Publisher Mobility in Distributed Publish/Subscribe Systems

Vinod Muthusamy[†], Milenko Petrovic[†], Dapeng Gao[†], Hans-Arno Jacobsen[†‡]
Middleware Systems Research Group
[†]Department of Electrical and Computer Engineering
[‡]Department of Computer Science
University of Toronto
{vinod,petrovi,gilbert,jacobsen}@eecg.toronto.edu

Abstract

The decoupling of producers and consumers in the publish/subscribe paradigm lends itself well to the support of mobile users who roam about the environment with intermittent network connectivity. This paper presents the first quantitative evaluation of publisher mobility in a distributed publish/subscribe system. Our results indicate that publisher mobility breaks a fundamental assumption of publish/subscribe systems and has a significant performance impact. We formalize publisher mobility algorithms for a distributed publish/subscribe system, and develop and evaluate optimizations to the mobile publisher algorithms.

1 Introduction

User-friendly tools such as blogs and wikis make it increasingly easier for non-computer oriented users to publish information on the Internet, and the number of information publishers has grown considerably. With advances to and increasing pervasiveness of portable wireless devices with Internet access, we foresee the publishing trend on the Internet to intensify in the portable computing space.

The publish/subscribe (pub/sub) paradigm has been studied in the context of selective information dissemination on the Internet [6, 8, 16]. Pub/Sub systems have a number of desirable characteristics for mobile information dissemination applications. They can efficiently filter and disseminate large amounts of data to a large number of users [8]. Also, they decouple communication, both in time and space, allowing publishers and subscribers to communicate without having to be connected simultaneously or having to know about each other. Therefore, the publish/subscribe paradigm naturally supports mobile publishers.

Most existing research assumes that both publishers and subscribers are not mobile [1, 2, 8]. While some research has studied subscriber mobility [4, 5, 7], we are not aware of any that examines publisher mobility. We will see that this is important because publisher mobility breaks a fundamental assumption of pub/sub systems, namely that the number of advertisement messages is much less than other messages. The most important contributions in this paper are the formalization of publisher mobility algorithms for distributed pub/sub systems, and the development and experimental evaluatation of optimizations that reduce the costs associated with mobile publishers.

This work is part of the ToPSS (Toronto Publish/Subscribe System) research projects [4, 10, 11, 13, 14]; especially the Mobile-ToPSS effort investigating support for mobility in publish/subscribe systems.

Section 2 provides an overview of the pub/sub paradigm. Section 3 describes the algorithm to support mobile publishers, and develops a number of optimizations to this algorithm. Section 4 presents an experimental evaluation of the mobility algorithm and proposed optimizations. Section 5 discusses related work and puts our work in perspective. Finally, Section 6 concludes the paper and discusses directions for future work.

2 Background

The pub/sub paradigm is effective at supporting information dissemination applications [1, 2, 7, 8]. Information producers (*publishers*) send information with publication messages, while information consumers (*subscribers*) express their interest in publications using subscription messages. The central component of a pub/sub system, the *broker*, records all subscriptions, and matches publications against all subscriptions. On a match, the broker notifies the corresponding subscribers. It is important to note that messages from the publishers (publications) do not contain any address; instead, they are routed through the system based on their content (for a content-based system). The broker ar-

chitecture can be centralized or distributed.

Publish/subscribe systems can be based on the notion of *topics* (or *subjects*), *types*, or *content*. In topic-based pub/sub, clients can subscribe to several topics and receive notifications about all publications within these topics. *Type-based* pub/sub systems are similar to topic-based, but use publication types instead of topics for matching. *Content-based* pub/sub systems improve expressiveness by allowing subscriptions to contain complex queries on the publication content. The subject space model [9] improves the expressiveness of subscriptions by persisting both subscriptions and publications.

Two main optimizations were introduced in the literature in order to increase the performance of these forwarding algorithms: subscription covering and advertisements [6].

Subscription Covering: Given two subscriptions s_1 and s_2, s_1 covers s_2 if and only if all the publications that match s_2 also match s_1. When a broker B receives a subscription s, it will send it to its neighbours if and only if it has not previously sent them another subscription s', that covers s.

