Radiator - efficient message propagation in context-aware systems

Pedro Alves[1]* and Paulo Ferreira[2]

Abstract

Applications such as Facebook, Twitter and Foursquare have brought the mass adoption 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 (e.g., Foursquare has millions of users sharing their location on a regular basis, with almost 1 million updates per day).

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-known 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 within their applications. Based on each user's 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.
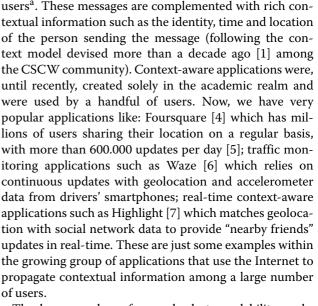
Keywords: Context propagation; Scalability; Publish-subscribe; Multicast trees; Peer-to-Peer; Aggregation

1 Introduction

Context-aware systems take into account the user's current context (such as location, time and activity) to enrich the user interaction with the application [1,2]. In the last decade, this topic has seen numerous developments that demonstrate its relevance and usefulness; this trend was further accelerated with the recent widespread availability of powerful mobile devices (such as smartphones) that include a myriad of sensors which enable applications to capture the user environment for a large number of users [3].

Following on this trend, we are watching a radical change in the type of packets that travel on the Internet. Facebook and Twitter (among others) brought the mass adoption of personal short messages or posts, distributed in (soft) real-time to a potentially large number of users[a]. 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 [1] among the CSCW community). Context-aware applications were, until recently, created solely in the academic realm and were used by a handful of users. Now, we have very popular applications like: Foursquare [4] which has millions of users sharing their location on a regular basis, with more than 600.000 updates per day [5]; traffic monitoring applications such as Waze [6] which relies on continuous updates with geolocation and accelerometer data from drivers' smartphones; real-time context-aware applications such as Highlight [7] which matches geolocation with social network data to provide "nearby friends" updates in real-time. These are just some examples within the growing group of applications that use the Internet to propagate contextual information among a large number of users.

The huge number of users leads to scalability problems as can be seen in several news articles (e.g., Twitter

*Correspondence: pedro.h.alves@gmail.com
[1]INESC-ID, Technical University of Lisbon, Opensoft, Rua Joshua Benoliel, 1, 4C, 1250 Lisbon, Portugal
Full list of author information is available at the end of the article

admitted that "record traffic" and "unprecedented spikes in activity" led to problems with the site [8]). In addition to the huge number of users and messages exchanged, context propagation creates unique challenges in the realm of distributed systems [9]: it is highly dynamic, does not require user intervention, and has different levels of urgency. We now detail each one of these challenges.

1.1 Highly dynamic

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) [10]. 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 his 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 [11] fall on the same problem: they assume that distribution rules do not 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 [12] 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 alone should be enough to convey all the information we need, e.g., the new location, in the "friends nearby" application).

1.2 Does not require user intervention

Context propagation does not usually require explicit user intervention — it happens in the background thus increasing the usability and effectiveness of the application [9]. Thus, context-aware applications continuously monitor and propagate the user's context. Moreover, context information is transmitted unattended, i.e., without the user having to explicitly give that command [13]. Contrast that with the kind of traffic we are used to watch until recently. Be it an email, a website or an FTP session, the communication is always deliberately initiated by the user. This has been changing and the immediate consequence of unattended communication is that it will happen a lot more frequently. Humans can only send a small number of messages in a given period of time but computers do not have these limits. To provide the best user experience, these applications will try to propagate their context as much and as often as possible, since they do not have to rely on the user explicitly initiating the communication. This inevitably leads to a huge number of messages being sent to the server at a high rate, thus reducing the system's scalability.

Note that, although smartphone OSs and synchronization tools (e.g., Dropbox) already do non-triggered updates, they are not as prone to scalability issues as context-aware systems. Non-triggered updates on smartphones are typically initiated by the server (e.g., a new version of an installed app is available or a new email has arrived). Synchronization tools are triggered after a human interaction (e.g., editing or copying a file) so they are bound by definition to the number of actions a human can do. Context-aware systems are triggered by changes in the user's context such as his location, speed or heart rate, which can potentially lead to a much larger number of messages than the previous examples.

1.3 Different levels of urgency

Finally, the urgency of context delivery is also highly dynamic. An application that enhances the cellphone's contact list with the current availability of others [14] must propagate context as soon as possible while an application that provides a noise map of the city [15] does not require immediate propagation (specially if it's not the city where the user is currently located). Even within the same application there exists different urgency levels. For example, Cenceme [16] captures users's activity (e.g., dancing at a party with friends) and shares those in the social network. It makes sense to propagate those activities to close friends as fast as possible while acquaintances only receive those updates a few hours later. This behavior resembles traditional relaxed consistency systems [17] with the

problematic difference of having some users requiring strong consistency while others tolerate some temporary inconsistencies.

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 [18,19], multicast trees [11] or gossip-based protocols [20] are not adequate to handle these dynamics, because they assume the matching rules are fixed or change infrequently (therefore changes are too expensive). Also, since such classic approaches do not 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's responsibility.

We propose an adaptable middleware, called Radiator, where context propagation is controlled by functions that, given the context of the recipient, dictate in which conditions a given context message should 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. Also note that *aggregability* functions are provided by the application programmer. Even though the programmer may take into account the user's input, it is not the user's responsibility to provide such functions.

Finally, we are also able to avoid some limitations of the centralized approach (e.g., less scalability due to resource usage concentration) by allowing a hybrid mechanism: using a centralized approach for defining the message propagation strategy (e.g., deciding whether a message should be retained or propagated) and a peer-to-peer approach for the actual message propagation. Thus, the decision of which clients receive the message and when they do so is still responsibility of the server, but most of the propagation is done through direct connections between the clients following a *p2p* approach, therefore reducing the outbound network bandwidth needs of the server and increasing scalability.

Moreover, the propagation path is completely dynamic: the set of recipients of each message is continuously changing based on the result of the *aggregability* function. This results in a more efficient use of the available network bandwidth.

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 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 in Section 4 its implementation. Section 5 presents evaluation results of Radiator's implementation and finally, in Sections 6 and 7, we relate Radiator with previous work and draw some conclusions, respectively.

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 a 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 others but it can be grouped before being propagated. In this case, the user does not care about individual context updates given that 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 does not break user expectations because, for some contextual information, he does not mind receiving it with delay. For example, a spot does not become popular in seconds and it certainly does not stop being so 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 5). 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) [21].

