Publisher Relocation Algorithms for Minimizing Average Delivery Delay and System Load [⋆]

Alex King Yeung Cheung and Hans-Arno Jacobsen

Middleware Systems Research Group
University of Toronto, Toronto, Ontario, Canada
`http://www.msrg.utoronto.ca`
{`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 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 performance 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 68%, average message rate by 85%, and average delivery delay by 68%.

Keywords: Publish/subscribe, publisher migration, content-based routing, delivery delay minimization, system load minimization, bit vectors.

1 Introduction

Many filter-based publish/subscribe systems assume that publishers and subscribers join the broker federation by connecting to the closest broker [11, 26, 10] or to any broker with no restrictions [8, 13, 6, 28]. 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 offset any advantages gained from the "short" client-to-broker link. The problem of reducing in-network processing and transmission delays on publication messages has previously been addressed by reconfiguring the broker topology [4], clustering subscribers into multicast-groups to limit publication propagation only among interested peers [36, 1, 24, 29, 30, 9, 38], or incorporating multicast-groups with filter-based approaches [7].

In this paper, we present two new algorithms that intelligently relocate publishers while keeping the broker overlay intact to minimize both the average

[⋆] CSRG-597, Middleware Systems Research Group, University of Toronto, 2009.

end-to-end delivery delay and system load. Our first algorithm is POP *(Publisher Optimistic Placement)* which is a fully distributed algorithm that relocates publishers to areas of the overlay with the highest number of publication deliveries. POP is able to reduce the average broker input utilization by 64%, system message rate by 85%, and delivery delay by 63% on PlanetLab. Our second algorithm, GRAPE *(Greedy Relocation Algorithm for Publishers of Events)* is an evolution of POP that not only reduces POP's message overhead by up to 91%, but also allows the prioritization of minimizing average delivery delay, system load, or any combination of both metrics simultaneously. GRAPE is able to reduce the average broker input utilization by 68%, system message rate by 85%, and delivery delay by 68% on PlanetLab.

Both POP and GRAPE are primarily targeted at enterprise-grade messaging systems consisting of hundreds to thousands of dedicated servers [33, 16]. The type of messaging system that we focus on is content-based publish/subscribe which enables individual entities to communicate asynchronously with each other in a loosely-coupled manner [15]. A motivating application scenario involves traveling business agents who have to bid on the cost of raw materials in different countries to build a car. The employer company of these agents has a publish/subscribe cloud that consists of a federated network of brokers residing in major office locations around the world. Assuming that each branch office in the world is responsible for designing a specific component of a car, each branch office will have many end-users (subscribers) who are interested in (subscribes to) competitive prices (a subset of the publications) reported (published) by agents (publishers) on selected raw materials. Whenever agents have information to report on, they will simply connect their PDA to the publish/subscribe broker cloud and publish their information. With GRAPE set to minimize solely on average delivery delay, GRAPE will reconnect a publisher, say on electrolyte prices, to the broker at the branch office where many engineers are interested in inexpensive lithium-ion battery packs. On the other hand, with GRAPE set to minimize solely on system load, GRAPE will reconnect the publisher to connect to the head office which has a database that sinks all of the agent's traffic. With POP, the outcome can be anywhere between the two extremes of GRAPE. Other application scenarios of POP and GRAPE include workflow management [13], RSS dissemination [31], network and systems monitoring [22], and more.

Both POP and GRAPE follow the same 3-Phase operational design: (1) gather publication delivery statistics on the publishers' publications, (2) identify the target broker to relocate the publisher, and (3) transparently migrate the publisher to the target broker. However, the actual methodologies to realize Phases 1 and 2 are completely different between POP and GRAPE. For example, the core of GRAPE follows a centralized design that is much easier to implement and verify while preserving the same level of robustness as POP. Together, these three phases contribute to our 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 a distributed design that scales with the number of brokers and publishers in the network. Both are *robust* because the distributed operations 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. Another common property of POP and GRAPE is that both use purely publish/subscribe messaging for internal communications.

The main contributions of this paper are: (1) the POP algorithm with its publication tracking, broker selection, and publisher migration algorithms presented in Section 3; (2) the GRAPE algorithm with its own unique publication tracking and broker selection algorithms described in Section 4; and (3) extensive experiments on PlanetLab and a cluster testbed that quantitatively validate our two approaches as illustrated in Section 5.

