adapt to such changes to maintain optimal throughput.

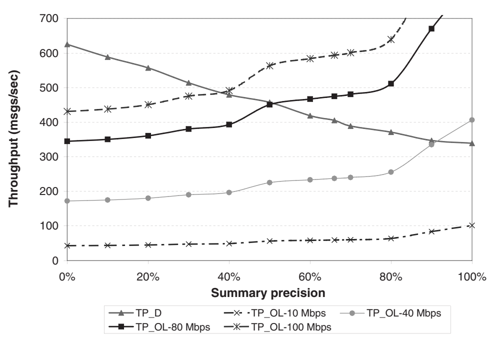


Fig. 13. System throughput as available network bandwidth changes.

We next study the throughput behavior as the (single-processor) dispatcher CPU speed changes. The lowest curve in Figure 14 shows that, with the XML messaging layer that we are currently using and with 80Mbps available network bandwidth, a 800MHz dispatcher would have been unable to catch up with the network and shifted the optimal operating point to the far left and render summary-based routing not useful. In contrast, a 3GHz dispatcher will move the CPU-bound throughput curve upwards. This allows the system to scale up to handle event schemas with more attributes and hence higher routing time T_d, which would bring the CPU-bound throughput curve back down and be able to benefit from the enhancement provided by imprecise summaries.

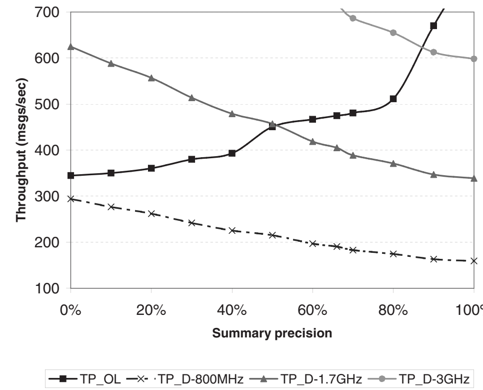


Fig. 14. System throughput as dispatcher CPU speed changes.

Finally, the two lower curves in Figure 15 illustrate the effect of non-trivial per-copy sending overhead T_c on system throughput. This represents the case where either the messaging layer does not optimize for multicast sending, or the application requires different processing for events forwarded to different servers (such as encrypting each message with a different key). In addition to increasing per-event processing time and hence lowering the throughput curve, a higher T_c has a secondary effect that decreases the slope of the curve: if a significant portion of the time that we save from routing with a lower precision is "wasted" on the T_c for forwarding the additional false-positive traffic, the benefits of imprecise summaries may be greatly reduced.

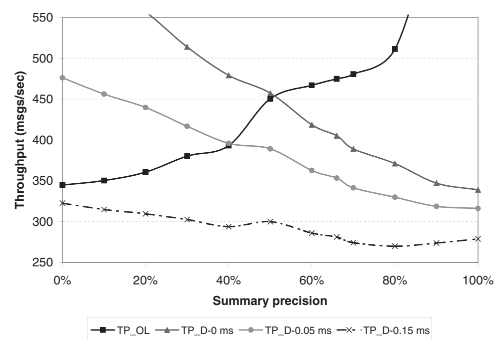
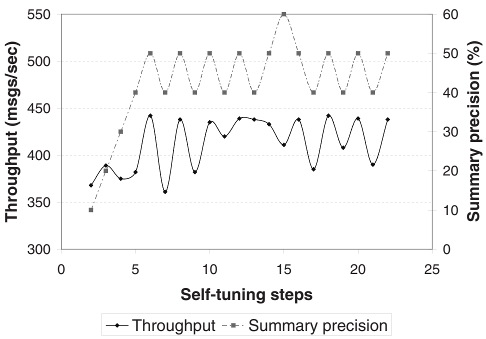


Fig. 15. System throughput as per-copy overhead in multicast changes.

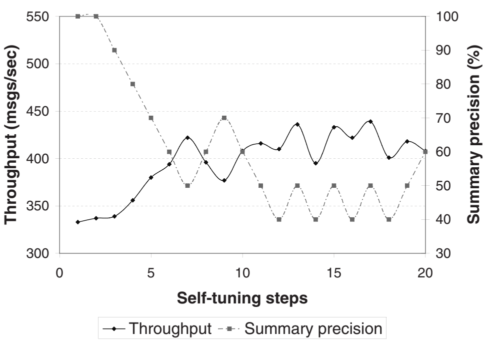
C. *Self-tuning Experimental Results*

We illustrate the self-tuning behavior of our system with two sets of experiments, both of which use the setting corresponding to the "TP-Measured" curve in Figure 12 as the basis. We set the granularity x of summary precision change to 10%, and use 96% as the threshold for determining whether the dispatcher is CPU-bound. For simplicity, we chose $n = 1$ in the self-tuning algorithm, which corresponds to not dampening oscillations. (In the final version, we will include the results using the dampening optimization described in Section IV-B.)

