Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware

M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. P. Buchmann
Databases and Distributed Systems Group
Darmstadt University of Technology
Darmstadt, Germany
{cilia,fiege,haul,az,buchmann}@informatik.tu-darmstadt.de

ABSTRACT

Publish/subscribe (pub/sub) middleware facilitates loosely coupled cooperation and fits well the needs of spontaneous, ad-hoc interaction. However, newly started mobile applications have to be bootstrapped to interpret the current flow of notifications correctly and commence normal operation. This problem is aggravated in mobile environments where disconnections and context changes occur frequently.

In this paper, we propose two forms of subscriptions that allow consumers to subscribe to past events to improve the bootstrapping process. The first form uses logical mobility to harness possible client movements and subscribe in future locations to bootstrap virtual counterparts before the application needs the data. The second form is based on buffers and offers a way to integrate data repositories distributed in the network.

1. INTRODUCTION

The increasingly popular publish/subscribe (pub/sub) paradigm allows processes to exchange information without explicit knowledge about any particular destination address (e.g. IP address and port number) where producers or consumers can be found. This is founded on the principle that producers simply make information available and consumers place a standing request for information by issuing subscriptions. The notification service is then responsible for making information flow from a producer (publisher) to one or more interested consumers (subscribers). A publish/subscribe notification service provides asynchronous communication, it naturally decouples producers and consumers, makes them anonymous to each other, and allows a dynamic number of publishers and subscribers. The *loose coupling* of producers and consumers is the prime advantage of pub/sub systems and seems promising in the context of spontaneous, ad-hoc and pervasive environments.

One major characteristic of pervasive applications is *mobility*, which intrinsically requires appropriate support from the pub/sub system. This includes mechanisms to support roaming clients, e.g., to bridge phases of disconnection, and a notion of location tailored for efficient location-dependent information delivery.

DEBS'03 2003, San Diego/CA, USA

The phenomenon we focus on in this paper is that many event-based applications are characterized by having an initial phase of *notification observation* in order to get into a consistent state. In mobile scenarios this phase is basically needed to "adapt" the application to current contextual information which is only available locally. This initial stage is what we call the *bootstrapping latency*. Before this stage is finished an application might not be able to work properly.

Unfortunately, in pervasive scenarios where applications rely on *location-dependent information*, the problem is aggravated by the fact that whenever a client reaches a new location, consequently, a new bootstrapping phase needs to be started in order to collect location-dependent information which is required to bootstrap location-aware applications. As pervasive environments are characterized by a rather dynamic behavior, including mobility and frequent context switches, this situation is not an isolated occasion.

However, because of the asynchronous and data driven nature of the pub/sub paradigm, a mobile application cannot make assumptions about the time it will take before notifications required for bootstrapping are published. For instance, when a client spontaneously appears in a new location, it cannot rely on notifications being published as soon as it enters. In this kind of scenario, system responsiveness is not only degraded, but the time window in which a client application can actively "listen" is naturally constrained by the duration it stays at a particular location.

Our approach for enhancing mobile pub/sub middleware concentrates on reducing the bootstrapping latency. Based on the assumption that a consumer can be initialized by a sequence of notifications, recently published notifications are delivered to new consumers to make old notifications available to them as if they were created earlier. This is achieved by extending the pub/sub infrastructure to store recently published notifications in the network. Two complementary approaches are proposed that utilize per-client proxies and distributed buffers, respectively.

We devote the next section to a brief discussion about possible solutions to access recent information before we present our mobile pub/sub platform in Section 3. Section 4 refers to our initial approach of dealing with mobility when possible consumer movements are predictable. In Section 5 we complement this with a more detailed discussion of another algorithm which deals with, e.g., spontaneous appearing of mobile consumers and takes this into account for the minimization of bootstrapping latency. We close the paper with a review of related work (Section 6) and a conclusion/open issues in Section 7.

2. USING PAST NOTIFICATIONS

Consider for instance the following scenario: Client applications

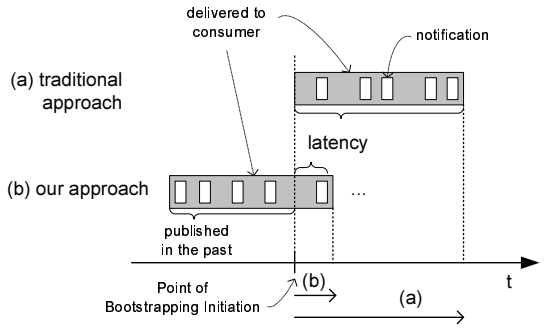

Figure 1: Bootstrapping latency

(e.g. vehicles) want to obtain the current status of the traffic light in their vicinity. In order to provide the required functionality at least the following approaches are possible.

By means of a traditional request/reply interaction a consumer needs to find whether its current location is contained in the range of any traffic light nearby. In this case, an explicit handle is needed to contact the traffic light or its proxy for requesting the current status. Although simple, querying external sources conceptually requires an infrastructure that supports directory lookups and remote queries. Moreover, the client is responsible for polling the required information with all well-known consequences. And from the viewpoint of a traffic light, the tightly-coupled nature of request/reply seems inferior to the anonymous, loosely-coupled "publish" in a pub/sub system.