2 Background and Related Work

2.1 Publish/Subscribe Systems

Two main classes of distributed content-based publish/subscribe systems exists today: filter-based [13, 6, 32, 20, 8, 11, 28, 26, 10] and multicast-based [1, 24, 29, 30, 9, 38]. 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 [7].

By reconnecting publishers to areas of most populated and/or highest-rated matching subscribers in filter-based approaches, POP and GRAPE reduces 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 focuses 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 [21, 25], subscription merging [37], and subscription summary [35, 19] optimizations.

A number of approaches are found in literature that also try to reduce the overlay distance between publishers and subscribers. Baldoni *et al.* [4] 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 [5] 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 [36] 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. There are other peer-to-peer publish/subscribe systems as well [3, 34, 17, 2], but their network architectures are fundamentally different from our work.

2.2 Publisher Migration Protocols

Muthusamy *et al.* [23] proposed several publisher migration protocols with different optimization techniques to study the effects of publisher 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 [18]. In terms of overhead message count, our approach generates $O(M)$ messages, whereas the approach in [23] generates $O(N)$ messages, where M is the number of brokers along the *migration path* and N is the total number of brokers in the network. 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 [23] 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.

One of the secondary goals of our publisher migration protocol is to minimize the amount of overhead generated as a result of migrating publishers. Picco *et al.* [27] developed a protocol to minimize the amount of overhead generated from network reconfiguration as a result of failing broker links. However, their scheme does not apply in our context since our objective is not to reconfigure routing tables to adapt to unexpected broker topology changes.

3 Publisher Optimistic Placement

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*, *B7* and *B8* are downstream brokers while *B5* and *B1* are upstream brokers.

The following subsections show POP's 3-Phase operation in detail. Section 3.1 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 3.2 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 3.3 presents Phase 3, which involves transparently migrating the publisher to the broker identified in Phase 2 with minimal routing table updates. We have included message sequence diagrams detailing each Phase under Figure 1's scenario as further clarification for the reader. 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.

3.1 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.

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

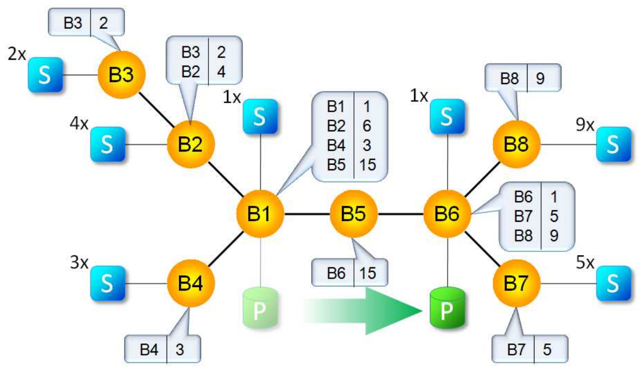


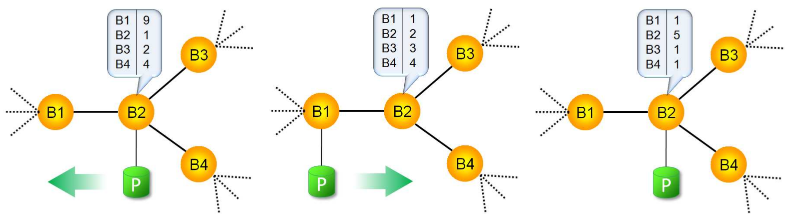
Fig. 1. Example of *Publisher Profile Table*

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. In contrast to the *KER* protocol in [14], POP's reply messages (or *TRM*s) report back aggregated data.

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.

3.2 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 sends a *Relocation Request Message (RRM)* to itself. 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.



(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 decentralized broker selection algorithm

When a broker handles 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. 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 *brokers 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 *B6*'s value in the *PPTable* is greater than the value from *B1*'s *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 15 subscribers on the link to *B5*. Since no neighbor is able to satisfy the *closest* condition, *B6* concludes itself to be the *closest* broker and sends a *Relocation Answer Message (RAM)* back to *B1* to initiate Phase 3.

3.3 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 will intercept 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 sends two control messages to the publisher to complete the migration. One to notify the publisher to resume publishing. Two, to notify the publisher to disconnect from the *source* broker.

