Electronic Communications of the EASST
Volume 43 (2011)

Proceedings of the 4th International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2011)

Volatile Sets: Event-driven Collections for Mobile Ad-Hoc Applications

Dries Harnie, Elisa Gonzalez Boix, Andoni Lombide Carreton, Christophe Scholliers and Wolfgang De Meuter

12 pages

Guest Editors: Gabriel Hermosillo, Russel Nzekwa, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Volatile Sets: Event-driven Collections for Mobile Ad-Hoc Applications

Dries Harnie[*], Elisa Gonzalez Boix[*], Andoni Lombide Carreton[†], Christophe Scholliers[†] and Wolfgang De Meuter

{dharnie,egonzale,alombide,cfscholl,wdmeuter}@vub.ac.be
Software Languages Lab
Vrije Universiteit Brussel, Brussels, Belgium

Abstract: In mobile peer-to-peer applications, a common pattern is to maintain a collection of remotely-hosted objects. Traditional approaches require programmers to manually track the connectivity state of these remote objects and add them or remove them from local collections on a per-object basis. Because this happens concurrently with the rest of the application code, it hinders the composability of such collections and leads to subtle and hard to find bugs. In this paper, we propose an abstraction called *volatile sets* that allows the contents of the set to be specified *intensionally*. Additionally, volatile sets offer an event-driven API that signals when remote objects appear, disappear or change. Finally, volatile sets can be easily and efficiently composed through traditional set operations. We show how volatile sets ease the development of a collaborative peer-to-peer drawing application.

Keywords: mobile applications, ambient-oriented programming, distributed collections, language abstractions

1 Introduction

In recent years we have witnessed a tremendous growth in the number of mobile applications that are being developed. In a keynote at the Mobile World Congress 2011 Eric Schmidt announced that the Android market has reached 150,000 applications, a threefold growth over the previous year. This growth is driven by the recent advances in wireless technologies and miniaturization of hardware such as increased battery life, more computational power, near field communication (NFC). This paper focuses on developing applications that run on wirelessly interconnected mobile devices. Such mobile applications are decentralized by nature and allow the user to communicate and interact with nearby users or devices in their surrounding in an without relying on central infrastructure.

The structure of mobile ad hoc applications is very similar to traditional distributed applications: first the application seeks out interesting remote objects to include into the interaction (discovery), then it participates in the interaction itself (maintenance), and finally at a certain moment in time the interaction is stopped and the references to the remote objects are removed from the application's data structures (garbage collecting). These phases, **discovery**, **mainte-**

[*] Funded by the Prospective Research For Brussels program of IWOIB-IRSIB.
[†] Funded by a doctoral scholarship of the IWT-Flanders, Belgium.

nance and **garbage collecting**, look very sequential in nature, but they actually run concurrently. Most mobile ad hoc network programming middleware provide the programmer with a means to discover objects in the environment. Due to the nature of the discovery process, objects are discovered and reported back to the programmer one by one. The programmer will then add each newly discovered object to the application data structures. Because this discovery process runs concurrently with the main program, new objects can be added at any time. At the same time, already discovered objects can get disconnected at any time, because the devices that host them can move out of communication range. During the implementation of a number of mobile ad hoc applications, such as Flikken [SGDD10] and Urbiflock [GLS+11], we found ourselves writing recurring code to keep these collections synchronized with context changes in the environment such as disconnections of devices, changes on the properties of the discovered objects, etc. Additionally, interactions with one communication partner run concurrently with the garbage collecting of other remote objects. Needless to say the concurrent nature of these applications has a huge impact on the application.

These issues open up a class of bugs that are hard to reproduce and hard to find. They form the main motivation to explore data structures that represent collections of remote (volatile) objects to allow programmers to abstract from such complexities when writing mobile ad hoc applications. In this paper, we propose *volatile sets*: an event-driven data structure to maintain an intensionally defined set of objects in the environment, keeps track of changes in these objects, and can be composed with other volatile sets using the standard mathematical set operations. Interested remote parties can be notified whenever changes occur to the volatile set or to the individual objects they contain.

