A Pro-active Mobility Extension for Pub/Sub Systems

Abdulbaset Gaddah
Carleton University
1125 Colonel By Drive
Ottawa, Ont., Canada K1S 5B6
agaddah@sce.carleton.ca

Thomas Kunz
Carleton University
1125 Colonel By Drive
Ottawa, Ont., Canada K1S 5B6
tkunz@sce.carleton.ca

ABSTRACT
In this paper, we propose a novel and efficient mobility extension based on a *pro-active* approach (i.e., the context transfer/caching occurs prior to the subscriber movement) with the objective to extend existing pub/sub systems to the mobile environments. We also describe the notion of *neighbor graph*, which forms the basis for pre-loading the subscriber context one hop ahead of its current location. We have investigated the adequacy of our proposed pro-active approach in supporting mobile subscribers and compared its behavior with a *durable subscription-based* approach adapted by JMS-based pub/sub systems. The experimental results show that our pro-active approach reduces the message loss by more than 50% and the message duplication to zero compared to the durable subscription-based approach. It also achieves better throughput results with low cost in terms of mobility extension overhead.

Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems – distributed application; C.4 [**Computer Systems organization**]: Performance of Systems

General Terms
Algorithms, Design, Performance, Experimentation

Keywords
Pub/Sub systems, mobility, JMS, Wireless network, middleware

1. INTRODUCTION
Computing devices with wireless connectivity are rapidly gaining popularity, as our dependence on the data accessed through them is growing. Users anticipate to access different information and services while they are roaming. Due to the limited and dynamic resources of the mobile computing systems, users may voluntary or involuntary disconnect from the network (i.e., running out of battery, loss of connectivity, commuting between locations). They expect that data disseminated while they are disconnected can be persistently buffered and delivered upon their reconnection. These constraints raise the demand for middleware infrastructure, based on a flexible and scalable interaction style, to meet the dynamic nature of mobile computing, and facilitate the development of innovative applications.

The pub/sub interaction paradigm has been widely used to model information dissemination applications [3], where *publishers* are event producers, *subscribers* are event consumers, and *brokers* are event dispatchers. Publishers notify the outside world about the occurrence of certain events. Subscribers express their interest in receiving a particular set of events by means of *subscriptions*. Upon receiving a new event, the event broker matches the event against all the subscriptions and then forwards it to all interested subscribers. The decoupling of publishers and subscribers in time, flow, and space along with the anonymous features of the pub/sub systems make them a good choice for supporting mobile, wireless systems in a natural manner. Most existing middleware systems are optimized for fixed environments (i.e., clients do not roam and the infrastructure itself is fixed). Hence, several add-on protocols are needed to extend such systems to support mobile subscribers.

One approach of extending pub/sub systems is based on the use of durable subscriptions along with persistent notifications [4][11]. In this approach, the network of distributed brokers buffers persistent notifications irrespective of their current active subscriptions until the notifications are no longer valid. When the mobile subscriber reconnects to the system, only the valid notifications are delivered to it in the published order. Such an extension may increase the overall overhead of distributed brokers due to the costly buffering process. This may result in degrading the system's performance. Also, the number of lost notifications can increase as the buffer space drains quickly or the notifications become invalid due to a large disconnection interval. Duplicated messages can be received when the subscribers reconnect to the previously visited brokers.

We propose a novel and efficient solution that is based on a pro-active approach, i.e. the context transfer/caching occurs before the subscriber movement. This is achieved with the help of a data structure, neighbor graph, which dynamically captures the subset of brokers to which the subscriber context should be forwarded. Our approach is based on the notion of a replicator proxy, that is, the proxy creates a dummy replica of the moving subscriber to act on its behalf. To minimize the overhead of state-transfer, we only transfer the subscriptions of each subscriber to the next-hop future brokers rather than its actual events. These subscriptions are only activated when the subscriber disconnects from the network and deactivated once it reconnects. During the deactivation process, events that belong to such subscriptions are garbage collected. Our experimental results show that the pro-active approach reduces the message loss by more than 50% and message duplication to zero compared to the durable subscription-based approach. Moreover, our approach decreases the overhead on the distributed brokers as the buffering process occurs upon demand.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MOBILWARE 2008, February 13-15, Innsbruck, Austria
Copyright © 2008 ICST 978-1-59593-984-5
DOI 10.4108/ICST.MOBILWARE2008.2762

We believe that our experimental results are valid for various pub/sub systems when they are deployed in a mobile, wireless environment.

With respect to previous work in extending the pub/sub systems, *Caporuscio et al.* evaluated the behavior of SIENA [1] in wired and GPRS-based networks. They mainly focused on measuring the performance of a mobility extension developed specifically for extending SIENA to mobile, wireless domains. Their experimental results investigated the number of duplicated and lost messages, but defined no other metrics to evaluate the system performance. *Ivana and Ignac* [11] proposed a mobility extension that delivers only valid events to the subscribers just after the activation of their subscriptions. Expired events are removed from their buffers. Each broker maintains a list of valid events that have been sent to the subscribers and neighbor brokers. Subscribers must provide a list of the previous received events to avoid duplicate events. This approach clearly creates extra traffic in the broker network, and increases the usages of broker memory and processing time. In contrast, our approach reduces this overhead as it propagates only the subscriptions to the neighboring brokers and buffers events on- demand. *Umar* et al. [4] presented their experience in evaluating the performance of a commercial JMS-based pub/sub system in wired/wireless networks. The nature of their work differs from ours as it mainly focuses on studying the effect of some mobility parameters on the system performance and did not propose a new mobility extension. The REBECA [7] has recently been extended to support mobility. The last visited broker plays the role of a proxy subscriber. When a subscriber reconnects to a new broker, it re-submits its subscriptions. The new broker finds the junction of delivery paths to the new and old brokers by inspecting its routing table and its list of received advertisements, and compares it to the received subscription. It then sends a fetching request to the old broker to retrieve the subscriber events. The events stored by the old broker are routed through the junction to reach the new broker, and then the subscriber. It has not justified why subscribers cannot maintain the information about the last visited broker. There are currently no results that evaluate the performance of the approach. The mobility extension in ELVIN [13] uses a central caching proxy server that mediates the original server and mobile units for caching events for disconnected subscribers. This approach creates a performance bottleneck at the proxy server as the subscribers must always reconnect to the central proxy. It also induces significant network traffic due to potential triangular routing. JEDI [2] has added an extension to support mobility that is based on explicit *moveIn* and *moveOut* operations to relocate subscribers. Subscribers explicitly trigger these operations during the handoff process, which can be problematic if a wireless connection breaks down unexpectedly due to physical mobility or interference.

