Context-aware Efficient Message Propagation

Pedro Alves
INESC-ID / Technical University of Lisbon / Opensoft
Rua Joshua Benoliel, 1, 4C, 1250 Lisboa
pedro.alves@opensoft.pt

Paulo Ferreira
INESC-ID / IST / Technical University of Lisbon
Rua Alves Redol, 9, 1000 Lisboa
paulo.ferreira@inesc-id.pt

ABSTRACT

Applications such as Facebook, Twitter and Foursquare brought the massification of personal short messages, distributed in (soft) real-time on the Internet to a large number of users. These messages are complemented with rich contextual information such as the identity, time and location of the person sending the message.

Such contextual messages raise serious concerns in terms of scalability and delivery delay; this results not only from their huge number but also because the set of user recipients changes for each message (as their interests continuously change), preventing the use of well-know solutions such as pub-sub and multicast trees. This leads to the use of non-scalable broadcast based solutions or point-to-point messaging.

We propose Radiator, a middleware to assist application programmers implementing efficient context propagation mechanisms on their applications. Based on each user current context, Radiator continuously adapts each message propagation path and delivery delay, making an efficient use of network bandwidth, arguably the biggest bottleneck in the deployment of large-scale context propagation systems.

Our experimental results demonstrate a 20x reduction on consumed bandwidth without affecting the real-time usefulness of the propagated messages.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communication Applications

General Terms

Algorithms, Human Factors, Measurement, Performance

Keywords

Context propagation, scalability, pub-sub

1. INTRODUCTION

We are watching a radical change in the type of packets that travel on the Internet. Facebook and Twitter (among others) brought the massification of personal short messages or posts, distributed in (soft) real-time to a potentially large number of users.[1] These messages are complemented with rich contextual information such as the identity, time and location of the person sending the message (following the context model devised more than a decade ago [5] among the CSCW community).

However, context propagation creates unique challenges in the realm of distributed systems [2] (in addition to their huge number) mostly related to its highly dynamic nature. To better understand how dynamic context can be, consider the case of capturing the geolocation of a moving person or the speed at which he is moving. To achieve a reasonable level of accuracy, the system must capture and propagate this information very frequently, probably at least once per minute. Since these systems usually have hundreds of thousands if not millions of users, we are talking about a huge volume of information being sent to the server (assuming a centralized topology which is the case in the vast majority of the commercial applications on this area). Moreover, the server must then be able to propagate this context to whoever may be interested. The problem lies on the dynamics of those interests. For example, if the user is interested in receiving information about friends nearby, there will be a matching rule between his location and the location of his friends. However, if he's moving, and his friends are also moving, the system has to continuously change that matching rule.

For this reason (the dynamics of context), traditional publish-subscribe approaches are unfeasible since they assume a relatively fixed set of matching rules. On these systems, users subscribe to topics (subject-based systems) or predicates (content-based systems) [6]. Then, users feed content into the system (publish) and the system distributes events matching subscribers interest with publisher content. Therefore, developing a "friends nearby" application using publish-subscribe requires each client to continuously change its interests. In fact, every time the user moves, the client application has to send three messages when just one should suffice: (1) publish the current location; (2) unsubscribe from the previous location, and (3) subscribe to the current location. This leads to wasted resources and poor scalability.

Application-level multicast tree approaches [4] fall on the same problem: they assume that distribution rules don't change very frequently. Although they still work on these conditions, the resources wasted by continuously rebuilding the multicast trees lead to poor scalability. For example, the Scribe system [3] relies on the following message types: JOIN, CREATE, LEAVE and MULTICAST. It is easy to see the resemblance with publish-subscribe messages - changing the matching rules implies the propagation of a LEAVE message, a JOIN message and a MULTICAST message (the latter

[1]As of 2012, there are 175 million tweets (twitter messages) being sent per day and some of these messages are distributed to over 19 million users (the number of followers of Lady Gaga) - http://bit.ly/zOiX8k

alone should be enough to convey all the information we need, e.g., the new location, in the "friends nearby" application).

In summary, context-aware applications have the potential to transmit a huge number of messages in a highly dynamic environment therefore raising hard challenges regarding scalability. We argue that current approaches such as publish-subscribe [9, 12], multicast trees [4] or gossip-based protocols [1] are not adequate to these dynamics, because they assume the matching rules are fixed or change infrequently (therefore changes are too expensive). Also, since such classic approaches don't know how to extract semantic meaning from the exchanged messages, they can't decide what is the most efficient way to distribute those messages - such a burden becomes the application programmer responsibility.