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 [1]. An attribute can be any name/value pair. For example, an application like

CenceMe [16] that shares social activities among a group of friends, might capture context as follows:

```
(("Alice"), 22:30..01:00,{"location":
        "Joe's Pub", "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's", "activity":"dancing"})
```

This context could be further aggregated with other contexts and so on and so forth. The advantages of this aggregation are: (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 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 five 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 of each message based on the current context of the users involved (sender and recipient). As already mentioned, this is achieved through an *aggregability* function. Let C_S be the current context of sender S and C_R the current context of receiver R. The aggregability function $G(C_S, C_R)$ represents how much aggregated the message must be before being transmitted to R, taking into consideration both C_S and C_R. G returns a tuple in the following format:

$$G(C_S, C_R) \rightarrow \{type : value\}, type \in (volume, time, people)$$

The *value* is an integer (or a function returning 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 senders 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[b].

Since *value* can be a function, the aggregation threshold can be very dynamic. For example, a certain application may want to immediately propagate a person's context to her friends but aggregate messages up to 40 seconds when they are being propagated to strangers. We could define such function as follows:

$$G(C_S, C_{R_i}) \rightarrow \{volume : 1\} \iff is\_friend(R_i, S)$$

$$G(C_S, C_{R_i}) \rightarrow \{time : 40\} \iff is\_stranger(R_i, S)$$

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 (1st row) or the number of retained messages (2nd row). Since traffic congestion occurs during a relatively long period of time it can be aggregated within 5 minutes (300 seconds) periods without losing its usefulness and relevance. More interesting scenarios are those in which the aggregation depends on contextual information such as the social distance (3rd and 4th rows), the geographical distance (5th row) or even the number of shares (6th row). This generality is possible because the *aggregability* function takes two arguments: the context of the sender and the context of the receiver. 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.

Listing 1 Aggregability function that aggregates messages based on how far the user is from the sender (implemented in Python)

```
def aggregability(Cs, Cr):
return { 'volume' : distance_in_kms
                    (Cs['attributes']['location'],
                     Cr['attributes']['location'])}
```

For example, the *aggregability* function for the scenario #5 (see Table 1) can be implemented in the Python language as shown in Listing 1. C_s represents the context of sender and C_r the context of the recipient. We assume that *distance_in_kms* is a function that returns an integer representing the number of kilometers between two geo-

locations. This aggregability function returns a *volume* that depends on that distance, i.e., messages become more aggregated as users become further away from each other.

The effective aggregation of messages is also performed by a function (*aggregation function*). If the developer does not provide any aggregation function, Radiator applies a simple concatenation of the messages to aggregate. To achieve higher compression levels (and therefore reduce network bandwidth), the developer should provide an aggregation function that takes into account the specific needs of his application. Table 2 shows some examples of such functions. In the traffic monitoring case (first row in Table 2), we are concerned about the average traffic speed within a geographical region: if the average speed is near zero, it is reasonable to assume that there is traffic congestion within that particular region.

3 Architecture

The Radiator architecture has two main components (see 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) **Client management** — Keeps track of all the clients (namely their identification and IP address). It also manages the connection with each one 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 key components of the Radiator architecture: *context aggregation* and *context propagation*.

Table 1 Different context propagation scenarios and the corresponding aggregability functions

#	Scenario	Description	Aggregability function
1	Traffic monitoring	Aggregate speedometer and GPS data within 300 second periods	{*time* : 300}
2	Road hazards detection	Aggregate vertical accelerometer and GPS data until 5 hazards detected	{*volume* : 5}
3	Popular spots + Friends' location	Aggregate location until 10 different people in the same spot but for friends send immediately (non-aggregated)	{*people* : 10} if stranger {*volume* : 1} if friend
4	Facebook likes	Aggregate likes from strangers within 300 seconds periods, likes from friends of friends until there are 5, and likes from direct friends with a maximum delay of 30 seconds	{*time* : 300} if stranger {*volume* : 5} if friend_of_friend {*time* : 30} if friend
5	Friends' location in crowded spaces (concerts, street markets)	Aggregate location based on how far you are from the recipient (further away implies more aggregation)	{*volume* : `distance`}
6	Stock market alerts	Aggregate stock market information during a period of time proportional to the number of shares owned by the recipient (higher number implies less aggregation)	{*time* : 1000/(1 + `num_of_shares`)}

3.1 Server — context aggregation

The "Context Aggregation" component at the server is responsible for applying an aggregability function to every incoming context message. Depending on the result of the aggregability function, the message may be put on the immediate propagation queue or on a queue associated with the threshold that will trigger the propagation. When one of these queues satisfy the associated threshold (e.g., if the threshold is *volume* : 5, and the associated queue has 5 elements), its items are moved into the immediate propagation queue. Algorithm 1 presents the pseudo-code for this process.

There is a global data structure that stores all the pending messages per client (*pending*). For each received context *C*, the Radiator server calls the algorithm, which may decide, depending on the aggregability function (provided by the application programmer) to append it to the *pending* data structure or return it through the *to_send* variable. The *to_send* variable stores all the messages ready for immediate propagation (line 20) and is passed by the Radiator server to the context propagation component (detailed in the next section).

Note that, as in many context-aware systems, Radiator propagates every message to everyone (albeit some can
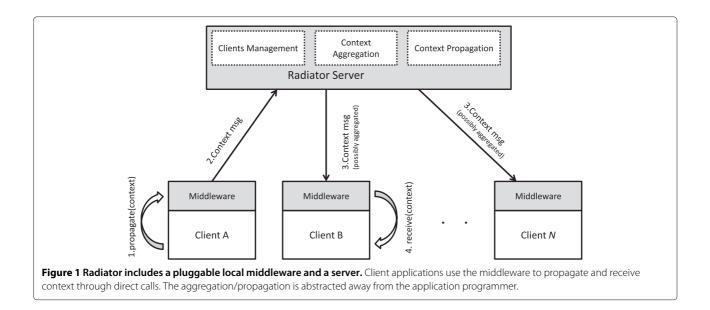
Table 2 Different context propagation scenarios and the corresponding aggregation functions (see Table 1 for corresponding scenarios)

#	Scenario	Aggregation function
1	Traffic monitoring	`avg`(speed) by location
2	Road hazards detection	`sum`(hazards) by location
3	Popular spots	`count`(people) by location
4	Facebook likes	`sum`(likes) by object
5	Friends location in crowded spaces	`list`(people) by distance
6	Stock market alerts	`newest`(values) by share

