Publisher Placement Algorithms in Content-based Publish/Subscribe

Alex King Yeung Cheung and Hans-Arno Jacobsen
University of Toronto, Middleware Systems Research Group (MSRG.org)
Email: {cheung,jacobsen}@eecg.utoronto.ca

***Abstract*—Many publish/subscribe systems implement a policy for clients to join to their physically closest broker to minimize transmission delays incurred on the clients' messages. However, the amount of delay reduced by this policy is only the tip of the iceberg as messages incur queuing, matching, transmission, and scheduling delays from traveling across potentially long distances in the broker network. Additionally, the clients' impact on system load is totally neglected by such a policy. This paper proposes two new algorithms that intelligently relocate publishers on the broker overlay to minimize both the overall end-to-end delivery delay and system load. Both algorithms exploit live publication distribution patterns but with different optimization metrics and computation methodologies to determine the best relocation point. Evaluations on PlanetLab and a cluster testbed show that our algorithms can reduce the average input load of the system by up to 68%, average broker message rate by up to 85%, and average delivery delay by up to 68%.**

Keywords: Content-based publish/subscribe, publisher migration, publisher repositioning, delivery delay minimization, load minimization

I. Introduction

Many filter-based publish/subscribe systems assume that publishers and subscribers join the broker federation by connecting to the closest broker [1], [2], [3] or to any broker with no restrictions [4], [5], [6], [7], [8]. The former assumption may minimize the transmission delay between the client and broker, and the latter may provide more freedom of choice for the client. Regardless, both policies introduce an unpredictable number of overlay network hops between the publisher and subscriber that may hinder system performance and result in high delivery delays. This problem is particularly important in commercial publish/subscribe systems that cannot tolerate server overloads and unexpected response times such as GooPS [9], Google's publish/subscribe system that integrates its web applications; SuperMontage [10], Tibco's publish/subscribe distribution network for Nasdaq's quote and order-processing system; and GDSN (Global Data Synchronization Network) [11], a global publish/subscribe network that allows suppliers and retailers to exchange timely and accurate supply chain data. Reducing in-network processing and transmission delays on publication messages has previously been addressed by reconfiguring the broker topology [12], clustering subscribers into multicast-groups to limit publication propagation only among interested peers [13], [14], [15], [16], [17], [18], [19], or incorporating multicast-groups with filter-based approaches [20].

Published in the 30th IEEE International Conference on Distributed Computing Systems (ICDCS), Genoa, Italy, 2010.

In this paper, we show that strategic placement of publishers in a content-based publish/subscribe network can improve system scalability, robustness, and performance. We present two different placement algorithms, POP (*Publisher Optimistic Placement*) and GRAPE (*Greedy Relocation Algorithm for Publishers of Events*), to intelligently relocate publishers while keeping the broker overlay intact to minimize both the average end-to-end delivery delay and system load. Both POP and GRAPE follow a 3-Phase operational design: (1) gather publication delivery statistics on the publishers' publications, (2) identify the target broker to relocate the publisher to, and (3) transparently migrate the publisher to the target broker. Each phase of POP and GRAPE contribute to the algorithms' dynamic, scalable, robust, and transparent properties. Both algorithms are *dynamic* by periodically making relocation decisions based on *live* publication delivery patterns. Both are *scalable* thanks to the use of distributed design that scales with the number of brokers and clients in the network. Both are *robust* because an instance of POP or GRAPE runs on every broker to rule out the possibility of any single point of failure. Lastly, both are *transparent* to application-level publish/subscribe clients as publication statistics gathering and publisher migration all happen behind the scenes; neither require the application's involvement nor introduce any message loss.

However, POP and GRAPE are different from each other due to design decisions that trade off simplicity for flexibility. (1) POP uses one optimization metric, the average number of publication deliveries downstream, whereas GRAPE uses two optimization metrics, the end-to-end delivery delay and total broker message rate, to compute the relocation target. (2) GRAPE allows the prioritization of minimizing average delivery delay, system load, or any combination of both metrics simultaneously whereas POP does not give the user this flexibility. (3) In Phase 1, POP retrieves optimization metrics once per each traced publication message whereas GRAPE retrieves optimization metrics once per broker selection cycle, thus giving POP and GRAPE different tradeoffs between algorithm response time and message overhead. (4) In Phase 2, POP performs its broker selection in a distributed manner hop-by-hop towards the target broker, whereas GRAPE performs its broker selection in a centralized manner all locally at the publisher's first broker. (5) GRAPE is easier to debug and test relative to POP because its core computations are centralized.

POP and GRAPE are primarily targeted at enterprise-grade messaging systems consisting of hundreds to thousands of dedicated servers [9], [21], [22]. These systems include the commercial publish/subscribe systems mentioned previously as well as other systems enabled by publish/subscribe such as workflow management systems [5], decentralized business process execution [23], automated service composition [24], RSS dissemination [25], [26], network and systems monitoring [27], resource discovery [28] and more.

A motivating application scenario can be found in the GDSN [11] commercial publish/subscribe system where retailers subscribe to product information/updates published by suppliers. To illustrate our point, we derive a real business scenario from [29] that focuses on the dynamic pricing of soft drinks. Retailers on GDSN such as Walmart, Target, SUPERVALU, Metro, Associated Grocers, and many others are likely subscribed to events published by suppliers that report moderate to large changes in the cost of soft drinks currently sold (due to peak season, supplier competition, currency fluctuations, etc.) so that shelve prices can be updated as soon as possible to maximize profit and minimize loss. However, retailers are not interested in minute price changes because price adjustment on the cent-level for all on-the-shelf products is not economically feasible. Suppliers of soft drink products on GDSN, Coca-Cola Enterprises and PepsiCo, are likely subscribing to all price updates published by themselves for record keeping and by their competitor for close monitoring. If the GDSN is equipped with GRAPE set to minimize solely on average delivery delay, GRAPE will reconnect the suppliers' publishing agents close to the hundreds of retailers' subscriber agents that sink a subset of the publishers' events. The result is quicker price update deliveries to the subscriber agents for more timely price adjustments. On the other hand, with GRAPE set to minimize solely on system load, GRAPE will reconnect the publishing agents close to their own and the direct competitor's subscriber agent that sink all of the publishers' events. The result is brokers become less loaded, which adds stability and further capacity to the GDSN backbone. If the GDSN is instead equipped with POP, the outcome can be anywhere between the two extremes of GRAPE: minimizing price update transmission delays and/or chances of system down-times, both of which are cost saving measures critical to businesses. These illustrated behaviors of GRAPE and POP are in fact what we observed in our experiments.

The main contributions of this paper are: (1) POP's Phase 1 algorithm which probabilistically traces publication messages and retrieves trace information through replies with data aggregation, (2) POP's Phase 2 algorithm which selects the target broker in a fully decentralized manner using only partial trace data, (3) GRAPE's Phase 1 algorithm which traces publication messages and stores trace results into space-efficient bit vectors for later retrieval through replies with data aggregation, (4) GRAPE's Phase 2 algorithm which selects the target broker in a centralized manner based on the specified prioritization metric and weight, (5) POP and GRAPE's Phase 3 algorithm which transparently migrates the publisher from the original to the target broker while introducing minimal message overhead, and (6) extensive experiments using real-world data on PlanetLab and a cluster testbed that quantitatively validate and compare our two approaches. Our results show that POP and GRAPE are able to reduce the average broker input utilization by up to 64% and 68%, average broker message rate by up to 85% and 85%, and average delivery delay by up to 63% and 68% on PlanetLab, respectively.

The rest of this paper is organized as follows. Section II puts our work in the context of related approaches. Sections III and IV describe in detail the architecture and operation of POP and GRAPE, respectively. Section V evaluates the performance of both POP and GRAPE implementations in the PADRES publish/subscribe system [1], [23], [28], [30], [31], [32], [33] running on PlanetLab and on a cluster testbed.

II. Related Work

A. Publish/Subscribe Systems

Two main classes of distributed content-based publish/subscribe systems exists today: filter-based [1], [2], [3], [4], [5], [6], [7], [32], [34] and multicast-based [14], [15], [16], [17], [18], [19]. In the filter-based approach, advertisements and subscriptions are propagated into the network to establish paths that guide publications to subscribers. Each publication is matched at every broker along the overlay to get forwarded towards neighbors with matching subscriptions. Consequently, the farther the publication travels, the higher is the delivery delay. In the multicast-based approach, subscribers with similar interests are clustered into the same multicast group. Each publication is matched once to determine the matching multicast group(s) to which the message should be multicasted, broadcasted, or unicasted. As a result, matching and transmission of a publication message happens at most once, thus incurring minimal delivery delay. However, compared to the filter-based approach, subscribers in a multicast group may receive unwanted publications because subscribers with even slightly different interests may still be assigned to the same group. One possible solution to this problem is the introduction of filter-based functionality within each multicast group [20].