We propose an adaptable middleware where context propagation is controlled by functions that, given the context of the recipient, dictate in which conditions should a given context message be propagated. These functions are, by nature, **dynamic matching rules** which change automatically if the involved clients change their context. Moreover, the retained messages (messages for which the functions have decided that they should not be propagated immediately) are aggregated into single compressed messages that can yield a substantial reduction on the consumed network bandwidth. For this reason, these functions are called *aggregability* functions because they tell whether a message should be aggregated or not, and to which level the aggregation should occur.

It is important to note that the aggregability functions (and therefore the propagation timing) are not only dependent on the message itself but also on the current context of both the sender and the receiver. This is a crucial difference over other generic message propagation approaches: since we know that messages contain the contexts of their senders, we have more information to make decisions about their propagation.

In short, this paper makes the following contributions:

- We present a model for context-aware applications that relies on the concept of *aggregability*, a function that tells how much aggregated a message can be before being propagated. This function takes into account the current context of both the sender and the receiver, making a more efficient use of the network bandwidth and significantly improving the system scalability.
- We present a hybrid dynamic propagation mechanism, where a server decides if a message should be retained or transmitted (based on the result of the *aggregability* function) and clients communicate directly between them to propagate it.
- We implement and evaluate the scalability of Radiator, a pluggable local middleware and a server that support the above mentioned model, i.e. it supports the hybrid propagation mechanism while still abstracting away from the application programmer the underlying communication and context management.

In the remainder of the paper, we start by describing Radiator's context aggregation model. In Section 3, we present Radiator's architecture and Section 4 presents experimental results. Finally we draw some conclusions.

2. CONTEXT AGGREGATION MODEL

In this section, we start by explaining the concept of context aggregation and then we describe in detail the model supported by Radiator.

2.1 What is context aggregation?

To help understand the concept of context aggregation, consider a "popular spots" application example. This application shows the most popular spots (e.g., pubs, restaurants, discos) nearby the user's current location, where a popular spot is a place where a large number of users is currently located. The context of those users (in this case, the location) must be propagated to the others but it can be grouped before being propagated. The user doesn't care about individual context updates since, in this case, he only wants to know popular spots, not who's in there. So, instead of propagating N messages, each one saying "user U is now at location L", we can wait until there are N users at location L and only then propagate a **single** message saying "users $U_1..U_N$ are now at location L" or (in case privacy is an issue) "N users are now at location L". In other words, we are delaying the propagation of the first *N - 1* users' location to improve the efficiency of the system, hence the concept of delayed propagation. Note that the delay doesn't break user expectations because, for some contextual information, he doesn't mind receiving it with delay. For example, a popular spot doesn't become one in seconds and it certainly doesn't stop being one in seconds, so a lag of some minutes is perfectly acceptable between the time when a spot becomes popular and the time a user is informed.

However, if a friend is in one of those spots, the user may no longer tolerate a delay - he may want to receive that information as soon as possible. So, the model has to accommodate multiple delay levels, depending on the user's context (the user's friends are part of his context).

All the messages that are not immediately propagated are said to be retained. The fact that these messages are retained allows the system to aggregate them in the most efficient way possible, thus increasing its scalability. For example, if a group of users share a certain context attribute (e.g., location or interest), we can aggregate their messages based on that attribute. In some cases, this aggregation leads to tremendous decreases in the messages' size, thus increasing the system's scalability (more details in Section 4). Also, the aggregation reduces the cognitive load that users typically suffer when using this kind of applications (caused by the huge number of messages received) [8].

Radiator is a context propagation middleware that combines the concepts of *Delayed Propagation* and *Aggregation* to improve the performance and scalability of context-aware applications. Moreover, these concepts are applied in a completely dynamic manner: each message may be subject to different aggregation levels, depending on the current context of the users involved.