The paper is organized as follows. Sect. 2 presents the pro-active context distribution algorithm that is based on the notion of neighbor graph. Sect. 3 describes the implementation of the pro-active approach. Sect. 4 illustrates the experimental setup. Sect. 5 evaluates and compares the behavior of pro-active and durable subscription-based approaches. Sect. 8 concludes this paper.

2. PRO-ACTIVE APPROACH

The core idea of our pro-active approach is largely based on a mechanism that intelligently transfers/buffers subscriber context (subscriptions/messages) one hop ahead of its current broker prior to the actual movement of the mobile subscriber. We also present the notion of neighbor graph that forms the basis of this approach as it dynamically captures the candidate subset of brokers to which subscriber-context should be pro-actively forwarded and buffered.

2.1 Pro-active Context Distribution Algorithm

Pro-active Context Distribution Algorithm

Step 1: IF subscriber S connects to B_j THEN
        FOR all B_i Є Neighbor(B_j) DO
            Propagate_Sub(B_j, S, B_i)
        ENDFOR
    ENDIF
Step 2: IF subscriber S disconnects from B_j THEN
        FOR all B_i Є Neighbor(B_j) DO
            Activate_Sub(B_j, S, B_i)
        ENDFOR
    ENDIF
Step 3: IF subscriber S reconnects to B_j THEN
        FOR all B_i Є Neighbor(B_j) DO
            Deactivate_Sub(B_j, S, B_i)
            Remove_Msgs(B_j, S, B_i)
        ENDFOR
    ENDIF
Step 4: IF subscriber S reconnects to B_k from B_j THEN
        Propagate_Msgs(B_j, S, B_k)
        FOR all B_i Є Neighbor(B_j) DO
            Remove_Sub(B_j, S, B_i) /* $i \neq k$ */
            Remove_Msgs(B_j, S, B_i)
        ENDFOR
    ENDIF
Step 5: IF subscriber S reconnects to B_j from B_k THEN
        IF Sub(S) is not in Buffer(B_j) THEN
            Obtain_Sub(B_k, S, B_j)
            Obtain_Msgs(B_k, S, B_j)
        ENDIF
        FOR all B_i Є Neighbor(B_j) DO
            Propagate_Sub(B_j, S, B_i)
        ENDFOR
    ENDIF
Step 6: IF subscriber S unsubscribes from B_j THEN
        FOR all B_i Є Neighbor(B_j) DO
            Remove_Sub(B_j, S, B_i)
            Remove_Msgs(B_j, S, B_i)
        ENDFOR
    ENDIF
Step 7: IF B_j receives Sub(S) or Msgs(S) from neighbors THEN
        Store(B_j, Sub(S)) or
        Store(B_j, Msgs(S))
    ENDIF
Step 8: IF B_j triggers Timeout(S) THEN
        FOR all B_i Є Neighbor(B_j) DO
            Remove_Sub(B_j, S, B_i)
            Remove_Msgs(B_j, S, B_i)
        ENDFOR
    ENDIF

Figure 1. The pro-active context distribution

We next describe our pro-active context distribution algorithm. The algorithm can be decomposed into the following three steps: (1) propagate/buffer subscriptions to/at the neighbor brokers; (2) activate subscriptions and buffer messages locally; (3) deliver messages and reset buffer. Figure 1 presents the pseudo code for the algorithm that is executed on broker B_j.

The following notation is used in the description:

- ***S***: denotes a subscriber who is potentially mobile.
- $\boldsymbol{B_j}$: denotes the initial hosted broker for subscriber *S*.
- $\boldsymbol{B_i}$: denotes the next-hop broker to B_j.
- **Neighbor($\boldsymbol{B_j}$)**: refers to the set of neighbor brokers of B_j.
- **Timeout(*S*)**: refers to a chosen *T* time to keep handling Sub(*S*) and Msgs(*S*) of a disconnected subscriber *S*. when *T* expires, subscriber context is garbage collected. ($T \geq$ average handoff interval at all brokers).
- **Sub(*S*)**: denotes the subscriptions related to subscriber *S*.
- **Msgs(*S*)**: denotes the messages related to subscriber *S*.
- **Propagate_Sub($\boldsymbol{B_{from}}$, *S*, $\boldsymbol{B_{to}}$)**: denotes the propagation of Sub(*S*) from B_{from} to B_{to}.
- **Propagate_Msgs($\boldsymbol{B_{from}}$, *S*, $\boldsymbol{B_{to}}$)**: denotes the propagation of Msgs(*S*) from B_{from} to B_{to}.
- **Obtain_Sub($\boldsymbol{B_{from}}$, *S*, $\boldsymbol{B_{to}}$)**: denotes that B_{to} obtains Sub(*S*) from B_{from}.
- **Obtain_Msgs($\boldsymbol{B_{from}}$, *S*, $\boldsymbol{B_{to}}$)**: denotes that B_{to} obtains Msgs(*S*) from B_{from}.
- **Remove_Sub($\boldsymbol{B_{old}}$, *S*, $\boldsymbol{B_{nghbr}}$)**: denotes that B_{old} sends a notification message to B_{nghbr} in order to remove Sub(*S*) from the B_{nghbr} buffer.
- **Remove_Msgs($\boldsymbol{B_{old}}$, *S*, $\boldsymbol{B_{nghbr}}$)**: denotes that B_{old} sends a notification message to B_{nghbr} in order to remove Msgs(*S*) from the B_{nghbr} buffer.
- **Store($\boldsymbol{B_j}$, Sub(*S*))**: store the subscriptions of subscriber *S* into the buffer of B_j.
- **Store($\boldsymbol{B_j}$, Msgs(*S*))**: store the messages of subscriber *S* into the buffer of B_j.
- **Buffer($\boldsymbol{B_j}$)**: denotes the buffer maintained at B_j.
- **Deactivate_Sub($\boldsymbol{B_{old}}$, *S*, $\boldsymbol{B_{nghbr}}$)**: B_{old} sends a notification message to B_{nghbr} in order to deactivate Sub(*S*) at the B_{nghbr}.
- **Activate_Sub($\boldsymbol{B_{old}}$, *S*, $\boldsymbol{B_{nghbr}}$)**: B_{old} sends a notification message to B_{nghbr} in order to activate Sub(*S*) at the B_{nghbr}.

This algorithm makes a few assumptions about the target pub/sub system. It assumes a set of brokers organized in a general graph (or a *peer-to-peer*) topology to form a distributed communication service. The peer brokers directly communicate with each other to exchange subscriptions and messages. We also assume that all the generated messages are sent to all the brokers in the system. This assumption is based on [10] that strongly recommends this strategy in highly mobile environments due to the practical limit on the number of multicast addresses and related overheads. Our algorithm is only limited to manage subscriber mobility, and is not concerned about publisher mobility explored by others [8]. A stepwise description of this algorithm is given next.

Step 1 of the algorithm is started when a subscriber *S* connects to a certain broker B_j in the system. The broker B_j propagates a passive copy of the subscriber's subscriptions *Sub(S)* to all the brokers B_i that are neighbors of the broker B_j *Neighbor*(B_j). Each neighbor broker B_i locally stores *Sub(S)*. In the meantime, the subscriber *S* consumes its messages from the broker B_j through its active subscription. The pro-active algorithm is based on the notion of a neighbor graph, which will be described later in this section. The neighbor graph is automatically build and maintained by each broker in the system.

Step 2 of the algorithm is started as the subscriber *S* temporarily disconnects from the network due to poor network connectivity or handoff procedure. Since the broker B_j does not receive an acknowledgement from the subscriber *S*, after some re-transmit attempts it considers *S* as temporarily disconnected and thus sends an *Activate_Sub*(B_j, *S*, B_i) request to all the brokers B_i that are in *Neighbor*(B_j). The brokers B_i acknowledge the receipt of this request and activate the *Sub(S)*. Accordingly, all the neighbor brokers B_i will locally buffer all the incoming messages *Msgs(S)* that match the *Sub(S)*. It should be noted that the message ID of the last message the subscriber *S* consumed (for each subscription) is enclosed with the activation request and the neighbor brokers B_i only buffer the messages after the message with this ID. Similarly, the broker B_j keeps buffering the messages for the subscriber *S* as it may reconnect to it again.

Step 3 of the algorithm is started when the subscriber *S* reconnects to the same broker B_j. This results in sending a *Deactivate_Sub*(B_j, *S*, B_i) message to all neighbor brokers B_i requesting them to deactivate the *Sub(S)* and to terminate the buffering process. Then, a *Remove_Msgs*(B_j, *S*, B_i) message is followed requesting to clean up the local buffer of each neighbor broker B_i. In the meantime, the broker B_j delivers all the buffered messages to the subscriber *S*.

Step 4 of the algorithm is started when the subscriber *S* reconnects to a different broker B_k. The broker B_k informs the broker B_j that the subscriber *S* reconnected to it. First, the broker B_j sends all the messages missed during the process of *Activate_Sub*(B_j, *S*, B_i) process to the broker B_k as discussed in step 2. Then, it sends two messages, *Remove_Sub*(B_j, *S*, B_i) and *Remove_Msgs*(B_j, *S*, B_i), to all the *Neighbor*(B_j) excluding the broker B_k. These messages remove the *Sub(S)* and *Msgs(S)* from the local buffer of all the *Neighbor*(B_j). The broker B_k is excluded from receiving these messages as the subscriber *S* is connected to it. One point worth mentioning here is that the *Remove_Sub*(B_j, *S*, B_i) semantic will change as the neighbor graph is built. Both brokers B_k and B_j exchange their neighbor graph tables in order to reduce the overhead of deleting/inserting *Sub(S)* requests. Throughout the neighbor graph table, the broker B_j decides which *Sub(S)* should be deleted and which *Sub(S)* should be deactivated for later use by the broker B_k. Similarly, the broker B_k can identify which *Sub(S)* should be propagated to its neighbors.