In the first set of experiments, we started the system with 10% and 100% precisions, and expected the self-tuning algorithm to bring the system to the optimal operating point of around 50% precision. Figure 16 illustrates the throughput and summary precision behavior for the two experiments. When started with a 10% precision, the system was quickly brought to the 50% point and stayed around the 40% and 50% precisions. Occasionally, the CPU usage at the 50% point dropped below the threshold and prompted the algorithm to try the 60% point to see if the optimal operating point had shifted (e.g., Steps 14 and 15). Since we did not vary any system or network parameters in this experiment, the algorithm correctly switched back to the near-optimal precision points. The second experiment starting with a 100% precision, shown in Figure 16(b), took a bit longer to converge. Between Steps 6 and 7, a transient condition falsely suggested that the optimal operating point might be between 60% and 50%. Once that transient condition disappeared, the algorithm correctly stabilized around the 40% and 50% precisions.



(a) Starting from 10% precision



(b) Starting from 100% precision

Fig. 16. Self-tuning system throughput and summary precision.

In the second set of experiments, we changed the event distribution from uniform to a normal distribution with a mean of 9.5 and a standard deviation of 0.25, and expected the self-tuning algorithm to bring the system to a new optimal operating point. As shown in Figure 17, the system was started with a 10% precision and brought to the near-optimal precisions of 40% and 50%; then, the event distribution was changed around Steps 13 and 14. The disturbance triggered the algorithm to search for a new optimal point, first incorrectly towards higher precisions and then in the reversed direction. Eventually, the algorithm stabilized around the 10% and 20% summary precisions, achieving a higher throughput than that at the previous optimal operating point.

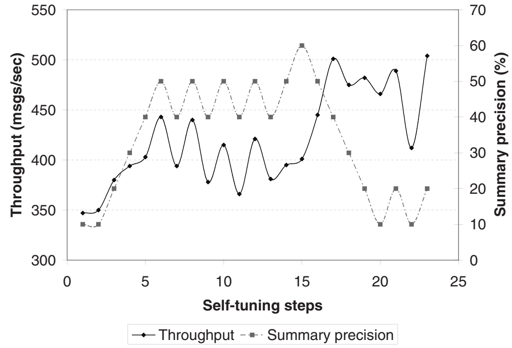


Fig. 17. Self-tuning system throughput and summary precision in response to changing event distribution.

VI. Related Work

Related work on pub/sub systems can be classified into four categories: (1) commercial products such as TIBCO's TIB Rendezvous, Talarian's SmartSockets, etc.; (2) interface standards in CORBA and Java; (3) single-node subscription filtering algorithms such as [14], [2], [11], which are orthogonal to our work; (4) prototype event notification services such as Elvin [25], Gryphon [4], Hierarchical Proxy Architecture [31], Ready [15], SCRIBE [24], and SIENA [9].

The EDN summary-based routing was inspired by the quench expressions in Elvin, the hybrid matching schemes in Ready, the vector annotation in Gryphon, the filters poset in SIENA, and the subscription merging in Hierarchical Proxy Architecture. Similar ideas have also appeared outside the pub/sub literature [26]. The more recent SIENA fast forwarding paper [8] described a fast content-based forwarder, which performs particularly well when the total number of distinct attribute names is large and the number of attributes per message is relatively small.

Almost all of these work, however, couple the subscription paths tightly with the notification paths (e.g., use reverse path forwarding to set up forwarding table). In comparison, we focus on event notification systems in which the subscription paths are completely decoupled from the notification paths. The latter architecture is commonly used in many commercial pub/sub systems. Under this new architecture model, we generalize the existing proposals by supporting imprecise summaries to optimize an additional performance metric – system throughput.

C. Y. Chan *et al.* [10] proposes a new index structure for regular expressions, called RE-tree. RE-trees is similar in spirit to R-trees, but handle regular expressions rather than multi-dimensional rectangles. The notion of imprecise summary is also useful to RE-trees, when they are used in a distributed fashion.