2.2 Model

Context-aware applications start by capturing context in the following form, assuming that P is a person, t a timestamp and A an attribute:

$Context = (P_1..P_n, t_1..t_n, \{A_1..A_n\})$

This triplet represents the attributes that characterize the situation of P_1 to P_n during the time span between t_1 and t_n, roughly following the context definition coined by Dey in his seminal paper [5]. An attribute can be any name/value pair. For example, an application like CenceMe [10] that shares social activities among a group of friends, might capture context like this:

$((`Alice'), `22:30`..`01:00`, \{`location' : `Joe'sPub`, `activity' : `dancing'\})$

A crucial concept in the Radiator design is the possibility of aggregating multiple contexts into a single one while retaining its basic format. For example, if Alice and Marc are both dancing together at Joe's Pub, their context can be aggregated as follows:

$((`Alice', `Marc'), `22:30`..`01:00`, \{`location' : `Joe'sPub`, `activity' : `dancing'\})$

This context could be further aggregated with other contexts and so on and so forth. The advantages of this aggregation are twofold: (1) it reduces the cognitive load on the user by presenting a summary of what's going on instead of multiple single activities and (2) it

Scenario	Description	Aggregability function
Traffic monitoring	Aggregate speedometer and GPS data within 300 seconds periods	$\{time : 300\}$
Road hazards detection	Aggregate vertical accelerometer and GPS data until 5 hazards detected	$\{volume : 5\}$
Popular spots + Friends location	Aggregate location until 10 different users in the same spot but for friends send immediately (non-aggregated)	$\{people : 10\}$ if stranger $\{volume : 1\}$ if friend
Friends location in crowded spaces (concerts, street markets)	Aggregate location based on how far the user is from the recipient (further away implies more aggregation)	$\{volume : \texttt{distance}\}$

Table 1: Different context propagation scenarios and the corresponding aggregability function

significantly reduces the necessary network bandwidth, specially if combined with a compression algorithm.

Related to aggregation, the Radiator also introduces the concept of *delayed propagation*, based on the principle that some context messages may be temporarily retained before being propagated while still fulfilling user expectations. For example, Paul won't mind receiving a message saying that Alice and Marc are dancing at Joe's Pub with a 5 minutes delay unless he's just passing nearby, in which case the delay could prevent him from stopping by (when he receives the message he's already too far from the pub). In fact, the urgency level depends on many factors: location, social distance (e.g., if it's a friend or an acquaintance), current activity, mood, etc.

Radiator allows programmers to define the tolerable propagation delay based on the current context of the users involved in each message (sender and recipient). As already mentioned, this is achieved through an *aggregability* function. Let C_P be the current context of a given person P and C_x the context of someone else to be propagated to P. The aggregability function $G(C_P, C_x)$ represents how much aggregated C_x must be before being transmitted to P, taking into consideration his current context C_P. G returns a tuple in the following format:

$$G(C_P, C_x) \rightarrow \{type : value\}\ , type :: [volume|time|people]$$

The $value$ is an integer representing a threshold of aggregated messages. This threshold may represent a quantity ($volume$), a time range ($time$) or the number of different users contained in the aggregation ($people$). If the type is $time$, context messages will be aggregated until the number of seconds between the oldest and newest retained message is equal or greater than $value$. The types $volume$ and $people$ are similar in the fact that they represent the maximum number of aggregated messages: $volume$ is the number of different messages while $people$ represents the number of different users involved on those messages. For example, if G returns $\{people : 4\}$, the system will aggregate messages until there are four different users involved, before propagating them. The $people$ type is useful to implement k-Anonymity [13] style privacy mechanisms.[2]

To better illustrate the generality of the *aggregability* concept, Table 1 shows some examples of aggregability for real-world scenarios. For simple propagation needs, such as traffic monitoring or hazards detection, we define a simple threshold for the maximum delay (1^{st} row) or the number of retained messages (2^{nd} row). More interesting scenarios are those in which the aggregation depends on contextual information such as the social distance (3^{rd} row) or the geographical distance (4^{th} row). This is possible because the *aggregability* function takes two arguments: the context of the sender and the context of the receiver. Based on that context, it is possible to infer factors such as the ones previously mentioned. This gives great flexibility to the application programmer who can easily fit the specific requirements of his application into a single function and start benefiting from the Radiator middleware without further effort.

[2]It is out of the scope of this paper to analyze how privacy can be achieved using Radiator; the idea is to aggregate as many messages as needed to ensure anonymity.

