The last hop of global notification delivery to mobile users: matching preferences, context, and device constraints

Dmitrii Zagorodnov Dag Johansen
Dept of Computer Science
University of Tromsø, Norway
{dmitrii,dag}@cs.uit.no

Abstract

Events injected by publishers into a publish/subscribe system may reach users through a variety of devices: a stationary desktop, a laptop, a mobile phone, etc. We argue that the "last hop" – from the network to the output device – has unique properties, owing to the mobile nature of these devices, and as such demands special consideration. In particular, user's preferences and location may limit ***what*** *should be forwarded to a device. Furthermore, technological constraints, such as network bandwidth availability and battery power, suggest that the decision* ***when*** *to forward messages is also important for optimizing user's experience. We describe a new publish/subscribe system with volume-limiting mechanisms and explain how user preferences, context, and device constraints can be accommodated in such a system. Notably, based on results of simulations, we propose a simple algorithm for low-cost "prefetching" of notifications to mobile devices in cases when network bandwidth is insufficient.*

1 Introduction

Users often view the Internet as a massive client-server infrastructure. Nevertheless, this pull-based structuring technique is gradually being complemented with push-based structures. E-mail is an excellent example of a pervasive application where messages are pushed to the inbox of recipients. Another emerging example application is network news. Users can subscribe to news services like, for instance, New York Times and be alerted when certain headline news stories are published. The financial industry also depends more and more on being the first to receive certain subscriptions. A stock broker exclusively receiving push-based information through, for instance Reuters, might have enough seconds to take action and outperform other competitors depending on the same information. Seconds count in this financial competition. Publish/subscribe systems like, for instance, Siena [3], SCRIBE [4], and Gryphon [1] are also being developed to target this new type of push applications.

We built and deployed the first academic sensor network in the world, the StormCast system, more than a decade ago [6, 7]. Currently, there is a large body of work in this area. The lessons we learned from a series of deployed StormCast sensor networks in the Arctic resemble lessons being learned in similar systems today: push-based communication complements the pull-based structuring scheme.

Large players in the computer industry are also moving towards push-based infrastructures. Microsoft already has their Alert engine as vital in their .Net infrastructure. The proactive initiative by Intel [15] and CMU [14, 12] also suggests wide deployment of push-based infrastructures in the future. In essence, the idea is to develop proactive software that anticipates our needs and produces answers before they are required.

However, there are a number of fundamental problems with a push infrastructure. One is that of scale; how can one support both high recall and precision when a large number of users are involved. Filtering close to the data source is one approach to this, typically being supported by a system like Siena. Another one is that of precision, where spam detection is an example illustrating this problem.

We are advocating a structuring technique where remote servers are programmed (extended) with user code and user profiles for high expressiveness [2]. This way, data is filtered as close to the data source as possible. Next, we structure a spooling service close to the user, much the way an e-mail client is being deployed and used today. This spooling service has high bandwidth and connectivity to the Internet. The last structure in our architecture is a mobile device, be it a cellular or a PDA, where connectivity is intermittent and resources are scarce [11]. The focus on this paper is on the aspects related to this last jump, where we need to understand some of the problems and issues involved in a context-sensitive, precise push of information to a light, mobile client.

The rest of this paper is structured as follows. In section 2, we present our WAIF infrastructure. Next, in section 3, we describe our publish/subscribe system. In section 4 we present our simulation results. In section 5 we outline

directions for future work and in section 6 we conclude.

2 Wide Area Information Filtering

In WAIF[1], we conjecture that extensible computational resources will be embedded throughout the Internet [9]. Focus is on how this next generation Internet can be made programmable and extensible with personalized, mobile software. Our goal is to replace the old, time-consuming pull-based Internet, with a push-based one delivering high-precision information in a timely manner. Servers initiating information dissemination can be programmed by accepting entrant client code and data for execution [2].

For many years, computers systems were structured to accommodate many users per computer. In our pervasive computing model, though, we structure systems initially so that many computers serve a single user. For each user, we create a personal overlay network system (PONS). This ad hoc network serves a single user by filtering, fusing and pushing information based on personal preferences. In addition, a PONS provides a distributed personal file system and private compute resources.

A publish/subscribe system like, for instance, Siena achieves expressiveness by evaluating filtering predicates close to data sources. We advocate a similar, but even more extreme design principle for our pervasive systems. We actually program the servers by deploying client code at the data sources. This resembles how we used mobile agents successfully in our TACOMA mobile agent system [8, 10]. As such, a PONS can be created by extending servers with (mobile) client code. The infrastructure is much like in Oxygen [5], but we use this pervasive infrastructure embedded in our environment as a virtual remote computer network.