Notice that our publisher migration protocol limits the amount of computational and message overhead to *O(M)*, where M is the set of brokers along the *migration path*. Since the network we consider is always a tree, there is always only one path. 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 transparent 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 on PlanetLab takes no more than 5 s.

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*, which includes brokers *B1*, *B5*, and *B6*: broker *B6* updates publisher P's advertisement last hop to a local destination, broker *B5* updates P's advertisement last hop to *B6*, and broker *B1* updates P's advertisement last hop to *B5*. Operation #2 applies to brokers *B1* and *B5*, where they have to remove subscription(s)[1] from the 15 subscribers that reside on the right of broker *B5*. Operation #3 applies to brokers *B1* and *B5*, where they have to forward subscription(s)[1] from the 10 subscribers that reside on the left of *B5* to broker *B6*.

4 Greedy Relocation Algorithm for Publishers of Events

Like POP, GRAPE follows the same 3-Phase operational design and uses the publisher migration protocol presented in Section 3.3. However, the data structures and algorithms that GRAPE uses in Phases 1 and 2 are completely dif-

[1] Depends if subscriptions are unique and subscription covering is available.

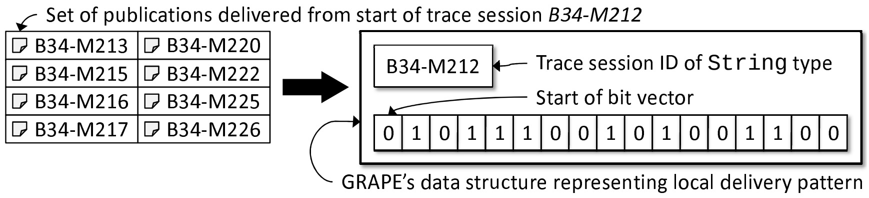

Set of publications delivered from start of trace session B34-M212
B34-M213
B34-M220
B34-M215
B34-M222
B34-M216
B34-M225
B34-M217
B34-M226
B34-M212
Trace session ID of String type
Start of bit vector
0 1 0 1 1 1 0 0 1 0 1 0 0 1 1 0 0
GRAPE's data structure representing local delivery pattern

Fig. 3. `String` and bit vector representation of delivered publication messages

ferent. As we will show in our evaluation, GRAPE uses 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 worst case.

4.1 Phase 1: Distributed Publication Tracing

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. One is 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. Two is 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 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.

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 its own and all downstream brokers' trace information in the payload. Brokers with no downstream neighbors can immediately reply back with a *TIA* message to the upstream neighbor with a payload containing the following information about itself: (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. Please see our online Appendix [12] 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*. As a result, GRAPE's message overhead is reduced by up to 91% even though more information is retrieved.

4.2 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 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 benefit of adopting a centralized approach is the ease of design, implementation, and verification which allows GRAPE to be no more complex but still more powerful than POP.

GRAPE allows the user to prioritize between minimization of average end-to-end delivery *delay* or system *load* (in the form of message rate), and also specify a 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. Because ping packets are filtered on certain PlanetLab nodes, publishers instead invoke an API on the brokers to obtain round trip times. The

Algorithm 1 calcAvgDelay(deliveryStats, cumulativeDelay, currBroker, prevBroker)

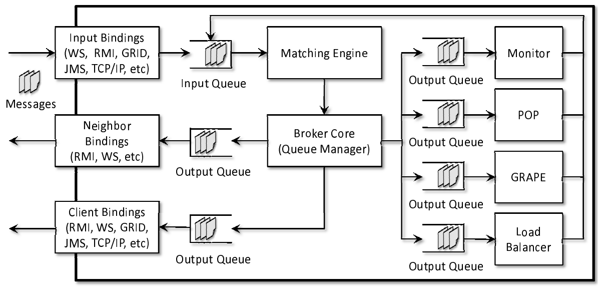
```
// Get delivery statistics from clients on the current broker
$deliveryStats.totalDelay += currBroker.totalDeliveries \times (cumulativeDelay +$
  $currBroker.queueAndMatchDelaysTo(client))$
$deliveryStats.totalDeliveries += currBroker.totalDeliveries$
for $neighbor$ in $currBroker.neighborSet$ do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

Algorithm 2 calcTotalMsgRate(currBroker, prevBroker)

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

Algorithm 3 calcDownstreamBV(currBroker, prevBroker)