Step 5 of the algorithm is started as the subscriber *S* reconnects to the broker B_j from the broker B_k. The broker B_j first checks if the context of the subscriber *S*, i.e. *Sub(S)* and *Msgs(S)*, is available in its buffer. If the subscriber's context is not in the buffer then the broker B_j will inform the broker B_k to send the subscriber's context. This may happen in two different scenarios: (1) the broker B_k is not a neighbor of the broker B_j. Therefore, the broker B_j has no information about the subscriber *S*. (2) the subscriber *S* is the first to visit the broker B_j from its neighbor B_k. If the subscriber's context is found in the buffer of the broker B_j, similar actions to step 1 will take place.

Step 6 of the algorithm is started as the subscriber *S* deletes one of its subscriptions. As a result of this request, the broker B_j issues two messages, *Remove_Sub*(B_j, *S*, B_i) and *Remove_Msgs*(B_j, *S*, B_i), to all its neighbors. These two messages will respectively delete the subscriber's subscription *Sub(S)* and its related messages if there are any in the buffer.

Step 7 of the algorithm is started when the broker B_j receives the subscriber's context from the broker B_k. The subscriber's context will be stored in a persistent buffer.

Step 8 of the algorithm takes place as the subscriber *S* disconnects from the broker B_j for good. When the disconnected time reaches the timeout period, the broker B_j sends a request to its neighbors to remove the subscriber's context from their buffers. This is a necessary step as buffering and managing the subscriber's context can severely affect the broker performance.

2.2 Neighbor Graph

The neighbor graph is an undirected graph with a set of edges that represent mobility paths between the vertices (or brokers). Hence, the neighbors of a given vertex *v* in the graph correspond to the set of potential next brokers. As it is difficult to predict the subscriber movement, we need to identify the candidate subset of next future brokers in order to transfer/buffer the subscriber-context prior to the occurrence of the handoff process (*pro-actively*). The neighbor graph provides the abstractions to achieve this goal.

2.2.1 Neighbor Graph Generation

The neighbor graph can be generated either in a static manner, i.e., manually constructed once and never changes over time, or in a dynamic manner, i.e., automatically generated and adaptively changes according to the mobility graph. A static neighbor graph is problematic as it fails to approximate the mobility graph which changes dynamically over time. Hence, we chose to dynamically construct an adaptive neighbor graph.

There are two complementary methods for the brokers to learn the edges in the graph. The first method is to embed the address of the old broker with the reconnection request sent by the subscriber to the new broker, thereby establishing the reconnection relationship between the two brokers. The second method is to use the request for message transfer received from another broker to establish the relationship. Each broker locally manages the edges in a Least Recently Used (LRU) approach. This is essential to remove the outlier edges, i.e. the ones that do not model recently-used relationship. Such edges may temporarily occur when a mobile subscriber puts its terminal in power save mode and moves to different locations to reconnect to any other broker in the network topology. As a result, a timestamp based LRU method ensures the freshness of the neighbor graph, and removes the outlier edges over time. The autonomous creation of the neighbor graph makes it adaptive to dynamism in the reconnection relationship (i.e., adding and removing brokers, network topology changes).

Each broker locally stores its neighbor graph, i.e., the list of its neighbor brokers. The whole graph thus is stored in a distributed manner. The following pseudocode is used to generate the local view of the graph that is executed at each broker. Here, we refer to the broker that executes the algorithm as $B_{current}$.

- *Receive a reconnection request*: As a mobile subscriber *S* reconnects to $B_{current}$ from B_i, $B_{current}$ adds B_i to its list of neighbors.
- *Receive a subscriber-context transfer*: As a broker $B_{current}$ receives a context transfer from B_i, it adds B_i to the list of neighbors.
- *Entity-deletion*: If none of the above addition operations takes place during a given timeout interval *T*, the B_i entity will be removed from the neighbor graph. $T \geq$ the average frequency interval of addition operations at all broker nodes.

It should be noted that the first subscriber to cross over an edge will receive its context in *reactive* fashion. This will be gradually changed to *pro-active* fashion as the edges are added to the graph. The edge degree of a given broker (vertex) in a neighbor graph is the number of outward edges from that broker. It determines the overhead of context transfer/caching in the pro-active approach. To control this overhead, the broker's degree can be bounded by a fixed upper bound (*M*).

3. IMPLEMENTATION

The pro-active approach is implemented within an independent layer of *proxies* between the subscribers and their messaging brokers. This layer is mainly responsible for replicating dummy subscribers at the next future brokers to buffer messages on behalf of the moving subscriber. It also dynamically captures the mobility graph of the distributed brokers' network to identify the subset of next neighboring brokers. A single proxy process runs with each broker to manage user mobility from one broker to the other. Note that the proxy layer is completely transparent to the brokers and the applications. We have integrated a monitoring component with the broker process to transparently track the subscribers' states (i.e., connect, disconnect, handoff) as well as the ID of the last message consumed by the subscriber. Also, each subscriber has to keep track of the last broker to which it was connected to.

When a subscriber connects to a broker, the proxy will be notified and receives a copy of the subscriber's subscription. The proxy locally stores the subscription and uses it to instantiate an inactive dummy subscriber. In the meanwhile, it propagates a copy of the subscription to all neighbors, and instructs its peers to create the same dummy subscriber using the forwarded subscription. When the subscriber disconnects from the network, all the corresponding dummy subscribers at the neighboring brokers are activated to buffer messages on behalf of the disconnected subscriber. The dummy subscribers use the ID of the last received message of the actual subscriber to prevent message duplication. Therefore, only messages with higher ID are stored for the subscriber. When the subscriber reconnects to a new broker, the proxy at that broker notices this and informs the related dummy subscriber to stop storing messages and to return to inactive mode. The broker in the meanwhile starts delivering the stored messages to the subscriber. The set of neighbor graphs located at the old and new brokers now must be inspected to ensure that new dummy subscribers are created on all neighboring nodes of the new broker and old dummy subscribers are either removed or deactivated. In this implementation, tracking the address of the broker is necessary as it is needed during the creation process of the neighbor graph and when the reactive approach is applied

before the building of this graph. We also use the broker address to distinguish the handoff state from the reconnect state.