There have been several recent papers on using similarity-based clustering in the pub/sub setting. All of them are concerned with reducing the total number of required IP multicast addresses for notification delivery, while the EDN subscription partitioning algorithms aim at enabling compact summaries for event traffic reduction, before notifications are generated. The Group Approximation Algorithm described in [20] tries to combine actual multicast groups into approximate groups, while introducing the least amount of false-positive traffic. For multi-party applications, [29] proposed using the k-means method to group subscribers with similar sets of publishers that they are interested in, so as to minimize overall wasted event traffic. The grid-based clustering framework in [23] partitioned the event space into cells, and associated a feature vector with each cell to indicate the set of subscribers interested in events falling into that cell. The cells are then clustered to minimize the expected waste of event traffic. The paper briefly mentioned the use of R-trees to map an event to a multicast group.

Finally, in [28] we have examined how to partition equality predicates to reduce event traffic.

VII. Conclusions & Future Work

In this paper, we propose a summary-based routing framework for content-based event distribution.

To make summary compact and effective, we cluster subscriptions based on their similarity. Our evaluation shows that for range predicates, the R-tree algorithms are most effective: compared to random partition, the R-tree algorithms cut down event traffic by 20% to 60% when subscriptions follow a uniform distribution, and by up to 5 times when subscriptions follow a Zipf-like distribution.

To optimize system throughput, we propose using imprecise summary, which generalizes previous work on precise summaries and provides an important trade-off between routing overhead and false-positive traffic. Simulation results showed that summary-based routing could increase throughput by up to 200% and 67% compared to the uses of precise summaries and no summaries, respectively.

To assess the practical benefits of summary-based routing in the context of Web-based eventing, we have implemented the proposed techniques in an XML/SOAP-based framework that conforms to the proposed Web Services standards. Performance measurements from the actual implementation largely validated our throughput analysis, which was extended to accommodate various implementation parameters.

To help system engineers determine whether imprecise summaries are useful for their particular implementations, we give a detailed discussion on the behavior of system throughput as messaging overhead, available network bandwidth, and CPU speed varies. Most importantly, we have designed a self-tuning algorithm that can dynamically maintain optimal throughput by automatically adapting to changing operation conditions. Experimental results from the implementation confirm the self-tuning capability in practice. Although we have focused on range predicates in this paper, the general concept of self-tuning, imprecise summary-based subscription and event routing should be applicable to other types of publish/subscribe systems as well.

A number of avenues for future work remain. First, we are currently investigating another approach to subscription partitioning. To minimize the average number of servers that an event needs to reach, we can formulate the problem as a graph partitioning problem: each vertex represents a rectangle and the edge cost between two vertices reflects the penalty if the two rectangles reside on different servers. Given the number of servers N_s, the goal is to cut the graph into N_s clusters so that the sum of edge costs in the cut is minimized while avoiding overloading any server. This is a well-studied problem known as Capacitated Graph Partitioning [12] or Min-Cut Clustering [19]. There are good algorithms to achieve this, such as the multilevel Kernighan-Lin algorithm [17].

Second, we have focused on the distribution topology that has only one level of branching. It is interesting to extend summary-based routing for general distribution topologies.

Finally, so far we considered providing summaries for subscriptions after all the dimensions have been defined and subscriptions have been constructed in that space. An interesting direction for future work would be to consider how our summary mechanism can guide the definition of the space by offering system throughput as an additional criterion for evaluating design alternatives of the dimensions.

References

[1] A. Adya, P. Bahl, and L. Qiu. Characterizing alert and browse services for mobile clients. In *Proc. of USENIX Annual Conference*, Jun. 2002.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[3] M. F. Arlitt and C. L. Williamson. Internet Web Servers: Workload Characterization and Performance Implications. In *IEEE/ACM Transactions on Networking*, pages 631–645, 1997.

[4] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *International Conference on Distributed Computing Systems*, pages 262–272, 1999.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proc. 1970 ACM-SIGFIDET Workshop on Data Description and Access*, Nov. 1970.

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

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of INFOCOMM '99*, March 1999.

[8] A. Carzaniga, J. Deng, and A. L. Wolf. Fast forwarding for content-based networking. In *Technical Report CU-CS-922-01, Department of Computer Science, University of Colorado*, Nov. 2001.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[10] C. Y. Chan, M. Garofalakis, and R. Rastogi. RE-Tree: An efficient index structure for regular expressions. In *Proc. of VLDB '2002*, 2002.

[11] 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 *Proc. of SIGMOD Conference*, 2001.