WAIF also supports automating environment mobility [11]. An environment is the applications and services being used at the source computer when a user decides to move on. For instance, if a user is editing a text document, is listening to some mp3-based music in the background, and has his mailer and browser active on the desktop, all these tasks should be brought along. Next, upon reaching a destination computer, the same applications should be recreated with state as upon departure.

3 Volume-Limiting Pub/Sub System

Our publish/subscribe system is *topic-based*, in that a user subscribes to a topic published by a specific publisher (e.g. weather updates from a news outlet). This is different from the more complex *content-based* systems in which a subscription is expressed as a query and any events submitted into the system that match the query are forwarded to the user, regardless of their origin. We believe that for an Internet-scale publish/subscribe system a topic-based approach is more appropriate. Firstly, query processing has higher routing overhead, which inhibits scalability. Secondly, the use of queries requires agreement on a query language, which is difficult to achieve on a global scale. Thirdly, it is not clear how to prevent malicious parties, such as unsolicited advertisers, from abusing a global content-based messaging system by submitting events designed to match queries of users who are not interested in their content.

Although topic subscriptions would seem to restrict a user to the publishers that it knows about, this need not be the case any more than web surfers being limited to the URLs that they know about. Just as web crawlers index content of websites and allow one to find that content with a query, a special kind of publisher – an *event notification indexer* – can parse all incoming events and notify subscribers whose queries match the content of the event, regardless of their origin. In this model, to discover new publishers, a user would subscribe to a ”customized” topic published by the indexer in response to the query (e.g. any event matching ”weather” & ”Norway”). The topic would exist only for as long as there are users interested in a query.

To obtain events for parsing, the indexer can both subscribe to topics of regular publishers and also allow publishers to submit events directly to it in the hope of reaching a wider audience. With likelihood, malicious parties will be submitting deceitful content to such an indexer, but its operators have a strong incentive to do what they can to detect and filter out such content, just as web search engines do today with some success. It is outside the scope of this paper, but certainly among key concerns in our overall design, to consider how to assist indexers in this filtering.

3.1 Publisher Interface

Just as many newspapers are thrown away unread, we expect that many event notifications in a global publish/subscribe system will never reach the eyes of a user who is unable to keep up with the flow of events on all the topics he subscribed to. When our time is scarce, we typically prioritize the tasks that need to be done and perform them in the order of priority, ignoring the ones that have lost relevance by the time they are at the top of the queue. This is our motivation for allowing a publisher to attach two attributes to every event notification:

- $\mathcal{R}$*ank* – Indication of a notification’s importance in relation to other notifications on its topic.
- $\mathcal{E}$*xpiration* – Time after which a notification is no longer relevant and should be discarded from the queue.

Although publishers are not required to use these two attributes and they cannot be forced to use them correctly,

[1] WAIF is a joint project between University of Tromsø, Cornell University, and UC San Diego. *http://www.waif.cs.uit.no*. Funded by Norwegian Resource Council (IKT-2010 Program).

it is in their interest to do so. If, for example, a publisher of a weather topic fails to attach a high priority to a storm warning resulting in that message being lost among other weather updates, a user would likely consider switching to a different publisher. Similarly, since a weather forecast is relevant only for a few days, it is most prudent to attach an appropriate expiration time to it, lest the user mistakenly rely on outdated information.

While helpful in overcoming information overload, both $\mathcal{R}$*ank* and $\mathcal{E}$*xpiration* are also useful for efficient utilization of hardware resources – primarily network bandwidth and battery power – as will be shown in Section 4.

3.2 Subscriber Interface

In the most general sense, a subscriber needs a way to specify *what* event notifications to receive and *when* to receive them. Although a subscription to a topic unambiguously identifies a set of notifications the user is interested in, when time is scarce the user may want to limit how many events from that topic are delivered. Our system offers two complementary volume limits for this purpose:

- $\mathcal{M}$*ax* – Deliver at most this many highest-ranked event notifications at a time. This is a *quantitative* limit.
- $\mathcal{T}$*hreshold* – Only event notifications with the rank at or above this threshold are deemed acceptable. This is a *qualitative* limit.

To illustrate, if one wanted to subscribe to the "Slashdot" topic, the two limits used in concert would allow one to request the highest-ranked stories and comments above threshold 4.5 (out of 5 maximum), but not more than 30 at a time. Provided that the stories do not expire too quickly, one can come back from a month-long vacation and read the most important bits from the past month. The volume limits are especially important for indexer queries because they aggregate content from multiple publishers and can result in an event arrival rate beyond the processing capacity of any user.

