The Last Hop of Global Notification Delivery to Mobile Users: Accommodating Volume Limits 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, or a mobile phone. 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, the user's preferences and location may limit ***what*** *should be forwarded to the device. Furthermore, technological constraints, such as network bandwidth availability and battery power, suggest that the decision* ***when*** *to forward messages is also important for maximizing the quality of service. We describe a new publish/subscribe system with volume-limiting mechanisms and explain how the user's 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

The marriage of publish/subscribe messaging and mobile networking [4, 2] is a mutually beneficial one: On one hand, portable devices are natural destinations for time-sensitive notifications such as traffic, weather, or stock updates. On the other hand, the push-based communication model is well-suited for reconciling the limitations of these devices: intermittent connectivity, rated network access, finite battery life, etc. However, this relationship is beneficial *only* when most of the messages sent to the mobile device are useful to the subscriber. Pushing of unwanted traffic can turn the mobile device into a nuisance that produces distracting notifications and wastes resources on superfluous data transfer and processing. Concern over the transmission of what we call *vain* data (notifications that are not useful) underlies this work.

At first sight, the subscription query is solely responsible for ensuring that only wanted notifications arrive. But this overlooks the dynamic nature of notifications: if the rate at which they arrive – and that is rarely something that a user can predict when subscribing – exceeds the rate at which the user can process them, some notifications will inevitably be left unconsidered. Similar to how many newspapers are thrown away unread, we expect that many event notifications in a global pub/sub system will never reach the eyes of a user who is unable to keep up with the flow. When our time is scarce, we weigh the importance of tasks 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 pile. If the piles get too big, we indiscriminately chuck them. This is our motivation for augmenting the classic pub/sub interface – namely, the *publish ( )* and *subscribe ( )* methods – with *volume-limiting* parameters, which allow the subscriber to set qualitative and quantitative thresholds for notifications which can help reduce vain traffic when publishers properly annotate it.

The key contribution of this paper is its evaluation of how volume-limiting mechanisms in a pub/sub system affect the vain traffic on the "last hop" – the link between the wired infrastructure and the portable device – where wasteful transfers can be particularly expensive. We consider the trade-off between waste of resources (chiefly network bandwidth), and quality of service, which we define in terms of lost or delayed notifications. We propose a simple algorithm for keeping the waste low and the quality high. Although we explore these issues in the context of a new pub/sub system, this work is complementary to the bulk of pub/sub research and particularly to the projects that consider subscribers with mobile devices. Among those, our use of a proxy for relaying messages between fixed infrastructure and mobile nodes, makes us closer to the systems that also use proxies – Elvin [9], JEDI [3], Siena [1], and Pronto [12], among others – rather than the systems that do not: STEAM [7] and MoPS [8].

Since proxies are common in contemporary commercial messaging systems, such as *iBus//Mobile Gateway* from Softwired and Broadbeam's *ExpressQ*, these ideas are applicable in the industry, as well.

2. Volume-Limiting Publish/Subscribe

In this section we present the publish/subscribe system that we are building as a part of the WAIF [6] infrastructure.

Our notification infrastructure is *topic-based*, in that subscriptions identify a topic from a specific publisher (e.g. weather updates from a news outlet). This is different from the more flexible and 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. Given our goals of building an Internet-scale pub/sub system, we consider a topic-based approach more appropriate. Firstly, query processing has higher routing overhead, which inhibits scalability. Secondly, deployment of such a system 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.

Having said that, we share with most prior systems, including content-based ones, the approach to mobility: we interface all devices via a *proxy* that can collect notifications on behalf of the client when the latter is disconnected. The algorithm for relaying notifications from the proxy to the user, which is developed in Section 3, is applicable to topic- and content-based systems alike. The fundamental differences lie inside the routing substrate, discussion of which is outside the scope of this paper. Here we can treat the network infrastructure as a black box that offers the standard pub/sub operations: advertising (or withdrawing) topics, publishing notifications, and subscribing to (or unsubscribing from) them. We are agnostic with respect to how the routing is implemented, we only require that notifications and subscription notices are each able to carry a pair of additional parameters, described next.