In the rest of this paper, we will show the problems that led us to explore volatile sets (Section 2), how the volatile set abstraction tackles these problems at a high level, and how programmers can use volatile sets to quickly develop mobile applications (Section 3). We will explain how volatile sets are implemented (Section 4), show a discussion of related work and how we intend to improve volatile sets in the future (Section 5).

2 Problem Statement

In mobile ad hoc networks, the number of devices participating in an interaction is not known a priori, but it varies as devices join and leave the network as they move about. Typically, applications are interested in communicating only with a specific group of those *proximate* objects which are *discovered* at runtime. For example, in a chat application, users can join or leave a chat room at any moment in time. In order to reflect the communication state of the users in the chat room and to allow communication with them, the programmer has to manually maintain a collection of remote objects. Such a collection of remote objects that is continuously fluctuating because of the volatile nature of the connections to the devices hosting these objects is what we call a *volatile set*.

At the software level, we have identified two ad hoc ways commonly used to implement volatile sets. A first approach when using an object-oriented distributed programming language is to discover and store the remote objects in an internal data structure. Once discovered the programmer must manually iterate over references to these remote objects to communicate with

them by means of remote method invocations or asynchronous message passing. All over the program it is the programmer's responsibility to make sure that these objects are still connected by making use of try-catch blocks or other failure handling.

A second strategy is to use an event-based distributed system such as a publish/subscribe architecture. In this case, volatile sets become groups of objects classified under a topic (topic-based publish/subscribe) and potentially filtered on their content (content-based publish/subscribe) using predicates [EFGA03]. Although this allows subscribing to the appearance and modification of interesting objects *intensionally*, objects are *detached* from their publishers: if a publisher goes offline its objects still remain in the network. This obliges programmers to bypass the publish/subscribe middleware periodically to detect whether the publishers are still connected and verify that their published objects are still "live" (i.e. in case of a crash). Additionally, the events signaled by the publish/subscribe middleware must still be manually converted to additions and removals on local collections. This hinders straightforward and efficient composition of such volatile sets.

As such, both solutions do not deal with the many issues that arise when maintaining and communicating with a collection of *volatile* objects. In what follows, we detail these issues.

P1. Time-varying, volatile collections. In a decentralized peer-to-peer setting, applications need to first discover other applications and services running in the environment to interact with them. A natural approach to dealing with this problem is to maintain a set of "currently available" objects in which the application is interested. Because devices can appear and disappear at any moment in time, we regard a set of such objects as highly *volatile*. Therefore, specifying the contents of the set *extensionally* (i.e. on a per-element basis) is problematic as the contents of the set can change at any point in time.

A second part of this problem is not the *specification* of the collections, but the *interaction* with these collections. To interact with these collections, the programmer must use constructs such as indexes, iterators, associations, etc. These constructs exhibit undefined behavior if the collection changes underneath them and thus do not map well to time-varying, volatile collections.

P2. Composition of volatile collections. A natural operation on collections of objects is to compose them with other, similar collections, using the standard set operators (union, intersection, difference). Composing volatile collections is not an atomic action: volatile sets can grow or shrink several times while their union or intersection is being computed.

Additionally, as the constituent collections evolve, the composed collection diverges from "the union of these two collections". This means that programmers have to write additional code to ensure the composed collection remains synchronized with its constituent collections. It also implies that compositions of volatile collections are themselves volatile. This requirement is a serious deviation from the traditional notion of composing collections, where a composed collection does not bear any relationship to its constituents.

P3. Synchronizing state replicated across several devices. Objects residing in a volatile collection are local copies of objects published by a remote device. Whenever the owner changes an object, the changes should be synchronized to all the copies spread across the network. This

can be accomplished through a publish/subscribe framework. For example, a user changing her nickname in a chat application should be represented in the other chat applications.