With a "traditional" pub/sub mechanism consumers need to update their subscriptions explicitly and are only notified if they are inside the traffic light's range at the time of the state change publication [18, 22]. As a consequence, consumers are forced to wait until the next notification is published, which leads to non-negligible delays and considerable initialization latency, which might not be tolerable for mobile applications and erodes the reactivity of the pub/sub approach. An alternative is to publish a client's request for past events and route it according to existing filters to subscribers with appropriate buffers; a flexible solution, although it introduces ordering and duplication problems.

Another technique is to let traffic lights publish their status (and not their status change) with a pre-specified frequency [1]. In the worst case, applications need to wait a full update period. Hence, the choice of the frequency is a crucial parameter that affects not only applications but also resource usage. For instance, a mobile device with scarce resources, like low bandwidth wireless link and limited power supply, might suffer from heavy traffic on the wireless link when the frequency of broadcasts is too high.

The different approaches above illustrates a very simple 'context detection' that merely relies on the last notification as a description of current state. In general, the kind of data necessary to bootstrap an application differs widely, but in the following we concentrate on the class of applications that commence normal operation after having seen a sequence of notification. The essential idea to diminish the bootstrap latency is to provide the consumer with a correct sequence of past notifications as if it had subscribed earlier. Figure 1 basically depicts a comparison of the "traditional" case (a) with our approach (b) of using past notifications in order to reduce bootstrapping latency.

A proper support by the infrastructure for this mobility scenario is to buffer published notifications somewhere in the network and to deliver them to newly subscribed consumers as required. Transparent delivery decouples clients and buffer management and allows for the integration of various implementation strategies, e.g., distributed caches, proxies, peer lookup, centralized stores, etc. Please note that the availability of an arbitrary number of past notifications cannot be assured by the infrastructure, which only mediates between applications and buffers. Consequently, in the worst case (e.g., not enough notifications were published) our approach cannot fully avoid bootstrap latency. But on the other hand, in this case no additional overhead is added and the bootstrap latency is not increased.

In this paper, two approaches are presented for buffering notifications for potential later use. They complement each other and may be combined with other methods to get the relevant current state of the context in which a client operates.

3. MIDDLEWARE FOR MOBILE PUB/SUB

The following discussion is based on the REBECA notification service [15, 10], which we use as basis for the proposed mobility support.

3.1 Architecture

Processes of a system based on pub/sub communication can act both as producers and consumers. They are clients of the underlying notification service. The communication interface to the service is rather simple and consists of *pub*, *sub*, *unsub*, and *notify* functions. The last one is a callback function called on the registered consumer to deliver a notification. A *notification* is a message that reifies and describes an event occurrence. Notifications are not published towards a specific receiver, but conveyed by the underlying notification service to those consumers that have registered a matching subscription.

Subscriptions can be seen as boolean functions over notifications (called *filters*). The most flexible scheme for specifying these filters is content-based filtering, which utilizes predicates on the entire content of a notification [14].

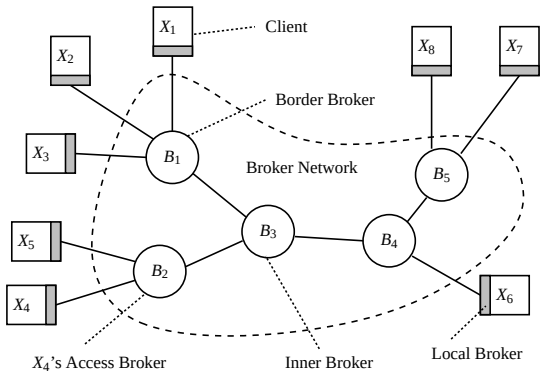

Figure 2: The router network of REBECA.

The service implementation is distributed to meet the mobility scenario and scalability considerations. The communication topology of the pub/sub system is given by a graph, which is assumed to be acyclic and connected (Fig. 2). The graph consists of brokers and clients. The edges are communication links that are point-to-point and obey FIFO ordering of messages. Brokers are processes that route notifications along multiple hops to the appropriate clients. Three types of brokers are distinguished: *Local brokers* constitute the clients' access point to the middleware and are part of the communication library loaded into the clients. A local broker is connected to at most one border broker. *Border brokers* form the boundary of the distributed communication middleware and maintain connections to local brokers, i.e., the clients. The border broker to which a client is connected is called its *access broker*. *Inner*

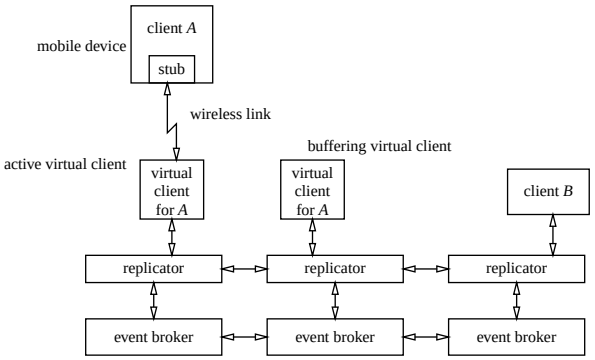

Figure 3: Transparent implementation of logical mobility on top of a mobile pub/sub system.

brokers are connected to other inner or border brokers and do not maintain any connections to clients.