2.1. Publisher Interface

A subscriber overwhelmed with vain traffic has no choice but to discard some of the messages. To do better than random dropping, some way of prioritizing notifications is needed. To that end, we allow a publisher to attach two *volume-limiting 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, 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 to humans 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 3.

2.2. 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 subscribing 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-limiting thresholds* 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 thresholds 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.

With most devices it is possible to either display event notifications *on-line*, i.e. as soon as they arrive, or accumulate them for *on-demand* display, when the user decides to check messages. This is a matter of the user's preference, which is likely to depend both on 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. The interface between the proxy and the device should allow 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, $\mathcal{R}$*ank* and $\mathcal{M}$*ax*, into account.

There are 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 proxy whether event delivery is on-line or on-demand.

2.3. 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 pub/sub system, changes in the location of a device 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 proxy 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 pub/sub 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 pub/sub 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 low-ranked unread messages 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 proxy 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.

3. Optimizing the Last Hop

To understand the dynamics of the last hop of a pub/sub system, we wrote a discrete-event simulator of a proxy attached to 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.

3.1. Prefetching

If the client device does not have 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 an 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 constraints but network connectivity is good, then it is appropriate to hold notifications for on-demand topics on the proxy until the user requests them. Such *on-demand* forwarding policy minimizes resource consumption and delivers equally good service for as long as the proxy is always reachable via a fast network.

As discussed in Section 2.3, 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 pub/sub system, waste can arise in two ways, both related to the volume-limiting parameters that we introduced in Section 2. Firstly, 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*. Secondly, 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 that would not have happened with 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 proxy and forwarded to the device as soon as the network was available. This is our definition for the best possible quality of service under the circumstances, which serves as the baseline for computing loss and as the cap for the maximum level of waste. In the *prefetching scenario*, we could experiment with a pure on-demand policy with no forwarding or an intermediate solution that forwards a portion of notifications based on some algorithm. To compute loss, upon the completion of a run, the set of messages read under a prefetching scenario was compared to the set of messages read under the on-line scenario.

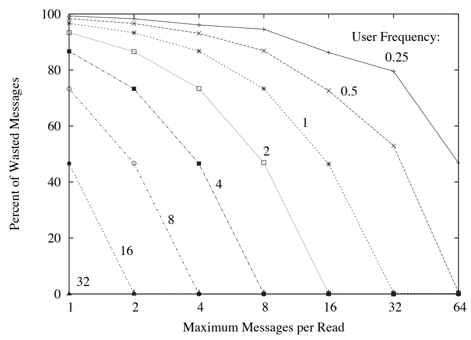

Figure 1. Waste due to overflow at different values of $\mathcal{M}ax$ and *user frequency* (*event frequency* = 32)

3.2. 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 shapes of these curves can be approximated very well 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 losses. In Figure 2 we show what those losses 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 work by suppressing of the forwarding of some notifications and both choose the highest-ranking notifications when they do forward. The intuition behind them was to adapt to the difference in production and consumption rates:

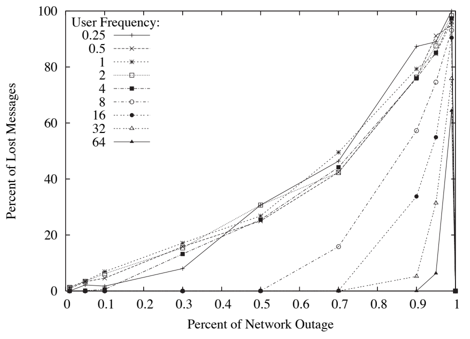

Figure 2. Loss due to overflow at different levels of network availability (*event frequency* = 32, $\mathcal{M}ax$ = 8)

- In the *buffer-based* approach the proxy 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 proxy 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 proxy 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.