With most devices it is possible to either display event notifications *on-line*, as soon as they arrive, or accumulate them for *on-demand* display, when the user decides to check messages. This is a matter of user's preference, which is likely to depend both on their personality and on the nature of notifications on a topic. Certain topics, such as urgent traffic updates, are likely candidates for on-line display, whereas others do not warrant interrupting the user and are best served on-demand. Our publish/subscribe interface allows the device to specify for each subscription how the user wants to receive the notifications. Notifications for on-line topics are forwarded over the last hop as soon as the connection allows. For on-demand topics, which we expect to be the majority, we optimize the use of the last hop by taking the volume-limiting parameters, such as $\mathcal{R}$*ank* and $\mathcal{M}$*ax*, into account. This is the subject of Section 4.

There is a number of potential refinements to the user interface for a topic, beyond a simple selector between on-line and on-demand display. For example, one can envision a hybrid model in which an on-line topic goes quiet (e.g. during a meeting) or an on-demand topic interrupts (e.g. a tornado warning on a weather topic). On-line topics could be configured to only deliver events at specific points during the day with a certain $\mathcal{M}$*ax* number of messages per day. Furthermore, some devices, such as SMS-enabled mobile phones, may not be capable of on-demand display. All of these are issues of user interface design – it only matters to the publish/subscribe system whether event delivery is on-line or on-demand.

3.3 Device Constraints

As receivers of event notifications, mobile devices on one hand open doors to innovative location-aware services, but on the other hand introduce limitations in processing and communication capabilities.

From the perspective of our publish/subscribe system, changes in device's location or, more generally, its *context*, lead to *changes in the set of subscriptions* that are forwarded to the device. For example, a subscription to a topic for traffic updates could be contingent upon the device being located in the home city of the user. Perhaps more ambitiously, such subscription could be "parameterized" to receive traffic updates for whatever city the user happens to be in. In other words, upon a context update from a GPS-enabled mobile device, the spooling server detects a change in context and re-subscribes the user to the traffic updates topic with the new location as a parameter. Despite a potentially unlimited variety of such services, in our publish/subscribe system their functionality can be mapped into a simple context update handler, which performs standard *subscribe( )* and *unsubscribe( )* operations.

Hardware limitations of mobile devices introduce performance challenges for the last hop of a publish/subscribe system:

- Even when network access is free or unrated, limited *battery power* adds a cost to every network transfer and every computation on the mobile device by effectuating a limit on network messages beyond which the device is inoperable.
- When *storage capacity* becomes scarce, the device may need to delete old or low-ranked notifications to make room for new ones. This deletion means that the messages were forwarded needlessly, thus contributing to battery drain.
- Limited *network capacity* may at worst prevent the user from accessing the spooling server or at best make receipt of notifications tedious.

The first two limitations argue in favor of minimizing the number of notifications forwarded to the device. The third

one, though, makes a case for *prefetching* some notifications to the mobile device in anticipation of poor network capacity. Although complete lack of connectivity may soon be a thing of the past in most corners of the globe, insufficient bandwidth will be a problem for the foreseeable future, given the small antenna sizes of many wireless devices and the large distances between them and base stations. We evaluate this trade-off between effectiveness of prefetching and waste of resources in the next section.

4 Optimizing the Last Hop

To understand the dynamics of the last hop of a publish/subscribe system, we wrote a discrete-event simulator of a spooling server and a mobile device and had the simulator report detailed statistics on the number of messages exchanged between them. During initialization the simulator is populated with three types of events:

- *Notification Arrivals* – Events on a topic arrive a certain number of times per day (*event frequency*), according to a Poisson distribution. Optionally, a portion of the events can be configured to expire within *expiration time*, according to a desired distribution (exponential, uniform, normal).
- *User Reads* – The user checks for new messages a certain number of times per day chosen from a normal distribution (*user frequency*), which are distributed randomly throughout the 16- to 17-hour period, also slightly randomized, that the user is awake. At most $\mathcal{M}ax$ messages are read at a time, and only messages with rank above $\mathcal{T}hreshold$ are read.
- *Network Outages* – The network link goes down with a configurable frequency (Poisson distribution with high variance) and can be specified to last long enough for cumulative network downtime of anywhere between 0 to 100%. Note that we view periods of unacceptably slow network performance as outages, so high outage percentages can represent users who are mainly on a slow but functioning link.

Each experimental run lasted for one "virtual" year, resulting in anywhere between 150 and several thousand user reads, depending on the configuration. Since studying interactions among different devices or different topics and the question of overall link utilization are outside the scope of this work, it was sufficient to model a single client device subscribed to a single topic.

4.1 Prefetching

If the client device does not have the constraints on storage and battery life and if network connectivity is inexpensive – then it is appropriate to forward all incoming notifications to the device as soon as they arrive, regardless of whether they belong to a on-line or on-demand topic. In the latter case the device will queue up the notifications until the user requests them. This *on-line* forwarding policy ensures the *best possible service* in that all notifications are delivered as soon as they can be, given the network conditions. If the device *does* have the constraints but network connectivity is good – then it is appropriate to hold notifications for on-demand topics on the spooling server until the user requests them. Such *on-demand* forwarding policy minimizes resource consumption and delivers service that is just as good for as long as the spooling server is always reachable via a fast network.