3.2 Possible routing strategies

Each broker maintains a routing table that determines in which directions a notification needs to be forwarded. Each table entry is a pair (F, L) containing a filter and the link from which it was received, signifying the link along which a matching notification has to be forwarded. Such data structures are used in many pub/sub services [6, 7]. The routing decision is assumed to be an atomic operation so that the end-to-end sender FIFO characteristic holds. The routing tables are maintained to correspond to the available information about active consumers and their subscriptions. Each broker forwards this information according to the routing algorithm used.

The basic form of routing is *simple routing*: filters are added to the routing table together with the link they were received from and they are forwarded in all other directions. This strategy may be improved by using *covering* and *merging* [15], but for the sake of simplicity we assume simple routing throughout this paper. Details of the subscription process are given together with the extensions we propose in Section 5.3. *Advertisements* are issued by producers to announce the kinds of notifications they are about to publish. They are distributed in the network and tell brokers into which directions a subscription have to be forwarded.

3.3 Roaming clients

As a first step towards a pub/sub middleware for mobile and pervasive systems, the REBECA notification service was extended to facilitate roaming clients, distinguishing *physical* and *logical mobility* [23, 9].

Mobile clients frequently disconnect from the network, e.g., due to power-saving requirements. And they often re-connect at a different location, possibly within a completely different physical or administrative context. The major concern for middleware support in such a setting is to make this change *transparent* to applications that are not aware of mobility.

Logical mobility addresses a different aspect of mobility in pervasive environments: *location awareness*. Information about the current location and the available information in the vicinity is a rich source mobility aware applications can draw from. In this context pure location transparency, like in the roaming case above, can be counterproductive. Most strikingly, location-based services (like pervasive tourist guides) rely on an explicit knowledge about the current location. A system supporting mobility should not only blend out unwanted phenomena, like disconnectedness, but should also facilitate *location awareness*. We have built support for location awareness into REBECA by introducing a special keyword *myloc* into subscriptions that is automatically adapted to the current location and is used to filter notifications. Hereby we made *location* a first-class citizen within REBECA [9], which is exploited in the next section for location-dependent buffer placement.

4. PRE-SUBSCRIPTION APPROACH

In this section we briefly discuss an algorithm which defines a mechanism for subscribing intelligently to information at locations a client might move to in the near future [8]. This is done before the client actually gets there (so called *uncertainty of movement*). The overall goal is to leverage information about the movement of a client to shorten significantly the time a client application needs to bootstrap at a new location.

4.1 Basic idea

The basic idea is to implement a distributed per-client caching at prospective future locations. It is based on the assumption that a client's movements will adhere to a *movement graph* that models the current location and its adjacent next hops. This graph was introduced for logical mobility and location awareness [9], but it may also predict physical movements. This topology information must be available in the infrastructure and may either be derived automatically or must be supplied by some external entity, such like an external location service, providing appropriate information about the location of a client.

While the client moves in the real world, the proposed mechanism instantiates a *virtual client* (VC) at every such location and *pre-subscribes* at possible future locations. The intended semantics is that the client listens and buffers before it actually reaches the location in question.

The time frame Δt refers to the time span the VC is listening before a client actually gets there. Within Δt the VC receives and buffers all notifications the client would have received. Once the client reaches any such location and is connected to its VC, buffered notifications are delivered to the client in the same order as if it has been there, which is semantically the same as if the subscriptions were issued in the past.

4.2 Architecture

The basic approach is to use an additional layer of *Replicators* between client and pub/sub system that are responsible for monitoring clients according to the constraints given by the movement graph of the underlying location model (cf. Fig. 3). Replicators implement the movement graph and they are connected to the adjacent peers in the graph. Whenever it is likely that a client will move to a location within a set of possible locations, the layer of replicators will instantiate and maintain *virtual clients* at those next locations. These virtual clients re-subscribe to the client's current subscription, which are subject to location-dependent automatic adaption, and buffer information for a client for potential future use.

The importance of virtual clients instantiated at possible future locations increases when location-dependent filters are used to select only locally distributed notifications (cf. Geocast [18]). Intuitively, the client would like to experience the notion of being subscribed for its interests "everywhere, all the time" and increase the reactivity of the system to moving clients. Whenever a mobile client wants to take into account location-dependent information and the replicators were not able to predict this location change the client may choose to "fall-back" to waiting for interesting notifications. We deal with this problem in Section 5.

4.3 Discussion

It is guaranteed that the VC is delivering notifications in the same order as the actual client would have received them, per-sender FIFO ordering holds. More importantly, when a client is connected to the virtual client and according to the requirement of responsiveness of the system, notification delivery only takes a very short amount of time and is almost instantaneously, hence maintaining minimal the time needed for bootstrapping. Furthermore, bootstrapping is not done for individual subscriptions, but the virtual client acts as a proxy and may even combine and preprocess multiple streams of events; this is, however, an open future issue. For many situations this proxy solution is desirable, yet in certain situations it falls short in at least two aspects:

1. Accuracy of the results depends on the time Δt the virtual client is "listening" at a new location.

2. Interests of the client may change, i.e., new subscriptions for location-dependent information are made, or the client was suspended and is reconnecting at an unforseen location.