Algorithm 1 Pseudo-code of the context aggregation and propagation process

```
 1: procedure PROPAGATE(in C, in/out pending)
 2:     to_send ←NEW_DICTIONARY()
 3:     for all recipient ∈CLIENTS_TABLE do
 4:         pending[recipient].append(C)
 5:         pending_queues ←NEW_DICTIONARY()
 6:         for all C_pending ∈ pending[recipient] do
 7:             result ←aggregability(C_recipient, C_pending)
 8:             pending_queues[result].append(C_pending)
 9:
10:             if result["volume"] then
11:                 propagate                            ←
    (len(pending_queues[result]) >= result["volume"])
12:             else if result["users"] then
13:                 propagate                            ←
    (users(pending_queues[result])) >= result["users"])
14:             else if result["time"] then
15:                 propagate                            ←
    (time_range(pending_queues[result]) >= result["time"])
16:             end if
17:
18:             if propagate then
19:                 C_aggr ←join(pending_queues[result])
20:                 to_send[C_aggr].append(recipient)
21:                 for all c ∈ pending_queues[result] do
22:                     pending[recipient].remove(c)
23:                 end for
24:                 pending_queues[result].clear()
25:             end if
26:         end for
27:     end forreturn to_send
28: end procedure
```

receive it with delay). In line 3, we can see the beginning of the loop that iterates through all the clients previously registered. For each client (a recipient, in this case), the message to propagate is first appended to its global

Figure 1 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.

pending list. Then, this list is iterated (from the oldest to the newest) while the *aggregability* function is applied to each pending message (line 7). We have to reiterate from the beginning (line 6) because the current context of the recipient may have changed since the last time the *propagate* function was called. That is, the result of applying the *aggregability* function to the same message may change with time so we can never rely on previous results (due to context's dynamic nature). Note that if there is data loss, either the message that was lost is recovered by the underlying network layer or the application deals with that message loss.

In addition, the result of the *aggregability* function may vary with each message in the *pending* list, so we have to group them by result. This is done using the variable *pending_queues*, which stores every result of the *aggregability* function along with the corresponding list of messages (line 8). Lines 10–16 test the different scenarios that can trigger the propagation: reaching the maximum number of retained messages (*volume*), reaching the maximum number of different users (*people*), or reaching the maximum time span since the oldest retained message (*time*).

If one of these thresholds is reached, the *propagate* variable will have the value *True.* In this case, the messages that were retained because of that particular threshold will be aggregated into a single message (line 19). The *join* function has a default behavior: it joins all the users involved into a single set, calculates the global time range based on the difference between the oldest and the newest message in the aggregation, and joins all the attributes into single lists. This behavior can be overridden to achieve more efficient aggregations. For example, if the messages contain geo-location coordinates, the application

programmer may decide to adopt a minimum bounding box approach to condense multiple locations into a single square that contains all the locations. (More examples of aggregation functions can be found in Table 2.)

The aggregated message is then appended to the *to_send* variable (line 20). This variable stores a list of messages for immediate propagation and for each message it stores the list of its recipients. This data structure is optimized for the context propagation component described in Section 3.2. Finally, the messages elected for immediate propagation are removed from the pending queues (lines 21–24).

3.2 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 sooner 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., those that are 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 (in the sense, that it automatically adapts itself to the current context of the sender and receiver, which can change very frequently). 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. Afterwords, 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). 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*).
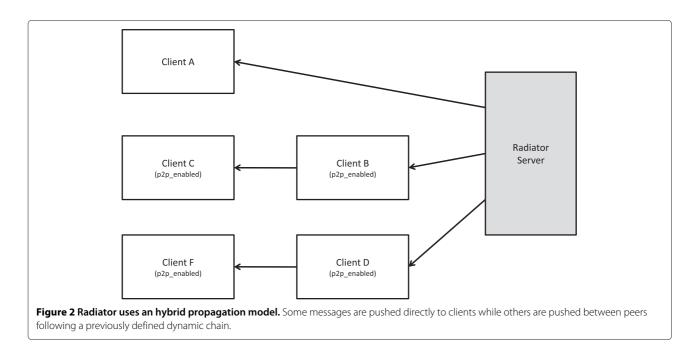
Figure 2 shows a possible scenario: there are five recipients for a given message where only one of them is not *p2p_enabled* (Client A). In this case, the chain of recipients size is setup to be 50%. We can see that the server pushes the message directly to client A (not *p2p_enabled*) and divides the remaining recipients in two groups. Then, it pushes the message to client B that should push that message further to client C, that does the same for client D, which must push the message 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):

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

From this follows that the smaller the *k*, the fewer messages the server has to push although this comes at a cost. The messages must contain the full chain of recipients for each group so, if the group is very large, the chain significantly increases the payload size therefore defeating our main purpose: reducing server outbound bandwidth. However, regarding scalability, this is not significant for two reasons: (1) given that most of the messages are aggregated, the relative weight of the recipients chain generally decreases, since the main payload is much bigger, and (2) compression is highly effective for this chain of recipients (for example, increasing the chain from 10 to 100 recipients only yields a 5× increase in the compressed payload — more details in Section 5).

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.

If a client is unable to forward the context message to the next in the chain of recipients, it informs the server accordingly. Then, the server tries the next one in the chain. The server will not include the unreachable client in the next propagation to prevent wasting unnecessary resources unless the client shows any sign that it is still alive (e.g., by sending a message to the server).

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

Even though we included some mechanisms to cope with network data loss such as the one explained above, this is not the focus of the paper for two main reasons: (1) there are well-known distributed systems techniques that deal with this problem that can easily be used within Radiator, and (2) some information that may be lost does not have to be re-sent given its semantics (i.e., it is superseded by the next messages).

4 Implementation

A prototype of Radiator is implemented in Python. The Python language has excellent support for list manipulation and anonymous functions (lambda functions), very useful for defining aggregability functions in a compact form. Its interpreter is also very lightweight (when compared to Java or .NET, for example) which enables experiments where we launch thousands of threads in the same machine. Note that, in order to use Radiator, the client application does not have to be implemented in Python. The Radiator server provides an API through HTTP Web Services which any application can easily use[c].

4.1 API

The local middleware is a pluggable component that runs along the client application and is responsible for abstracting away all the details of context aggregation and propagation. It provides a very simple API with 4 functions:

setup(*server_address*) — Initializes the local middleware and stores the server address that will be used for subsequent API calls. Application programmers must first call this function before using other functions of the API.

register_client(*client_id*, *receive_callback*, *attributes*) — Upon calling this function, the local middleware opens a local HTTP port for receiving messages. It then sends a registration message to the server indicating the *client_id* and the local HTTP port. The server stores this information in a clients' table that will be used to propagate context. The *receive_callback* parameter defines the function that will be called when the client receives a message (see below for further explanation).