```
$aggregatedBV = currBroker.bitVector$
// Take local deliveries (above) and OR with the downstream broker deliveries
for $neighbor$ in $currBroker.neighborSet$ do
  if $neighbor$ equals $prevBroker$ then
    continue
  end if
  $downstreamBV = currBroker.getDownstreamBVTo(neighbor)$
  if $downstreamBV$ is $NULL$ then
    // Calculate the downstream bit vector and store for future retrieval
    $downstreamBV = calcDownstreamBV(neighbor, currBroker)$
    $currBroker.setDownstreamBVTo(neighbor, downstreamBV)$
  end if
  $aggregatedBV \mid= downstreamBV$
end for
return $aggregatedBV$
```


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

(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
POP trigger	50	100
GRAPE trigger	50	100

(b) Deployment specs

Fig. 4. Experimentation details

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. After invoking `calcAvgDelay()` on each candidate, GRAPE will normalize the candidates' delivery delays and drop 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. `calcTotalMsgRate()` uses the helper function `calcDownstreamBV()` (shown in Algorithm 3) to aggregate downstream broker bit vectors to compute the total message rate. The candidate that offers the lowest system-wide 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 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.

5 Experiments

Both POP and GRAPE are implemented on top of PADRES [11], a Java-based 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 follow 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.

5.1 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 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 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 actual 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,18.62],[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 publisher's traffic, 25% are medium-rated, meaning they sink about 50% of their publisher's traffic, and 5% are high-rated, meaning they sink *all* of their publisher's 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 *clustered*, 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 clustered 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. The number of publishers, subscribers, brokers and all other settings

[2] Data set available at `www.msrg.utoronto.ca/∼cheung/grape/stockquotes.zip`

we used on PlanetLab and the cluster testbed are shown in Figure 4b. Unless otherwise stated, default POP and GRAPE settings are used.

5.2 Experiment Results

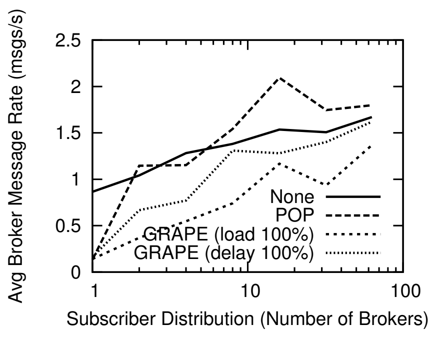
We evaluated POP and GRAPE under random and clustered workloads while varying the subscriber distributions, minimization weights for GRAPE, and number of samples for triggering broker selection. For graphs without time as the x-axis, the plotted values are obtained at the end of the experiment, which 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 subset of the graphs here with the rest in an online Appendix [12]. Nevertheless, we summarize all of our results here in this paper. From this point on, we will use the notation *load75* (*delay25*) to denote GRAPE's configuration to prioritize on minimizing average system load (delivery delay) with 75% (25%) weight.