[12] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey. The node capacitated graph partitioning problem: A computational study. In *Mathematical Programming*, 1996.

[13] Y.J. Garcia, M.A. Lopez, and S.T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *Univ. of Denver Computer Science Tech. Report #97-02*, 1997.

[14] J. Gough and G. Smith. Efficient recognition of events in a distributed system. In *Proc. of the 18 th Australasian Computer Science Conference*, Feb. 1995.

[15] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

[16] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD Conference*, 1984.

[17] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. of Supercomputing*, 1995.

[18] A. K. Jain and Dubes. Algorithms for clustering data. 1988.

[19] Weighted min cut. *http://riot.ieor.berkeley.edu/riot/Applications/WeightedMinCut/*.

[20] L. Opyrchal, M. Astley, J. S. Auerbach, G. Banavar, R. E. Strom, and D. C. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *Proc. of Middleware*, pages 185–207, 2000.

[21] V. N. Padmanabhan and L. Qiu. The Content and Access Dynamics of a Busy Web Site: Findings and Implications. In *Proceedings ACM SIGCOMM 2000*, August 2000.

[22] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of IEEE INFOCOM*, Jun. 2002.

[23] A. Riabov, Z. Liu, J. L. Wolf, P. S. Yu, and L. Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proc. of ICDCS*, 2002.
[24] A. I. T. Rowstron, A. M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *International Workshop on Networked Group Communication*, pages 30–43, 2001.
[25] B. Segall and D. Arnold. Elvin has left the building: A publish /subscribe notification service with quenching. In *Proc. of AUUG*, 1997.
[26] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-based content routing using xml. In *ACM Symposium on Operating System Principles*, Oct. 2001.
[27] Y. M. Wang, P. Bahl, and W. Russell. The SIMBA user alert service architecture for dependable alert delivery. In *IEEE Int. Conf. on Dependable Systems and Networks (DSN)*, 2001.
[28] Y. M. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. J. Wang. Subscription partitioning and routing in content-based publish/subscribe networks (Brief Announcement). In *Proc. of DISC*, Oct. 2002.
[29] T. Wong, R. H. Katz, and S. McCanne. An evaluation on using preference clustering in large-scale multicast applications. In *Proc. of INFOCOM*, pages 451–460, 2000.
[30] Web Services standards. *http://www.webservices.org/index.php/standards/*.
[31] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for internet-scale event services. In *Proc. of WETICE*, 1999.

APPENDIX

We use the following notation to analyze system throughput (referring to Figure 1):

- N_s: the total number of servers receiving event messages from the dispatcher.
- T_d: dispatcher's average per-server per-event routing time. It is calculated as the average per-event routing time of the summary-based router, divided by N_s.
- R: the average hit ratio per server (always less than one). It is calculated as the per-event average number of servers receiving events from the dispatcher, divided by N_s.
- T_s: average per-event processing time on a single-node filtering engine.
- F: server's load expansion factor. In addition to invoking the filtering engine to process each event, a server needs to generate and deliver notifications to all subscribers with a matching subscription. We use $T_s \cdot F$ to represent the overall per-event processing time at the server. The factor F is close to the minimum value of 1 if T_s dominates the processing time. This corresponds to the case where either a separate scalable notification delivery network is available and so the servers only need to send a copy of the event message and the IDs of the matching subscribers to the delivery network, or the application allows delaying notification deliveries until the system is less loaded, for example, during off-peak hours.
- T_{dp}, R_p and T_{sp}: the values of T_d, R, and T_s, respectively, when precise summaries are used.
- T_{do}, R_o, and T_{so}: the values of T_d, R, and T_s, respectively, at the optimal operating point.
- T_{dn} and T_{sn}: the values of T_d and T_s, respectively, when no summary is used. (Note that $R_n = 1$.)
- T_l: the average per-message transmission time from a communication link. Let B be the link bandwidth expressed as the number of bytes per second; let S_m be the average message size in terms of number of bytes. We have $T_l = \frac{S_m}{B}$. For ease of presentation, we will assume that all servers and the dispatcher have the same bandwidth, and hence the same T_l for all incoming and outgoing links. The analysis can be easily extended to accommodate different values of T_l for different links.
- T_p: the average time interval between two consecutive event messages sent to the dispatcher by the publisher(s). In the analysis below, we assume that all published events are sent to a single dispatcher. Multiple dispatchers can be used to further enhance throughput either by partitioning the set of events or by partitioning the set of servers among the dispatchers. The analysis would be similar.