As discussed in Section 3.3, many mobile devices face capacity constraints in combination with insufficient network connectivity. In such a setting, an on-line forwarding policy may waste resources and a pure on-demand policy may result in a lower quality of service. To make this precise, we define two *inefficiency* metrics: **wasted messages** are those that were sent to the device, but never read by the user; and **lost messages** are those that would have been read by the user under an on-line forwarding policy (i.e. the best possible service), but never reached the user under the policy in effect. A pure on-demand policy has no waste because only the messages explicitly requested are transferred to the device. An on-line forwarding policy has no losses, by definition.

In our publish/subscribe system, waste can arise in two ways, both related to the volume-limiting mechanisms that we introduced in Section 3. First, when the arrival rate of notifications on a topic (as modeled by *event frequency*) exceeds the rate at which the user can read them (product of *user frequency* and $\mathcal{M}ax$), some notifications never reach the user. We call this condition *overflow*. Second, notifications may expire before the user gets to them (as modeled by *expiration time*). When network connectivity is good, both overflow and expirations would cause the same messages to remain unread regardless of the forwarding policy. But when network outages are present, different forwarding policies lead to differences in the set of messages available to the user at any moment. For example, if a user checks an on-demand topic during an outage, no notifications will be available on the device. By the time the network is up, some notifications may expire, resulting in a loss as compared to the on-line forwarding policy.

To investigate the trade-off between waste and loss, we configured the simulator to execute two scenarios for each randomized set of discrete events. In an *on-line scenario*, notifications were queued up at the spooling server and forwarded to the device as soon as the network was available. This defined the best possible user experience under the circumstances – the baseline for computing loss – and also the maximum level of waste. In the *prefetching scenario*, we could experiment with a pure on-demand policy with no forwarding or a intermediate solution that forwarded a portion of notifications based on some algorithm. To compute loss, upon the completion of a run, the set of messages

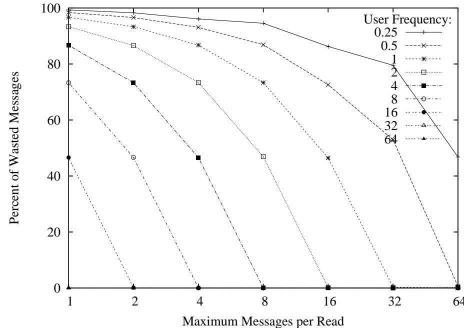

Figure 1. Wasted due to overflow at different values of $\mathcal{M}ax$ and *user frequency* (*event frequency* = 32)

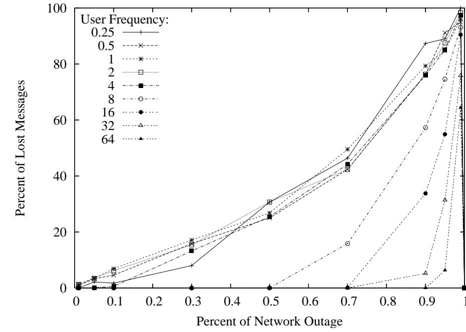

Figure 2. Loss due to overflow at different levels of network availability (*event frequency* = 32, $\mathcal{M}ax$ = 8)

read under a prefetching scenario was compared to the set of messages read under the on-line scenario.

4.2 Inefficiency Due to Overflow

We first turn to the waste and loss caused by overflow, or mismatch in event production and consumption rates. In this section we assume that event notifications do not expire. Figure 1 shows the percentage of waste (i.e. the fraction of unread forwarded messages) at different values of $\mathcal{M}ax$ and *user frequency*. Without loss of generality, *event frequency* was fixed at 32 notifications per day. The results are predictable: a user that reads a maximum of 32 messages once a day will not cause any waste, but if $\mathcal{M}ax$ is reduced to 4, then 88% of the forwarded messages are wasted. The shape of these curves is dictated by a simple formula:

$$\text{Waste \%} = 1 - \frac{\textit{user frequency} * \mathcal{M}ax}{\textit{event frequency}}$$

The point to observe is that users who do not check messages frequently and do not have the time to read much, risk burdening their mobile device with a high level of waste – thus shortening battery life and incurring extra connectivity costs – under an on-line forwarding policy. With pure on-demand forwarding the waste can be eliminated, but at the price of some loses. In Figure 2 we show what those loses are at different levels of network availability. As the portion of the time that the network is unavailable increases, the losses grow exponentially to the point just below 100% before dropping back to 0 at the point of no connectivity (on-line and on-demand policies are equally powerless at that point). Although we only show losses at $\mathcal{M}ax$ = 8, the shape of the curves with low *user frequency* is much the same with $\mathcal{M}ax$ anywhere between 1 and 64.