Advertisements: Advertisements are used by publishers to announce the set of publications they are going to publish. Advertisements look like subscriptions but are used to build the routing path from the publishers to the interested subscribers. An advertisement a determines a publication e if and only if all attribute-value pairs match some predicates in the advertisement. An advertisement a intersects a subscription s if and only if the intersection of the set of the publications determined by the advertisement a and the set of the publications that match s is a non-empty set.

Upon receiving a subscription, a broker forwards it only to neighbours that previously sent advertisements that intersect the subscription. Thus, subscriptions are forwarded only to brokers with potentially interesting publishers.

Distributed Pub/Sub: In a distributed broker architecture, a network of brokers collaborate to route information based on its content [6, 7, 2]. In one popular distributed pub/sub algorithm [6], advertisements from publishers are flooded throughout the network, and build a distributed advertisement tree. Subscriptions flow in the reverse path of intersecting advertisement trees, and result in a distributed multicast tree. Finally publications from a publisher follow the reverse path of matching subscriptions (i.e. the multicast tree) and are delivered to interested subscribers.

A fundamental assumption of pub/sub systems is that the number of advertisements is much less than the number of subscriptions which is much less than the number of publications. This justifies the flooding of advertisements. However, we will show that this assumption is not valid when publishers are mobile.

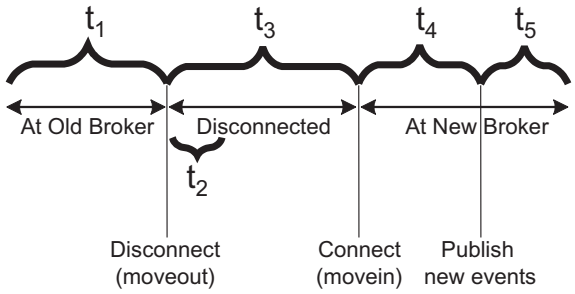

Figure 1. Publisher mobility timeline

Algorithm *receive*(Advertisement a)
($*$ Store and foward an advertisement. $*$)
1. ($*$ Store in table $*$)
2. inAds $\leftarrow$ inAds $\cup$ $(a, a.sender)$
3.
4. ($*$ Forward ad to neighbours $*$)
5. **for** each neighbour n where $n \neq a.sender$
6. **do if** $\exists(ad, n, true) \in$ outAds where $ad.covers(a)$
7. **then** outAds $\leftarrow$outAds $\cup$ $(a, n, false)$
8. **else** outAds $\leftarrow$outAds $\cup$ $(a, n, true)$
9. send(a, n)
10.
11. ($*$ Send matching subscriptions back towards ad $*$)
12. **for** each $(s, n) \in$ inSubs where $s.intersects(a)$ and $n \neq a.sender$
13. **do if** $\exists(sub, n, true) \in$ outSubs where $sub.covers(s)$
14. **then** outSubs $\leftarrow$outSubs $\cup$ $(s, a.sender, false)$
15. **else** outSubs $\leftarrow$outSubs $\cup$ $(s, a.sender, true)$
16. send$(s, a.sender)$

Figure 2. Advertisement handler

3 Publisher Mobility Algorithms

Client mobility is based on "movein" and "moveout" operations [7] that offer clients the ability to disconnect from and reconnect to the system. To support disconnected operation of subscribers, brokers must store publications for the subscriber, and replay them to the subscriber when it reconnects to the network. However, there are no special algorithms to handle publisher mobility. Unlike disconnected operation with subscribers, there is no information that the publisher would have missed while disconnected. It is perhaps for this reason that no new algorithms have been proposed for handling publisher mobility. We will show in Section 4 why this is a problem.

Below we describe the standard publisher mobility algorithm as well as some optimizations that we propose.

Standard Algorithm: Figure 1 illustrates a timeline of a mobile publisher. During period t_1, the publisher is connected to Broker 1, and the publisher rooted advertisement and multicast trees have been built. At the end of period t_1, the publisher disconnects from Broker 1 and reconnects after period t_3 to Broker 2. Period t_2 is used by the PREFETCHING optimization below. Period t_4 is required to complete the reconnection phase, which involves rebuild-

Algorithm *receive*(Subscription s)
($*$ Store and foward a subscription. $*$)
1. ($*$ Store in table $*$)
2. inSubs $\leftarrow$ inSubs $\cup$ $(s, s.sender)$
3.
4. ($*$ Forward toward reverse path of advertisements $*$)
5. **for** each $(a, n) \in$ inAds where $a.intersects(s)$ and $a.sender \neq s.sender$
6. **do if** $\exists(sub, n, true) \in$ outSubs where $sub.covers(s)$
7. **then** outSubs $\leftarrow$outSubs $\cup$ $(s, a.sender, false)$
8. **else** outSubs $\leftarrow$outSubs $\cup$ $(s, a.sender, true)$
9. send$(s, a.sender)$

Figure 3. Subscription handler

ing advertisement and multicast trees.