Clustered Workload. Under the enterprise scenario, Figure 5a shows that GRAPE's *load100* yields the lowest average system message rate compared to GRAPE's *delay100*, POP, and baseline with neither GRAPE nor POP enabled. This is true 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 brokers where all the matching subscribers reside. Lower average message rate translate directly to lower average broker input (Figure 5b) and output utilizations [12]. *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 *load100* 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 are lower rated. As a result, average delivery delay and hop count are sacrificed as shown in Figures 5c and 5d, respectively. Although not shown, results on the cluster testbed reveal that GRAPE's *load100* achieves minimal message rate at the cost of higher delivery delay than even the baseline case.

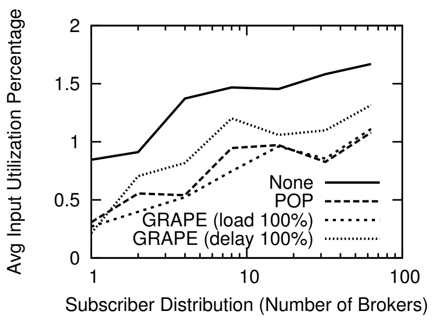
With GRAPE's *delay100* setting, the system achieves lowest average delivery delay across almost all subscriber distributions as shown in Figure 5c. This is due to placing the publishers closest to the subscribers with the highest number of publication deliveries, which dominates what is shown in Figure 5d. However, because the publishers are further away from the high-rated subscribers, the system has to transmit more messages overall (Figure 5a) which leads to higher input (Figure 5b) and output utilizations. The same observations are witnessed on the cluster testbed as well [12].

POP minimizes both average system load and delivery delay simultaneously without offering a choice of which metric to prioritize and by how much. From Figures 5a to 5d, POP's input utilization, delivery delay, and hop count is between GRAPE's *load100* and *delay100*. However, the average message rate

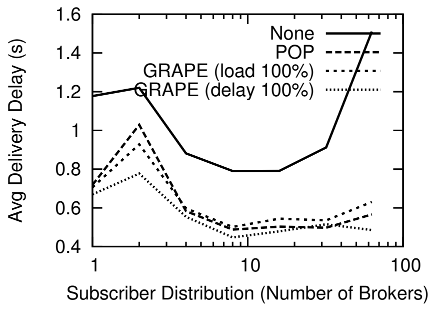
[3] Please see [11] for definitions of these metrics.



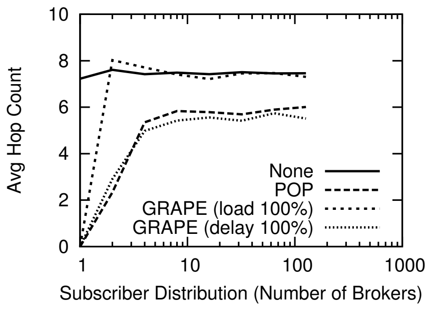
(a) Broker message rate on P. Lab



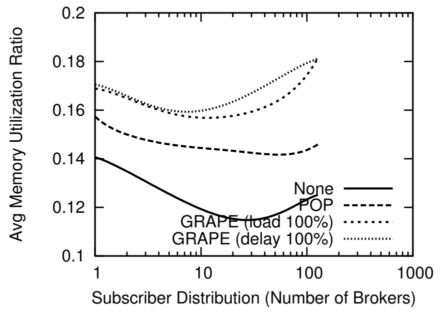
(b) Input util percentage on P. Lab



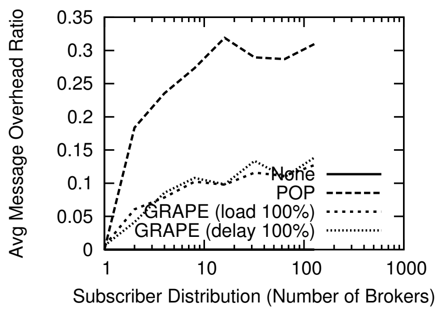
(c) Delivery delay on P. Lab



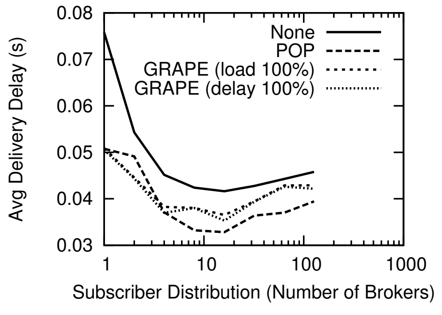
(d) Hop count on cluster



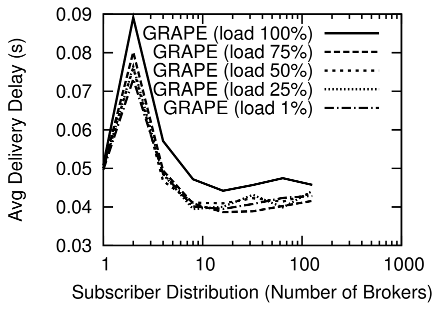
(e) Memory util ratio on cluster



(f) Message overhead ratio on cluster



(g) Del. delay on cluster (rand. workload)



(h) Delivery delay on cluster

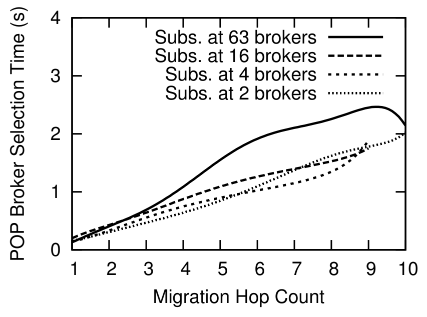
Fig. 5. Experiment results on PlanetLab (P. Lab) and the cluster testbed

of POP exceeds that of GRAPE due to the increased message overhead from *TRM*s.

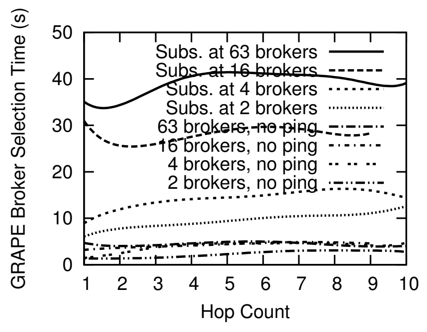
As we expected, the advantages of both POP and GRAPE comes at a cost, as Figures 5e and 5f reveal that both approaches introduce additional memory and message overhead. With a ratio of 1.0 representing 512 MB of memory, both GRAPE settings use up to an additional 58% (or 31 MB) of memory compared to the baseline. 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. Message overhead wise, 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 5f are specific to the publishers' publication rates in the experiment and trigger settings of each algorithm. The latter parameter is further analyzed below.