Volatile collections take this one step further: some changes made to an object can result in it being removed from or added to volatile sets with certain predicates. For example, in a chat application, a user that decides to move to another chat room will change the "current chat room" property. This change will cause some volatile collections to remove her user object from the old chat room, while other collections will add her to the new chat room. Thus, remote parties defining a volatile set should have a means to subscribe to changes on properties of already-published objects.

Programmers find themselves implementing an abstraction to express these volatile collections over and over. In the next section, we introduce a novel abstraction representing such volatile collections named *volatile sets* which solve the issues outlined above.

3 Volatile Sets

In this section, we explain volatile sets and how they address the problems when maintaining and composing groups of volatile objects. We solve problem **P1** by allowing developers to specify an *intensional definition* of the objects they want to interact with, together with an *event-driven API* that signals events whenever an object is added to or removed from the volatile set. Composing volatile sets (**P2**) does not require breaking the abstraction of the sets by looking at their contents at a certain moment in time, but instead can happen using mathematical *set operations* (intersection, union, difference) that rely only on the intensional definition of these sets. Finally, the event-driven API allows these volatile sets to *signal modifications* that are performed to the elements they contain (**P3**).

We have prototyped volatile sets in the distributed programming language AmbientTalk [VMG+07]. The rest of this section describes how each problem is solved by illustrating volatile sets in the context of a concrete application called *weScribble*.

3.1 The weScribble Application

In this section we describe how volatile sets are used in the implementation of a collaborative drawing application for the Android platform, called weScribble[1]. The application allows users to dynamically participate in a drawing session with other people nearby, assuming no other infrastructure than mobile Android devices and wireless ad hoc connections between these devices.

At startup, weScribble presents the user with a list of drawing sessions available in the environment with an indication of the amount of people drawing in each session. The user can then join an existing session or create a new one. A drawing session consists of a number of participants and a shared canvas on which they can draw. When a user joins a drawing session, the application synchronizes the canvas with the existing participants to fetch the drawing elements already drawn and displays them on the screen. Then, the user can draw on the canvas of that session. Those changes on the canvas are propagated to the other participants of the session who

[1] WeScribble is available from the Android Market at http://bit.ly/eOxpLg.

update their canvas accordingly. If a user temporarily disconnects from the network, he or she can keep drawing and those changes will be synchronized with the other users in the session upon reconnection.

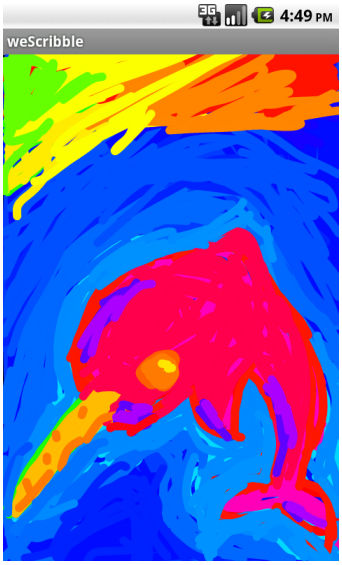

Figure 1: The weScribble application for Android.

WeScribble uses volatile sets for two different purposes: first, a session consists of a volatile set that contains the shapes drawn by users. Whenever a user adds a new shape to a session, it will be added to the other participants' volatile sets and be shown on their screens. The event-driven interface makes it easy for the weScribble programmer to handle changes published by other people. At the same time, the fact that volatile sets are also plain collections makes both the initial synchronization process and merging after a disconnection relatively painless.

3.2 Volatile Sets at Work

In this section we explain how volatile sets tackle the problems listed in the previous section in the implementation of weScribble.