Note that from this moment on, the client may start receiving messages propagated by others. It is also able to send messages using the *propagate_context* function (see below). The client is uniquely identified by the pair (ip address, client id) so name collisions are infrequent. If a collision occurs, the function raises an exception and the client application must try again with a different identifier. The *attributes* parameter is optional and is used to inform about specific information related to this client. One of these attributes is the "p2p_enabled" attribute which, as previously described, informs the server that the client is able to connect directly to other peers (either by manual configuration or by testing a connection with a dummy client on well-known location/port).

unregister_client(*client_id*) — It signals that the client is no longer interested in receiving messages propagated by others (e.g., logout). This closes the local HTTP port that was opened to receive incoming messages and sends a message to the server to delete the client from the client's table.

propagate_context(*context*) — The application calls this function whenever it wants to propagate the context associated with a given client to others. The *context* parameter is an object containing a set of clients, a time range and a list of attributes (name/value pairs).

This object is marshalled into JSON format [22] and sent to the server. Depending on the aggregability function, the server may decide to propagate the message immediately or aggregate it with other messages. In any case, this is transparent to the client application — the function returns as soon as it is able to deliver the message to the server.

The application programmer must implement *receive_callback* (the second parameter of the *register_client* function) according to the following interface.

receive_callback(*context*) — This function is called in the client application for every received message. Note that the *context* parameter is an object of the same type of the one sent using *propagate_context*.

Also, the JSON message received from the server is conveniently unmarshalled into an object before invoking the callback. This process occurs asynchronously, following the event-driven paradigm and it is highly convenient for application programmers who want to refresh the application's UI with incoming messages.

4.2 Server — client management

The server keeps a table for each active client. For each client, it stores its identification, IP address and local callback port as well as some attributes that may be sent using the *register_client*. This table is kept in memory for fast access (we opted for storing the table in memory since the volume of data to store for each client is small and the cost of memory is decreasing). If for some reason the server is no longer able to establish a connection or send a message to a client, the client is considered unreachable and is therefore deleted from this table to prevent further wasting of resources. The local middleware can reissue a *register_client* anytime (e.g., if it does not receive any messages for a certain amount of time). If it already exists in the clients' table it will overwrite the previous entry (e.g., it may have new local callback port or a different attribute).

The last context message received for each client is also stored in the table. This effectively represents (possibly with some delay) the current context of each client, which is used by the aggregability function to decide its result.

4.3 Communication protocol

Radiator uses HTTP to communicate between the server and the clients and between clients themselves. Besides being a well-understood widely-adopted protocol, it has several advantages over other protocols: (1) multiple available libraries in many programming languages, (2) works well within security-constrained environments (e.g., behind firewalls), (3) it does not require a permanent connection between the client and the server which would hinder the server's scalability, (4) it supports encryption (HTTPS) and compression (through the *accept-encoding* header).

Our prototype uses a fast and lightweight embeddable web-framework called Bottle [23] in conjunction with Paste [24], a high-performance multi-threaded HTTP middleware.

The connections are accomplished through an HTTP request initiated by the sender (the server or client) to the local TCP port that every recipient opened during its initialization (to receive incoming messages). This port is registered in the clients table, as described in Section 4.2. Note that, by default, all communications are compressed using gzip. The use of the HTTP protocol allows clients behind firewalls, and since the server initiates the connection (push approach) there is no need for clients to poll the server for new messages, causing unnecessary traffic and delays.

4.4 Compression

As already mentioned in Section 2, most of the messages compression results from applying an *aggregation* function to each message. If the developer provides such function, it can achieve high compression levels by taking into account the specific context attributes of his application (e.g., using the average speed instead of individual speed information in a traffic monitoring application).

Besides the *aggregation* function, Radiator also uses standard compression mechanisms (e.g., gzip) to increase the efficiency gains of aggregating multiple messages into one. The HTTP protocol supports compression through the headers *Accept-Encoding* and *Content-Encoding*. There is a negotiation between client and server where the client sends an HTTP Request containing the header Accept-Encoding: gzip and, if the server supports the request compression scheme, responds with a gzip [25] compressed response along with the header Content-Encoding: gzip. Note that this only works for responses — requests are never compressed. Since most of the HTTP

traffic lies on responses, this does not usually constitute a problem.

However, in Radiator, requests make up the biggest slice of traffic, because it follows a push model: clients push context to the server, which then pushes that context to other clients. Also, when a chain of recipients is used (*p2p* propagation), clients push messages between them. The HTTP protocol does allow a *Content-Encoding: gzip* in the request but most of the implementations ignore this header. So, we had to develop: (1) an extension to the standard HTTP python lib that optionally compresses request on the client, and (2) a plugin to handle compressed requests on the server.

5 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 chained (*p2p*) and non-chained (broadcast) message propagation scenarios using the same settings
- We evaluate several chained message propagation scenarios using different *chain of recipients* sizes

5.1 Experimental setup

We developed a traffic monitoring and hazard detection application (scenarios #1 and #2 in Table 1) 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 [26]) 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) — we used a standard python random function (within bounds) for each of the context variables: location, speed and number of hazards. 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 $2\times$ 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). All these machines have their clocks synchronized using NTP.

Several metrics such as CPU, memory and network bandwidth consumption are captured using the sysstat tool [27]. 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).

5.2 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 (threads) across 6 different machines, each client propagating 1000 messages (each one ranging between 300 and 500 bytes) at 1 msg/sec rate. We experimented with 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.
- **Vol: 50** — Messages are retained in the server until there are 50 pending messages, which are then propagated in a single message.

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 should have the same effect w.r.t. the performance values obtained.

Figure 3 and Table 3 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 [28].
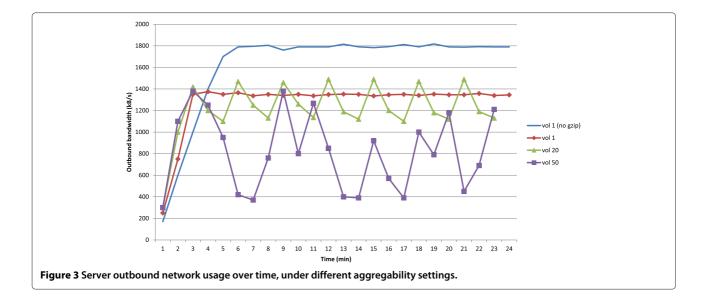
We can also see in Figure 3 that the consumed bandwidth is much more uniform on the unaggregated scenarios, because messages are immediately propagated

(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 (around the values presented in Table 3).