We experimented with two prefetching approaches in the attempt to find a compromise between waste and loss due to overload. Both approaches suppress forwarding of some notifications and both chose the highest-ranking notifications when they do forward. The intuition behind both was to adapt to the difference in production and consumption rates:

- In the *buffer-based* approach the spooling server ensures that the client device never has more than a fixed *prefetch limit* of notifications in its buffer. When the buffer is full, no forwarding occurs. Once the user has read some notifications, room in the buffer opens up and more notifications can be forwarded.

- In the *rate-based* approach the spooling server dynamically calculates the ratio between the event arrival rate and the read rate of the user. The ratio is used to forward messages with a certain frequency. For example, with a ratio of 0.2, forwarding takes place at the arrival of every 5th message.

We found that both approaches were good at reducing waste and loss to a few percentage points, but the buffer-based approach turned out to be more effective and, incidentally, simpler. In Figure 3 we show loss and waste with buffer-based prefetching under different prefetch limits. As the limit increases from 1 to 16, the loss percentage drops down very close to 0; as the limit goes beyond 64, the waste percentage starts growing exponentially before leveling off at 50%. (With *event frequency* = 32, $\mathcal{M}ax$ = 8, and *user frequency* = 2 we expect half of all messages to be wasted in the worst case.) Between 16 and 64, both waste and loss are below 1%. The low end of this range corresponds to the average number of messages a user reads per day.

Therefore, in cases of overflow, a buffer-based prefetching algorithm can be highly effective. To help determine the prefetch limit, a spooling server needs to keep track of several past user reads and calculate a moving average. It is safe to set the prefetch limit to twice that amount.

4.3 Inefficiency Due to Expirations

If a user fails to read an event notification before it expires, then forwarding of this notification is wasteful. If we

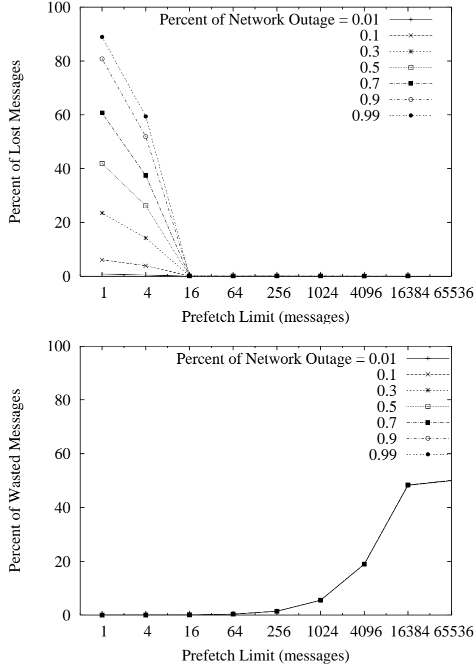

Figure 3. Loss and waste with buffer-based prefetching under different prefetch limits and levels of network availability (*event frequency* = 32, $\mathcal{M}ax$ = 8, *user frequency* = 2)

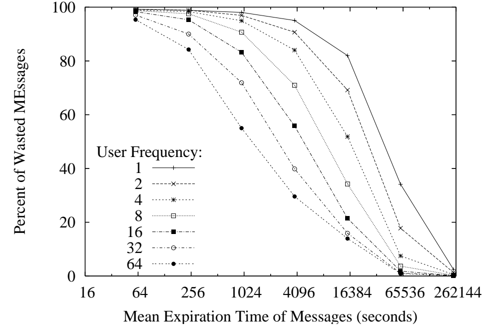

Figure 4. Waste due to expirations with different values of *user frequency* and expiration periods from 16 seconds to 3 days (*event frequency* = 32)

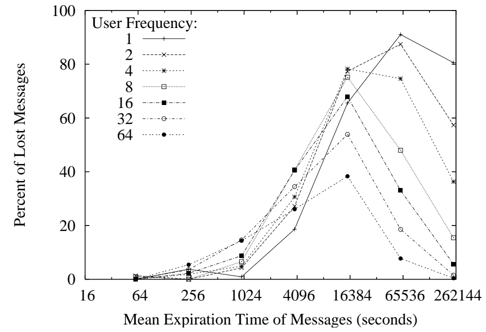

Figure 5. Loss due to expirations with different values of *user frequency* and expiration periods from 16 seconds to 3 days (*event frequency* = 32, network outage 95% of the time)

assume for now that the user is willing to process all notifications in the queue every time (i.e. $\mathcal{M}ax = \infty$), then the fraction of wasteful notifications is determined by *event frequency*, mean expiration time, and *user frequency*. Figure 4 shows this fraction at different values of *user frequency* and different expiration times. As can be expected, most short-lasting notifications expire before the user gets to them, but when the user checks messages with frequency below the expiration time, waste disappears.