The first situation can occur either if information is disseminated seldom and the Δt is too small to ensure reception of such a notification, or the client is moving too "fast" with respect to the *uncertainty* maintained by the system, i.e., $\Delta t \to 0$. The other situation occurs whenever a client application "spontaneously" subscribes to new location-dependent information or does not obey the movement graph. Obviously, at the current location the new subscription must be propagated through the broker network exactly like in a non-mobile pub/sub system.

5. SUBSCRIBING INTO THE PAST

In some sense, the approach introduced in this section is complementary to the one of Section 4. While the above approach is optimized to maximize the responsiveness of the system by presubscribing to information needed in the future, the approach in this section is to "trade" responsiveness for consistency and completeness by using subscriptions aiming backward "in history." Especially in situations where the above approach falls short, i.e., the Δt for listening is too small and unforeseen subscriptions come into the broker network, applications can greatly benefit from "backward" angled subscriptions-into-the-past. By accessing information which was already delivered but is stored within the broker network applications can reach a consistent state and be set-up without listening for notification in the future. By maintaining recent published notification within the broker network the bootstrapping phase might be sped-up significantly.

In the above example of the traffic-light, a (hypothetical) driving assistant should warn a driver in case the driver is about to cross the traffic-light when the status is "red." For some reason the associated virtual client (as in Section 4) was not active when the traffic light disseminated its state change from "green" to "red". The only way for the assistant to determine whether or not to warn the driver is to have access to the last sent notification, which was already delivered in the past. Obviously, here, the complimentary nature of this approach is highly beneficial.

5.1 Basic idea

We have extended the subscription method in order to provide consumers/subscribers with the possibility to express their interest in happenings occurred in the past. In this case, and in addition to the subscription filter that expresses their interest, they can specify a number of *n* notifications they want to access from the past. As in the standard case, the pub/sub system delivers the last *n* notifications that match the subscription to the subscriber through the notify callback method as in normal operation. This makes opaque to the consumer that those received notifications have already been delivered in the past. After sending the solicited notifications stemming from the past, standard delivery of present and future notifications commence operation. Note that the system cannot guarantee that it can deliver the total number of notifications specified in the subscription. This depends on the individual policies for buffering notifications and what notifications are available in the broker network. As a consequence, the client application should not assume that the first *n* notifications in fact are part of the "past."

5.2 Prerequisites

In order to keep track of past notifications, buffers are implemented within the broker network and they are accessed by the different versions of the algorithm presented in the following. A buffer is assumed to be simply a circular log of bounded length that stores notifications matching a filter assigned to the buffer. Any broker may install a history buffer as a cache of the latest notifications forwarded through this broker. In connection with a specific subscription other temporary buffers may be created as well. If we want to ensure a minimum number of notifications available for replay, border brokers have to buffer this number of notifications published by any locally attached producer. In general, any broker may maintain history buffers to improve data placement localities, though buffers only listen to passing notifications and are empty before the first subscription initiates delivery.

Notifications travel through the broker network from the producers to the consumers along delivery paths. We refer to the publishing direction as being directed *downstream*, while subscriptions and some administrative messages are directed *upstream* towards the producer. The *replay message* is of administrative kind and contains a set of notifications. Clients of the pub/sub service need not to expose any unique identifiers, but the pair (C, F) of a consumer and its issued subscription filter is presupposed to be unique; at least a unique ID can be assigned by the access broker.

5.3 Algorithm Outline

The following basic approach to subscribing with buffer replay extends the subscription process available in the REBECA pub/sub service. A subscription can now also include past notifications; the routing configuration is updated as before, but delivery of new notification is postponed; matching buffered notifications are fetched from the network; and finally, the fetched data has to be delivered before new notifications.

Issued subscriptions have to contain the number of past notifications that are to be delivered, according to the semantics given in Sect. 2. For example, assume X_1 at broker B_1 to be a publisher matching the interests of subscriber X_6 at B_4. This establishes a delivery path $\{B_1 \to B_3 \to B_4\}$ that is described by the respective routing tables. Consider now that X_7 subscribes also to the same kind of notification. As before, subscriptions are propagated through the broker network to update routing tables to direct matching notifications towards the consumer. Subscription forwarding stops at a broker that already carries an identical filter.[1] The above exemplary subscription is forwarded to update B_5 and B_4, but not any further. This is the already known subscription propagation as implemented, e.g., in the REBECA pub/sub service.