Let TP_{IL}, TP_D, TP_{OL}, and TP_S denote the individual throughput components of the incoming link, the dispatcher CPU, the outgoing link, and the servers, respectively. The overall system throughput TP is then determined by the minimum of these four components, which are calculated as follows.

$$TP_{IL} = 1/max(T_p, T_l)$$

$$TP_D = 1/(T_d \cdot N_s)$$

$$TP_{OL} = 1/(R \cdot T_l \cdot N_s)$$

$$TP_S = 1/(R \cdot T_s \cdot F)$$

$$TP = min(TP_{IL}, TP_D, TP_{OL}, TP_S)$$

We would like to support as high event rate from the publisher as possible, so we can assume $T_p < T_l$. We are mainly interested in systems with $N_s \geq 10$ and $R \geq 0.1$, so we have $TP_{OL} = \frac{1}{R \cdot T_l \cdot N_s} \leq \frac{1}{T_l} = TP_{IL}$ and

$$TP = min(\frac{1}{T_d \cdot N_s}, \frac{1}{R \cdot T_l \cdot N_s}, \frac{1}{R \cdot T_s \cdot F})$$

Clearly, the first term (inverse of T_d) decreases monotonically and the second term (inverse of R) increases monotonically as summary precision increases. The third component has both a hit ratio and a time component T_s. We will show that it also increases monotonically with summary precision. Given a hit ratio R_2 corresponding to server processing time T_{s2}, suppose the hit ratio increases by x percent by using a lower summary precision, R_1. The new T_{s1} can be calculated as $\frac{T_{s2} \cdot 100 + T_{s'} \cdot x}{100+x} > \frac{T_{s2} \cdot 100}{100+x}$ where $T_{s'}$ is the average per-event processing time for those additional x percent of false-positive traffic. We then have $R_1 \cdot T_{s1} > \frac{R_2 \cdot (100+x)}{100} \cdot \frac{T_{s2} \cdot 100}{100+x} = R_2 \cdot T_{s2}$.

Let $TP' = min(TP_{OL}, TP_S)$, i.e., the minimum of the two monotonically increasing terms. We distinguish three cases. First, if $TP' < TP_D$ for all summary precisions, then we have $TP = TP'$ and precise summary offers the optimal throughput. In this case, the dispatcher's CPU is not the bottleneck even with precise summaries, so imprecise summaries are not useful. Second, if $TP' > TP_D$ for all summary precisions, then we have $TP = TP_D$ and the no-summary operating point is the optimal one. (When the route time T_d approaches zero, the messaging layer overhead for receiving and sending is no longer negligible. This will be discussed in more details in the implementation section.)

In the third case, the TP' and TP_D curves intersect and the summary precision corresponding to the intersection provides the optimal throughput. If $TP' = TP_{OL}$, the optimal operating point happens at $\frac{1}{T_{do} \cdot N_s} = \frac{1}{R_o \cdot T_l \cdot N_s}$. The optimal *Relative*

ThroughPut (RTP) with respect to the precise-summary operating point is

$$\begin{aligned} RTP_p^o &= \frac{Optimal\ throughput}{Precise\ summary\ throughput} \\ &= \frac{T_{dp}}{R_o \cdot T_l} \end{aligned} \tag{5}$$

(where the subscript "p" stands for "precise summary" and the superscript "o" indicates that TP_{OL} is considered), which would be more significant when T_{dp} is large and imprecise summaries can reduce it to T_{do} (which must be less than T_l because $R_o \leq 1$) while maintaining a low R_o.

The optimal relative throughput with respect to the no-summary operating point (i.e., $R = 1$) is

$$\begin{aligned} RTP_n^o &= \frac{Optimal\ throughput}{No\ summary\ throughput} \\ &= \frac{1/(R_o \cdot T_l \cdot N_s)}{1/(T_l \cdot N_s)} = \frac{1}{R_o} \end{aligned} \tag{6}$$

Note that, since we have $1 \geq R_o \geq R_p$, RTP_n^o would be insignificant if R_p is already close to 1.

Similarly, if $TP' = TP_S$, the intersection occurs at $\frac{1}{T_{do} \cdot N_s} = \frac{1}{R_o \cdot T_{so} \cdot F}$, and the optimal RTP's are $RTP_p^s = \frac{T_{dp}}{T_{do}}$ and $RTP_n^s = \frac{T_{sn}}{R_o \cdot T_{so}}$.