By reconnecting publishers to areas of most populated and/or highest-rated matching subscribers in filter-based approaches, POP and GRAPE reduce the in-network processing and transmission overhead to the level of multicast-based approaches. At the same time, our algorithms guarantee no false-positive publication delivery and do not have to manage and partition subscribers into multicast groups. A key distinguishing feature of our work is that both POP and GRAPE focus on bringing publishers closer to their subscribers on the overlay network rather than virtually grouping subscribers together. POP and GRAPE are applicable to filter-based approaches that use tree overlays with or without subscription covering [35], subscription merging [36], [37], and subscription summary [38], [39] optimizations. POP and GRAPE can also be adapted to systems where clients take on both publisher and subscriber roles by separating the network connections between the two entities.

A number of approaches are found in the literature that also try to reduce the overlay distance between publishers and subscribers. Baldoni *et al.* [12] dynamically reconfigure inter-broker overlay links to allow publications to skip over brokers with no matching subscribers. On the other hand, POP and GRAPE relocate publishers to the location of highest-rated or populated matching subscribers while preserving the broker overlay. Hence, our work is suited to policy-driven broker networks where inter-broker links are statically and tightly controlled by administrators. TERA [40] utilizes random walks and access point lookup tables to have publications delivered directly from the publisher to clusters of peers with matching subscriptions. In contrast to our work, TERA operates on a peer-to-peer architecture, clusters subscribers of similar interests, and supports a topic-based rather than a content-based language. Through epidemic-based clustering, SUB-2-SUB [13] clusters subscribers of similar interests and propagates publications only among interested peers. Similar to our work, SUB-2-SUB supports a content-based language. However, SUB-2-SUB's peer-to-peer architecture is fundamentally different from the broker-based architecture that POP and GRAPE are designed for: each peer in SUB-2-SUB is associated with exactly *one* subscription, while each broker in our system is associated with *any* number of subscriptions, depending on the number of subscriber clients attached to the broker. This distinction allows SUB-2-SUB to route multiple events from the same publisher only to interested peers even if the events are delivered to different sets of peers.

B. Publisher Migration Protocols

Muthusamy *et al.* [31], [41] proposed several publisher migration protocols with different optimization techniques to study the effects of publisher and subscriber migration on system performance. The publisher migration protocol that we propose in this paper is different in three ways. First, instead of rebuilding the advertisement tree rooted at the new broker, we simply revise the last hop of the existing advertisement only on brokers along the *migration path* as in [31]. In terms of overhead message count, our approach generates $O(\log N)$ messages, whereas the approach in [41] generates $O(N)$ messages, where N is the total number of brokers in a tree-overlay network with typical fan-out greater than one. Second, the advertisement/subscription tree rebuilding period is known in our approach. This allows our publishers to know precisely the earliest time to resume publishing at the new broker with assurance that those messages will be delivered to all matching subscribers in the network. Third, the objective in [41] is to analyze the impact of supporting mobile publishers on system performance, whereas here, our objective is to minimize average end-to-end delivery delay and system load by relocating publishers.

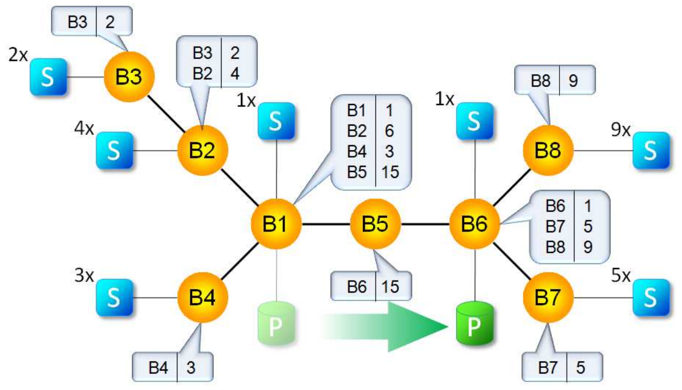


Fig. 1. Example of *Publisher Profile Table*

III. The POP Placement Algorithm

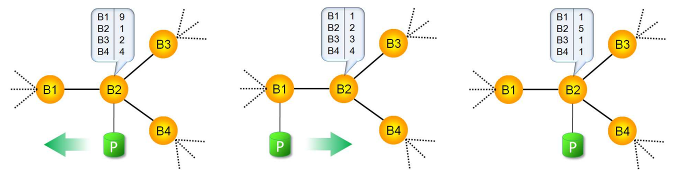
The rest of this paper makes use of the terms *downstream* and *upstream* to identify other brokers relative to an arbitrarily referenced broker and a publisher. Downstream brokers are those that receive publication messages from the referenced broker, directly or indirectly over multiple neighbors. In other words, downstream brokers are those farther away from the publisher compared to the referenced broker. The opposite definition holds for upstream brokers. Using Figure 1 as an example, if the referenced broker is *B6* and the publisher is at *B1*, then *B7* and *B8* are downstream brokers while *B5* and *B1* are upstream brokers.

The following sections show POP's 3-Phase operation in detail. Section III-A presents Phase 1, where POP probabilistically traces each publisher's *live* publications to discover the location and number of matching subscribers in the network. Section III-B describes Phase 2, where POP uses trace information obtained from Phase 1 to pinpoint the broker closest to the highest number of matching subscribers that the publisher should connect to. Section III-C presents Phase 3, which involves transparently migrating the publisher to the broker identified in Phase 2 with minimal routing table updates. Message sequence diagrams detailing each phase under Figure 1's scenario are included in our online Appendix [42] for further clarification. Our evaluation shows that POP's data structures use no more than 34% (or 19 MB) of additional memory, and message overhead varies between 6% and 57% at two most extreme POP configurations.

A. Phase 1: Distributed Trace Algorithm

The goal of Phase 1 is to gather the average number of subscribers downstream of each brokers' neighbor links for each publisher client. To realize this goal, we developed (1) an algorithm to tag publication messages to trace where they got delivered, (2) a reply protocol to notify upstream brokers of the number of subscribers to which the publication was delivered at downstream brokers, and (3) a data structure to store and aggregate results from the traces.

1) Probabilistic Publication Tagging: POP utilizes a special publication tagging technique to reduce both message and computation overhead from publication tracing. Whenever the



(a) Neighbor *B1* satisfies the *closest* condition (b) No neighbors satisfy *closest* condition. *Closest* broker is *B2* (c) The original broker is the *closest* broker

Fig. 2. All possible outcomes of POP's broker selection algorithm

publisher's first broker handles a publication message from the client, it can choose to trace the message by tagging/setting the `trace` header field to `true`, or disable tracing by leaving `trace` at its default value of `false`. Tagging is based on P_{trace}, which is defined by the function: $P_{trace} = 1 - \frac{T}{N}$. Here, T is the number of messages already tagged for tracing in the current time window W, and N is a configurable parameter that limits the maximum number of publication messages traced in time window W. By default, N is set to 50 and W to 60 s. Each publisher is associated with its own value of T. The advantages of using the P_{trace} function over a constant function are: (1) the number of publications traced within W is bounded by N, (2) for extremely low-rated publishers, at least one publication message is tagged with 100% probability in each time window, and (3) for high-rated publishers, this equation offers a higher chance of tagging publication messages sent near the end of each time window.

2) Trace Result Notification: On handling a publication message with a `true` value in the `trace` header field, the broker has to send back to the upstream broker a *Trace Result Message (TRM)*. A *TRM* contains two fields: (1) *publisher's advertisement ID* obtained from the publication's header and (2) *cumulative subscriber count*, which is the total number of subscribers at and downstream of the reporting broker. A broker can only send a *TRM* to the upstream broker if any of the following two conditions are satisfied: (1) the publication message is *only* delivered to subscribers or (2) a corresponding *TRM* is received from each neighbor to which the publication message is sent.

3) Publisher Profile Table: POP stores trace results for each publisher into a *Publisher Profile Table (PPTable)* which has two columns: (1) *downstream broker* and (2) the *average number of subscribers*. A running average is used to maintain the average number of subscribers because it has the benefit of efficiently aggregating multiple values to conserve space. By default, the running average gives a weight of 0.25 to the newest value and 0.75 to the last average value. Figure 1 shows an example of the *PPTable* at each broker after tracing one publication message that got delivered to all illustrated subscribers.