Another important insight from these results is the impact of the different aggregability settings on the average message propagation delay. Table 3 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 many 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. Moreover, Radiator is able to perform this retention without breaking user expectations because, for some contextual information, he does not mind receiving it with delay.

5.3 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 3.2 under 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 3.2):

- no chain — server sends messages to every client individually.
- $k = 0.02$ — 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.

Figure 3 Server outbound network usage over time, under different aggregability settings.

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 (aprox. 26%). Note that, even if the server sends messages to 12× less clients, we incur in 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 send fewer messages. Obviously, the outbound bandwidth consumption on the other (client) 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 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 is worthy
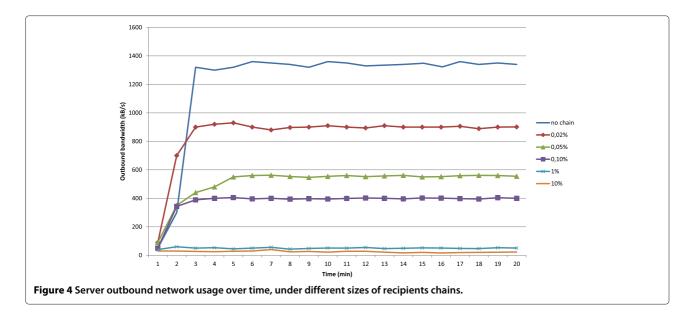
noting that even though message size increases (due to the inclusion of the recipients chain), this allows a highly dynamic reconfiguration capability of the propagation paths (the propagation path is calculated for each individual message); this solution is better than using a specific protocol (with specific control messages) for the propagation paths reconfiguration (e.g., as in multicast trees — more details in Section 6), specially if it occurs very frequently.

It is also important to understand the impact in the propagation time as we change the sizes of recipients chains. Table 4 shows the average lag in seconds under 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% (from 42 ms to 70 ms) even though the message has to travel through 60 clients (k=0,1%).

Even propagating the message among 600 clients (k=1%) does not add much to the average lag (76% over the non partitioned scenario, i.e., from 42 ms to 74 ms). This is because this propagation occurs in 6 nodes that do not include the server node (which is the node that is being stressed out with these experiments). Although we do not have yet real-world results, we expect this delay to be even less significant in those conditions, where each node corresponds to a single client. If we grow too much the size of the chain (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

Table 3 Average outbound network usage (from the server) and lag under different settings

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

Figure 4 Server outbound network usage over time, under different sizes of recipients chains.

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 5. 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 many 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 out, allowing a greater throughput.

Although this is a simulated experiment (due to the lack of publicly available real traces), we believe it reflects a real-world scenario of a traffic monitoring application capturing and propagating the location and speed of cars in transit on a medium-sized city. We conducted the experiment with 60.000 nodes which is close to the average number of cars on US expressways, according to USGS data [26].

**Table 4 Average lag under different
chain of recipients sizes (vol 1)**

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

6 Related work

In this section, we review the literature for the usage of aggregation on context-aware systems and some large-scale context propagation techniques.

6.1 Aggregation

The term "aggregation" is usually identified with the inference/reasoning component of typical context-aware systems. Although this component's primary goal is to provide application developers with data on a higher level of abstraction [9], this usually materializes on some kind of semantic aggregation. A classic example is the Activity widget provided by the Context Toolkit [2] that senses the current activity level at a location such as a room, using a microphone. Instead of producing raw audio data captured by the microphone, it provides a high-level attribute "Activity Level" with three possible values: *none*, *some* or *a lot*. In a sense, we can say that a sample of raw audio data was semantically aggregated into a single attribute.

Another type of aggregation can occur at the distribution stage, when context was already captured and processed, and is now in the process of being sent to the client application. This aggregation consists of gathering into a single large message the content of multiple smaller messages before propagating it. Even though the vast majority of the context-aware frameworks propagate

**Table 5 Average lag with chain size 1% under different
volumes**

Chain size	Settings	Avg. lag (sec)
1%	Vol 1	74
1%	Vol 20	41
1%	Vol 50	53

context in single disaggregated messages [9], recent experiments with aggregation have achieved good results; for example ReConMUC [29] combined a delayed approach with aggregation resulting in a more efficient use of resources on multi-user chat applications; similarly, Dyck [30] referred aggregation as a commonly used technique to improve scalability in network games that could be equally effective on groupware applications. In both cases, we are talking about syntactic aggregation: blindly concatenating single packets into a larger one and possibly compressing it. Although this yields good compression levels for text-based messages, we believe a higher-level semantic aggregation would be much more effective with other types of information (e.g., raw audio or image).

Radiator is able to perform both syntactic and semantic aggregation based on the *aggregation* function implementation. Moreover, since the *aggregation* function is executed on the server, the aggregation can be reused by multiple client applications (e.g., an aggregation of geo-localized messages by city region can be reused by a traffic monitoring and a "popular spots" application).

6.2 Large-scale selective propagation

Even though Radiator is built upon the principle that every client eventually receives every message from other clients (i.e., as in a broadcast system), the propagation can occur at different times and assume different forms (i.e., more or less aggregated) depending on the current context of the users involved. Thus, it makes sense to study current solutions to the large-scale selective propagation problem.

The most popular approach is to rely on a publish-subscribe paradigm. The publish-subscribe paradigm [10] is a loosely coupled form of interaction suitable for large scale settings. It consists of three components: *publishers*, who generate and feed the content into the system; *subscribers*, who receive content based on their interests in a topic or pattern; and an infrastructure that distributes events matching subscriber interests with publishers content. Matching can be done either through a subject-based approach or a content-based approach. Subject-based approaches (e.g., Herald [31]) assume some predefined channels/topics to which both publishers and subscribers can connect. On content-based approaches (e.g., Elvin [19] and Siena [32]), subscribers can issue sophisticated queries or predicates to perform the matching on a message-by-message basis. Note that the publish-subscribe model can be implemented on a centralized topology (e.g., Elvin [19]) or a distributed topology (e.g., [33]).

Content-based matching is usually made at the syntactic level (e.g., exact comparison of keywords) but this requires that publishers and subscribers use the same terminology defeating the decoupling nature of the publish/subscribe approach [34]. Some systems overcome this limitation by performing the matching at the semantic level (e.g., matching different keywords with the same meaning such as *school* and *university*) [35,36]. Radiator does not assume a specific kind of matching — it is up to the developer to implement the *aggregability* function (responsible for matching and aggregating) using the most appropriate techniques for his specific needs. Fulcrum [37] also allows developers to provide their own matching functions but does not provide any solution for delayed propagation - the events are either discarded or propagated immediately.