Figures 6a and 6b show the broker selection time for POP and GRAPE with *delay1* setting. The *delay1* setting is the worse case for GRAPE since it requires the publisher to obtain ping times to all relevant downstream brokers and GRAPE has to calculate the system-wide message rates for all those brokers as well. Note that the subscriber distribution does not affect POP as much as the length of the *migration path*. Compared to POP, GRAPE spends more time to select a broker. However, much of that time is spent by the publisher obtaining ping times to brokers as shown by the top four lines in Figure 6b. If we subtract the ping time retrieval from the total broker selection time, then it takes at most 5 s to fetch data from all relevant downstream brokers and perform the localized computation. Still this is higher than POP because GRAPE has to obtain delivery statistics from all downstream brokers. Note that the length of the *migration path* has negligible impact on GRAPE's selection time. Figure 6c illustrates that the average time a publisher waits while migrating to the target broker is directly proportional to the number of brokers on the *migration path*. The longer the path, the more routing tables are updated, the longer is the waiting time.

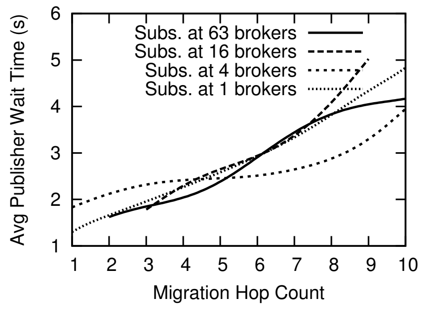
Random Workload. With high-rated subscribers randomly assigned to different brokers, there are virtually no visible differences between the average message rate, input and output utilizations, delivery delay, hop count, message overhead, and memory utilization of GRAPE's *load100* and *delay100*. Only the delivery delay graph is included here in Figure 5g, with the rest in the Appendix [12]. GRAPE's average broker message rate and input utilization matches that of the baseline, with the output utilization of GRAPE surpassing the baseline 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 approximately 50% higher message rate than GRAPE due to the message overhead from *TRM*s. Figure 5g illustrates that POP's metric for broker selection is more effective than GRAPE's *delay100* for minimizing delay under the random workload. On



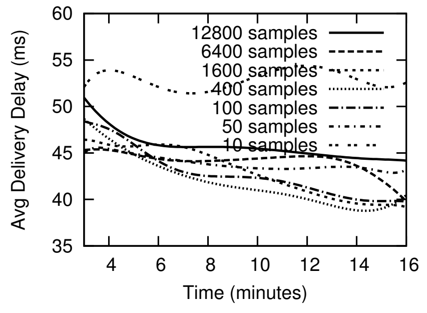
(a) POP selection time on P. Lab



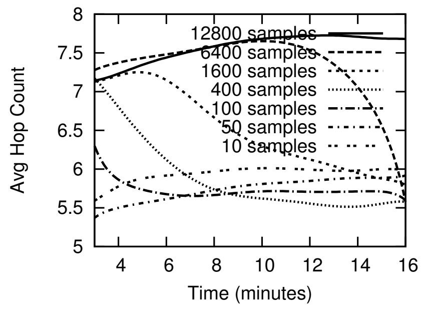
(b) GRAPE selection time on P. Lab



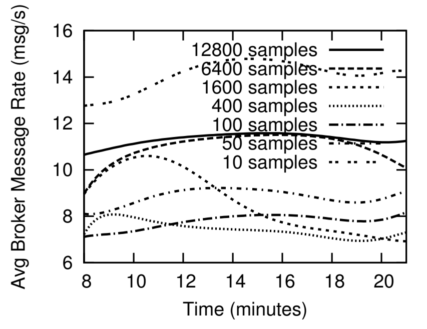
(c) Publisher wait time on P. Lab



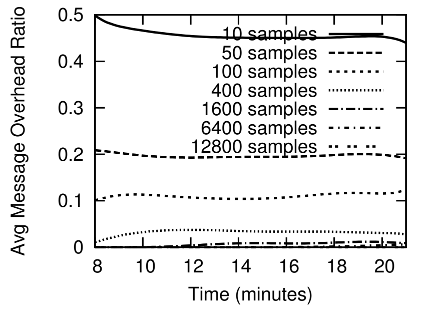
(d) GRAPE delivery delay



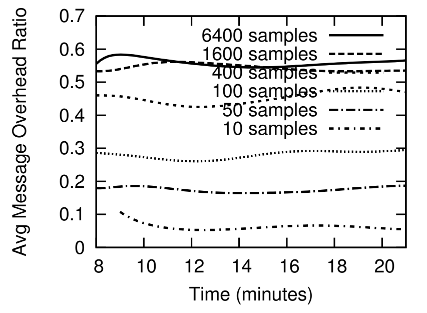
(e) GRAPE hop count



(f) GRAPE broker message rate



(g) GRAPE message overhead ratio



(h) POP message overhead ratio

Fig. 6. Experiment results on PlanetLab (P. Lab) and the cluster testbed

PlanetLab, we experienced overloads at about 1-3 internal brokers in our baseline, POP, and GRAPE's *delay100* experiments. However, no overloads ever happened to GRAPE's *load100* experiments. This shows that GRAPE's *load100* setting is effective in minimizing or preventing the chances of overloading brokers in an unstable environment.

Impact of Varying 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, there are no distinct differences in performance regardless of the prioritization metric nor weight setting under the random workload. However, GRAPE responds differently under the clustered workload. Interpreting from the graphs in our Appendix [12] and Figure 5h, 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 message rate, and input and output utilizations. Moreover, the results of *load1* are virtually identical to *delay100*. Similarly, as GRAPE is varied from *delay100* to *delay1*, we observe an increase in average delivery delay and hop count, and decrease in system load and message rate. Likewise, *delay1* is virtually equivalent to *load100*. No notable differences in memory usage and message overhead are observed over different minimization weights.