The objective of a publisher mobility algorithm is to reconfigure the advertisement and multicast trees to account for publisher mobility. When the publisher disconnects from it, Broker 1 will send an unadvertisement message to initiate the teardown of the publisher rooted advertisement tree, which will induce the teardown of corresponding multicast trees. This tree teardown occurs during period t_2 after which there is no state associated with the publisher in the system. At the end of period t_3 the publisher connects to Broker 2. During period t_4 the advertisement and multicast trees are rebuilt. It is this mobility induced tree teardown and reconstruction that makes the assumption of few advertisements invalid in this context. Publications sent during period t_4 may not be delivered to interested subscribers since the multicast tree has not been rebuilt yet.

Note that in this algorithm, there is no way to know when period t_4 is complete; Broker 2 does not know for certain when all subscriptions that match the newly sent advertisements have been received and the multicast tree has been rebuilt. This is a fundamental problem arising from the decoupling of publishers and subscribers in the pub/sub model. Since the length of period t_4 is unknown, we would like to minimize this period so as to minimize the probability that a publication sent soon after reconnection is not delivered to an interested subscriber.

Each broker has an `inAds` table to store $(ad, nodeid)$ pairs, where ad is an advertisement received at a broker from neighbor $nodeid$. The `outAds` table stores $(ad, nodeid, sent)$ tuples, where ad is an advertisement forwarded by the broker to neighbor $nodeid$, and $sent$ is true if ad actually was sent to $nodeid$. There are also corresponding `inSubs` and `outSubs` tables for subscriptions. The `inAds` (`inSubs`) table is used as a routing table for the forwarding of subscriptions (publications). Figures 2 and 3 show pseudo-code for handling advertisements, and subscriptions, respectively. Handling unadvertisements and unsubscriptions follows in a similar manner. Notice that (un)advertisement propagation can induce (un)subscription propagation. That is, the (de)construction of the advertisement tree induces the (de)construction of the multicast tree.

Pefetching Algorithm: The PREFETCHING algorithm exploits knowledge of future mobility patterns. It is similar to the STANDARD algorithm except that the advertisement and multicast trees are rebuilt during period t_2 instead of period t_4. Therefore, the length of period t_4 is now zero, and any publications sent immediately after reconnection are delivered to interested subscribers.

This algorithm has the advantage of hiding tree rebuilding time from the publisher since it occurs while the publisher is disconnected. Also, since the old tree (at Broker 1) is torn down concurrently with the building of the new tree (at Broker 2), it may occur that the new tree grafts onto the old tree before it is torn down, obviating the need to tear down the old tree completely. The DELAYED algorithm below tries to force this case, which only occurs by chance in the PREFETCHING algorithm.

Proxy Algorithm: The assumption here is that publishers tend to move within a restricted area. For example, a taxi driver may service only certain regions of a city. (The taxi may be publishing location updates to a dispatcher or potential customers.) The PROXY algorithm assigns a set of brokers to act as proxies for the publisher. These proxies always maintain a tree for the publisher. This way, there is no teardown or rebuilding of the tree when the publisher disconnects from or connects to one of its proxies.

Delayed Algorithm: The DELAYED algorithm exploits the fact that the old tree (rooted at Broker 1) and the new tree (rooted at Broker 2) have significant overlap. The teardown of the old tree at Broker 1 is delayed for some time after moveout, to allow the publisher to reconnect to another broker, and graft the new publication tree to the old one. After the delay, the old broker tears down only the extraneous portions of the combined tree.

3.1 Discussion

It should be noted that these optimizations make minimal assumptions about the underlying system. They can be used with any type of distributed pub/sub system. Moreover, the optimizations can be combined. For example, PREFETCHING can be combined with DELAYED to potentially achieve even better performance.

It is instructive to notice that the STANDARD algorithm does not distinguish between a moving publisher and a publisher that leaves and enters the system. Therefore it discards all state (advertisement and multicast trees) associated with a publisher on moveout, and must completely rebuild it on movein. Our optimizations address this issue.