4. EXPERIMENTAL SETUP

For our experimental study, we have chosen Java Message Service (JMS) [12] as our base pub/sub platform. We have incorporated our pro-active extension into the selected JMS implementation to explore its behavior and compare it with the *durable subscription-based* approach supported by JMS implementations. JMS offers several modes that lend themselves well to the mobile, wireless environments. It adapts two subscription modes, *nondurable* and *durable*. In the nondurable mode, messages are forwarded only to subscribers who are presently online while in the durable mode messages are also forwarded to subscribers that are not currently connected. JMS also offers two communication modes, *point-to-point* and *pub/sub*. In this study, we consider the durable and pub/sub modes. Detailed descriptions of the JMS features can be found in [12].

We performed all our experiments on an overlay network of six Intel based Pentium 4 nodes running RedHat Linux 9, inter-connected by a 100 Mbps switch. Two nodes were used for running two instances of the JMS broker. A router node was used for running a wireless network emulator. One node was used for running a single, stationary message publisher. The remaining two nodes were used for running the mobile subscribers. Subscribers that share the same machine run in separate threads and establish separate connections, but use the same Java Virtual Machine and JMS Client library. The JVM used for running the brokers and the clients is Sun SDK 1.4.2, started with the options –Xms64m and –Xmx256m as a minimum and maximum heap size. Although this is a limited configuration, it is sufficient for the purpose of this paper: evaluating different mobility support extensions.

The JMS supports content-based filtering with the help of *message selectors* (conditional expressions). It allows subscribers to specify their selectors as an argument when they create their subscriptions to express their interest in receiving a certain set of messages. In our setup, each subscriber exploits a specific selector range that is randomly chosen to be $1/5^{th}$ of the total selector range. Similarly, the message publisher assigns a single selector value ranging from 0 to 99 with each generated message.

A mobile subscriber in this setup represents an application running on a mobile terminal that transparently moves from one broker to the other. It initially registers with one of the two JMS brokers by sending a single durable subscription. Through a mobility scenario written in Java, the subscriber keeps migrating between the two brokers during the course of the experiments. In our experiments, subscriber threads were created and executed on two stationary machines. Subscribers are initially split evenly between the two JMS brokers. However, due to mobility, the number of subscribers at each broker fluctuates over time, resulting in brokers serving a relatively lager number of subscribers at times while at other times the broker may serve only a small number of subscribers.

A Java program is implemented to model subscribers' mobility. Each subscriber goes through different mobility states as described next. The *connect state* is the starting point for all subscribes in our mobility model: a subscriber is connected to one of the two JMS brokers and consumes messages. Each subscriber remains in this state for a randomly generated, exponentially distributed time with a mean β=60 seconds. With an equally and randomly selected probability, a subscriber either moves to disconnect or handoff state. The *disconnect state* reflects the case of signal breakdowns due to poor network connectivity. A subscriber remains in this state for a randomly generated, exponentially distributed time with a mean δ=10 seconds. With a similar probability, the subscriber moves either back to the connect state and reconnects to the same broker or goes to the handoff state. The *handoff state* corresponds to the case where a subscriber moves out of the covered range of its previous broker. After staying in the handoff state for a randomly generated, exponentially distributed time with a mean δ=10 seconds, the subscriber moves back to the connect state and connects to a different broker.

All the communications between the subscribers and the brokers are tunneled via an emulated wireless channel that is created using a network emulator called NistNet [9]. NistNet is a popular software tool that is implemented as a kernel module extension to the Linux operating system. It can be used to emulate various network environments. We used NistNet to model the characteristics of an IEEE 802.11 wireless LAN network based on a set of configuration parameters such as packet delay, packet loss, packet duplication, and network bandwidth. All these parameters were set to the most commonly used values reported for IEEE 802.11 wireless LAN networks [6][14].

The reported results were captured from the measurement data obtained under different workloads. Each experiment was run for a duration that was long enough to reach a steady state. We ensured that the publisher/subscriber machines were not the bottlenecks in our experiments. We kept both CPU and memory utilizations at less than 65%, thereby preventing publisher and subscriber bottlenecks from impacting the overall system performance. Each broker machine was fully dedicated to running a single instance of the JMS broker. Before running any experiment, topic destinations and message stores were purged and reinitiated to start each test with a clean slate. All subscribers were consuming messages in asynchronous manner. Each subscriber was using a separate connection to receive its messages. Network latency for creating subscribers' connections was not included in our results.

4.1 Performance Measures

- *Subscriber throughput* (*Ts*): Total number of messages received per second. It is obtained by adding up the number of messages received by individual subscribers and dividing by the total duration of the experiment.
- *Percentage of message loss* (*L*): Percentage of missed messages by all the subscribers. It was obtained by calculating the difference between the total published and received messages and then dividing by the total published messages.
- *Percentage of message duplication* (*D*): Percentage of duplicated messages received by all the subscribers. It is obtained by dividing the total duplicated messages by the total received messages.
- *Message processing time* (*Ls*): Average processing time that it takes the broker to process messages. It is obtained by adding up the processing time of each message and then dividing the total by the total number of received messages.
- *Handoff latency (H):* Time between sending the reconnect request and receiving the first message of the corresponding

subscriber at its new broker.