Even though content-based publish-subscribe systems offer a reasonable level of flexibility and could be used to implement context-aware applications, they lack a very important feature — their predicates are fixed expressions that do not depend on the current context of the publisher or subscriber. For example, a subscriber can't issue a predicate saying "I want to receive messages from users (publishers) who are at most 1 km away from me". His only option is to issue a predicate saying "I want to receive messages from users within a 1 km radius of the coordinates (34.567, 3.566768)" assuming he's currently located in that position; but then, if he's moving, he has to continuously update the predicate. This causes severe scalability problems as the network is flooded with subscribe and unsubscribe messages since each predicate change implies an unsubscribe from the previous predicate and a subscribe to the new predicate.

Berkovsky [38] proposes a context-aware publish-subscribe system that introduces the notion of automatic subscription based on the user's location, personal preferences and interests. The system automatically translates the user's context (which may be changing frequently) into a semantic query to perform the matching. In Radiator, the *aggregability* function is responsible for this translation with two important differences: (1) it may decide to delay its propagation, and (2) it may aggregate multiple events into a single one. These two characteristics are key to the scalability improvements shown in Section 5.

Another approach for selective propagation (or multicast) of messages to multiple recipients is to use structured peer-to-peer (p2p) overlays that support many members with high scalability [11]. In general, all proposals fall into one of two categories: flooding or tree building.

The flooding approach creates a separate overlay network per multicast group. It leverages the routing information already maintained by a group overlay to broadcast messages within that group (e.g., CAN-Multicast [39]). So, for each set of recipients a separate mini-overlay network is constructed, usually a costly operation involving JOIN messages and exchange/splitting of neighborhood tables with the existing nodes. So, once

again, this approach does not scale well for frequently changing multicast trees.

The tree approach uses a single overlay and builds a spanning tree for each group, on which the multicast messages for the group are propagated. Some examples of this approach are Bayeux [40] and Scribe [12]. Multicast messages related to a group are propagated through its associated spanning tree. This form of application-level multicast leverages the object location and routing properties of the overlay network to create groups and join groups. For example, Bayeux uses Tapestry's [41] unicast routing to build a multicast tree using 4 message types: JOIN, LEAVE, TREE and PRUNE. A node joins a multicast tree by first sending a JOIN message towards the root node which responds a TREE message towards the joining node. The TREE message sets up the forwarding state at intermediate application-level nodes (in this case acting as routers). The LEAVE/PRUNE messages reverse this operation. Even though this approach scales well for stable multicast trees, the amount of exchanged messages that would be necessary to maintain highly dynamic trees (possibly changing every second) seriously reduces its scalability.

Finally, it is also possible to implement gossip-based multicast [42] by using membership protocols that manage the gossiping strategy, more specifically the nodes to whom gossip messages are sent. Again, the problem with this approach is that membership protocols are not well-suited to dynamic situations where the set of recipients change very frequently. Every time a member joins or leaves, the membership tables must be updated in multiple nodes, generating a lot of traffic.

In short, Radiator does not suffer from the rigidity of these approaches, which either assume a fixed set of matching rules or a set of recipients that does not change very frequently, making them unsuitable for context-aware applications.

6.3 Distributed stream processing systems

Traditional database management systems (DBMS) assume a pull-based model: users submit queries to the system and an answer is returned. In these systems, users play the active role, and the system plays the passive role. In contrast, stream processing systems (SPS) assume a push-based model: data is pushed into the system (as soon as it become available) and it is evaluated in response to detected events. Query answers are then pushed to a waiting user or application [43]. This push-based approach allows real-time processing of events, where query processing is performed directly on incoming messages. Therefore, unlike in DBMS, messages are processed before (or instead of) storing them [44]. Queries are built from a standard set of well-defined operators that accept input streams, transform them in some way and produce one or more output streams. For example, Aurora, a well-known SPS, supports a simple unary operator (Filter), a binary merge operator (Union), a time-bounded windowed sort (WSort), and an aggregation operator (Tumble) [45].

Distributed stream processing systems (DSPS) extend SPS to distribute their operators across multiple machines, providing several benefits: (1) stream processing performance can be scaled to handle increasing input loads; (2) it enables high availability because the machines can monitor and take over for each other when failures occur; and (3) they can take advantage of geographic and administrative distribution that is inherent to certain SPS such as wireless sensor networks [46]. Some examples of DSPS include Medusa [46], Borealis [47] and Stream Mill [48]. More recently there have been proposals for implementing DSPS on Cloud infrastructure such as Stormy [49], taking advantage of its elastic characteristics (i.e., easily adding and removing nodes from the system). Additionally, systems such as Naiad [50] combine DSPS with batch processing techniques, allowing complex incremental computations on streaming data.

DSPS can, and have already been used to, handle message context propagation in distributed systems. However, DSPS are generic systems (they can handle all kinds of data), which is simultaneously their biggest strength and weakness. In fact, since they do not know how to extract semantic meaning from context messages, they can't infer the current context of their users. All the dynamism in Radiator is a consequence of knowing and using the current context of both the sender and recipient, which not only allows more complex propagation scenarios (see Table 1) but also improves scalability in applications where context is frequently changing. To illustrate the last point, take for example how Borealis handles dynamic query modification. If we use Borealis in a context-aware system, every time the user's context changes we have to potentially change the attributes of one or several queries. Borealis supports these changes through a mechanism called control lines [44], which carries messages with revised parameters and functions for the deployed operators. Hence, as in other large-scale selective propagation systems (described in Section 6.2), there is always an overhead associated with transmitting the new parameters. In Radiator, the message itself already carries those parameters, since the current context is implicit in the message.

6.4 Summary

We now summarize the conclusions from this section into Table 6, where we compare the main feature of each system with Radiator.