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 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 6d to 6g indicate that increasing $G_{threshold}$ will increase the algorithm's response time and decrease the amount of message overhead. At the extreme setting of 10, not only is the system message rate higher than the baseline case, but also the final average hop count is higher than all other cases. 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 (Figure 6h).

6 Conclusions

In this paper, we have presented two new algorithms, POP and GRAPE, that relocates publishers in the network to minimize average delivery delay and system load. While the POP algorithm is simple, GRAPE further allows the prioritization of minimizing average delivery delay, system load, or any combination of both metrics simultaneously. This extra functionality does not increase the complexity of GRAPE in terms of design, implementation, or verification because, unlike POP, 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 algorithm efficiently traces and stores publication delivery information, in Phase 2 the algorithm precisely selects the target broker to which the publisher should relocate, and in Phase 3 the algorithm orchestrates the publisher migration in a transparent fashion. Together, the three phases give our solutions robust, scalable, dynamic, and transparent properties. Extensive experimental results confirm that our algorithms are effective under clustered and random workloads on both PlanetLab and a cluster testbed. GRAPE's *load100* setting reduced the average input load of the system by 68% and average message rate by 84%, while GRAPE's *delay100* setting reduced the average delivery delay by 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, *load100* and *delay100*, except for the random workload where POP produced the lowest average delivery delay. Broker selection took up to 17× longer in GRAPE than in POP due to a large portion of the time spent obtaining ping times between the publisher and candidate brokers. Overhead-wise, 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. Our results show that GRAPE's message overhead was lower than POP by up to 91%. In the future, we plan on developing a relocation algorithm for subscribers that work together with GRAPE to further minimize average delivery delay and system load.

References

1. M. Adler et al. Channelization problem in large scale data dissemination. In *ICNP*, 2001.
2. I. Aekaterinidis and P. Triantafillou. Internet scale string attribute publish/subscribe data networks. In *CIKM*, 2005.
3. I. Aekaterinidis and P. Triantafillou. PastryStrings: A comprehensive content-based publish/subscribe DHT network. In *ICDCS*, 2006.
4. R. Baldoni et al. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *TCJ*, 2007.
5. R. Baldoni et al. TERA: Topic-based event routing for peer-to-peer architectures. In *DEBS*, 2007.
6. G. Banavar et al. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, 1999.
7. F. Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *INFOCOM*, 2004.