3.3. 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 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

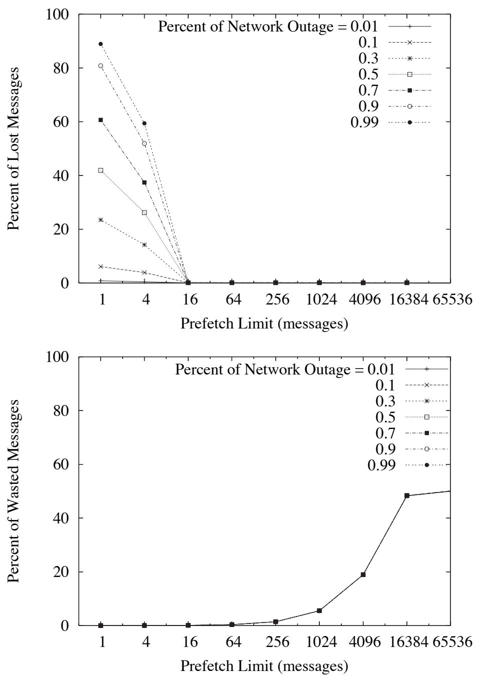

Figure 3. Loss and waste with buffer-based prefetching under different prefetch limits and levels of network availability (*event frequency* = 32, $\mathcal{M}ax$ = 8, *user frequency* = 2)

shows this fraction at different values of *user frequency* and different expiration times. The lifetime of notifications and, in fact, whether expirations are used at all, is very application specific. So, we included a wide range of possibilities in our experiments: from notifications that last 1 minute to those that practically never expire. As can be expected, most short-lasting notifications typically 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 losses due to expirations are harder to minimize. Fortunately, as the expiration time increases, notifications stick around long enough to be picked up eventually with on-demand forwarding, so the loss percentage starts dropping

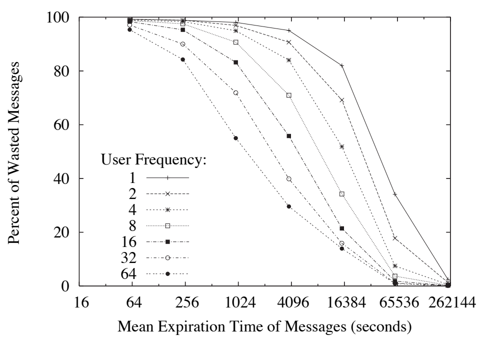

Figure 4. Waste due to expirations with different values of *user frequency* and expiration periods from 16 seconds to 3 days (*event frequency* = 32)

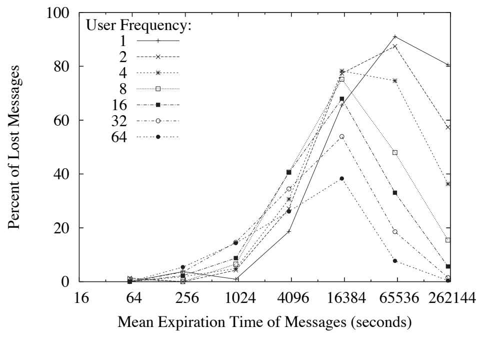

Figure 5. Loss due to expirations with different values of *user frequency* and expiration periods from 16 seconds to 3 days (*event frequency* = 32, network outage 95% of the time)

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.[1] 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

[1] It may be possible to 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 we are conducting.

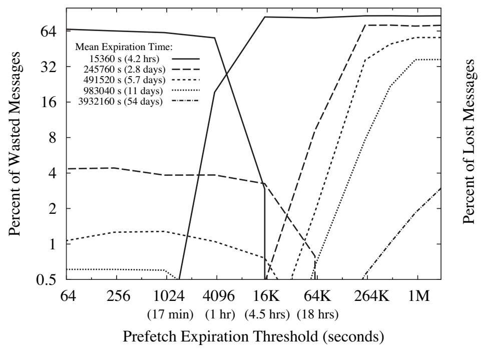

Figure 6. Waste (curves starting on the left) and loss (curves ending on the right) due to expirations, at different prefetch expiration thresholds (*event frequency* = 32, *user frequency* = 2, network outage 90% of the time). Note that **both** axis are logarithmic.

of this threshold in Figure 6. For these experiments we used a challenging configuration: network downtime of 90%, *user frequency* of 2/day, and a set of expiration times from 4.2 hours (this is one of the top points in Figure 5) and up. 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 tradeoff between waste and loss. But when the system detects that average message expiration time is considerably higher than the average 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/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.