It is clear from Table 6 that Radiator does not have the limitations of traditional CSCW systems on how

Table 6 Comparison between radiator and the surveyed systems

System	Main feature	How it compares to radiator
Context Toolkit [2]	Inference leads to semantic aggregation	Radiator performs semantic aggregation both at the inference layer and at the distribution layer
ReConMUC [29] and Dyck [30]	Syntactic aggregation at the distribution layer	Radiator is able to perform both syntactic and semantic aggregation at the distribution layer
Content-Based Pub-Sub	Matching is made at syntactic level or at limited semantic level (keywords with the same meaning)	In Radiator, the matching algorithm is more flexible because it can be defined as a function provided by the developer.
Fulcrum [37] (Pub-Sub)	Allows developers to provide matching functions	Radiator also allows events propagation to be delayed.
Berkovsky [38] (Pub-Sub)	Automatic subscription based on user context	Radiator adds to the automatic subscription the possibility of aggregating and delaying messages, improving scalability.
Multi-cast trees	Message propagation using P2P overlays	Radiator does not incur the cost of rebuilding P2P overlays every time the context changes.
DSPs	Efficient push-based propagation of messages	Radiator is able to extract semantic meaning from messages and infer the current context of its users, therefore adapting its propagation characteristics more efficiently.

context data is aggregated since the aggregation is performed by functions defined by the programmer. Aditionally, the aggregability function (equivalent to the matching function in Pub-Sub systems) is also defined by the programmer, allowing Radiator to have much more flexible rules and to take advantage of the current context of its users. Unlike Pub-Sub systems, Radiator also introduces the notion of delayed propagation, improving the applications' scalability. Finally, unlike generic distributed systems approaches (Pub-Sub, Multicast trees, DSPS), Radiator is able to extract semantic meaning from the exchanged messages thus does not need to perform costly "reconfiguration" operations every time the context changes.

7 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 should a message 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 chains of recipients 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 20× (when compared to the usual centralized and non-aggregated approach) without penalizing the average propagation delay in a given scenario (*partition 1%* and *vol 20*).

Regarding future work, we envisage to also use the current clients' context to build more efficient chains of recipients. The system will accept a *chainability* function (similar to the *aggregability* function) that can decide if two clients should belong into the same chain (e.g., based on their network-level proximity). We also plan to experiment parallelizing the context aggregation and propagation algorithm (Algorithm 1), since it can potentially become a bottleneck for systems with many clients.

Endnotes

[a]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.

[b]The *people* type can be useful to implement k-Anonymity [51] style privacy mechanisms; the idea is to aggregate as many messages as needed to ensure anonymity. It is out of the scope of this paper to analyze these mechanisms.

[c]The source code for the prototype is available at https://bitbucket.org/anonymousJoe/radiator.

Author details
[1]INESC-ID, Technical University of Lisbon, Opensoft, Rua Joshua Benoliel, 1, 4C, 1250 Lisbon, Portugal. [2]INESC-ID, IST, Technical University of Lisbon, Rua Alves Redol, 9, 1000 Lisbon, Portugal.

References
1. Dey A, Abowd G (1999) Towards a better understanding of context and context-awareness In: Handheld and ubiquitous computing. Springer, Berlin Heidelberg, pp 304–307. http://link.springer.com/chapter/10.1007/3-540-48157-5_29.
2. Salber D, Dey A, Abowd G (1999) Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit. ACM, New York, p 441. doi:10.1145/302979.303126, http://portal.acm.org/citation.cfm?id=302979.303126.
3. Capra L, Quercia D (2011) Middleware for social computing: a roadmap. J Internet Serv Appl 3(1): 117–125. doi:10.1007/s13174-011-0045-8, http://www.springerlink.com/index/10.1007/s13174-011-0045-8.
4. Foursquare. http://www.foursquare.com.
5. Foursquare's ups and downs, Foursquare blog. http://blog.foursquare.com/post/607883149/foursquares-ups-and-downs.
6. Waze. http://www.waze.com.
7. Highlight. http://highlig.ht/.
8. Twitter Suffers WORST Month Since October: Here's Why, Huffington Post. http://www.huffingtonpost.com/2010/06/15/twitter-down-time-explain_n_613433.html.
9. Baldauf M, Dustdar S, Rosenberg F (2007) A survey on context-aware systems. Int J Ad Hoc Ubiquitous Comput 2(4): 263. doi:10.1504/IJAHUC.2007.014070, http://www.inderscience.com/link.php?id=14070.
10. Eugster P, Felber P, Guerraoui R, Kermarrec A (2003) The many faces of publish/subscribe. ACM Comput Surv 35(2): 114–131. doi:10.1145/857076.857078, http://portal.acm.org/citation.cfm?doid=857076.857078.
11. Castro M, Jones M, Kermarrec A (2003) An evaluation of scalable application-level multicast built using peer-to-peer overlays. IEEE INFOCOM 2: 1510–1520. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1208986.
12. Castro M, Druschel P, Kermarrec A, Rowstron A (2002) SCRIBE: a large-scale and decentralized application-level multicast infrastructure. IEEE J Sel Area Comm 20(8): 1489–1499. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.299.
13. Priyantha B, Lymberopoulos D, Liu J (2011) Little rock: enabling energy efficient continuous sensing on mobile phones. IEEE Pervasive Comput 10(2): 12–15
14. Raento M, Oulasvirta a, Petit R, Toivonen H (2005) ContextPhone: a prototyping platform for context-aware mobile applications. IEEE Pervasive Comput 4(2): 51–59. doi:10.1109/MPRV.2005.29, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1427649.
15. Rana R, Chou C, Kanhere S, Bulusu N, Hu W (2010) Proceedings of the 9th ACM/IEEE international conference on information processing in sensor networks. ACM, pp 105–116. http://dl.acm.org/citation.cfm?id=1791226.
16. Miluzzo E, Lane N, Fodor K, Peterson R, Lu H (2008) 6th ACM conference on Embedded network sensor systems. ACM Press, New York, p 337. doi:10.1145/1460412.1460445, http://portal.acm.org/citation.cfm?id=1460445.
17. Saito Y, Shapiro M (2005) Optimistic replication. ACM Comput Surv 37(1): 42–81. doi:10.1145/1057977.1057980, http://dl.acm.org/citation.cfm?id=1057977.1057980.
18. Mathur A, Hall RW, Jahanian F, Prakash A, Rasmussen C (1995) The publish / subscribe paradigm for scalable group collaboration systems. Ann Arbor 1001(313): 48,109. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.1487&rep=rep1&type=pdf.
19. Segall B, Arnold D (1997) Elvin has left the building: a publish/subscribe notification service with quenching In: Proceedings of AUUG 1997. Brisbane, September, pp 3–5
20. Allavena A, Demers A, Hopcroft JE (2005) 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. ACM Press, New York, p 292. doi:10.1145/1073814.1073871, http://dl.acm.org/citation.cfm?id=1073814.1073871.
21. Hudson J, Christensen J, Kellogg W, Erickson T (2002) Proceedings of the SIGCHI conference on human factors in computing systems: changing our world, changing ourselves. ACM, pp 97–104. http://portal.acm.org/citation.cfm?id=503376.503394.
22. Introducing JSON. http://www.json.org/.
23. Bottle: Python Web Framework. http://bottlepy.org/docs/dev/.
24. Python Paste. http://pythonpaste.org/.
25. Deutsch LP (1996) GZIP file format specification version 4.3. http://tools.ietf.org/html/rfc1952.
26. Gruteser M, Grunwald D (2003) Anonymous usage of location-based services through spatial and temporal cloaking In: Proceedings of the 1st international conference on Mobile systems, applications and services - MobiSys '03, pp 31–42. doi:10.1145/1066116.1189037, http://portal.acm.org/citation.cfm?doid=1066116.1189037.
27. Sysstat documentation. http://sebastien.godard.pagesperso-orange.fr/.
28. Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27: 379–423. doi:10.1145/584091.584093, http://dl.acm.org/citation.cfm?id=584093, 9411012.
29. Alves P, Ferreira P (2011) ReConMUC - adaptable consistency requirements for efficient large-scale multi-user chat In: Proceedings of the 2011 ACM conference on computer supported cooperative work. ACM, pp 553–562
30. Dyck J, Gutwin C, Graham T, Pinelle D (2007) Beyond the LAN: techniques from network games for improving groupware performance In: Proceedings of the 2007 international ACM conference on supporting group work. ACM, pp 291–300. http://portal.acm.org/citation.cfm?id=1316669.
31. Cabrera L, Jones M, Theimer M (2001) Herald: achieving a global event notification service In: Proceedings of the eighth workshop on hot topics in operating systems, pp 87–92. doi:10.1109/HOTOS.2001.990066, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=990066.
32. Carzaniga A, Rosenblum D, Wolf A (2000) Achieving scalability and expressiveness in an Internet-scale event notification service In: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00, pp 219–227. doi:10.1145/343477.343622, http://portal.acm.org/citation.cfm?doid=343477.343622.
33. Tam D, Azimi R, Jacobsen H (2004) Building content-based publish/subscribe systems with distributed hash tables In: Databases, Information Systems, and Peer-to-Peer Computing. Springer, Berlin Heidelberg, pp 138-152. http://www.springerlink.com/index/65HRU54EERAPK0AP.pdf.
34. Burcea I, Petrovic M (2003) I know what you mean: semantic issues in Internet-scale publish/subscribe systems In: Proceedings of the SWDB 2003, pp 51–63. http://arxiv.org/abs/cs/0311047.
35. Petrovic M, Burcea I, Jacobsen H (2003) S-topss: Semantic toronto publish/subscribe system In: Proceedings of the 29th international conference on very large data bases-Volume 29. VLDB Endowment, pp 1101–1104. http://dl.acm.org/citation.cfm?id=1315559.
36. Wang J (2004) A semantic-aware publish/subscribe system with RDF patterns In: Proceedings of the 28th annual international computer software and applications conference, 2004 COMPSAC 2004, pp 141–146. doi:10.1109/CMPSAC.2004.1342818, http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1342818.
37. Boyer R (2005) Fulcrum - an open-implementation approach to internet-scale context-aware publish/subscribe In: 2005 HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on. IEEE, pp 275a–275a. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1385802.
38. Berkovsky S, Eytani Y (2005) Semantic platform for context-aware publish/subscribe M-commerce In: Applications and the Internet Workshops, 2005. Saint Workshops 2005. The 2005 Symposium on. IEEE, pp 188–191. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1620008.
39. Ratnasamy S, Francis P, Handley M (2001) A scalable content-addressable network In: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications. ACM, pp 161–172. http://dl.acm.org/citation.cfm?id=383072.