Immediately before the new link is activated to start delivery

[1]With merging and covering it already stops when an existing broader filter ensures that notifications matching the new subscription are forwarded to this broker and so to the consumer.

```
/** upon receiving subscription (C, F) for past p events
 * via link L_N */
void receiveSub(C, F, p, L_N) {
  if (localClients.contains(C)) {
      routeTable.setHold(C, F)
      buffers.newBuffer(C, F, p) // create temp. buffer for replay
  }
  if (routeTable.includes(C, F) {
      replay(C, F, L_N, p) // prepare replay message and send
  } else {
      routeTable.add(C, F, L_N)
      propagate(C, F, p, L_N) // to all neighbor brokers with
  } // matching advertisements except L_N
}

/** upon receiving notifi cation n from B_j */
void receiveNotif(n,B_j) {
  routeTable.route(n)
  historyBuffers.append(n)
  for(∀b ∈buffers with matching assigned F)
      b.append(n)
}

/** upon receiving replay(C, F, [n_1, ..., n_m]) */
void receiveReplay(C, F, [n_1, ..., n_m]) {
  if (buffer.exists(C, F)) { // i.e. this is access broker
      b := buffers.get(C, F)
      b.prepend([n_1, ..., n_m])
      if(b is completely fi lled) {
        deliver b
        buffers.remove(C, F)
        routeTable.clearHold(C, F)
      }
  } else {
      routeTable.route(C, F, [n_1, ..., n_m]) // route replay towards consumer
  } // according to unique (C, F)
}

/** upon timeout of buffer (C, F) */
void receiveTimeOut(C, F){
    buffers.get(C, F).deliver()
    buffers.remove(C, F)
}
```

Figure 4: Basic algorithm for subscriptions into the past.

of passing notifications, the buffer-fetching functionality is called. The broker selects as much of the most recent notifications from a locally kept history as necessary to meet the subscription request. These are sent as a replay message down the new link towards the consumer's access broker. We will later suggest more advanced strategies that include history buffers at other brokers.

The access broker unpacks and delivers the replay notifications to the consumer before delivering new notifications. In general, new notifications must be delayed until the replay message is sent in order to deliver the buffered notifications first. In this simple approach, however, the replay is prepared only at the nearest branch on the delivery path, so new notifications cannot overtake the replay. Unfortunately, the desired number of past notifications may not be available at this broker.

The algorithm presented in Figure 4 sketches the core algorithm common to all presented extensions made in the following paragraphs. For all extensions, only two methods need to be changed as it is presented in Figure 5. This figure particularly shows the refinements for the simplest case as outlined above.

In order to provide a framework for the general case, the algorithm explicitly blocks delivery at the access broker in case new notifications arrive before the replay. For the basic version, this is not necessary. The timeout stops waiting for replays and starts delivering new notifications if not enough replay data was received in time; this is only necessary if multiple replays are expected.

5.4 Algorithm

```
void replay(C, F, L_N, p) {
    b := historyBuffers.get(C, F)
    if (b.length > 0) {
        b.sendReplay(C, F, p)
    }
}

void propagate(C, F, p, L_N) {
    for (∀n ∈ localClients\{L_N} ) {
        requestReplay(n, C, F, p)
    }
}
```

Figure 5: Specification and refinements of the `replay` and `propagate` methods.

The algorithm presented in Sect. 5.3 is very naive in its restriction to search only for notifications at the nearest buffer and therefore it needs to be extended for practical relevance. If more buffers are considered for preparing the replay, two problems arise. First, new notifications and replays are concurrent and must be ordered correctly. And second, multiple replays may cover different producers so that reordering is not possible without identifying producers and individual notifications (a strong requirement we deliberately avoided so far). In the next subsections we suggest a number of improvements that search for more buffered notifications, cope with concurrent new notifications, join buffers of multiple producers, and reduce traffic by using sequence numbers.

Largest History Buffer

We return to the example given in Sect. 5.3 and specify the subscription process in more detail.

For now assume that every broker has no more than one link with a matching advertisement. In the basic approach the reduction of bootstrapping time for a consumer depends on the size of the history buffer of the first broker that is discovered on an existing delivery path. Consider for instance the network as presented in Figure 2 where X_7 issues a subscription and this subscription involving past notifications depends on the broker B_4.

In order to provide a better solution the first extension includes history buffers at other upstream brokers: another broker (B_1) further upstream may have a larger buffer that could be used instead. However, new notifications may be in transit while the first broker (B_4) requests a replay of B_1's history buffer. Thus, the first broker (B_4) needs to hold notifications for the consumer (X_7), or its access broker (B_5), until a replay message is received. The replay needs to be shortened by the number of held notifications to avoid duplicating notifications. After the replay content is passed to the consumer, held notifications are delivered.

By including the contents of buffers at other brokers, potentially a larger fragment of recent history can be accessed. This assumes however, that buffers have different sizes and that is possible to find a larger buffer in brokers upstream.

Merged Histories

Looking for a broker in the delivery path with enough notifications (as it was proposed above) involves communication costs and time. Considering that during this searching time new notifications may arrive implies that less notifications need to be specially delivered from other brokers. This leads to the next modification of the algorithm: All queried brokers send as much of their history buffer as is available in the hope, that enough notifications were issued in the meantime to fulfill the requested number of notifications.

Coming back to the example, the first broker B_4 might decide to

stop waiting for history replays and start delivering held notifications. Another broker B_3 between B_4 and B_1 might have a larger history than B_4 but not sufficiently large to satisfy the requested number of notifications. However, it could send a replay anyway increasing the probability that B_4 is able to fulfill the request based on newly received notifications plus received replays so far. B_4 needs to keep track of outstanding replay requests unless there is a timeout defined.