Since publisher mobility causes expensive reconstruction of advertisement and multicast trees, it may be tempting to eliminate advertisement flooding and flood subscriptions instead. However, subscribers typically outnumber publishers, so the savings in publisher mobility induced tree

rebuilding cost do not justify subscription flooding. Furthermore, subscriber mobility will now cause multicast tree reconstruction; this reconstruction is much more expensive than when advertisements are used, since the multicast trees now span the whole network rather than being a minimal tree from the subscriber to interesting publishers.

4 Evaluation

Below we describe our experimental setup and metrics, and then discuss the experiments in detail.

4.1 Methodology

We performed all our experiments using the ns2 network simulator [3], extended with the STANDARD mobility algorithm and the three optimizations presented in Section 3.

The experiments simulate a city-wide pub/sub scenario, with a network of 85 brokers organized in a tree of height 4 and degree 4. While our algorithms work in general graph topologies, we feel a tree topology is common in metropolitan area networks. Conceptually, the 64 leaf brokers in this topology are distributed across a city, and publishers and subscribers connect to one of these 64 brokers. Each broker services a 0.5km range, so the 64 brokers service a 32km wide city. The brokers are communicating with each other over a 256kbps link with 10ms latency. The clients have a 128kbps link to the brokers.

Each publisher is assigned a random unique publication and an advertisement that is identical to its publication. (Unique advertisements are justifiable in content-based pub/sub because advertisements can be very expressive and customized for each publisher.) Each subscriber randomly subscribes to one of the publications assigned to a publisher. Unless otherwise stated, there are 50 publishers and 500 subscribers in the system.

The publishers randomly move at speeds of 5km/h (walking), 50km/h (city driving) or 100km/h (highway driving). Publishers connect and disconnect to adjacent brokers as they move, and the disconnection time when moving between brokers is 3 seconds. We keep the subscribers stationary in order to isolate the effects of publisher mobility. All clients connect to a random leaf broker during a warm-up phase, during which no measurements are made. Subscribers subscribe immediately after connecting to a broker and never unsubscribe. Publishers start publishing 200ms after connecting, and publish a total of 20 publications over a 4 second period. These publications are used to probe the system to determine when tree rebuilding is complete.

For the PROXY algorithm, the five closest brokers to the broker to which a publisher first connects are assigned as its proxies. For the DELAYED algorithm, the previous broker tears down the tree 10s after the publisher disconnects. PREFETCHING assumes perfect mobility prediction.

4.2 Metrics

The cost of supporting mobile publishers is measured in terms of the effects on the network and the clients. The network effect is measured as message load introduced by tree rebuilding. Publications are not counted as part of tree rebuilding cost, only (un)advertisements, (un)subscriptions, and any other control messages. Each hop a message travels is counted. The effect on the user is measured as the average time from the publication of an event to its delivery at a subscriber. Note that a publication at t_1 that is delivered to two subscribers at t_2 and t_3 is counted as two delivery times of $t_2 - t_1$ and $t_3 - t_1$.

We measure the tree rebuilding speed indirectly. Recall that it is difficult to determine when a tree has been rebuilt. Instead, we exploit the fact that when the tree is rebuilt, all publications will be delivered to all interested subscribers. We count the delivery of probe publications sent after reconnection. The faster the tree is rebuilt, the sooner events get delivered to all interested subscribers.

The state at the brokers is measured in terms of the number of entries in its advertisement and subscription tables.

4.3 Experiments

Publisher Scalability: In this experiment we evaluate the scalability of the algorithms with an increasing number of publishers (50, 100, 150, 200, 250), and hence an increase in aggregate mobility.

Figure 4 shows the tree building message cost for the four algorithms. The cost of tree rebuilding grows approximately linearly for the algorithms. However, there is a substantial difference in the cost of the algorithms. The STANDARD and PREFETCHING algorithms have more than ten times the message cost as the PROXY and DELAYED algorithms. STANDARD performs poorly because every move-out (movein) causes the whole advertisement and multicast trees for the moving publisher to be torn down (rebuilt). The same is true for PREFETCHING but with an additional cost of having the old broker inform the new broker to start rebuilding the tree. Therefore PREFETCHING is not building the new tree fast enough to graft onto the old tree that is being torn down. On the other hand, PROXY and DELAYED both graft onto existing trees. The reason DELAYED performs better than PROXY is because in DELAYED the old tree is rooted at a nearby broker (recall that the publishers move along adjacent brokers), so the distance an advertisement must travel to graft onto an existing tree is short. In PROXY, the distance to the old tree depends on the position of the publisher relative to its fixed proxies.