5. PERFORMANCE EVALUATION AND COMPARISON

This section evaluates and compares the behavior of the pro-active and durable subscription-based approaches in terms of mobility service overhead, handoff latency, message loss/duplication, and overall throughput.

5.1 The Mobility Extension Overhead

The overhead of using the pro-active and durable subscription-based extensions is evaluated in terms of two different metrics: the average processing time of the messages and the aggregated throughput of the subscribers. These metrics provide a good indicator of the overhead incurred by both extensions. To study this overhead under different load conditions, we have varied the total number of subscribers that can be served by the brokers.

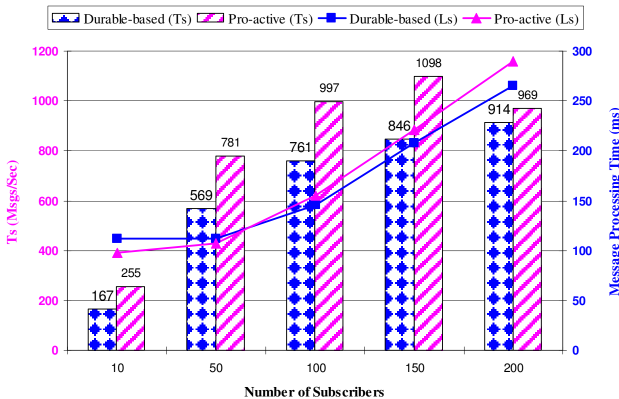

Figure 2: The mobility extension overhead

Figure 2 shows the incurred message processing time (*Ls*) and the overall throughput (*Ts*) as the subscriber population increases. From the figure, it can be observed that there is a proportional relationship between the message processing time and the number of served subscribers. This is an expected behavior of the system since increasing the number of subscribers adds extra load on the system, thereby the message processing time increases. The graph shows that the pro-active approach experiences a relatively higher

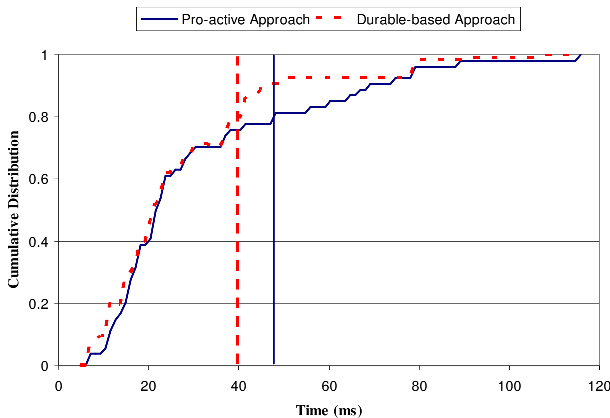

Figure 3. The cumulative distribution of handoff times.

overhead in terms of message processing time compared to the durable subscription-based approach. This can be attributed to the additional load on the brokers to serve dummy subscribers, which is not present in the durable subscription-based approach. This load includes managing extra connections, (de)activating dummy subscribers, and purging buffers. On the other hand, the pro-active approach shows better throughput than the durable subscription-based approach as it prevents message duplication and minimizes message loss. The pro-active approach therefore offers a tradeoff between the message serving overhead and the overall throughput.

5.2 The Handoff Latency

We evaluate the handoff latency under different load conditions imposed by the number of served subscribers (10, 50, 100, 150, and 200). We define the handoff latency as the time between sending the reconnect request and receiving the first message of the corresponding subscriber at its new broker. Figure 3 shows the cumulative distribution graph of the handoff latency observations.

From the figure, we observe that the pro-active and durable-based approaches show approximately similar handoff latency. Almost 80% of the handoffs are performed in less than 48 and 40 ms with the pro-active and durable-based approaches respectively. This indicates that the pro-active approach imposes low handoff latency as the subscriber context is always available at its new location prior to its movement. A portion of the handoff latency in the pro-active approach is a result of the switchover process involved between the actual and dummy subscriber before the buffered messages start being forwarded. While the actual subscriber takes over the dummy one should be deactivated and all neighbors should be notified about the arrival of the actual subscriber. Another portion of this latency, which also exists in the durable-based approach, is attributed to the preparation time for the broker to start delivering the stored messages. This time is mainly based on the broker's load conditions.

5.3 Overall Performance

We evaluate the behavior of pro-active and durable subscription-based approaches in terms of message loss/duplication, and overall throughput. The results of these metrics are given as a function of publication rate, and queue size. This allows us to study both mobility extensions under different system load levels.

Figure 4 (a) and 4 (b) show the percentage of message loss and duplication, along the left y-axis and the overall throughput along the right y-axis. The results are measured as the publication rate increases up to the maximum, the rate that the system can sustain. The publication rate has a direct impact on the percentage of message loss as we have a limited queue size. This can be seen in the graph where the message loss increases almost linearly with the increase of publishing rate. From the graphs, we can note that the pro-active approach reduces the message loss by more than 50% compared to the durable subscription-based approach. This approach suffers from high message loss because it forces the brokers to continue storing messages for disconnected subscribers. This will lead to overflowing the brokers' buffers and hence many messages will be overwritten. In contrast, the pro-active approach buffers messages on demand, that is, only when the subscribers disconnect from the network. This can optimize the buffer usage and hence decreases the message loss. Also, subscribers that move to a new broker for the first time will miss