B. Phase 2: Decentralized Broker Selection Algorithm

The goal of Phase 2 is to use the *PPTables* gathered in Phase 1 to incrementally pinpoint the broker closest to the highest number of matching subscribers, which we refer to as the *closest* broker from here on. POP's Phase 2 algorithm is initiated after two conditions are met: (1) the number of publications traced meets the threshold $P_{threshold}$ and (2) the publisher does not have any outstanding publication trace results (so as to prevent trace data inconsistency among brokers). By default $P_{threshold}$ is set to 100. On Phase 2 initiation, POP in the publisher's first broker creates a *Relocation Request Message (RRM)*. The *RRM* contains three values: (1) *publisher's advertisement ID*, (2) *total number of subscribers* down the link from which this request is sent, and (3) *list of brokers* traversed by this *RRM* in decreasing order of *closest* location. The latter field identifies brokers on the *migration path* that need routing table updates in Phase 3.

When a broker creates or receives a *RRM*, it has to determine the next *closest* neighboring broker to forward the message to. The next *closest* neighboring broker is:

> *the one whose number of downstream subscribers is greater than the sum of all other neighbors' downstream subscribers plus the local broker's subscribers.*

If no neighbor broker satisfies the *closest* condition, then the *closest* broker is itself. Note that each broker handling the *RRM* has its own definition of downstream as will be shown through an example in the next paragraph. The closest condition can be extended to include a threshold parameter to dampen any potential ping-pong effect when the difference is just one. However, we will leave this extension for future work and just focus on POP with minimal optimizations in this paper. Figure 2 summarizes all three possible outcomes of the broker selection algorithm. If the *closest* broker is *not* the originator of the *RRM*, then a *Relocation Answer Message (RAM)* is sent back to the originator with the *publisher's advertisement ID* and the *list of brokers* traversed by the *RRM* including the *closest* broker itself. Otherwise, the publisher is already at the *closest* broker, in which case Phase 3 is aborted and Phase 1 will be initiated again after getting $P_{threshold}$ new trace results.

Using broker *B1* in Figure 1 as an example, since *B5*'s sum of 15 from the *PPTable* is greater than the sum of *B2*, *B4*, and *B1*, $6 + 3 + 1 = 10$, *B5* is the next *closest* broker. As a result, *B1* updates and forwards the *RRM* to *B5*. Specifically, *B1* adds itself to the head of the *broker list* and increments the *total number of subscribers* field by 10, which is the number of subscribers at the local broker *B1* plus the number of subscribers at and downstream of all *non-closest* neighbors, namely *B2* and *B4*. Upon receiving the *RRM* from *B1*, *B5* finds that the number of subscribers downstream to *B6* (according to the *PPTable*) is greater than the number of subscribers downstream to *B1* (according to the *RRM*). Therefore, *B5* updates the *RRM*'s broker list to [*B5*, *B1*] and forwards the message to *B6*. Upon receiving the *RRM* from *B5*, *B6* discovers that there are 10 subscribers downstream to *B5*. Since no neighbor is able to satisfy the *closest* condition, *B6* determines itself to be the *closest* broker and sends a *RAM* back to *B1* to initiate Phase 3.

C. Phase 3: Publisher Migration Protocol

On receiving a *RAM* from the *closest* (or *target*) broker, the publisher's first (or *source*) broker initiates the migration by informing the designated publisher to (1) temporarily pause publishing or buffer its messages locally and (2) submit a migration advertisement, which is an advertisement with the *RAM* as payload, to the *target* broker. POP at the *target* broker intercepts the special advertisement message from entering the matching engine and sends a *Migration Update Message (MUM)* to itself carrying the *list of brokers on the migration path* and the publisher's *advertisement ID* obtained from the advertisement's payload. Each broker handling the *MUM* updates its own routing tables to reflect the publisher's new location, clears the *PPTable* entry for this publisher, and forwards the *MUM* to the next broker along the *migration path*. Once the *MUM* reaches the *source* broker and finishes updating the routing tables, the *source* broker sends a *Migration Complete Message (MCM)* to the *target* broker to end the migration. The purpose of sending the *MCM* to the *target* broker over the *migration path* instead of the publisher directly is because the arrival of this message there guarantees that all subscriptions forwarded by any brokers on the *migration path* will have reached the *target* broker. At that point, the *target* broker completes the migration process by notifying the publisher to resume publishing and disconnect from the *source* broker.

Notice that our publisher migration protocol limits the amount of computational and message overhead to the set of brokers along the *migration path*. In a tree network consisting of *N* brokers with typical fanout greater than one, there exists only one migration path and the overhead complexity is bounded by *O(log N)*. Brokers outside of the *migration path* do not participate because the state of those brokers before and after the migration remains the same. The routing table update operations at the individual brokers in Phase 3 include: (1) updating the last hop of the publisher's advertisement to reflect the migrated position, (2) removing subscriptions that no longer match any advertisement, and (3) forwarding subscriptions that match the updated advertisement. To reduce the amount of matching overhead in operation #2, only subscriptions with a last hop equal to the advertisement's new last hop need to be checked for removal. The entire migration session is transparent to the application as all migration activities are handled by a thin software layer built into the publish/subscribe client. Subscribers are also isolated from the migration as the migration protocol is completely lossless, though subscribers may notice a short delivery interruption while the publisher migrates. Our evaluation shows that a 10 hop migration takes 5 s on PlanetLab and 1.5 s on the cluster testbed.

For clarification, the following explains how each of the above update operations apply to the scenario given in Figure 1. Operation #1 applies to all brokers along the *migration path* where broker *B6* updates publisher *P*'s advertisement last hop to a local destination, broker *B5* updates *P*'s advertisement

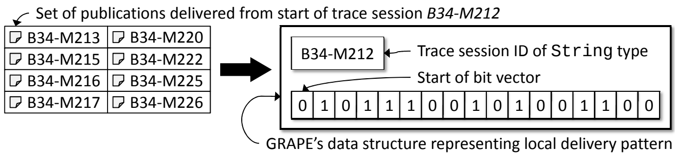

Set of publications delivered from start of trace session B34-M212
B34-M213
B34-M220
B34-M215
B34-M222
B34-M216
B34-M225
B34-M217
B34-M226
B34-M212
Trace session ID of String type
Start of bit vector
0 1 0 1 1 1 0 0 1 0 1 0 0 1 1 0 0
GRAPE's data structure representing local delivery pattern

Fig. 3. String and bit vector representation of delivered publications

last hop to *B6*, and broker *B1* updates *P*'s advertisement last hop to *B5*. Operation #2 applies to brokers *B1* and *B5* where subscription(s) from the 15 subscribers that reside on the right of broker *B5* are removed[1]. Operation #3 applies to brokers *B1* and *B5* where subscription(s) from the 10 subscribers that reside on the left of *B5* to broker *B6* are forwarded[1] .

IV. The GRAPE Placement Algorithm

Like POP, GRAPE follows the same 3-Phase operational design and uses the publisher migration protocol presented in Section III-C. However, the data structures and algorithms that GRAPE uses in Phases 1 and 2 are completely different. As we will show in our evaluation, GRAPE uses up to an additional 58% (or 31 MB) of memory with message overhead ranging between 0% and 46% at two extreme GRAPE configurations. Compared to POP, that is 24% (13 MB) more memory but 20% less message overhead in the extreme worst case.

A. Phase 1: Distributed Publication Tracing

1) Logging Publication Delivery Statistics: GRAPE tracks publications from publishers only within *trace sessions*. In a trace session, $G_{threshold}$ publications are traced. By default, $G_{threshold}$ is 100. Each publisher is associated with its own trace session as managed by its first broker. Trace sessions are identified by the message ID of the first trace-enabled publication in that session. Message IDs are uniquely generated by prefixing the value of an incrementing counter with the ID of the publisher's first broker. Publications published within a trace session carry the same trace session ID in the `traceID` header field. A publication that is not trace-enabled has `traceID` set to `null`.

During a trace session, brokers handling a trace-enabled publication capture two pieces of information. (1) The *total number of local subscribers that matched this publication*, or simply the *total number of local deliveries*. This value is used in Phase 2 to estimate the average end-to-end delivery delay of all matching subscribers when GRAPE tries to place the publisher at different brokers. (2) The set of *publication messages delivered to local subscribers*. This information allows Phase 2 to accurately estimate the amount of traffic that flows through each broker when GRAPE tries to place the publisher at other brokers. Instead of storing a set of publication messages, we developed a novel scheme that utilized one `String` and one bit vector variable. The `String` variable records the trace session ID, whose suffix signifies the starting index of the bit vector. On delivering a trace-enabled publication with message ID $M + \Delta$ for a trace session with

[1]With subscription covering, the number of subscriptions removed/forwarded may not equal the number of subscribers.

identifier M, GRAPE will set the Δ-th bit of the bit vector. An example is demonstrated in Figure 3 with $M = 212$. Use of the bit vector comes with many advantages, including space efficiency, ease of aggregating multiple bit vectors with the `OR` bit operator, and the direct proportional relationship between cardinality and message rate. Unlike POP where publications are probabilistically selected for tracing, GRAPE has to trace consecutive publications in a trace session to minimize the size of the bit vector.