In periods of network outage, expirations can also contribute to loss. When expiration time is short relative to *user frequency*, loss is negligible because most notifications expire before the user gets to them and consequently users have little to read during outages, regardless of the forwarding policy. As the expiration time increases, so does the percentage of loss, because notifications that expire during a network outage are potentially readable under on-line forwarding, but not under on-demand forwarding. Unlike the overflow case described in the previous section, where spooled notifications are in theory available to the user indefinitely, once a notification has expired, it has no chance of being fetched by a user after a network outage. Thus loses due to expirations are harder to minimize. Fortunately, as the expiration time grows further, notifications stick around long enough to be picked up with on-demand forwarding eventually, so the loss percentage starts dropping back down. This is illustrated is Figure 5, where loss is shown for different expiration times on a network that is down 95% of the time (with better network availability the height of these curves is lower).

If Figure 4 and Figure 5 are superimposed, however, it is evident that only at long expiration times – points at the right end of the scale and beyond – are both waste and loss sufficiently low. This is a clue to why devising a prefetching algorithm that accommodates notification expirations is difficult. Ideally, such an algorithm would only forward notifications that will not expire by the time of the next user read. Although expiration times are known, user behavior is unpredictable.[2]As a result, the best one can do is pick an *expiration threshold* and abstain from forwarding any notifications that expire over a shorter period of time.

We show how the system behaves with different values of this threshold in Figure 6. For these experiments we used

[2] It may be possible devise statistical methods that predict user behavior with sufficient accuracy, but this can only be done by a comprehensive study of real users and not by a simulation-based preliminary evaluation that we are conducting.

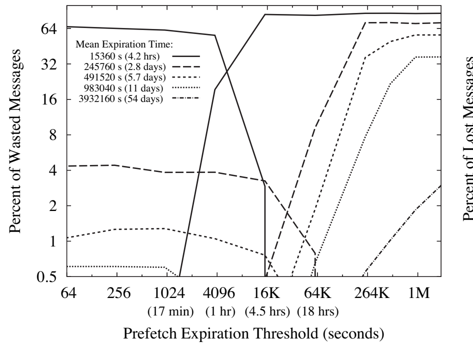

Figure 6. Waste (curves starting on the left) and loss (curves ending on the right) due to expirations, at different prefetch expiration thresholds (*event frequency* = 32, *user frequency* = 2, network outage 90% of the time). Note that **both** axis are logarithmic.

a challenging configuration: network downtime of 95%, *user frequency* of 2, and expiration times starting from 4.2 hours (this is one of the top points in Figure 5). The graph shows both waste and loss for five message expiration intervals. In each pair of curves, the waste is high with short expiration thresholds (because many frivolous messages get past the thresholds) but then sharply drops to zero. Conversely, the loss is nonexistent at first, but then climbs up to a high percentage and stabilizes there (too high of a threshold is as bad as no prefetching at all).

This plot reveals that there are configurations – roughly those in which message expiration time is in the vicinity of the intervals between user reads – that result in high levels of waste or loss no matter what threshold is chosen (e.g. the solid curve in the figure). In those cases, we believe it is most appropriate to let the user decide on the best trade-off between waste and loss. But when the system detects that average message expiration time is considerably higher than interval between user reads, it can set the expiration threshold automatically. Our experiments indicate that when the expiration time is an order of magnitude higher than the *time interval between reads*, as in the case of the 5.7-day curve in Figure 6, then there is a range of values where loss and waste are very small, visible as a gap between the descended waste curve and the ascending loss curve. That range includes the value of the interval between reads, making it the natural choice for the expiration threshold. For example, *user frequency* of 2 per day results in an average interval between reads of 8 hours – an expiration threshold value that is within the gap of the 5.7-day curve and all others with longer expiration times.

4.4 Inefficiency Due to Rank Changes

As an additional refinement of the volume-limiting mechanisms in our system, we allow the rank of an event notification to change over time. A positive change can be used to boost popularity of a useful notification based on recommendations from other users. A negative change can help retract notifications of malicious users *after* they reach mailboxes of subscribers, but before the notifications are read. Details of mechanisms for adjusting ranks are outside the scope of this article, but here we consider the implications of rank changes on waste and loss.

On the last hop the lowering of a rank in combination with prefetching can lead to overhead, since notifications may fall below the threshold after being prefetched (needlessly). This is similar to expirations, except now the expiration time is not known in advance, so there is no point in establishing an expiration threshold. We instead propose delaying all events on a topic that suffers from rank reductions for a period of time that is just long enough to eliminate most of them. Assuming that "bad" messages are detected quickly, this can be a useful option for allowing the user to trade off timeliness for quality. It is clear that this delay would be computed based on the expiration history of past events, but finding the right formula demands data from a deployed publish/subscribe system. Therefore, this is a topic that we plan to come back to in the future.