In order to merge replays with the local history buffer, B_4 needs to reduce the received replay by the number of notifications in its own history buffer to avoid duplicates. Since sender FIFO is guaranteed, all replays are aligned to the beginning of the first broker's (B_4) buffer. In other words, the most recent notifications are present in B_4's buffer as well as in the replay. Figure 6 shows the content of two brokers' history buffers, B_3 and B_4, with administrative messages and a new notification "g" is being published at X_1. At time $t+1$, B_4 receives a subscription requesting four past notifications while B_3 receives notification "g". B_4 allocates a new buffer large enough and copies the content of the history buffer to this new buffer. Next, at $t+2$, B_3 forwards "g" to B_4 while B_4 requests a replay of four notifications from B_3. Notification "g" is added to the new buffer at B_4. In $t+3$ the replay is sent to B_4. It can be seen that B_4's buffer contains the same leftmost (most recent) notifications as the replay message. The result after removing the duplicate notifications is shown at $t+4$.

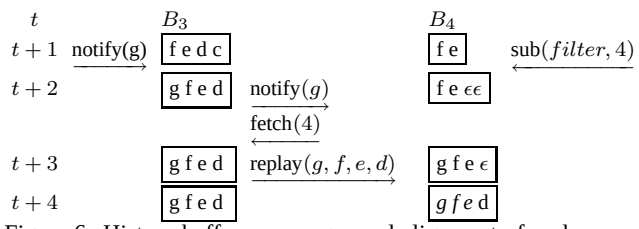

Figure 6: History buffers, messages and alignment of replay contents with history buffer contents

With this modification, the bootstrapping delay of a consumer is guaranteed to be no larger than without recent history in the worst and better in the average case.

Junctions

Up to this point we have assumed a single matching advertisement per subscription, which we will relax now. *Broker IDs* help to distinguish notification sources and allow elimination of duplicates (notice that the simple alignment as depicted in Fig. 6 does not hold anymore).

Junction brokers (e.g. B_3) are brokers that are connected to more than one peer with matching advertisements. Upon a replay request, junction brokers should query all peers in order to get a good estimation of past notifications. In general, a specific ordering cannot be assumed between replays of different producers. Replay processing may be based on first-come-first-serve, random interleaving, or timestamp ordering if logical or global-time clocks are presupposed. However, only replays from the junction broker's own history buffer will have the order of notifications the consumer would have observed if it were subscribed earlier.

When receiving replays from multiple brokers they are not guaranteed to be aligned with the local history buffer anymore thus removing duplicates is more difficult. Therefore, globally unique broker IDs must be added to a notification envelope when it is published. Since the ordering of notifications relative to the source is fixed, replays will include the same most recent notifications which allows to drop as many notifications from a replay as already present in the local buffer for this source. Notifications are always stored in the history buffer with the complete envelope.

Sequences

Although the algorithm and the extension presented above eliminates duplicates before delivering notifications to the consumer, notifications may be sent multiple times over the network since a very recent notification will be included in every replay. *Sequence numbers* can be used to shift the task to detect and eliminate duplicates from the access broker to the infrastructure.

In the same way border brokers connected to producers add their broker ID to the notification envelope, a sequence number is added. Such sequence numbers can be leveraged to eliminate duplicate sending. Requests for history replays include a list of tuples (b, s) with b as broker ID the producer is connected to and s as the sequence number of the oldest notification from this producer still contained in the local history buffer. As an additional benefit, the requested number of replay notifications can be reduced by the number of notifications found in the local history buffer that match the request.

Putting it all together

In this section we describe an algorithm that takes into consideration those remarks made previously in this section. Additionally, three new message types are introduced for inter-broker communication: *fetchHistory(expression, list, #past notifications)*, *replay()*, and *expect(tagname)*.

Handle Subscription Message: Upon receiving a subscription including past notifications p, a broker checks first if it is the responsible border broker or *access broker* for the issuing consumer. A buffer H with size p is created to assemble an estimation of past notifications, a counter r for outstanding requests, and a map L for lowest sequence numbers are allocated. Delivery for the requesting consumer according to this subscription is set to hold.

If the current access broker does not carry the requested subscription, the request is sent to all connected brokers that have sent a matching advertisement, hence are upstream. r is incremented for each request.

If the broker already has a matching subscription and a local history buffer matching the subscription expression exists, notifications are copied to H until the end is reached or the request is fulfilled. At the same time L is filled with lowest sequence numbers per originating broker. The number of expected notifications is decreased with each notification.

If more past notifications are needed, a *fetchHistory* message is sent to all connected brokers with matching advertisements and r is incremented for every request. If more than one broker is contacted, a junction marker is set.

Any inner broker that does not already have a matching subscription re-sends the subscription request to all upstream brokers. For each request an *expect* message is sent back downstream and a junction marker if more than one broker is contacted. In addition, an empty *replay* message is sent downstream.

If a non-access broker has a matching subscription, a local buffer H' with size p' is allocated. The broker scans the local history buffer looking for notifications that have a lower sequence number than indicated in the request. If it is the case then it copies them to H' and updates the number of outstanding notifications and the sequence number list L. All notifications in H' are included in a *replay* message and sent downstream. Whenever the number of outstanding past notifications is non-zero, corresponding *fetchHistory* messages using L' are sent to all connected brokers that have matching advertisements. *expect* messages are sent as described in

the case when no matching subscription was found. Buffer H' is removed.