S1. Intensionally defined volatile collections Volatile sets allow developers to discover specific remote objects based on custom criteria. These criteria come in two granularities: objects can be annotated with *type tags* to coarsely select objects related to the application and *predicates* to select objects on an individual basis. Type tags are a lightweight classification mechanism to categorise remote objects explicitly by means of a nominal type. They can best be compared to a topic in publish/subscribe terminology or marker interfaces in Java. Programmers can use both type tags and predicates to give an intensional definition of a volatile set.

WeScribble uses volatile sets to collect all the shapes that belong to a drawing session in which the user is currently participating:

```
def currentSessionShapes := volatileSetOf: Shape where: { |s| s.session == mySession };
```

`volatileSetOf:where:` creates a set of objects tagged with the type tag `Shape` for which the predicate is true. Whenever a `Shape` object is discovered in the network, the predicate is first applied to it and if it returns true the shape is added to the volatile set. The predicate given is executed locally and can be arbitrarily complex. This solves one part of problem **P1**, namely selecting specific objects that meet certain criteria from the ambient.

Note that a volatile set abstracts the contents of the collection by presenting the set as an opaque data structure. However, sometimes (e.g. for debugging purposes) the current contents of such a volatile set need to be accessed. In this case, developers can take a peek at the contents of the volatile set at a certain moment in time by means of the `snapshot()` method.

To solve the second part of **P1** (event-driven notification of changes to the set), we provide developers with a means to react whenever the volatile set changes by registering a listener. The weScribble application registers such a listener to render new drawings on the screen as follows:

```
def listener := object: {
    def elementAdded(shape)   { GUI.show(shape) };
    def elementRemoved(shape) { GUI.hide(shape) };
};
currentSessionShapes.addListener(listener);
```

Here we assume the existence of a `GUI` object that knows how to render the individual elements in the volatile set, `currentSessionShapes`. The `elementAdded` method will be invoked whenever an object is discovered that satisfies the intensional definition of the `currentSessionShapes` volatile set. The `elementRemoved` method will be invoked whenever one of the objects in the set is removed. In general, this can happen due to two different reasons: either the device hosting the removed object disconnected from the network, or the device hosting the object performed some changes on the object such that it no longer satisfies the intensional definition of the volatile set. In weScribble, `elementRemoved` is called when a participant disconnects from the drawing session either due to a network disconnection or because he leaves this drawing session to join another one. This solves the second part the problem **P1**.

S2. Composition of event-driven collections As mentioned earlier, one of the problems arising with volatile data structures is that their volatility makes them hard to compose in meaningful structures, because it requires accessing the individual elements of the set. Volatile sets provide mathematical set operators to compose volatile sets. The volatile and event-driven nature of volatile sets is at the heart of these set operators.

In weScribble, for example, organizing all drawn shapes in a single volatile set makes it difficult to track which user is drawing what. To cater for this issue, weScribble organizes all drawn shapes in volatile sets dedicated to a single user in the session. These volatile sets are specified as follows:

```
def getShapesOfUser(user) { volatileSetOf: Shape where: { |s| s.user == user } };
```

In the application's GUI, the user can tick other users in a list to choose from whom he wants to display drawings. In response, the user's screen displays the contents of the volatile set that is

the union of all shapes of all users that the user ticked in the GUI. This volatile set is composed using the `union:` operator as shown below.

```
def unionedSet := VolatileSet.new();
tickedUsers.each: { |user|
    unionedSet := unionedSet.union: getShapesOfUser(user);
};
```

Important to note is that the `union:` operator never requires the programmer to deal with individual set elements, nor does it require applying all the predicates in the intensional definitions of the constituting volatile sets to every single discovered or changed object. Instead, it extracts the intensional definitions of the unioned sets and constructs a composed intensional definition for the unioned set, that is applied as a whole to newly discovered or changed objects.