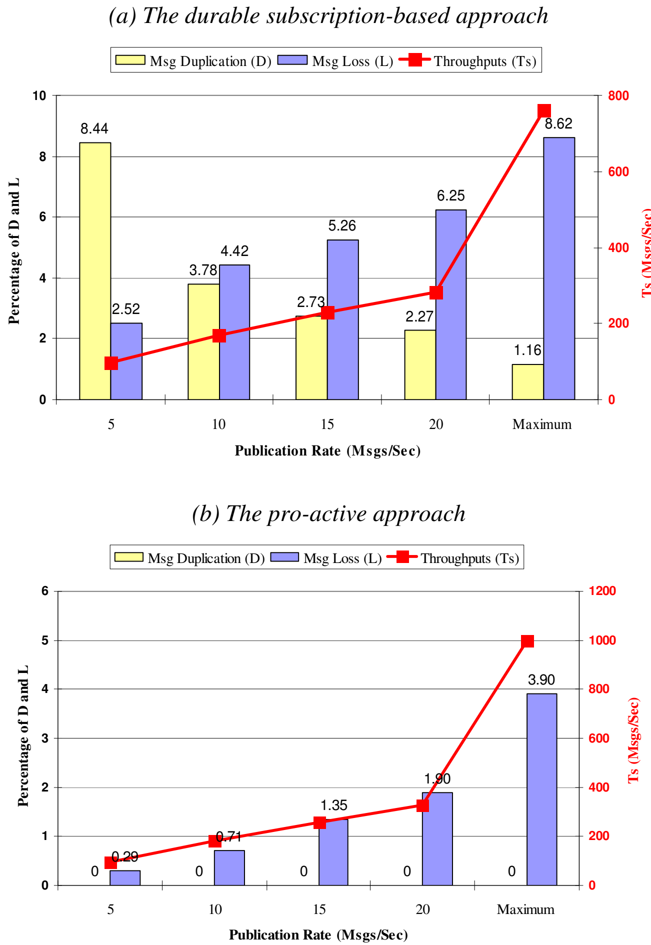

Figure 4. System performance at given publishing rates

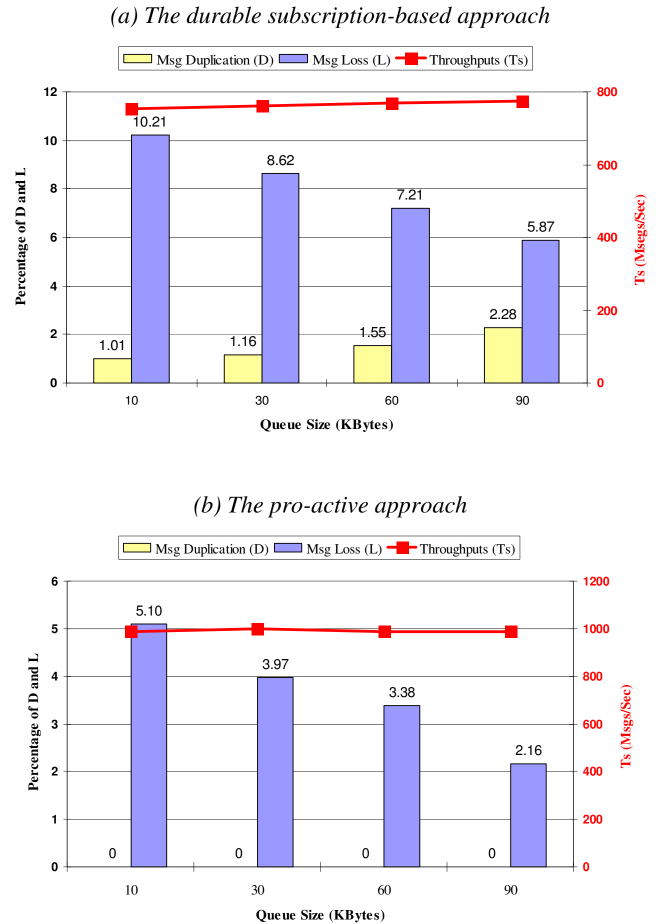

Figure 5. System performance at given queue sizes

the published messages during their handoffs.

The pro-active approach shows zero message duplication in all cases as it involves only one broker in delivering the messages of the reconnected subscribers. In this approach, the ID of the last received message is used to prevent the impact of race conditions during the subscriber handoffs. In contrast, a number of duplicated messages occur in the durable subscription-based approach as it requires the old and new brokers to keep storing messages for the moving subscriber. This results in receiving duplicated messages when the subscriber moves between the brokers.

From the graphs, we note that the pro-active approach achieves relatively higher throughput results than the durable subscription-based approach. This is a result of preventing message duplication, reducing message loss, and eliminating the overhead of buffering messages endlessly at every visited broker.

Figure 5 (a) and 5(b) respectively present the results of message loss, duplication, and overall throughput for the pro-active and durable subscription-based approaches with an increase of the queue sizes (10, 30, 60, and 90Kbytes). The figures show an inversely-proportional relationship between the queue size and the percentage of message loss. As the queue size increases, more messages can be accommodated and remain longer in the queue. Thus, the percentage of message loss decreases. This implies that the queue size has a direct impact on the system performance and should be well selected. It should be noted that increasing queue size beyond a certain threshold will not result in any further reduction of message loss, because a portion of the loss can be attributed to the handoff protocol and the characteristics of the wireless network. From the graphs, we note that the pro-active approach reduces message loss by nearly 50% compared to the durable subscriptions-based approach in all queue sizes. This is a result of the same reason described in Figure 4.