All the algorithms require the brokers to maintain about the same state, with PROXY needing slightly more state.

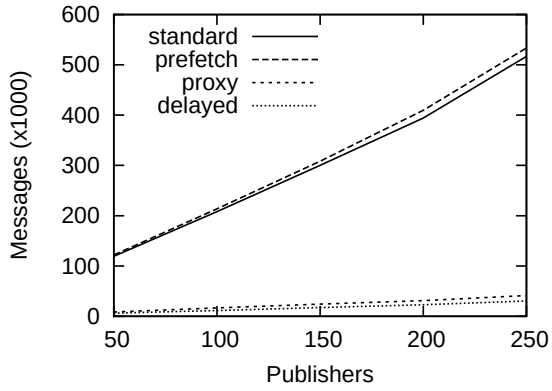

Figure 4. Tree rebuilding

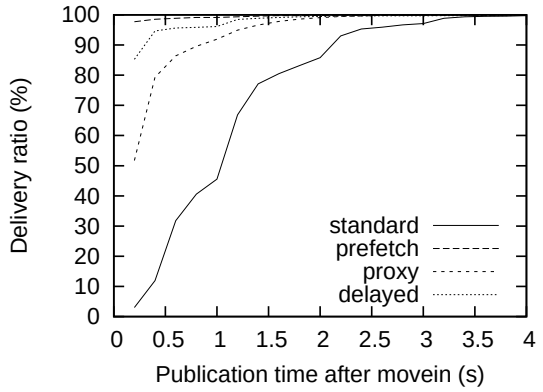

Figure 5. Probe delivery (250 publishers)

This state increases approximately linearly with increasing publishers. Due to the nature of our topology and workload, brokers closer to the root need to maintain more state. This may not be the case with different workloads or topologies.

We now look at the subscriber perceived cost. The subscriber is concerned with timely delivery of publications it is interested in. Figure 5 shows the delivery ratio of each probe publication. The ith probe is the ith publication sent after connecting to a broker. Recall that probes are sent every 200ms. We expect that probes shortly after reconnection will not be delivered to all subscribers since the multicast tree is still being rebuilt. We see that the STANDARD algorithm takes nearly 4 seconds to rebuild the multicast tree. At the other extreme, PREFETCHING takes almost no time to rebuild the tree. This is because PREFETCHING initiates tree rebuilding when the publisher disconnects from the previous broker. The DELAYED algorithm also rebuilds the tree relatively quickly.

A limitation of using probes to determine tree rebuilding time is that the probes are loading the network while tree reconstruction is taking place, and slow down tree reconstruction. We plan to address this in future work.

In Figure 6, we see a cumulative distribution function of

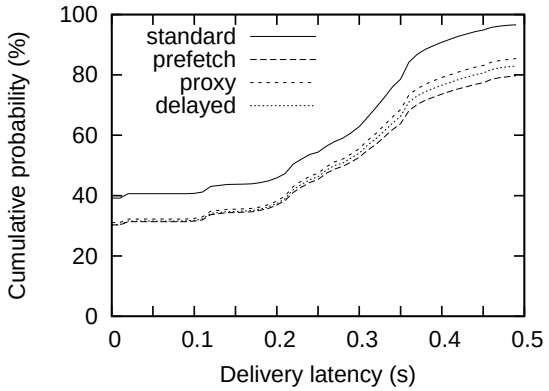

Figure 6. Delivery time (250 publishers)

the delay between a publication and its delivery to an interested subscriber, in the case of 250 publishers. The median publication is delivered in about 0.3 seconds. There is not much difference among the algorithms. The differences are due to congestion caused by tree rebuilding traffic. Hence, the STANDARD algorithm, which has a high tree building cost (as seen in Figure 4) has the worst delivery latency. Note, while PREFETCHING has a higher rebuilding cost than STANDARD, its cost occurs while the publisher is disconnected; tree rebuilding is essentially complete when the publisher reconnects (3s after disconnection in this experiment) and hence PREFETCHING has the lowest delivery latency.