S3. Synchronizing state replicated across several devices In drawing editors, users can change properties of shapes after creation e.g. color, size, position. Those changes are a good example of local changes that need to be synchronized with remote parties. To capture changes to objects belonging to a volatile set, we require developers to publish objects in the network as *synchronizing isolates*. Synchronizing isolates are special objects that have no surrounding lexical scope (i.e. similar to *structs*, but they can have methods defined on them). This way, they can be easily *copied* over the network and *cached* in the volatile sets of subscribed remote peers. This design allows developers to model elements of a volatile set in an object-oriented fashion while the underlying implementation implicitly synchronizes state across the different devices using the volatile set.

Although every device hosting a volatile set has the illusion that it stores a local copy of the elements in the collection, elements can only be modified by their *owner*: the device that created them and published them to the network. Finally, they allow listeners to be registered on them, such that whenever the original host of a synchronizing isolate *modifies* it, all parties hosting a copy of this synchronizing isolate are notified and their corresponding listeners are invoked. These notifications are signaled through reliable, asynchronous messages to make sure that intermittent network connectivity does not cause missed updates.

Below is an example of a `rectangle` synchronizing isolate that can be used by the weScribble application.

```
def rectangle := syncIsolate: {
    def type  := 'Rectangle;
    def color := 'Blue;
    def p1 := Point.new(10, 10);
    def p2 := Point.new(80, 80);

    def drawOn(canvas) {
        // Draw the rectangle on canvas
    };
};
publish: rectangle as: Shape;
```

The above `rectangle` can be discovered by a volatile set that collects `Shape` objects. Running weScribble applications can redraw such rectangles or other shapes whenever they change (i.e. whenever *any* of their fields are mutated) by registering a listener that listens for changes to these synchronizing isolate objects in the volatile set, as shown below:

```
def changeListener := object: {
    def elementChanged(old, changed) {
        GUI.canvas.remove(old);
        changed.drawOn(GUI.canvas);
    };
};
currentSessionShapes.addListener(changeListener);
```

Here we create a `changeListener` listener object of which the `elementChanged()` method is invoked every time an element of the volatile set changes. Synchronizing isolates thus solve problem **P3**.

We already briefly hinted how synchronizing isolates are published and discovered and how modifications performed by the publisher are signaled to subscribed parties. In the section below, we further elaborate on the implementation of volatile sets and synchronizing isolates .

4 Implementation

In this section we will explain how volatile sets are organized internally. As previously mentioned, we have prototyped volatile sets in the distributed object-oriented programming language AmbientTalk. They have been built on top of the discovery and communication mechanism already present in AmbientTalk. It is possible to implement volatile sets in another language with object discovery and asynchronous communication like Java, but the complexity of the implementation is greatly increased.

In AmbientTalk, applications publish objects to the environment using type tags. Other applications can discover these objects by using the type tag. AmbientTalk objects are classless and come in two variants: objects and isolates. Whenever an object is discovered, the application receives a *remote reference* to it. Any message sent to a far reference that is disconnected is buffered locally until the device hosting the far reference rejoins the network. Isolates, on the other hand, are objects that have no surrounding lexical scope and are sent by copy over the network.

4.1 Volatile Sets

A volatile set is implemented[2] as an advanced bookkeeping abstraction over an object discovery handler. AmbientTalk offers primitive event handlers that are fired whenever objects of a given type tag are discovered in the environment. These event handlers yield far references which are first tested for any property required by the volatile set and then added to a local data structure. Additionally, the volatile set installs disconnection and reconnection listeners on each far reference, which are signaled whenever the device hosting the remote object disconnects from the network or reconnects to it. Each time a far reference disconnects or reconnects, the far reference is removed from or added to the local data structure and the appropriate event listeners are fired.

In a typical environment, several disconnections and reconnections can happen at the same time, which can cause race conditions and other concurrency problems in languages like Java or C#. AmbientTalk avoids these concurrency problems because its concurrency model is based on event loops (like in JavaScript) rather than threads. Whenever an discovery or disconnection

[2] The volatile sets implementation can be downloaded with the AmbientTalk distribution at http://tiny.cc/dd8bb.

event is triggered, it is reified into a message and inserted into the mailbox of the actor hosting the volatile set. The actor then processes the messages one by one and updates the volatile set. As this effectively serializes these events, no concurrency problems are possible.