3.4. Inefficiency Due to Rank Changes

As an additional refinement of the volume-limiting parameters in our system, we allow the rank of an event notification to change over time. A positive change can be used to boost the popularity of a useful notification based on recommendations from other users. A negative change can help retract the notifications of malicious users *after* they reach the mailboxes of subscribers, but before the messages are read. Essentially, rank changes allow dynamic tuning of volume limiting, further improving its efficacy. 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 in this case the expiration time is not known in advance, so there is no point in establishing an expiration threshold. We instead propose that if a topic sees rank reductions, all events may be optionally delayed for a period of time long enough to separate the wheat from the chaff. 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 pub/sub system. Therefore, this is a topic that we plan to come back to in the future.

3.5. 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 proxy 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 auxiliary 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 okay 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 the 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 bounds.

4. Conclusions

We conclude that if publishers attach volume-limiting attributes ($\mathcal{R}$*ank* and $\mathcal{E}$*xpiration*) to notifications and if subscribers or devices acting on their behalf specify thresholds ($\mathcal{M}$*ax* and $\mathcal{T}$*hreshold*), then by using the prefetching algorithm described above, vain traffic on the last hop can be kept to a few percentage points of the overall traffic while the quality of service remains high. In particular, mismatch in production and consumption rates can be mitigated with a simple buffering scheme. The overhead due to message expirations, if the expiration times are sufficiently long in relation to *user frequency*, can also be minimized by not forwarding notifications that expire in less time than the average *user frequency*.

Two problems have escaped our attention for now: In the future we want to look into cooperation among multiple devices belonging to one user. Their interaction, perhaps with the aid of an ad-hoc 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 proxy a single point of failure, we will 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 that we are building at the University of Tromsø. We especially look forward to comparing results of simulations to the behavior of real publishers and subscribers under real network conditions. Scalability of proxies is of interest, too.

Having built the first academic sensor network in the world, the StormCast system [5], over 15 years ago, we acquired appreciation for both pervasive technologies and for push-based communication. We envision the future Internet as a more proactive and more personalized network, with custom filters running close to data sources, pushing data towards users in timely notifications. Many researchers, particularly those at Intel [11] and CMU [10], share our vision. Before it can become a reality, however, many technical problems have to be resolved. The problem of flooding a user of such an infrastructure with irrelevant data is high at the top of our list. By analyzing vain traffic in this paper, we make a step towards solving it. Within our WAIF[2]project, we are working on many others.

References

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

[2] G. Cugola and H.-A. Jacobsen. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33, 2002.

[3] G. Cugola, E. D. 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):827–850, Sep 2001.

[4] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile enviroment. In *Proc. 2nd ACM Intl Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, pages 27–34, Santa Barbara, California, USA, May 2001.

[5] D. Johansen and G. Hartvigsen. Convenient abstractions in StormCast applications. In *ACM SIGOPS European Workshop*, pages 11–16, Dagstuhl Castle, Germany, Sep 1994.

[6] D. Johansen, R. van Renesse, and F. Schneider. WAIF: Web of asynchronous information filters. *Lecture Notes in Computer Science: Future Directions in Distributed Computing*, 2584, Apr 2003.

[7] R. Meier and V. Cahill. STEAM: event-based middleware for wireless ad hoc networks. In *Proc. 22nd Intl Conf. on Distributed Computing Systems, 3rd Intl Workshop on Distributed Event-Based Systems (DEBS)*, pages 639–644, Vienna, Austria, Jul 2002.

[8] I. Podnar and I. Lovrek. Supporting mobility with persistent notifications in publish/subscribe systems. In *3rd Intl Workshop on Distributed Event-Based Systems (DEBS)*, Edinburgh, Scotland, UK, May 2004.