Handle fetchHistory Message: A non-access broker handles *fetchHistory* messages similar to *subscribe* messages.
An access broker should never receive a *fetchHistory* message.

Handle Notification: Upon receiving a matching *notification* for a consumer that is set to hold, an access broker copies it to the local history buffer and delivers it to all other consumers that are not set to hold. In addition, it is queued in H and the number of expected notifications is decreased. If the number of notifications is zero, all queued notifications are delivered and all helper structures are removed. The hold marker is removed.

A non-access broker transmits the notification downstream.

Handle Replay Message: Upon receiving a *replay* message, an access broker appends all included notifications to H and r is decremented. If it has been signalled before that a junction has been encountered, ordering of notifications from different producers cannot be reconstructed after this point. The access broker could try to order notifications from beyond the junction based on the notification's production timestamps.

If no replays are outstanding, all notifications from H are delivered to the consumer and the hold marker as well as all helper structures H, L, r are removed.

A non-access broker relays the *replay* downstream.

Handle Expect Message: Upon receiving an *expect* message, the access broker increases r. An included junction marker is stored. A non-access broker relays an *expect* message downstream.

5.5 Considering the time dimension

Subscriptions in the past can be also specified with a time bound. In this case, a subscription into the past asks for notifications that have been published *m* minutes in the past (relative to subscription time). A first approximation could be achieved by synchronizing the clocks of all border brokers. In this way, at *publishing-time* border brokers attach a timestamp to notifications in order to represent the time when they have entered into the pub/sub system.

At *subscription-time* the border broker sets the time bound by simply subtracting the relative bound of the subscription from its local, synchronized time. This time reference is then used by the algorithm to search in the broker network for matching notifications with a newer timestamp.

Moreover, a combination of number of notifications and time into the past could also be useful, i.e., the last ten notifications within the last five minutes. This combination constrains the search in the buffers of the broker network. That means that there are two criteria to stop the search: (a) once the number of notifications within the reference achieves the solicited number, or (b) once a timestamp older than the reference is found.

5.6 Discussion

The algorithm introduced in this section was designed to complement the algorithm shown in Section 4. In situations in which pre-subscriptions fall short, the access to buffers in the broker network and the past notifications possibly stored there can compensate for a rather small Δt.

The tradeoff between the two techniques is one of responsiveness versus flexibility. Subscriptions into the past have in general not the same responsiveness as a system relying on pre-subscriptions, because a subscription has to go through the broker network at least until an appropriate buffer is found or the algorithm terminates without success. On the other hand, this approach does not rely on any movement restrictions and can draw from localities within the broker network. This is especially important for location-dependent subscriptions that are typically of interest to more than one subscriber. Here, close-by history buffers possibly carry the necessary past notifications so that they can be fetched fast.

Due to the complementary nature of the algorithms a combination of both approaches is obvious: while pre-subscriptions are optimal for per-client buffering of information and in situations where a Δt is sufficiently large, subscriptions into-the-past are per-subscription caches which are getting more important in spontaneous settings or when the uncertainty maintained by the system is small, e.g., when the number of clients is high and the number of virtual clients per real client must be restricted. Fortunately, in such scenarios with small Δt, the likelihood that another client already has subscribed to the same location-dependent information is correspondingly higher.

6. RELATED WORK

We are only aware of some pub/sub systems offering support for mobile clients to some extent. The Java Message Service (JMS) includes durable subscriptions that store intermediate messages during disconnections [20]. They can be reactivated by subscribing with the unique ID assigned to each of them, and thus our approach and this feature of JMS are based on a similar idea. However, in JMS the semantics of reconnecting in a distributed system is not specified. In IBM's MQseries retained publications are valid until a follow-up is published and the last one is also delivered to all new subcriptions announced in the meantime. Huang and Garcia-Molina [11] provide a good overview of possible options for supporting mobility. An extension to Elvin exists that allows for disconnectedness using a central caching proxy [21] but is not used for location dependent subscriptions. CEA [2] and JEDI [7], too, tackle problems of mobility. JEDI uses explicit *moveIn* and *moveOut* operations to relocate clients, which is problematic if wireless communication just breaks down when moving, and it has no explicit notion of location as a first class concept for pub/sub systems. The mobility extensions of SIENA [3] are very similar to the JEDI approach. Probably the most related work is STEAM [12], a middleware service designed for wireless local area networks using the ad-hoc network model where there are no access points and system wide services. Subscribers only consume events produced by geographically close-by publishers. For this it relies on proximity-based group communication [13]. As a result it is not clear how this approach can be applied in an application like weather forecast for a particular remote location produced by a forecasting service. Other communication paradigms facilitating loose coupling, like Linda tuple spaces [5], were also investigated for their potential support of mobility (e.g., in LIME [17] and CORELIME [4]), but are out of the scope of this paper.

7. CONCLUSIONS AND OPEN ISSUES

This paper is motivated by the use of pub/sub notification services in pervasive applications where mobility plays a prime role. In order to adapt to context changes, moving clients requires an initialization phase to commence normal operation from a valid state. However, without proper countermeasures in the infrastructure the latency of a client's bootstrapping phase has the potential to severely impair the usability of the pub/sub paradigm in pervasive scenarios. We have proposed two enhancements of the pub/sub platform REBECA that provide applications with sequences of past notifications to optimize the bootstrapping phase.