40. Zhuang S, Zhao B, Joseph A, Katz R (2001) Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination In: Proceedings of the 11th international workshop on network and OS support for digital audio and video (January). ACM, pp 11–20. http://dl.acm.org/citation.cfm?id=378347.
41. Zhao B, Kubiatowicz J, Joseph A (2001) Tapestry: an infrastructure for fault-tolerant wide-area location and routing (April). http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.111.1818.
42. Kermarrec A (2007) Gossiping in distributed systems. ACM SIGOPS Oper Syst Rev 41(5): 2–7. doi:10.1145/1317379.1317381, http://dl.acm.org/citation.cfm?id=1317381.
43. Cherniack M, Balakrishnan H, Balazinska M (2003) Scalable distributed stream processing CIDR, vol. 3, pp 257–268. 2003. http://www.eecs.harvard.edu/~mdw/course/cs260r/papers/aurora-cidr03.pdf.
44. Cetintemel U, Abadi D (2006) The Aurora and Borealis Stream Processing Engines In: Data stream management: processing high-speed data streams. Springer-Verlag. http://dumay.info/pdf/StreamProcessor/8.pdf.
45. Carney D, Çetintemel U (2002) Monitoring streams: a new class of data management applications In: Proceedings of the 28th international conference on very large data bases. VLDB Endowment, pp 215–226. http://dl.acm.org/citation.cfm?id=1287389.
46. Balazinska M, Balakrishnan H, Stonebraker M (2004) Load management and high availability in the Medusa distributed stream processing system In: Proceedings of the 2004 ACM SIGMOD international conference on management of data - SIGMOD '04, p 929. doi:10.1145/1007568.1007701, http://portal.acm.org/citation.cfm?doid=1007568.1007701.
47. Ahmad Y, Tatbul N, Xing W, Xing Y, Zdonik S, Berg B, Cetintemel U, Humphrey M, Hwang JH, Jhingran A, Maskey A, Papaemmanouil O, Rasin A (2005) Distributed operation in the Borealis stream processing engine In: Proceedings of the 2005 ACM SIGMOD international conference on management of data - SIGMOD '05, p 882. doi:10.1145/1066157.1066274, http://portal.acm.org/citation.cfm?doid=1066157.1066274.
48. Thakkar H, Mozafari B, Zaniolo C (2008) Designing an inductive data stream management system: the stream mill experience In: Proceedings of the 2nd international workshop on scalable stream processing system. ACM, pp 79–88. http://dl.acm.org/citation.cfm?id=1379286.
49. Loesing S, Hentschel M (2012) Stormy: an elastic and highly available streaming service in the cloud In: Proceedings of the 2012 joint EDBT/ICDT workshops ACM. http://dl.acm.org/citation.cfm?id=2320789.
50. Murray D, McSherry F, Isaacs R (2013) Naiad: a timely dataflow system In: Proceedings of the twenty-fourth ACM symposium on operating systems principles ACM. http://dl.acm.org/citation.cfm?id=2522738.
51. Sweeney L (2002) k-anonymity: a model for protecting privacy. Int J Uncertainty Fuzziness Knowl- Based Syst 10(5): 557–570. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.391.6292.