8. A. Carzaniga et al. Design and evaluation of a wide-area event notification service. *ACM ToCS*, 2001.
9. E. Casalicchio and F. Morabito. Distributed subscriptions clustering with limited knowledge sharing for content-based publish/subscribe systems. In *NCA*.
10. Y. Chen and K. Schwan. Opportunistic Overlays: Efficient content delivery in mobile ad hoc networks. In *Middleware*, 2005.
11. A. K. Y. Cheung and H.-A. Jacobsen. Dynamic load balancing in distributed content-based publish/subscribe. In *Middleware*, 2006.

12. A. K. Y. Cheung and H.-A. Jacobsen. Appendix. www.msrg.utoronto.ca/∼cheung-/grape/appendix.pdf, 2009.
13. G. Cugola et al. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE TSE*, 2001.
14. G. Cugola et al. On adding replies to publish-subscribe. In *DEBS*, 2007.
15. P. T. Eugster et al. The many faces of publish/subscribe. *ACM CSUR*, 2003.
16. S. Ghemawat et al. The Google file system. *SOSP*, 2003.
17. A. Gupta et al. Meghdoot: Content-based publish/subscribe over P2P networks. In *Middleware*, 2004.
18. S. Hu et al. Transactional mobility in distributed content-based publish/subscribe systems. In *ICDCS*, 2009.
19. Z. Jerzak and C. Fetzer. Bloom filter based routing for content-based publish/subscribe. In *DEBS*, 2008.
20. G. Mühl. Generic constraints for content-based publish/subscribe systems. In *CoopIS*, 2001.
21. G. Mühl et al. Filter similarities in content-based publish/subscribe systems. In *ARCS*, 2002.
22. B. Mukherjee et al. Network intrusion detection. *IEEE Network*, 1994.
23. V. Muthusamy et al. Effects of routing computations in content-based routing networks with mobile data sources. In *MobiCom*, 2005.
24. L. Opyrchal et al. Exploiting IP multicast in content-based publish-subscribe systems. In *Middleware*, 2000.
25. A. M. Ouksel et al. Efficient probabilistic subsumption checking for content-based publish/subscribe systems. In *Middleware*, 2006.
26. S. Pallickara and G. Fox. NaradaBrokering: A middleware framework and architecture for enabling durable peer-to-peer grids. In *Middleware*, 2003.
27. G. P. Picco et al. Efficient Content-Based Event Dispatching in Presence of Topological Reconfigurations. In P. McKinley and S. Shatz, editors, *ICDCS*, May 2003.
28. P. R. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. In *DEBS*, 2002.
29. A. Riabov et al. Clustering algorithms for content-based publication-subscription systems. In *ICDCS*, 2002.
30. A. Riabov et al. New algorithms for content-based publication-subscription systems. In *ICDCS*, 2003.
31. I. Rose et al. Cobra: Content-based filtering and aggregation of blogs and RSS feeds. In *NSDI*, 2007.
32. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *AUUG*, 1997.
33. P. Strong. eBay - very large distributed systems (a.k.a. grids) @ work. BEinGRID Industry Days, 2008.
34. W. W. Terpstra et al. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.
35. P. Triantafillou and A. A. Economides. Subscription summaries for scalability and efficiency in publish/subscribe systems. In *ICDCS*, 2002.
36. S. Voulgaris et al. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *IPTPS*, 2006.
37. Y.-M. Wang et al. Summary-based routing for content-based event distribution networks. *SIGCOMM Comp. Comm. Rev.*, 2004.
38. T. Wong et al. An evaluation of preference clustering in large-scale multicast applications. In *INFOCOM*, 2000.