It is interesting to note that while PREFETCHING gives the best performance to subscribers (in terms of delivery rate and latency), it imposes the highest cost on the network.

Subscriber Scalability: We ran simulations with 500, 2500, and 5000 subscribers (all with 50 publishers) to measure performance as the number of participants in a multicast tree increases. Our results show that there is little impact in the delivery ratio and minimal increase in the tree rebuilding cost as the number of subscribers increases. This is because the second phase of the tree rebuilding (propagating subscriptions) can support additional subscribers with decreasing incremental cost due to the multicast nature of the subscription tree.

Publisher Mobility Speed: This experiment varies the speed of the publishers from 25km/h to 100km/h in 25km/h increments. Our results indicate that faster mobility leads to larger tree rebuilding cost, with STANDARD and PREFETCHING suffering from the steepest increases. However, an interesting phenomenon occurs where the STANDARD algorithm's delivery ratio (Figure 7) *increases* with speed. Similarly, the delivery latency (Figure 8) decreases with faster mobility for all the algorithms. We are not sure why this is occurring, and plan to investigate this in the future.

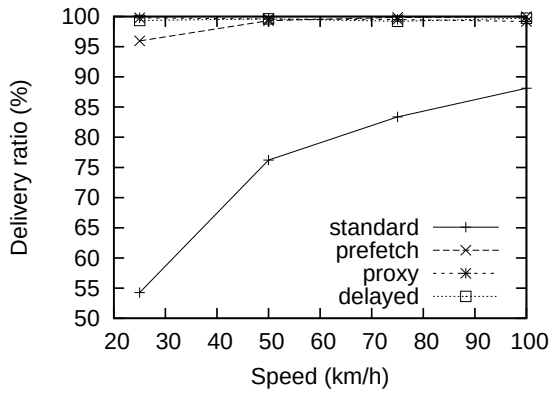

Figure 7. Delivery ratio (speed)

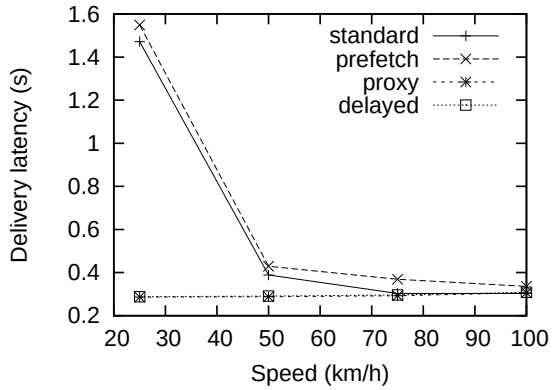

Figure 8. Delivery time (speed)

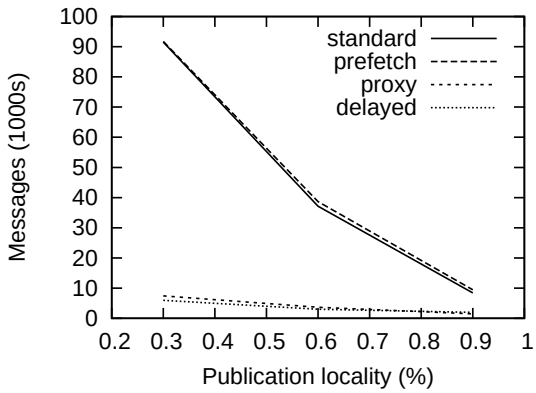

Figure 9. Tree rebuilding (locality)

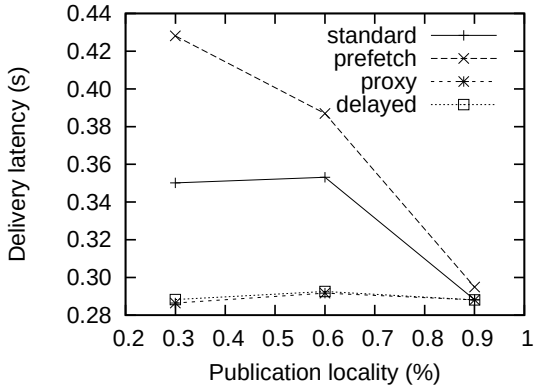

Figure 10. Delivery time (locality)

Proxy Locality: In this experiment we test how sensitive the PROXY algorithm is to choosing a good set of proxy brokers. Initially, publishers only move back and forth among their five adjacent proxy brokers. This may be the case of a policeman patrolling a few city blocks repeatedly. (The policeman may be publishing status reports, calls for backup, or accident report information.) Then we vary how much the publisher may overshoot this proxy set. An overshoot of δ means that the publisher may move δ brokers past its proxy set. As expected, our results show that tree rebuilding cost increases with δ. Incidentally, this cost stabilizes at $\delta = 5$. This is because our relatively small topology means that the length of the path that an advertisement takes to graft onto an existing static tree maintained by a proxy broker is limited. In general, we saw that PROXY is not very sensitive to δ. The message load, delivery time, delivery latency, and tree building time hardly change.