2) Retrieval of Delivery Statistics: When the required number of publication messages are traced as governed by $G_{threshold}$, GRAPE sends a *Trace Information Request (TIR)* message to all downstream neighbors that have received at least one publication from this publisher within this trace session. Brokers receiving a *TIR* message will (1) forward the *TIR* message to downstream neighbors that satisfy the previously stated condition, (2) wait to receive a *Trace Information Answer (TIA)* reply message from the same set of downstream neighbors, and (3) send an aggregated *TIA* reply message containing their own and all downstream brokers' trace information in the payload. Brokers with no downstream neighbors immediately reply back with a *TIA* message to the upstream neighbor with a payload containing the following information about themselves: (1) *broker ID*, (2) *neighbor broker ID(s)*, (3) *bit vector* capturing the delivery pattern, (4) *total number of local deliveries*, (5) *input queuing delay*, (6) *average matching delay*, and (7) *output queuing delay* to each neighbor broker and the client binding. The latter three figures can be measured outside of GRAPE by a monitor module as is the case in our implementation. For additional clarification, please see our online Appendix [42] for a message sequence diagram that shows GRAPE's trace retrieval protocol under the scenario illustrated in Figure 1. If we assume default GRAPE settings with each broker's ID to be around 10 characters, each broker having on average three neighbors, then the size of *one* broker's payload is only about 100 bytes. After sending a *TIA* message, the broker clears all data structures related to that trace session to free up memory. Compared to POP, GRAPE's *TIA* messages are very similar to POP's *TRM* messages. What is different, however, is that GRAPE sends a reply message after each *trace session* whereas POP sends a reply message after each *traced publication*.

B. Phase 2: Broker Selection Algorithm

With the statistical information from Phase 1, GRAPE in Phase 2 can estimate the average end-to-end delivery delay and system load if the publisher is moved to any one of the *candidate* brokers. The candidate brokers are the downstream brokers that replied with a *TIA* message in Phase 1. In POP, where the broker selection algorithm is distributed, the broker selection algorithm in GRAPE is entirely centralized at the publisher's first broker. Some may argue that the amount of processing will overwhelm a node or there exists a single point of failure, but both arguments are not quite true. The total processing time on PlanetLab never exceeded 70 ms in the worse case with subscribers residing on all 63 brokers in the network. As well, each publisher is managed by GRAPE running at the publisher's first broker. Therefore, if the first broker fails, the publisher can reconnect to any other broker and continue to be managed by another instance of GRAPE. The major benefits of adopting a centralized approach are the ease of design, implementation, and verification.

Algorithm 1 calcAvgDelay(stats, cumDelay, currBroker, prevBroker)

```
// Get delivery statistics from clients on the current broker
$stats.totalDelay += currBroker.totalDeliveries \times$
    $(cumDelay + currBroker.queueAndMatchDelaysTo(client))$
$stats.totalDeliveries$ += $currBroker.totalDeliveries$
for $neighbor$ in $currBroker.neighborSet$ do
    // Skip the upstream broker
    if $neighbor$ equals $prevBroker$ then
        continue
    // Accumulate this broker's processing and queuing delays
    $newCumDelay = cumDelay +$
        $currBroker.queueAndMatchDelaysTo(neighbor)$
    // Get delivery statistics of clients at downstream brokers
    $calcAvgDelay(stats, newCumDelay, neighbor, currBroker)$
return
```

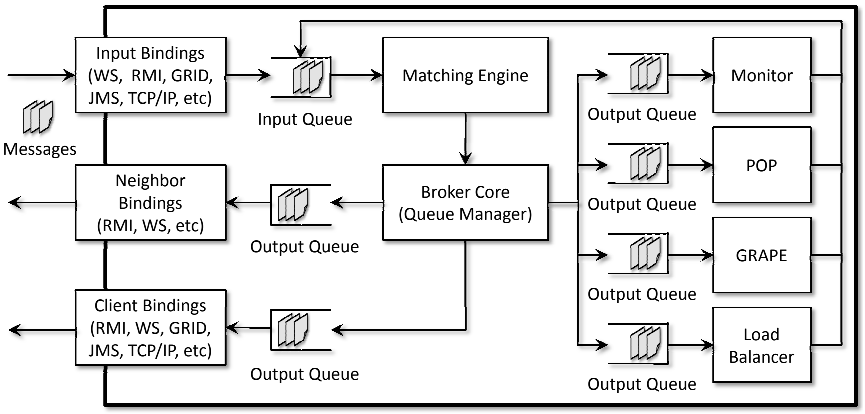
Algorithm 2 calcTotalMsgRate(currBroker, prevBroker)

```
// Get the message rate going through this broker
$localMsgRate =$
    $calcDownstreamBV(currBroker, prevBroker).cardinality$
// Get the total message rate at all downstream brokers
$dsMsgRate = 0$
for $neighbor$ in $currBroker.neighborSet$ do
    if $neighbor$ equals $prevBroker$ then
        continue
    $dsMsgRate$ += $calcTotalMsgRate(neighbor, currBroker)$
// Return the sum of all brokers' message rates
return $localMsgRate + dsMsgRate$
```

Algorithm 3 calcDownstreamBV(currBroker, prevBroker)

```
$aggregatedBV = currBroker.bitVector$
// Take local deliveries and OR with downstream deliveries
for $neighbor$ in $currBroker.neighborSet$ do
    if $neighbor$ equals $prevBroker$ then
        continue
    $dsBV = currBroker.getDownstreamBVTo(neighbor)$
    if $dsBV$ is $NULL$ then
        // Calculate and store the downstream bit vector
        $dsBV = calcDownstreamBV(neighbor, currBroker)$
        $currBroker.setDownstreamBVTo(neighbor, dsBV)$
    $aggregatedBV$ |= $dsBV$
return $aggregatedBV$
```

GRAPE allows the user to prioritize which of the two metrics to minimize: average end-to-end delivery *delay* or system *load* (in the form of message rate), and also specify a *minimization weight* from 0 to 100% to indicate how much of that metric to minimize. If the primary metric to minimize is *delay*, and the minimization weight is P, then GRAPE first asks the publisher to reply back with a set of ping times to each candidate broker. Due to fluctuating network conditions and unavailability of ping on PlanetLab, publishers instead invoke an API on each candidate broker five times to measure the round trip times. On our cluster testbed, publishers invoke the API once with multiple candidate brokers simultaneously. Landmark and multidimensional-scaling based latency estimation techniques such as Netvigator [43] and Vivaldi [44] can be substituted in place of ping, but they trade off faster turnaround time for less accuracy.


Input Bindings (WS, RMI, GRID, JMS, TCP/IP, etc)
Messages
Input Queue
Matching Engine
Output Queue
Monitor
POP
GRAPE
Load Balancer
Neighbor Bindings (RMI, WS, etc)
Broker Core (Queue Manager)
Client Bindings (RMI, WS, GRID, JMS, TCP/IP, etc)

(a) POP and GRAPE in the PADRES broker

Setting	PlanetLab	Cluster
Brokers	63	127
Publishers	20	30
Msgs per min per Publisher	10-40	30-300
Subscribers per Publisher	80	200
Total Subscribers	1600	6000
$P_{threshold}$	50	100
$G_{threshold}$	50	100

(b) Deployment specs

Fig. 4. Experimentation details

The ping times together with the queuing and matching delays from each broker allow GRAPE to estimate the average end-to-end delivery delay with the publisher located at any downstream candidate broker by using the `calcAvgDelay()` function as shown in Algorithm 1. This function recursively calculates the average end-to-end delivery delay from the publisher to each subscriber according to the number of deliveries made in the past and using actual queuing and matching delay measurements at each broker. After invoking `calcAvgDelay()` on each candidate, GRAPE normalizes the candidates' delivery delays and drops those candidates with delivery delays greater than $100 - P$. GRAPE then calculates the *total system message rate* with the publisher positioned at each remaining candidate by using `calcTotalMsgRate()` as shown in Algorithm 2. The total system message rate is the sum of the input message rates introduced by this publisher into every broker. Note that the input message rate at each broker may be different as each broker's message rate depends on both local and downstream subscriptions. To aid in this calculation, `calcTotalMsgRate()` uses the helper function `calcDownstreamBV()` (shown in Algorithm 3) to aggregate downstream broker bit vectors to accurately compute the input message rate at each broker. The candidate that offers the lowest total system message rate is the selected broker.