[9] B. Segall and D. Arnold. Elvin has left the building: a publish/subscribe notification service with quenching. In *Proc. 1997 Australian UNIX Users Group Technical Conf.*, 1997.

[10] J. P. Sousa and D. Garlan. Aura: An architectural framework for user mobility in ubiquitous computing environments. In *Proc. 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, pages 29–43, 2002.

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

[12] E. Yoneki and J. Bacon. Gateway: a message hub with store-and-forward messaging in mobile networks. In *Proc. 23rd IEEE Intl Conf. Distributed Computing Systems, Workshop on Mobile Computing Middleware*, pages 348–353, Rhode Island, USA, May 2003.

[2]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).

```
var topic // pointer to the current topic
var q $\leftarrow$ 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 $event.rank < topic.rank\_threshold \wedge event \in topic.history$ then
        q.holding $\leftarrow$ q.holding $\setminus$ event; // remove from holding queue
        q.prefetch $\leftarrow$ q.prefetch $\setminus$ event; // ditto for prefetch queue

        // if it has been forwarded to the client
        if $event \in topic.forwarded$ then
            q.outgoing $\leftarrow$ q.outgoing $\cup$ event; // tell client of rank drop
        else
            q.outgoing $\leftarrow$ q.outgoing $\setminus$ event; // don't bother client

    // if rank is above the threshold
    else if $event.rank \geq topic.rank\_threshold$ then
        if topic.type = "on-line" then
            q.outgoing $\leftarrow$ q.outgoing $\cup$ event; // send to client ASAP
        else if topic.type = "on-demand" then
            if $event.expires > 0$ then
                topic.exp_times $\leftarrow$ topic.exp_times $\cup$ event.expires;
                topic.avg_exp $\leftarrow$ moving_average(topic.exp_times);
                schedule(&expiration_timeout, event.expires, event);

            if $event.expires < topic.expiration\_threshold$ then
                q.holding $\leftarrow$ q.holding $\cup$ event;
            else if $topic.delay > 0$ then
                schedule(&delay_timeout, topic.delay, event); // delay it
            else
                q.prefetch $\leftarrow$ q.prefetch $\cup$ event;
    topic.history $\leftarrow$ topic.history $\cup$ event; // record all events
    topic.delay $\leftarrow$ delay_function(topic.history); // recompute delay
    try_forwarding ( );

READ ( N, queue_size, client_events ) // called when a user reads
    topic.old_reads $\leftarrow$ topic.old_reads $\cup$ N; // remember N
    topic.prefetch_limit $\leftarrow$ moving_average(topic.old_reads) * 2;
    topic.old_times $\leftarrow$ topic.old_times $\cup$ gettimeofday(); // timestamp
    time_between_reads $\leftarrow$ moving_average_difference(topic.old_times);
    topic.expiration_threshold $\leftarrow$ time_between_reads;
    topic.queue_size $\leftarrow$ queue_size;
    best $\leftarrow$ get_highest_ranked(N, q.outgoing $\cup$ q.prefetch $\cup$ q.holding);
    difference $\leftarrow$ get_highest_ranked(N, best $\cup$ client_events) $\setminus$ client_events;
    q.outgoing $\leftarrow$ q.outgoing $\cup$ difference;
    try_forwarding ( );

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

try_forwarding ( )
    if $topic.network \neq$ "up" then
        return;
    // first empty the outgoing queue
    for each $event \in topic.outgoing$ do do_forward(event);
    // then see if anything should be prefetched
    while $topic.queue\_size < topic.prefetch\_limit \wedge q.prefetch \neq \emptyset$ do
        event $\leftarrow$ get_highest_ranked(1, q.prefetch);
        do_forward(event);

do_forward ( event )
    forward(event);
    topic.queue_size $\leftarrow$ topic.queue_size $+ 1$;
    topic.forwarded $\leftarrow$ topic.forwarded $\cup$ event;

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

delay_timeout ( event ) // after a delay the notification can be prefetched
    q.prefetch $\leftarrow$ q.prefetch $\cup$ event;
    try_forwarding ( );
```

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