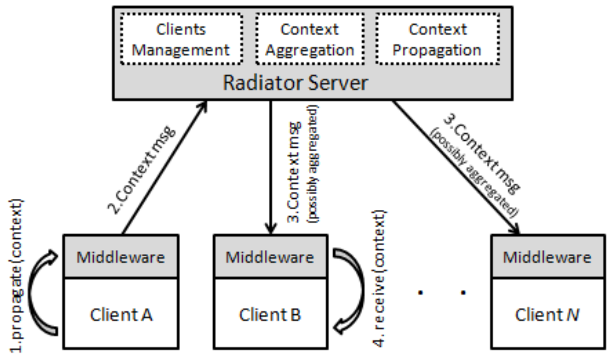

Figure 1: The Radiator includes a pluggable local middleware and a server. Client applications use the middleware to propagate and receive context through direct calls. The aggregation/propagation is abstracted away from the application programmer.

3. ARCHITECTURE

The Radiator architecture has two main components (Figure 1):

1. A **local middleware** that acts as a pluggable component to applications that completely abstracts away the application from the underlying propagation infrastructure;
2. A **server**, to which the local middleware connects, that assumes three responsibilities:
   (a) **Clients management** - Keeps track of all the clients of the application (namely their id and IP address). It also manages the connection with each of these clients: IP renewal, intermittent connectivity, dead/unreachable client detection, etc. Most importantly, it manages the current context of every client which is crucial to the context aggregation process.
   (b) **Context aggregation** - Applies the aggregability function to every incoming context message, providing both the sender and recipient's contexts. It also manages the list of retained messages and the thresholds at which messages are no longer retained and start being propagated.
   (c) **Context propagation** - Delivers the context messages to all clients triggered by the context aggregation component. The delivery can be done using direct connections to the clients, peer-to-peer propagation between clients or a combination of both.

We now describe in more detail the *context propagation* component.

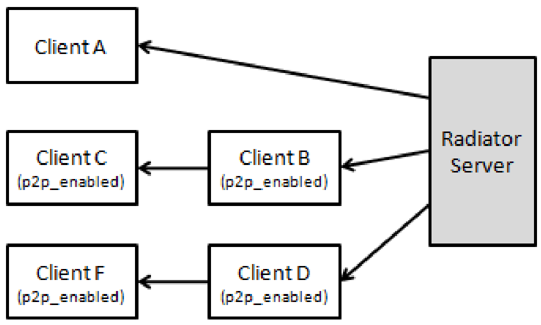

Figure 2: Radiator uses an hybrid propagation model. Some messages are pushed directly to the client while others are pushed between peers following a previously defined dynamic chain.

3.1 Server - Context propagation

The "Context Propagation" component at the server is responsible for distributing the context messages (possibly aggregated) to their recipients. Every client will eventually receive all context messages but, depending on the aggregability function, some may receive the messages faster than others.

The propagation can be done through direct connections from the server to every recipient or through peer-to-peer communication between recipients. In any case, the communication is always initiated by the sender (*push* approach) so there is no need for clients to poll the server or other clients for new messages (causing unnecessary traffic and delays).

The centralized approach, where the server is responsible for pushing messages to every client has the advantage of being simple to implement and allowing clients with network restrictions (e.g., behind a firewall). However, if the number of recipients is large, the server starts suffering from scalability problems, since it has to push the message to everyone.

Radiator introduces an alternative propagation mechanism that is highly dynamic. First, all clients that can communicate directly with other clients (i.e., are not subject to firewall restrictions) send an attribute $p2p\_enabled$ to the server when they register themselves into the system. Afterward, for every message ready for propagation, the server checks which of the recipients are $p2p\_enabled$. Those that are not $p2p\_enabled$ receive the message through a direct push as already described. The others are divided into groups of k elements (k is configurable as a percentage - if the percentage is 20% and there are 100 *p2p_enabled* recipients, then k is 20). Each group is processed as a chain of peers through which the message must get through. The message is propagated from the server to the first peer which then propagates to the second peer and so on and so forth. From now on, we will name these groups as *chain of recipients*. So, for each *chain of recipients*, the server sends only one message which is then disseminated directly between the recipients (*p2p propagation*).

Figure 2 shows a possible scenario: there are five recipients for a given message where only one of them is not $p2p\_enabled$. In this case, the chain of recipients size is setup to be 50%. We can see that the server pushes the message directly to client A (not $p2p\_enabled$) and divides the remaining recipients in two groups. Then, it pushes the message to client B telling him that it should push that message further to client C and does the same for client D which must push forward to client F. It is obvious from this example that the server must do only 3 pushes instead of 5 if there wasn't any p2p propagation. In fact, the server will always push N messages, where (k is the *chain of recipients* percentage):

$$N = N_{non\_p2p} + (N_{p2p} * k/100)$$

Note that, due to its dynamic nature, this chain of recipients is very flexible making it specially suitable to highly dynamic conditions such as those usually found in context-aware applications. Since these conditions may vary very frequently, Radiator continuously recalculates the chain of recipients for each message.