4.5 Unified Prefetching

To combine the ideas presented in this section and to present them more precisely, we show in Figure 7 a pseudocode for the spooling server in our system. The code consists of three main routines (shown in all caps) that are invoked in response to: arrival of notifications from outside, reads from the client device triggered by the user, and network status changes. These routines rely on many queues and several axillary routines (some of which were omitted for brevity) to pass messages along. We assume that a reader familiar with the set notation, which we use to concisely indicate operations on queues, will find most of the code self-explanatory. But several points demand explanation:

- Three main queues are used for temporarily storing events: the *outgoing queue* is filled with events that must be forwarded as soon as possible; the *prefetch queue* contains events that passed expiration checks and the delay stage, meaning they are OK to prefetch if there is room on the client; and the *holding queue* is for events deemed unacceptable for prefetching due to their short expiration time. Note that all three queues are tapped for events when the user requests a read.

- *schedule( )* is used to invoke a routine in the future, much like a signal handler does. It is used for both expiring notifications and delaying them, as described in the previous section.

- *READ( )* receives from the client three parameters: *N*, the number of items user wants to read; *queue_size*, the number of messages currently in the queue on the

client device, *including* the N that it is requesting; and *client_events*, a set of anywhere between 0 and N event identifiers that are the highest-ranked events on the client device (with effective prefetching this set may be better than anything available in queues on the server, making any transfer unnecessary). Essentially, a read is not a request for more data, but a request for "better" data if it exists.

- In addition to omitting implementations of certain routines, such as *moving_average( )*, we also did not include "garbage collection" that would have to operate in the background as certain queues (e.g. *topic.history*) grow without any bounds.

5 Future Work

Two interesting problems resulting from the marriage of publish/subscribe systems and mobile devices have escaped our attention for now: In the future we want to study the implications of cooperation among multiple devices belonging to one user. Their interaction, perhaps with the aid of an adhoc network, has the potential for reducing both loss and waste by allowing one device to use the cache of another. Also, to avoid making the spooling server a single point of failure, we want to consider approaches to replicating it.

On the practical front, we are in the process of implementing the ideas described in this paper in a real system. We look forward to comparing results of simulations to the behavior of real publishers and subscribers under real network conditions.

6 Concluding remarks

We conjecture a more push-based Internet in the future. Such a network has high potential for supporting timely alerts and the like, but is also burdened with a serious problem: spam and information overload. We target this problem by filtering data as close as possible to data sources and by building personal overlay networks fusing data towards the client.

The focus in this paper has been to study issues involved in the last communication path leading into the mobile client device. We argue that precision must be taken largely into account, and we suggest a context-sensitive precision mechanism to support this. This is our volume-limiting mechanism, which also has potential applicability in pervasive environments like, for instance, in GPRS and UMTS based cellular communication technology.

We have presented simulation results from this scenario, and our volume-limiting mechanism takes findings here into account. Currently, we have implemented our volume-limiting mechanism in one of our WAIF applications, a peer-to-peer recommender system where user recommendations are being pushed and ranked among (topic) peers. Also, we built aspects of it into our first WAIF web browser, where we learned that we needed a more holistic simulation as done in this paper to better understand the issues involved.

We are targeting to deploy our precision scheme in a public WAIF PONS soon operational for students at the northernmost university in the world. Naturally, this WAIF based push infrastructure has severe StormCast weather alerts as one of the services possible to subscribe to. Other personalized services or correlations of services include, for instance, bus routes, alerts when a certain topic is indexed by the search engine Google, Television program alerts, local concert subscriptions, news subscriptions based on mediator services, and a push-based auction service.

References

[1] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In proceedings of *the 18th ACM Symposium on Principles of Distributed Computing,* 53–61, May 1999.

[2] Ingar M. Arntzen and Dag Johansen. A programmable structure for pervasive computing. In proceedings of *IEEE International Conference on Pervasive Services,* 19–23, Beirut, Lebanon, July 2004.

[3] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a Wide-area event notification service. *ACM Transactions on Computer Systems,* 19(3):332–383, 2001.

[4] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *Journal on Selected Areas in Communications (JSAC),* 20(8):100–110, 2002.

[5] Michael L. Dertouzos. The future of computing. *Scientific American,* 281(2):36–39, August 1999.

[6] Gunnar Hartvigsen and D. Johansen. StormCast – A distributed artificial intelligence application for severe storm forecasting. *Distributed Computer Control Systems*, Pergamon Press, Oxford, England, 1989.

[7] Dag Johansen and Gunnar Hartvigsen. Convenient abstractions in StormCast applications. In proceedings of *ACM SIGOPS European Workshop,* pages 11-16, Dagstuhl Castle, Germany, September 1994.