On the other hand, if the primary metric to minimize is *load*, then the algorithm is reversed. GRAPE first calculates the total system message rate with the publisher placed at each candidate broker by using `calcTotalMsgRate()`, drops candidates with normalized message rates past $100-P$, fetches the set of publisher ping times, calculates the average delivery delay with the publisher positioned at each candidate broker by using `calcAvgDelay()`, and finally selects the broker that offers the least average delivery delay. At the very extreme case, if GRAPE is set to minimize load at 100%, then GRAPE will select the candidate where the publisher introduces minimal amount of traffic in the system without regards to the average delivery delay. If GRAPE is set to minimize delay at 100%, then GRAPE will select the candidate that offers the lowest average delivery delay without regards to the system load. The worst case runtime complexity of this algorithm is $O(N^2)$ where N is the number of brokers in the system. Our experiments on PlanetLab show that even when N is 63, GRAPE's broker selection algorithm took less than 70 ms.

V. Experiments

Both POP and GRAPE are implemented on top of PADRES [1], [23], [28], [30], [31], [32], [33], an open source distributed content-based publish/subscribe system developed by the Middleware Systems Research Group (MSRG) at the University of Toronto. Both POP and GRAPE are integrated into the PADRES broker as additional internal modules as illustrated in Figure 4a. The code structure of both approaches follows the same 3-Phase architectural design. POP required the addition of about 1,700 lines of code to PADRES while GRAPE required about 2,700.

A. Experiment Setup

We ran experiments on PlanetLab to show how POP and GRAPE behave under real-world networking conditions and on a cluster testbed to validate our PlanetLab results under controlled networking conditions. The cluster testbed consists of 20 nodes each with two Intel Xeon 1.86 GHz dual core CPUs connected by 1 Gbps network links. Deployments on both the cluster and PlanetLab are aided by the use of a tool developed by the MSRG called PANDA (*PADRES Automated Node Deployer and Administrator*). This tool allows us to specify the experiment setup within a text formatted *topology file* such as the time and nodes at which to run brokers and clients, as well as any process specific runtime parameters such as the neighbors for brokers. The topology file is fed into PANDA which then deploys the processes automatically. On PlanetLab, each broker process runs on a separate machine, while on the cluster testbed six to seven brokers run on one machine. Brokers and overlay links are verified to be up and running before clients are deployed. A publisher and its set of matching subscribers run on the same machine to accurately measure end-to-end publication delivery delay. Different publishers run on randomly chosen machines that also run broker processes. Each publisher publishes stock quote publications of a particular stock that are real-world values obtained from Yahoo! Finance containing a stock's daily closing prices.[2] A typical publication looks like this:

```
[class,'STOCK'],[symbol,'YHOO'],[open,18.37],
    [high,18.6],[low,18.37],[close,18.37],
    [volume,6200],[date,'5-Sep-96'],
    [openClose%Diff,0.0],[highLow%Diff,0.014],
    [closeEqualsLow,'true'],[closeEqualsHigh,'false']
```

We ran experiments on both PlanetLab and the cluster testbed using two different subscriber traffic distributions. One is *random* where 70% of the subscribers are low-rated, meaning they sink about 10% of their publishers' traffic; 25% are medium-rated, meaning they sink about 50% of their publishers' traffic; and 5% are high-rated, meaning they sink *all* of their publishers' traffic. Subscribers of each category are randomly assigned to X number of brokers, where X is varied in our experiments. The other distribution is referred to as *enterprise*, where 95% of the subscribers are low-rated and 5% are high-rated. All high-rated subscribers connect to one broker, while low-rated subscribers are randomly assigned to the other $X - 1$ brokers. The random workload represents a generic scenario, while the enterprise workload mimics an enterprise deployment consisting of a database sinking all traffic at the main office and many end-users subscribing to selected traffic at different branch office locations. An example of low, medium, and high-rated subscriptions to YHOO stockquotes are shown below:

```
low    - [class,=,'STOCK'],[symbol,=,'YHOO'],
           [highLow%Diff,>,0.15]
medium - [class,=,'STOCK'],[symbol,=,'YHOO'],
           [volume,>,1000000],[openClose%Diff,>,0.025]
high   - [class,=,'STOCK'],[symbol,=,'YHOO']
```

The overlay topology we used for all evaluations is a balanced tree with a fan-out of 2 or 4. The number of publishers, subscribers, brokers and all other settings we used on PlanetLab and the cluster testbed are shown in Figure 4b. Unless otherwise stated, default POP and GRAPE settings are used.

B. Experiment Results

We evaluated POP and GRAPE with real-world data sets on PlanetLab and a cluster testbed while varying the subscriber distributions, GRAPE's minimization metric (which is either the end-to-end delivery delay or system load), GRAPE's minimization weight, and number of samples for triggering broker selection. For graphs without time as the x-axis, the plotted values are obtained at the end of the experiment, which is 18 minutes after all clients are deployed. All graphs use average values across all brokers or clients in the system. Due to limited space, we can only include a small subset of the graphs here with the full set in an online Appendix [42]. Nevertheless, we summarize all of our results here in this paper. From this point on, we will use the notation load 75% (delay 25%) to denote GRAPE's configuration to prioritize on minimizing average system load (delivery delay) with 75% (25%) weight.

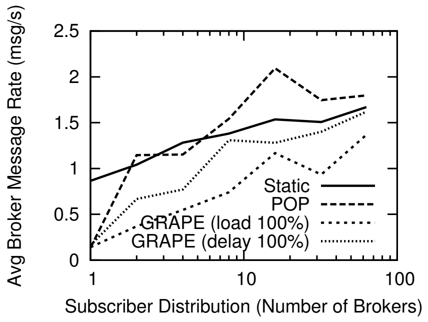
[2]Available at http://research.msrg.utoronto.ca/Padres/DataSets

1) Enterprise Workload: Under the enterprise scenario, Figures 5a and 5d show that GRAPE's load 100% yields the lowest average broker message rate compared to GRAPE's delay 100%, POP, and static (with neither GRAPE nor POP enabled). This is true on both testbeds and across all subscriber distributions except when all subscribers to each publisher are concentrated at one broker. In which case, both POP and GRAPE make the same relocation decision to migrate the publishers to the broker where all the matching subscribers reside. Lower average broker message rate translate directly to lower average broker input (Figures 5b and 5e) and output utilizations (Appendix [42]). *Input utilization ratio* captures the brokers' matching rate versus the flow of incoming traffic and *output utilization ratio* captures the output bandwidth usage.[3] However, because GRAPE's load 100% setting moves the publishers closest to the subscribers that subscribe to all of the publishers' traffic, the publishers are farther away from the majority of subscribers who sink a subset of the traffic. As a result, Figures 5c, 5f, and 6a show that average delivery delay and hop count are sacrificed, especially in the experiment on the cluster testbed where the delivery delay of load 100% is higher than static. We believe the latter behavior is due to the increased number of subscribers and maximum hop count in the cluster testbed setup over the PlanetLab setup. Figures 6c, 6d, and the Appendix [42] show that the performance trends of GRAPE and POP still hold as we increase the fanout of the network from 2 to 4 while keeping the number of brokers constant. However, since the maximum length of the network is now decreased, publishers start off closer to the subscribers before their relocation. Hence, the broker selection time, publisher wait time, and reductions in delivery delay and broker message rate of GRAPE and POP decrease as fanout increases.

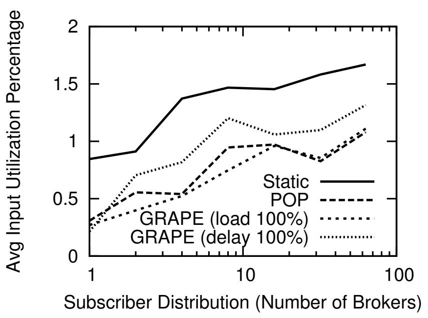
With GRAPE's delay 100% setting, the system achieves lowest average delivery delay across almost all subscriber distributions as shown in Figures 5c and 5f. This is due to placing the publishers closest to the subscribers with the highest number of publication deliveries, which dominates what is shown in Figure 6a. However, because the publishers are further away from the high-rated subscribers, the system has to transmit more messages overall (Figures 5a and 5d) which leads to higher input (Figures 5b and 5e) and output utilizations (Appendix [42]). On platforms where latency and processing delay are minimal already such as the cluster testbed, Figure 5f shows that there is not much more that POP and GRAPE can reduce.

POP minimizes both average system load and delivery delay simultaneously without offering a choice of which metric to prioritize and by how much. According to Figures 5b, 5c, 5e, 5f, and 6a, POP's average input utilization, delivery delay, and hop count is between GRAPE's load 100% and delay 100%. However, the average broker message rate of POP as shown in Figures 5a and 5d exceeds that of GRAPE due to the increased message overhead from *TRM*s (Figure 6b).