4. EVALUATION

This section presents results of several experiments to evaluate the scalability of the Radiator implementation.

In particular, we measure the tradeoff between network bandwidth consumption and the average propagation time (i.e., the time it takes for a message to go from the sender to the recipient). To study this tradeoff, we take three approaches:

- We evaluate different aggregability functions (using a non-chained approach)
- We evaluate a chained (*p2p*) and a non-chained (broadcast) scenarios using the same settings
- We evaluate several chained scenarios using different *chain of recipients* sizes

4.1 Experimental setup

We developed a traffic monitoring and hazard detection application because it is the kind of context-aware application that usually suffers from the problems outlined on this paper: huge number of messages (e.g., 70.000 cars per day on US expressways [7]) and highly dynamic matching rules (cars in transit are, most of the time, changing their location and speed).

For this experiment, the application produces random context messages (related to traffic information). Each message contains information about the current location, speed and number of hazards detected by the client. The application then uses the Radiator local middleware to propagate these messages to other clients. The experiment was conducted using 7 machines (each one is a 2x 4-Core Intel Xeon E5506@2.13GHz running Ubuntu 10.04.3) connected through a Gigabit LAN switch. The server runs on a dedicated machine; the other 6 machines run the application (with multiple threads where each thread simulates a client).

Several metrics such as CPU, memory and network bandwidth consumption are captured using the sysstat tool [3]. The average delay between message transmission and reception was also recorded (the average delay between the moment a client sends a message and the moment another client receives the message).

4.2 Aggregation/compression

To measure the impact of the aggregation on the system scalability (as related to the consumed network bandwidth), we launched 60.000 clients[4] (threads) across 6 different machines, each client propagating 1000 messages. We experimented different aggregability/compression settings:

- **Vol: 1 (no gzip)** - Messages are immediately propagated, uncompressed. All other scenarios are performed with compression turned on using gzip (the default settings). We decided to include an uncompressed experiment to understand the impact of compression on the bandwidth.
- **Vol: 1** - Messages are immediately propagated.
- **Vol: 20** - Messages are retained in the server until there are 20 pending messages, which are then propagated in a single message.

[3]Available at http://sebastien.godard.pagesperso-orange.fr/

[4]60.000 nodes is close to the average number of cars on US expressways, according to USGS data [7]

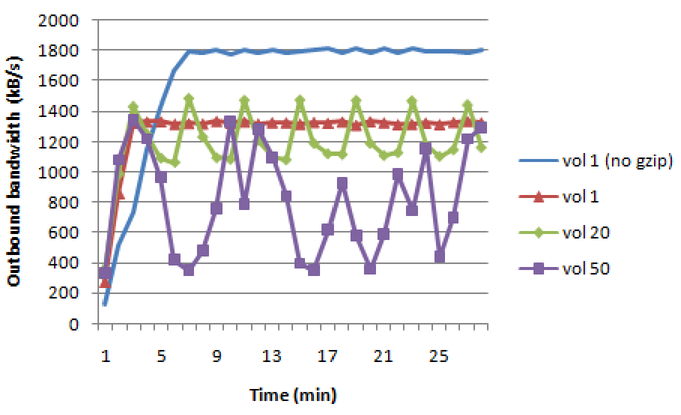

Figure 3: Server outbound network usage over time, for different aggregability settings

- **Vol: 50** - Messages are retained in the server until there are 50 pending messages, which are then propagated in a single message.

Settings	Avg. Outbound Bandwidth (kB/s)	Avg. Lag (sec)
vol 1 (no gzip)	1751	41
vol 1	1324	42
vol 20	1222	34
vol 50	797	36

Table 2: Average outbound network usage (from the server) and lag for different settings

We decided to change the volume parameter (as opposed to the time parameter, for example) because it is easier to manipulate. However, as we observed experimentally, changing the time and volume parameters redound in the same effect.

Figure 3 and Table 2 show the results concerning the server outbound bandwidth and average delay (between a client sending a message and another client receiving it), under these settings. As expected, the non compressed scenario is the worse performer in the experiment. We can see in Figure 3 that even without aggregation (*vol 1*), the mere act of compressing achieves a 25% reduction on consumed bandwidth. Aggregating with *vol 20* yields another 8% decrease and with *vol 50* we achieve a substantial reduction of 40% over the non aggregated compressed scenario. This is because, as we aggregate more messages, the compression algorithm becomes more effective because of the increased redundancy.