4.2 Synchronizing Isolates

An important aspect of volatile sets is that the elements of the volatile set are kept synchronized when the state of remote objects changes. In current state of the art object-oriented distributed systems, objects are passed either by reference or by copy. The elements in a volatile set are passed by copy so that the programmer can locally read the state of the object, but whenever the original object is changed the element in the set should be changed as well. In order to obtain this behavior a new abstraction called synchronizing isolates was created.

A device publishing a synchronizing isolate to the network under a certain type tag is called a *publisher*. A device creating a volatile set for this type tag is called a *subscriber* for synchronizing isolates published under this type tag. A schematic overview of a number of published and subscribed synchronizing isolates is shown in Figure 2.

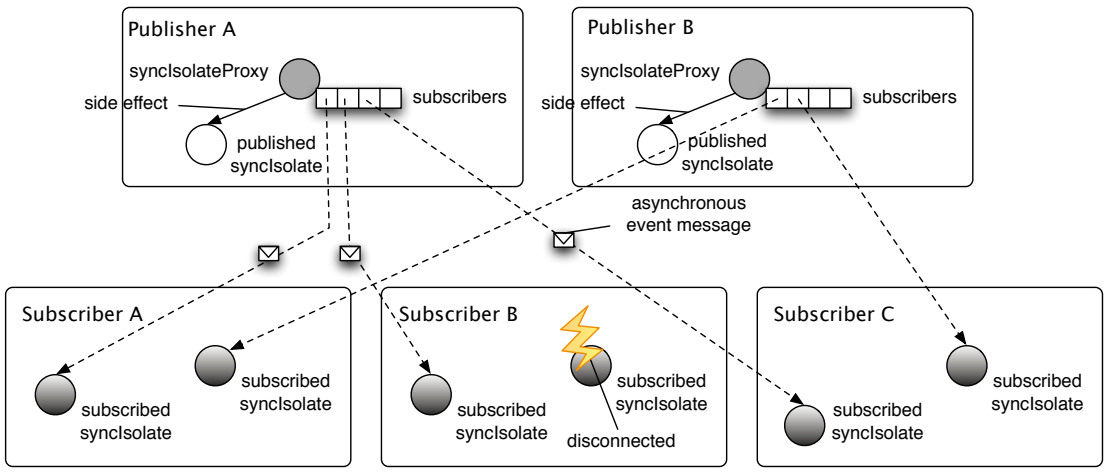

Figure 2: Schematic overview of synchronizing isolates.

Publisher A and B host and publish a synchronizing isolate into the network while subscribers A, B and C are subscribed to some of them. Whenever a synchronizing isolate is discovered, the subscriber sends a subscription message with a reference to a listener. Whenever the synchronizing isolate is modified, all such listeners are signaled using an asynchronous event. This happens by wrapping the synchronizing isolate object in a proxy object that traps all side effects and transforms them into asynchronous event messages that are signaled to all subscribers associated with the synchronizing isolate.

Note that developers using a synchronizing isolate are not exposed to all these registrations and will just observe that the discovered objects change from time to time. However, when the programmer wants to be notified of these changes he can also register a custom observer in the

volatile set as shown in the previous section.

5 Related Work

There has been much research into group abstractions for mobile applications in the past, but most of this research focuses on group communication which is not the focus of this work. In this section, we discuss related work that focuses on organizing remote objects in intermittently connected peer-to-peer applications.

Distributed Asynchronous Collections (DACs) [EGS00] is the closest work to volatile sets. DACs were originally devised as a way to marry publish-subscribe systems to traditional collection frameworks. They allow developers to subscribe to additions and removals that occur in the collections. However, DACs offer no support for tracking the connectivity of publishers that export objects, so the programmer still has to track this manually.