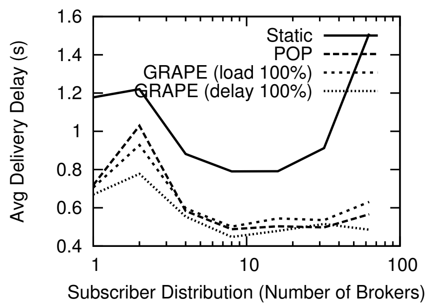
[3]Please see [32] for definitions of these metrics.



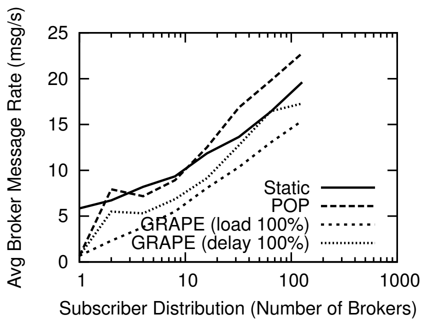
(a) Broker message rate on PlanetLab



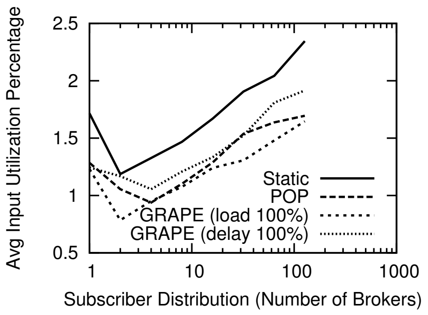
(b) Input util. percentage on PlanetLab



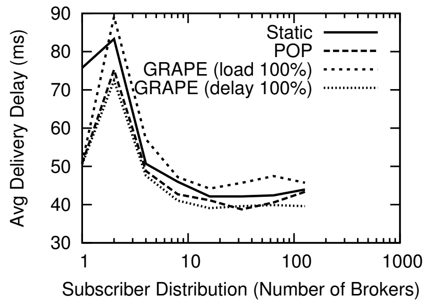
(c) Delivery delay on PlanetLab



(d) Broker message rate on the cluster testbed



(e) Input util. percentage on the cluster testbed



(f) Delivery delay on the cluster testbed

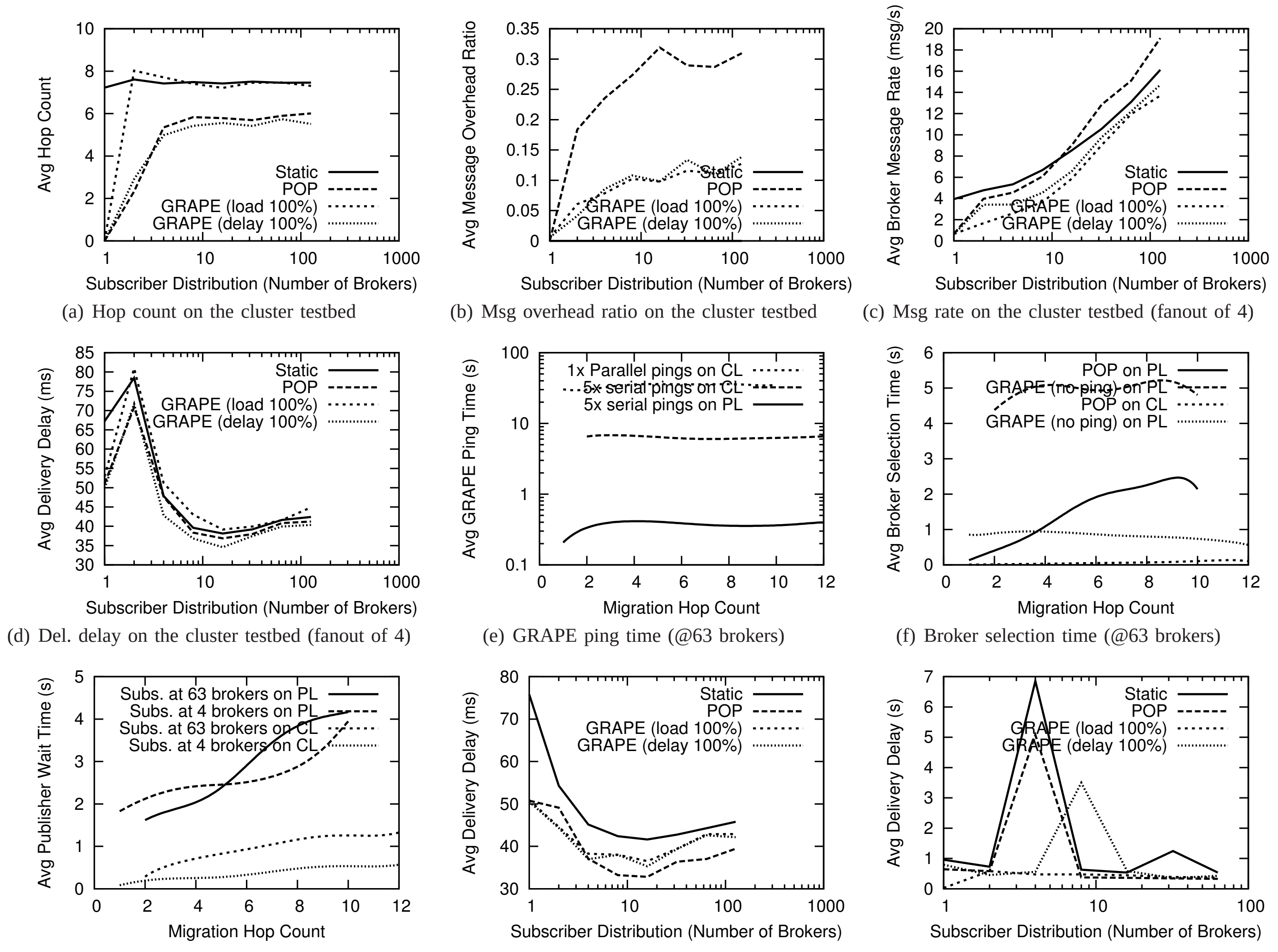
Fig. 5. Experiment results on PlanetLab and the cluster testbed

As expected, the advantages of both POP and GRAPE come at a cost, as Figure 6b and the Appendix [42] reveal that both approaches introduce additional message and memory overhead, respectively. GRAPE introduces far less control messages into the system than POP because, in GRAPE, a reply-like message is only generated after each trace session, whereas in POP, a reply-like message is generated after each traced publication message. The result is that maximum message overhead for POP is 32% and GRAPE is 16%. Note that the results in Figure 6b are specific to the publishers' publication rates in the experiment and trigger settings of each algorithm. The latter parameter is further analyzed below. In terms of memory overhead, both GRAPE settings use up to an additional 58% (or 31 MB) of memory compared to static. POP uses at most 34% (or 19 MB) more memory, thanks to the running average function used in *PPTables* to aggregate values from multiple traces.

Figure 6e shows the average time to obtain pings to 63 brokers for GRAPE with delay 1% setting. This setting is the slowest since it requires the publisher to obtain ping times to all relevant downstream brokers, whereas load 100% is the fastest setting since it requires no pings to be sent. The ping time should be part of GRAPE's broker selection time as shown in Figure 6f, but we have separated them into two graphs for a clearer analysis. Given the large network performance fluctuations on PlanetLab, each publisher takes the average of five pings to each relevant downstream broker with all pings done in serial to obtain the most accurate measurement. Using this approach, pinging 63 brokers required around 30 s on PlanetLab and 7 s on the cluster testbed. For more stable networks such as the cluster testbed, it is possible to ping each broker once with multiple ping threads to bring down the waiting time to less than 0.4 s without any performance degradation as shown in the Appendix [42]. Examining the average broker selection times without the ping times on Figure 6f, it takes GRAPE around 5 s on PlanetLab and 1 s on the cluster to fetch data from all 63 downstream brokers and perform the localized computation. This is higher than POP on both testbeds because GRAPE has to obtain delivery statistics from all downstream brokers. Thus, the subscriber distribution affects GRAPE but not POP. Conversely, the length of the *migration path* has negligible impact on GRAPE's selection time but has a linear effect on POP's selection time.

Figure 6g illustrates that the average time a publisher waits while migrating to the target broker is directly proportional to two variables: (1) the number of brokers on the *migration path* and (2) the distribution of subscribers. This observation makes sense because the longer is the migration path, the higher is the number of brokers that need updating. Likewise, the greater is the number of brokers with matching subscribers, the more routing table update operations are needed.