We can also see in Figure 3 that the consumed bandwidth is much more uniform on the unaggregated scenarios, because messages are immediately propagated (i.e., constant flow of data). On aggregated scenarios, messages are retained in the server and propagated in batches, originating big fluctuations on network usage. Nevertheless, the average consumption is relatively stable.

Another important insight from these results is the impact of the different aggregability settings on the average message propagation delay. Table 2 shows that even in the scenario with *vol 1* (where messages are not being retained in the server) there is already a substantial average lag of 42 seconds (between sending and receiving a message) caused by 60.000 clients continuously pushing information and overloading the server's outbound network link. The stress on the network link is key to explaining why the aggregated scenarios (*vol 20* and *vol 50*) actually decrease the lag even though messages are being retained at the server. By sending much fewer messages the server is reducing the stress in the outbound network link and increasing the throughput. In a sense, we can say that under heavy load, it is unavoidable that there will exist message retention on the network link so we might as well retain them at the server.

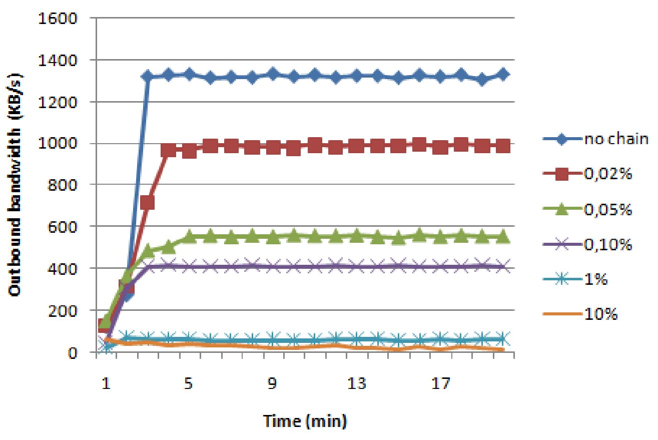

Figure 4: Server outbound network usage over time, for different sizes of recipients chains

4.3 Hybrid propagation

Even with aggregation, the server outbound bandwidth can easily become the bottleneck on large-scale distributed context-aware systems. We use the same setup (simulating 60.000 clients) to evaluate the hybrid propagation mechanism described in Section 3.1 for different *chain of recipients* sizes. As already mentioned, this size (represented as a percentage) is the maximum number of clients in a group (chain) for which the server sends only one message which is then disseminated directly between them (*p2p propagation*). We tested the following chain sizes (represented by the k parameter described in Section 3.1):

- no chain - server sends messages to every client.
- $k = 0.02$ - server sends messages to 5000 groups of 12 clients each.
- $k = 0.05$ - server sends messages to 2000 groups of 30 clients each.
- $k = 1$ - server sends messages to 100 groups of 600 clients each.
- $k = 5$ - server sends messages to 20 groups of 3000 clients each.
- $k = 10$ - server sends messages to 10 groups of 6000 clients each.

Without lack of generality, to simplify the analysis, we started by conducting the experiment without aggregation (vol 1).

Figure 4 shows the outbound network usage over time (during the experiment) under these chain sizes. Even with a chain size of 0.02% there is already a significant decrease over the no chain scenario (approximately 26%). Note that, even if the server sends messages to 12x fewer clients, we incur the overhead of including all the recipients' IDs and addresses in the message (messages become much bigger). As we increase the chain size, the server outbound bandwidth decreases, since the server sends fewer messages. Obviously, the outbound bandwidth consumption on the other machines increases but in a real scenario we expect this not to be a problem since each client will have his own machine/device to run the application. We can also see that the decrease is logarithmic - the reduction that we get when we go from a chain size of 0,02% to 0,05% is much greater than the reduction we get when we go from

Chain size	Avg. Lag (sec)
No chain	42
0,1%	70
1%	74
5%	223
10%	295

Table 3: Average lag for different *chain of recipients* sizes (vol 1)

Chain size	Settings	Avg. Lag (sec)
1%	vol 1	74
1%	vol 20	41
1%	vol 50	53

Table 4: Average lag with chain size 1% for different volumes

1% to 10%. This is because the cost of including information about the members of the group (increasing the message size) no longer justifies the gain of establishing fewer connections. It's worthy to note that even though messages size increases (due to the inclusion of the recipients chain), this allows a highly dynamic reconfiguration capability of the propagation paths; this solution is better than using a specific protocol (with specific control messages) for the propagation paths reconfiguration (e.g. multicast trees), specially if it occurs very frequently.