[8] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In proceedings of *the 5th Workshop on Hot Topics in Operating Systems,* 42–45, Orcas Island, WA, May 1995.

[9] Dag Johansen, Robbert van Renesse, and Fred Schneider. WAIF: Web of asynchronous information filters. In *Lecture Notes in Computer Science: "Future Directions in Distributed Computing,"* v. 2584. Springer-Verlag, Heidelberg, April 2003.

[10] Dag Johansen, Kåre Jørgen Lauvset, Robbert van Renesse, Fred B. Schneider, Nils P. Sudmann, and Kjetil Jacobsen. A

TACOMA retrospective. *Software Practice and Experience,* 605–619, 2002.

[11] Dag Johansen, Håvard Johansen, Robbert van Renesse. Environment mobility – Moving the desktop around. In proceedings of *the 2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing,* 18–22, Canada, October 2004.

[12] M. Kozuch and Mahadev Satyanarayanan. Internet Suspend/Resume. In proceedings of *the 4th IEEE Workshop on Mobile Computing Systems and Applications,* Calicoon, NY, June 2002.

[13] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications,* 10–17, August 2001.

[14] Joao Pedro Sousa and David Garlan. Aura: An architectural framework for user mobility in ubiquitous computing environments. In proceedings of *the 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance,* 29–43, 2002.

[15] David Tennenhouse. Embedding the Internet: Proactive computing. *Communications of the ACM,* 43(5):43–50, 2000.


```
var topic // pointer to the current topic
var q ← topic.queues // shortcut to queues for the topic
NOTIFICATION (event) // called when new outside event arrives
    // if rank has been lowered below the threshold
    if event.rank < topic.rank_threshold ∧ event ∈ topic.history then
        q.holding ← q.holding \ event; // remove from holding queue
        q.prefetch ← q.prefetch \ event; // ditto for prefetch queue

        // if it has been forwarded to the client
        if event ∈ topic.forwarded then
            q.outgoing ← q.outgoing ∪ event; // tell client of rank drop
        else
            q.outgoing ← q.outgoing \ event; // don't bother client

    // if rank is above the threshold
    else if event.rank ≥ topic.rank_threshold then
        if topic.type = "on-line" then
            q.outgoing ← q.outgoing ∪ event; // send to client ASAP
        else if topic.type = "on-demand" then
            if event.expires > 0 then
                topic.exp_times ← topic.exp_times ∪ event.expires;
                topic.avg_exp ← moving_average(topic.exp_times);
                schedule(&expiration_timeout, event.expires, event);

            if event.expires < topic.expiration_threshold then
                q.holding ← q.holding ∪ event;
            else if topic.delay > 0 then
                schedule(&delay_timeout, topic.delay, event); // delay it
            else
                q.prefetch ← q.prefetch ∪ event;

    topic.history ← topic.history ∪ event; // record all events
    topic.delay ← delay_function(topic.history); // recompute delay
    try_forwarding ( );


READ ( N, queue_size, client_events ) // called when a user reads
    topic.old_reads ← topic.old_reads ∪ N; // remember N
    topic.prefetch_limit ← moving_average(topic.old_reads) * 2;
    topic.old_times ← topic.old_times ∪ gettimeofday(); // timestamp
    time_between_reads ← moving_average_difference(topic.old_times);
    topic.expiration_threshold ← time_between_reads;
    topic.queue_size ← queue_size;

    best ← get_highest_ranked(N, q.outgoing ∪ q.prefetch ∪ q.holding);
    difference ← get_highest_ranked(N, best ∪ client_events) \ client_events;
    q.outgoing ← q.outgoing ∪ difference;
    try_forwarding ( );


NETWORK ( status ) // called when the status of the connection changes
    topic.network ← status;
    if status = "up" then
        try_forwarding ( );


try_forwarding ( )
    if topic.network ≠ "up" then
        return;
    // first empty the outgoing queue
    for each event ∈ topic.outgoing do do_forward(event);

    // then see if anything should be prefetched
    while topic.queue_size < topic.prefetch_limit ∧ q.prefetch ≠ ∅ do
        event ← get_highest_ranked(1, q.prefetch);
        do_forward(event);

do_forward ( event )
    forward(event);
    topic.queue_size ← topic.queue_size + 1;
    topic.forwarded ← topic.forwarded ∪ event;

expiration_timeout ( event ) // remove from all queues
    q.holding ← q.holding \ event;
    q.prefetch ← q.prefetch \ event;
    q.outgoing ← q.outgoing \ event;


delay_timeout ( event ) // after a delay the notification can be prefetched
    q.prefetch ← q.prefetch ∪ event;
    try_forwarding ( );
```

Figure 7. Pseudo-code for the prefetching algorithm used on the spooling server.