2) Random Workload: With high-rated subscribers randomly assigned to different brokers, there are virtually no visible differences between the average broker message rate, input and output utilizations, delivery delay, hop count, message overhead, and memory utilization of GRAPE's load 100% and delay 100%. Only the average delivery delay graphs for the cluster testbed and PlanetLab are included here in Figures 6h and 6i, respectively, with the rest in the Appendix [42]. GRAPE's average broker message rate and input utilization matches that of static, with the output utilization of GRAPE surpassing the static case. Message overhead of GRAPE is at most 5%, and is 91% lower than POP. Summarizing POP's results, POP achieves the same average hop count, and input and output utilizations as GRAPE, but introduces



(a) Hop count on the cluster testbed

(b) Msg overhead ratio on the cluster testbed

(c) Msg rate on the cluster testbed (fanout of 4)

(d) Del. delay on the cluster testbed (fanout of 4)

(e) GRAPE ping time (@63 brokers)

(f) Broker selection time (@63 brokers)

(g) Publisher wait time

(h) Delivery delay on the cluster testbed (random)

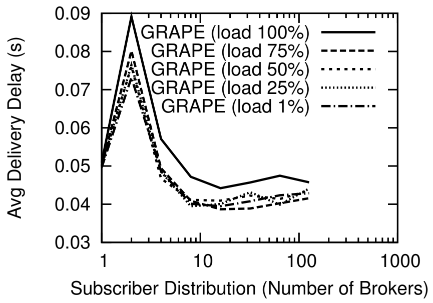
(i) Delivery delay on PlanetLab (random)

Fig. 6. Experiment results on PlanetLab (PL) and the cluster testbed (CL)

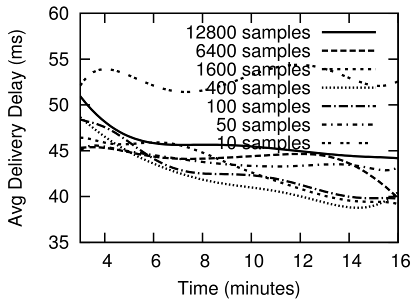
approximately 50% higher broker message rate than GRAPE due to the message overhead from *TRM*s. Figure 6h also illustrates that POP's metric for broker selection is more effective than GRAPE's delay 100% for minimizing delay under the random workload. On PlanetLab, we experienced overloads at about 1-3 internal brokers in our static, POP, and GRAPE's delay 100% experiments. During overloads, subscriber clients experience orders of magnitude higher delivery delays on publication messages as demonstrated by the spikes in Figure 6i. However, no overloads ever happened to GRAPE's load 100% experiments. This shows that GRAPE's load 100% setting is effective in preventing the chances of overloading brokers by minimizing broker load in an unstable environment.

3) Impact of GRAPE's Minimization Weight: We ran experiments with GRAPE set to prioritize minimization of both load and delay from 1% to 100% at increments of 25%. Similar to our previous observations with the random workload, GRAPE behaves indifferently regardless of the prioritization metric and minimization weight settings. However, the reverse is true with the enterprise workload. Interpreting from the graphs in our Appendix [42] and Figure 7a, as the weight on load minimization decreases from 100% to 1%, the average delivery delay and hop count decreases, putting more emphasis on minimizing delay. All the while, load minimization is sacrificed with increased average broker message rate, and input and output utilizations. Moreover, the results of load 1% are virtually identical to delay 100%. Similarly, as GRAPE is varied from delay 100% to delay 1%, we observe an increase in average delivery delay and hop count, and decrease in system load and average broker message rate. Likewise, delay 1% is virtually equivalent to load 100%. No notable differences in memory usage and message overhead are observed over different minimization weights.

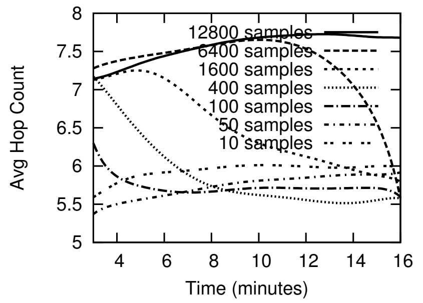
4) Impact of POP and GRAPE's Sampling Trigger: Before broker selection can occur in POP, or delivery statistics are retrieved in GRAPE, a required number of publications must be traced. Recall, that this number is $P_{threshold}$ for POP and $G_{threshold}$ for GRAPE. We experimented with altering the thresholds using the enterprise scenario on the cluster testbed with subscribers distributed among 16 brokers. For POP, we set the maximum number of traceable publications per time window to be half of $P_{threshold}$. Our results from Figures 7b to 7e indicate that increasing $G_{threshold}$ will increase the algorithm's response time and decrease the amount of message



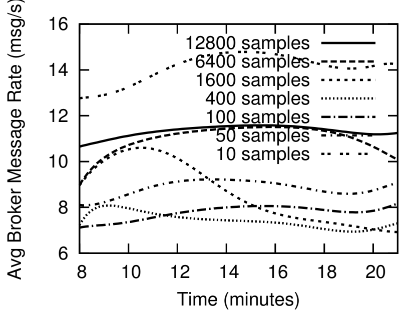
(a) Delivery delay with GRAPE



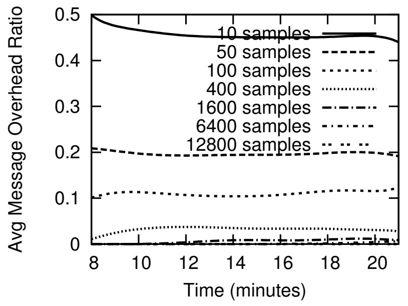
(b) Delivery delay with GRAPE



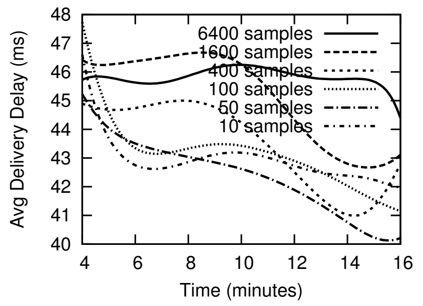
(c) Hop count with GRAPE



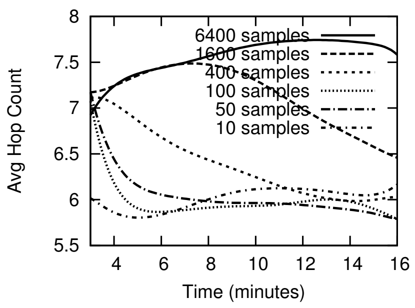
(d) Broker message rate with GRAPE



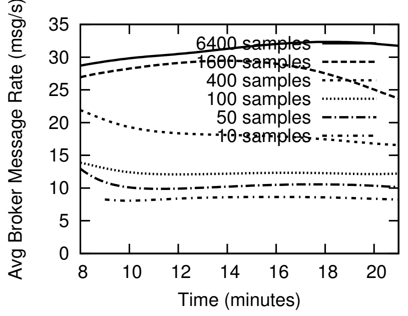
(e) Message overhead with GRAPE



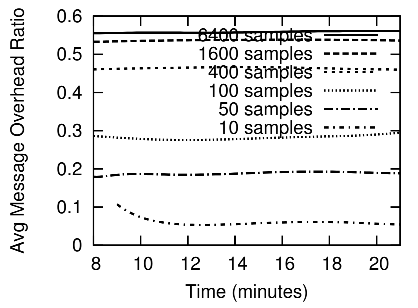
(f) Delivery delay with POP



(g) Hop count with POP



(h) Broker message rate with POP



(i) Message overhead with POP

Fig. 7. Experiment results on the cluster testbed

overhead. For example, at 12800 samples the overhead is lowest because GRAPE never performed a single relocation throughout the experiment. However, at 10 samples, not only is the message overhead high but also the average delivery delay and system loads are higher than if $G_{threshold}$ is set at 400 samples. Therefore, GRAPE performs best when $G_{threshold}$ is set within the hundreds range for best tradeoff between message overhead, response time, and performance gains.

On the contrary, POP's threshold setting behaves different from GRAPE as increasing $P_{threshold}$ will increase both the algorithm's response time and amount of message overhead according to Figures 7f to 7i. Our results indicate that POP achieves best performance gains with fast response time and minimal overhead when $P_{threshold}$ is set below 100.