Table 3 shows the average lag in seconds for different sizes of recipients chains. We can observe that there is an increase in the average lag as we go from a non chained (direct push) model to a chained (p2p) model. This is due to the fact that, in the latter, the messages must travel through the chain of recipients instead of being directly pushed from the server to each recipient. Nevertheless, the average lag only grows 66% and 76% even though the message has to travel through 60 clients (k=0,1%) and 600 clients (k=1%). If we increase the size of the chain too much (above 5%), the number of hops the message has to travel starts to severely penalize the average lag and the technique is no longer effective. In this case, a good equilibrium seems to be achieved with a chain size of 1%: the consumed outbound is reduced to 4,4% of the non-chained scenario while the corresponding delay only increases 76%.

Using the chain size of 1% (the one that reached the best compromise between bandwidth and lag), we experimented with different aggregability functions, more specifically, setting different volumes. The results are shown in Table 4. We can see that aggregating messages has the same effect it had in the non chained experiment: the lag decreases from *vol 1* to *vol 20*, and while it increases again with *vol 50*, it remains smaller than the *vol 1* lag. The explanation is the same: since the server is sending much fewer messages (and note that these messages are bigger as they now carry the chain of recipients), the server's network link is less stressed, allowing a greater throughput.

5. CONCLUSIONS

In this paper, we present Radiator, a dynamic adaptable middleware for efficient distribution of context messages. Unlike current selective message distribution approaches which rely on relatively stable sets of matching rules (the rules that dictate who receives a certain message), our approach relies on functions that, given the current context of sender and receiver, decide under which conditions a message should be distributed.

Moreover, we introduce the concept of propagation based on a chain of recipients that, unlike pub-sub and application-level multicast tree approaches, can quickly react to highly dynamic ever-changing rules. In fact, as our experiments have shown, the tree can be continuously rebuilt and still achieve significant bandwidth reduction and no penalty on the average propagation time. This is only possible because of our delayed propagation mechanism that, when paired with compressed aggregated messages, makes a much more efficient use of the network bandwidth.

By combining both techniques (aggregation/compression and *chain-based* propagation) we were able to reduce the server's outbound bandwidth 20x without penalizing the average propagation delay in a given scenario (*partition 1%* and *vol 20*).

6. REFERENCES

[1] A. Allavena, A. Demers, and J. E. Hopcroft. Correctness of a gossip based membership protocol. In *Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing - PODC '05*, page 292, New York, New York, USA, July 2005. ACM Press.

[2] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263, 2007.

[3] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[4] M. Castro, M. Jones, and A. Kermarrec. An evaluation of scalable application-level multicast built using peer-to-peer overlays. *IEEE INFOCOM*, 2:1510–1520, 2003.

[5] A. Dey and G. Abowd. Towards a better understanding of context and context-awareness. In *CHI 2000 workshop on the what, who, where, when, and how of context-awareness*, volume 4, pages 1–6, 2000.

[6] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[7] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services Through Spatial and Temporal Cloaking. *Proceedings of the 1st international conference on Mobile systems, applications and services - MobiSys '03*, pages 31–42, 2003.

[8] J. Hudson, J. Christensen, W. Kellogg, and T. Erickson. I'd be overwhelmed, but it's just one more thing to do: Availability and interruption in research management. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves*, number 4, pages 97–104. ACM, 2002.

[9] A. Mathur, R. W. Hall, F. Jahanian, A. Prakash, and C. Rasmussen. The Publish / Subscribe Paradigm for Scalable Group Collaboration Systems. *Ann Arbor*, 1001(313):48109, 1995.

[10] E. Miluzzo, N. Lane, K. Fodor, R. Peterson, and H. Lu. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *6th ACM conference on Embedded network sensor systems*, page 337, New York, New York, USA, 2008. ACM Press.

[11] D. Salber, A. Dey, and G. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, page 441, New York, New York, USA, 1999. ACM.

[12] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. *in Proceedings AVVG 1997, Brisbane, September*, 1997.

[13] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.