Publication Locality: In this experiment we vary the degree of similarity of publications in the system. We achieve $x\%$ locality by having $x\%$ of the publishers publish the same publication and the remaining publish different and unique publications. Our results indicate that with enough similarly between publications (90%), the STANDARD algorithm's performance approaches that of DELAYED and PROXY. Figures 9 and 10 show this is the case for the tree rebuilding cost and delivery time, respectively.

5 Related Work

To the best of our knowledge we are the first to evaluate the cost of mobile publishers in pub/sub middleware.

SIENA [5], JEDI [7], ELVIN [15], and M-ToPSS [4] add extensions to support mobile subscribers. The problem with mobile subscribers is the efficient storage and replay of publications missed by a disconnected subscriber. In this paper, however, we examine algorithms for supporting mobile publishers.

Mobile IP [12] uses the concept of a home agent to handle mobile clients. Each IP client has a home agent that mediates communication between the roaming client and other nodes. While this approach eliminates tree reconstruction by maintaining a tree at the home agent, M-ToPSS [4] shows that such an approach causes excessive traffic, because the gains of a multicast tree are lost to the unicast traffic from the mobile client to the home agent.

6 Conclusions and Future Work

The number of mobile information producers will increase in the future. However, there has been no evaluation of mobile publishers in pub/sub systems. This is important because mobility breaks a fundamental pub/sub assumption, namely that the advertisement load on the system is very low.

Our evaluation supports this intuition: the current STANDARD mobility algorithm causes excessive state reconstruction traffic. The DELAYED and PROXY algorithms perform well in terms of network load and user perceived performance. Usually DELAYED is preferable because, while it performs only slightly better than PROXY, it does not require the additional task of assigning proxies to a publisher. Interestingly, PREFETCHING has the highest load on the network but delivers the best performance to the user. We cannot recommend the STANDARD algorithm in any case.

For future work, we would like to vary more parameters and develop more optimizations. Furthermore, we plan to develop a less intrusive technique to determine tree rebuilding time, and try to understand why increasing mobility speed improves performance in some cases. In addition, we hope to study the impact when both publishers and subscribers are mobile.

References

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

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

[3] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, P. H. A. Helmy, S. Mc-Canne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33:59–67, May 2000.

[4] I. Burcea, H.-A. Jacobsen, E. de Lara, V. Muthusamy, and M. Petrovic. Disconnected operation in publish/subscribe middleware. In *IEEE Mobile Data Management*, pages 39–50, 2004.

[5] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, Dec. 2003.

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

[7] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), 2001.

[8] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. volume 30, pages 115–126. ACM Press, 2001.

[9] H. K. Y. Leung. Subject space: A state-persistent model for publish/subscribe systems. In *Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 2002.

[10] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. *International Conference on Distributed Computing Systems (ICDCS'05)*.

[11] H. Liu and H.-A. Jacobsen. Modeling uncertainties in Publish/Subscribe System. In *In Proceedings of ICDE*, 2004.

[12] C. E. Perkins and D. B. Johnson. Mobility support in IPv6. In *MOBICOM*, 1996.

[13] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS - a semantic publish/subscribe system. In *Very Large Databases (VLDB'03)*, Berlin, Germany, September 2003.

[14] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPSS - fast filtering of graph-based metadata. In *the 14th International World Wide Web Conference (WWW2005*, Chiba, Japan, May 2005.

[15] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness - transparent information delivery for mobile and invisible computing. In *CCGrid 2001 IEEE International Symposium on Cluster Computing and the Grid*, 2001.

[16] Talarian Inc. Publish-subscribe middleware helps direct traffic of Olympic proportions. http://messageq.ebizq.net/communications_middleware/talarian_2.html.