The first approach deals with the notion of subscribing in advance at locations a client might move to in the near future, according to a model of possible movements. Virtual counterparts

are installed, establishing per-client caches in the network to deliver stored notifications to the client if it eventually arrives at this location. In this way, past notifications are made instantaneously available as if the client has subscribed in the past.

Spontaneous movements and subscription changes, however, are not efficiently handled and motivated a complementary approach to establishing and searching additional buffers in the broker network. They are looked for on the paths towards producers and basically contain recently sent notifications. For bootstrapping purposes, new subscriptions may ask for a certain number of recent notifications, which are taken from these buffers. It must be noticed again that particularly in mobile environments the bootstrapping latency may occur frequently. Therefore the reduction of this latency is our main motivation. However, this buffering approach can be also applied to stationary applications since the bootstrapping period is common to most event-based applications.

From the infrastructure point of view the first approach requires the introduction of a new entity, the virtual client, which in fact plays the role of a client, but instead of processing with caching capabilities. The second approach involves changes in the subscription functionality due to the consumer now being able to specify how "far" in the past the system should go to start delivering notifications. Additionally, brokers need to provide buffer capabilities.

Besides attaching conventional data stores to the network and active database research, little (insufficient) attention is paid to the notion of the lifetime of events. Consequently, a number of open issues remain.

Scalability of efficient content-based filtering algorithms has been investigated for standard, i.e., non-mobile, environments [16]. Pervasive environments with their complex issues of different forms of mobility pose greater challenges both in the number of clients to support as well as in the dynamics of their behavior.

How can the efficiency of the underlying routing framework be best exploited? Tuning and adaption are of increasing importance in these systems. Buffer sizes and placement strategies determine efficiency, resource usage, and therefore scalability.

It is just a small step to generalize the concept of location-dependent subscriptions to "state-dependent" subscriptions, opening the whole area of context-awareness [19] to the domain of pub/sub middleware systems. How can systems implement or make use of such dynamic filters, which depend on a function of the local state of the client (not only its current location)? Can virtual counterparts offer more than just being an intelligently placed cache?

Acknowledgments

We thank Gero Mühl for his cooperation in the REBECA project, Felix Gärtner, Sidath Handurukande, and Oliver Kasten for many helpful discussions on the topic of mobility and publish/subscribe. The work was supported by the Deutsche Forschungsgemeinschaft as part of the PhD program "Enabling Technologies for E-Commerce."

8. REFERENCES

[1] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proc. of ACM SIGMOD*, pages 183–194, May 13–15 1997.

[2] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, 2000.

[3] M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of clients mobility in the Siena publish/subscribe middleware. Technical report, Department of Computer Science, University of L'Aquila, Oct. 2002.

[4] B. Carbunar, M. Valente, and J. Vitek. Corelime: a coordination model for mobile agents. In *Proc. of Intl Workshop ConCoord 2001*, 2001.

[5] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, Apr. 1989.

[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, 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] L. Fiege, F. C. Gärtner, S. B. Handurukande, and A. Zeidler. Dealing with uncertainty in mobile publish/subscribe middleware. In 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing, to appear, 2003.

[9] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *IFIP/ACM Middleware 2003 (to appear)*, 2003.

[10] L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *Proc. of ACM SAC'02*, pages 385–392, 2002.

[11] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proc. of MobiDE01*, May 2001.

[12] R. Meier and V. Cahill. STEAM: Event-based middleware for wireless ad hoc networks. In *Proc. of DEBS'02*, 2002.

[13] R. Meier, M.-O. Killijian, R. Cunningham, and V. Cahill. Towards proximity group communication. In Banavar:2001:MobileMiddleware, editor, *Advanced Topic Workshop Middleware for Mobile Computing (Middleware 2001)*, 2001.

[14] G. Mühl. Generic constraints for content-based publish/subscribe systems. In *Proc. of CoopIS '01*, volume 2172 of *LNCS*, pages 211–225. Springer-Verlag, 2001.

[15] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt Univ. of Technology, 2002.

[16] G. Mühl, L. Fiege, F. C. Gärtner, and A. P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. MASCOTS 2002*, 2002.

[17] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proc. of ICDCS-21*, pages 524–533, May 2001.

[18] J. C. Navas and T. Imielinski. Geocast - geographic addressing and routing. In *Proc. of the ACM/IEEE International Conference on Mobile Computing and Networking*, pages 66–76, 1997.

[19] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[20] Sun Microsystems, Inc. Java Message Service Specification 1.1, 2002.

[21] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness – transparent information delivery for mobile and invisible computing. In *Intl Symposium on Cluster Computing and the Grid*, 2001.

[22] C. L. Tan and S. Pink. Mobicast: a multicast scheme for wireless networks. *Mobile Networks and Applications*, 5(4):259–271, 2000.

[23] A. Zeidler and L. Fiege. Mobility support with REBECA. In *Proc. of the ICDCS Workshop on Mobile Computing Middleware*, 2003.