From the graphs, we note that the pro-active approach has reduced message duplication to zero in all cases. In contrast, the durable subscription-based approach suffers from message duplication due to the same reason described earlier. Message duplication shows a proportional relation with the queue size in this approach. This is because larger queue sizes can accommodate a larger number of identical messages. Note that larger queues may decrease message loss to a certain limit, but on the other hand increase message duplication noticeably.

The figures show that with the increase in the queue size, the achieved throughput does not change significantly. As the queue size increases, the publishing rate tends to decrease due to the overhead of the larger queue sizes. This includes the increased load of forwarding more messages to the subscribers and the frequent call to garbage collection due to the growth of heap

memory size within the JVM. As the publishing rate decreases, the subscriber throughput tends to decrease as well. On the other hand, larger queue sizes reduce message loss, which in turn increases the throughput. Hence, the subscriber throughput more or less remains constant until the queue size hits its threshold value. Beyond this value, the throughput will be negatively affected as the publishing rate decreases without any reduction in message loss. In all cases, the pro-active approach shows better throughput results compared to the durable subscription-based approach. This is an outcome of reducing message loss and duplication as well as the load of the endless buffering at every visited broker.

6. CONCLUSIONS AND FUTURE WORKS

We have presented a mobility support extension that is based on a *pro-active* approach to support mobility in pub/sub systems. The proposed approach ensures that the subscriber context is always one hop ahead of its current broker. We have described a relocation algorithm that provides the possibility to seamlessly extend pub/sub systems to mobile, wireless environments. We explored the adequacy of our proposed pro-active approach using a prototype implementation, presented evaluation results that investigate its performance and compared it to the solution based on durable subscriptions supported by the JMS implementations. The experimental results showed that our approach achieves better results compared to the durable subscription-based approach with respect to message loss/duplication, and overall throughput. The results indicate that our approach decreased message loss by more than 50% and message duplication to zero. It also achieves better throughput results under all scenarios with low costs in terms of mobility extension overhead.

We recognize that the experimental testbed described in this paper does not model a truly large network. We plan for the future work to deploy our approach on a large-scale network, especially with high frequency of handoffs, to explore scalability concerns.

7. ACKNOWLEDGMENTS

Our thanks to Carleton University and the Natural Sciences and Engineering Research Council of Canada (NSERC), for funding this project and giving us the opportunity to conduct this work.

8. REFERENCES

[1] Caporuscio, M., Carzaniga, A., and Wolf, A. Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications. *IEEE Transactions on Software Engineering, Vol. 29* (Dec. 2003), 1059-1071.

[2] Cugola, G., Di Nitto, E., and Fuggetta, A. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27, 9 (Sept. 2001), 827-850.

[3] Eugster, P., T., Felber, P., Guerraoui, R., and Kermarrec, M. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, Vol. 35, No. 2, June 2003, 114-131.

[4] Farooq, U., Parsons, E., and Majumdar, S. Performance of Publish/Subscribe Middleware in Mobile Wireless Networks. In *Proceedings of the 4th International Workshop on Software and Performance (WOSP04)*, Redwood City, CA, Jan. 2004, 278-289.

[5] Grigoras, D. Challenges to the Design of Mobile Middleware Systems. In *Proceedings of the international Symposium on Parallel Computing in Electrical Engineering (PARELEC'06),* Washington, DC, September 2006, 14-19.

[6] Gupta, N. and Kumar, P. R. A Performance Analysis of the IEEE 802.11 Wireless LAN Medium Access Control. *Communications in Information and Systems*, 2003 3(4), 279-304.

[7] Muhl, G., Ulbrich, A., Herrmann, K., and Weis, T. Disseminating Information to Mobile Clients Using Publish-Subscribe. In *Proceedings of the IEEE Internet Computing*, Vol. 8, No. 3, June 2004, 46-53.

[8] Muthusamy, V., Petrovic, M., and Jacobsen, H.-A. Effects of Routing Computations in Content-Based Routing Networks with Mobile Data Sources. In *Proceedings of the 11th annual international conference on Mobile computing and networking (MobiCom'05)*, (August 2005) New York, NY, USA, 103–116.

[9] National Institute of Standards and Technology. NIST Network Emulation Tool. *http://snad.ncsl.nist.gov/itg/nistnet*.

[10] Opyrchal, L. Astley, M., Auerbach, J. S., Banavar, G., Strom, R. E., and Sturman, D. C. Exploiting IP Multicast in Content-Based Publish/Subscribe Systems. In *Proceedings of 1FIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)* (April 2000), New York, NY, 185-207.

[11] Podnar, I. and Lovrek, I. Supporting mobility with persistent notifications in publish/subscribe systems. In *the Proceedings of Third International Workshop on Distributed Event-based Systems (DEBS 2004)*, Edinburgh, Scotland, UK, 24-25 May 2004, 80 -85.

[12] Sun Microsystems. Java Message Service (JMS) API Specification. *http://java.sun.com/products/jms*. 2002.

[13] Sutton, P., Arkins, R., and Segall, B. Supporting Disconnectedness-Transparent Information Delivery for Mobile and Invisible Computing. In *Proceedings of the 1st international Symposium on Cluster Computing and the Grid (CCGRID'01)* (May 15 - 18, 2001). IEEE Computer Society, Washington, DC, 277-285.

[14] Yin, J., Wang, X., and Agrawal, D. P. Modeling and Optimization for Wireless Local Area Network (WLAN). *Computer Communications Journal, Special Issue on Performance Issues of Wireless LANs, PANs, and Ad Hoc Networks*, *vol. 28*, (June 2005), 1204 -1213.