VI. Conclusions

In this paper, we have shown that strategically relocating publishers in a publish/subscribe network can improve system scalability, robustness, and performance. We have demonstrated this with two unique algorithms, POP and GRAPE, that relocate publishers in the network to minimize average delivery delay and system load. POP is a simple yet effective algorithm that uses only one optimization metric in its calculation, which is the number of average matching downstream subscribers. GRAPE is a more flexible algorithm that allows the prioritization of minimizing average delivery delay, system load, or any combination of both metrics simultaneously while taking a completely different computational approach than POP. For example, all of POP's computations are distributed while GRAPE's core computations are done in a robust yet centralized manner. Nevertheless, both algorithms adopt a 3-Phase architecture where in Phase 1 the algorithms efficiently trace and store publication delivery information, in Phase 2 the algorithms precisely select the target broker to which the publisher should relocate, and in Phase 3 the algorithms orchestrate the publisher migration in a transparent fashion. Together, the three phases give our solutions dynamic, scalable, robust, and transparent properties. Both algorithms are *dynamic* by periodically making relocation decisions based on *live* publication delivery patterns. We observed relocations in our experiments even when the subscriber set remained constant because publications matched different sets of subscribers over time. Both are *scalable* thanks to the use of a distributed design that scales with the number of brokers and clients in the network. We tested both POP and GRAPE using hundreds of brokers and thousands of clients and still witnessed fast algorithm response times with low overheads. Both are *robust* because every broker runs an instance of POP or GRAPE to rule out the possibility of any single point of failure. Our PlanetLab experiments revealed that GRAPE's load 100% setting was effective in preventing broker overloads that could have lead to broker crashes. Lastly, both are *transparent* to application-level publish/subscribe clients as publication statistics gathering and publisher migration all happen behind the scene. Our experiments showed no interruptions to message delivery while POP or GRAPE actively relocated publishers.

Extensive experimental results confirm that our algorithms are effective under enterprise and random workloads on both PlanetLab and a cluster testbed. GRAPE's load 100% setting reduced the average input load of the system by up to 68% and average broker message rate by up to 84%, while GRAPE's delay 100% setting reduced the average delivery delay by up to 68%. GRAPE was able to minimize both average delivery delay and system load simultaneously according to the specified priority and weight. POP consistently reduced both average delivery delay and system load on PlanetLab, but the reductions fell in between the two extreme settings of GRAPE, load 100% and delay 100%, except for the random workload where POP produced the lowest average delivery delay. POP's broker selection time is lower and is dependent on the length of the migration path. GRAPE's broker selection time is higher and is dependent on the number of brokers with matching subscribers. In terms of overhead, both approaches introduced no more than 58% (or 31 MB) more memory overhead. The amount of message overhead from both approaches depended upon the number of publications traced per trace session, which in turn controlled the response time of both approaches.

Taking all results and the unique features of POP and GRAPE into account, we recommend POP for publish/subscribe systems that strive for simplicity (such as GooPS [9]) and expect unpredictable subscription and traffic patterns. On the other hand, we recommend GRAPE for systems that strive to achieve minimal delivery delay (such as Tibco's Supermontage [10]), load usage (such as sensor networks), or require the flexibility to minimize delivery delay when there are available resources (i.e., during off-peak Internet hours) and minimize network traffic when servers are about to overload (i.e., during on-peak Internet hours).

ACKNOWLEDGEMENTS

Thanks to the PADRES team members, specifically Chunyang Ye, Young Yoon, and Vinod Muthusamy for their comments. We are also very grateful to Serge Mankovskii (CA), CA, Sun Microsystems, NSERC, OCE, CFI, and ORF for their support in this research.

REFERENCES

[1] G. Li *et al.*, "Adaptive content-based routing in general overlay topologies," in *Middleware'08*.
[2] S. Pallickara and G. Fox, "NaradaBrokering: a middleware framework and architecture for enabling durable peer-to-peer grids," in *Middleware'03*.
[3] Y. Chen and K. Schwan, "Opportunistic overlays: efficient content delivery in mobile ad hoc networks," in *Middleware'05*.
[4] A. Carzaniga *et al.*, "Design and evaluation of a wide-area event notification service," *ACM ToCS*, 2001.
[5] G. Cugola *et al.*, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE TSE*, 2001.
[6] G. Banavar *et al.*, "An efficient multicast protocol for content-based publish-subscribe systems," in *ICDCS'99*.
[7] P. R. Pietzuch and J. M. Bacon, "Hermes: a distributed event-based middleware architecture," in *DEBS'02*.
[8] R. S. Kazemzadeh and H.-A. Jacobsen, "Reliable and highly available distributed publish/subscribe service," in *SRDS'09*.
[9] J. Reumann, "Pub/sub at google," CANOE and EuroSys Summer School, 2009.
[10] Tibco, "Tibco software chosen as infrastructure for nasdaq's supermontage," 2001. [Online]. Available: www.tibco.com
[11] GS1. [Online]. Available: www.gs1.org/docs/gdsn/gdsn_brochure.pdf
[12] R. Baldoni *et al.*, "Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA," *TCJ*, 2007.
[13] S. Voulgaris *et al.*, "Sub-2-sub: self-organizing content-based publish subscribe for dynamic large scale collaborative networks," in *IPTPS'06*.
[14] M. Adler *et al.*, "Channelization problem in large scale data dissemination," in *ICNP'01*.
[15] L. Opyrchal *et al.*, "Exploiting IP multicast in content-based publish-subscribe systems," in *Middleware'00*.
[16] A. Riabov *et al.*, "Clustering algorithms for content-based publication-subscription systems," in *ICDCS'02*.
[17] ——, "New algorithms for content-based publication-subscription systems," in *ICDCS'03*.
[18] E. Casalicchio and F. Morabito, "Distributed subscriptions clustering with limited knowledge sharing for content-based publish/subscribe systems," in *NCA'07*.
[19] T. Wong *et al.*, "An evaluation of preference clustering in large-scale multicast applications," in *INFOCOM'00*.
[20] F. Cao and J. P. Singh, "Efficient event routing in content-based publish-subscribe service networks," in *INFOCOM'04*.
[21] P. Strong, "eBay - very large distributed systems (a.k.a. grids) @ work," BEinGRID Industry Days, 2008.
[22] S. Ghemawat *et al.*, "The Google file system," *SOSP'03*.
[23] G. Li *et al.*, "A distributed service-oriented architecture for business process execution," *ACM Trans. Web*, 2010.
[24] S. Hu *et al.*, "Distributed automatic service composition in large-scale systems," in *DEBS'08*.
[25] M. Petrovic *et al.*, "G-ToPSS: Fast filtering of graph-based metadata," in *WWW'05*.
[26] I. Rose *et al.*, "Cobra: content-based filtering and aggregation of blogs and RSS feeds," in *NSDI'07*.
[27] B. Mukherjee *et al.*, "Network intrusion detection," *IEEE Network*, 1994.
[28] "Efficient event-based resource discovery," in *DEBS'09*.
[29] K. Graham, "Hedging your bets: currency fluctuations & the supply chain," 2008. [Online]. Available: www.supplyexcellence.com
[30] PADRES. [Online]. Available: www.msrg.org/projects/padres/
[31] S. Hu *et al.*, "Transactional mobility in distributed content-based publish/subscribe systems," in *ICDCS'09*.
[32] A. K. Y. Cheung and H.-A. Jacobsen, "Dynamic load balancing in distributed content-based publish/subscribe," in *Middleware'06*.
[33] G. Li and H.-A. Jacobsen, "Composite subscriptions in content-based publish/subscribe systems," in *Middleware'05*.
[34] B. Segall and D. Arnold, "Elvin has left the building: a publish/subscribe notification service with quenching," in *AUUG'97*.
[35] A. M. Ouksel *et al.*, "Efficient probabilistic subsumption checking for content-based publish/subscribe systems," in *Middleware'06*.
[36] G. Li *et al.*, "Routing of xml and xpath queries in data dissemination networks," in *ICDCS'08*.
[37] Y.-M. Wang *et al.*, "Summary-based routing for content-based event distribution networks," *SIGCOMM CCR*, 2004.
[38] P. Triantafillou and A. A. Economides, "Subscription summaries for scalability and efficiency in publish/subscribe systems," in *ICDCS'02*.
[39] Z. Jerzak and C. Fetzer, "Bloom filter based routing for content-based publish/subscribe," in *DEBS'08*.

[40] R. Baldoni *et al.*, “TERA: topic-based event routing for peer-to-peer architectures,” in *DEBS’07*.

[41] V. Muthusamy *et al.*, “Effects of routing computations in content-based routing networks with mobile data sources,” in *MobiCom’05*.

[42] A. K. Y. Cheung and H.-A. Jacobsen, “Appendix,” 2009. [Online]. Available: research.msrg.utoronto.ca/twiki/pub/Padres/WebHome/appendix.pdf

[43] P. Sharma *et al.*, “Estimating network proximity and latency,” *SIGCOMM CCR*, 2006.

[44] F. Dabek *et al.*, “Vivaldi: a decentralized network coordinate system,” in *SIGCOMM’04*.