Tuple spaces [Gel85] allow distributed parties to publish tuples to a conceptually shared memory and remove tuples from it. Technologies such as LIME [MPR01] even allow distinct tuple spaces to merge if users are close to each other and allows programmers to define reactions on the appearance or disappearance of tuples. However, tuple spaces do not support the direct modification of tuples: tuples have to be removed first and new versions reinserted later. This requires application developers to write additional code to watch these remove/insert event pairs individually. Additionally, there is no support for creating a data structure from a tuple space, which forces programmers to update the derived collections manually.

Ambient references [CDM+06] allow discovering and communicating with homogenous groups of references to objects in the environment that change over time. This abstraction takes care of monitoring any disconnections and reconnections in the environment and even allows developers to take a snapshot of the current state. Ambient references do not allow programmerss to react on objects, joining, leaving or being modified in the group. Additionally, they are not composable.

SpatialViews [NKI04] allow developers to interact with services running on nearby devices that implement the same Java interface. Developers can write code that iterates over all the services available along two dimensions: space (services entering a certain location are added to the group) and time (services being discovered over a certain timespan are added to the group). If a portion of code requires a service that is not available locally, the running program is moved to a node that hosts this service. SpatialViews do not notify client code about services being removed from the group or changes to remote objects.

M2MI [KB02] introduces a data type called a *handle* that is used to denote a dynamic group of remote Java objects of the same interface. In M2MI, an omnihandle refers to all proximate objects of a certain interface. Such an interface is equivalent to the role of our type tags. Applications can send a message to an omnihandle which means, that every object implementing that interface, calls the corresponding method. However, M2MI invocations on objects that are not immediately reachable are lost. This is not the case in volatile sets. The synchronized isolate abstraction ensure that changes on the elements of a volatile set are guaranteed to be reflected. If a device disconnects, all its synchronized isolates will be removed from the existing volatile sets. However, if the device moves back online, its elements will be then added to the corresponding volatile sets reflecting changes on state effectuated while disconnected.

6 Conclusion and Future Work

In this paper we discussed a pattern which is ubiquitous in mobile ad hoc applications, namely discovering and maintaining collections of remote objects that evolve organically over time due to remote events such as user interaction on a remote device. Moreover, these remote objects might get disconnected at any moment in time. Maintaining these collections requires extra developer effort because developers need to manually reflect *changes* to the content of the collection or the individual elements. To solve this issues, we proposed *volatile sets*, a composable collection type which allows developers to react to changes in both the structure and its contents. Volatile sets allow programmers to intensionally define a set of remotely hosted objects and be notified upon object addition, removal, modification using an event-driven API. Volatile sets can be composed by means of traditional set operators.

As future work, we would like to introduce different types of collections that take additional constraints into account, such as the ordering of elements. Another improvement could be to batch the signaled change event messages sent to all subscribers in such a way that the amount of events and hence network traffic is reduced. Finally, we are investigating decentralized replication and consistency algorithms [TPST98, EM97, CK00] such that clients can modify the objects in the volatile sets and the changes are synchronized with the publisher or directly to other peers that have also discovered those objects.

Bibliography

[CDM+06] T. V. Cutsem, J. Dedecker, S. Mostinckx, E. Gonzalez Boix, T. D'Hondt, W. D. Meuter. Ambient references: addressing objects in mobile networks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. Pp. 986–997. ACM Press, New York, NY, USA, 2006.
doi:http://doi.acm.org/10.1145/1176617.1176757

[CK00] U. Cetintemell, P. Keleher. Light-Weight Currency Management Mechanisms in Deno. *International Workshop on Research Issues in Data Engineering* 0:17, 2000.
doi:http://doi.ieeecomputersociety.org/10.1109/RIDE.2000.836495

[EFGA03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.Kermarrec. The many faces of publish/-subscribe. *ACM Computing Survey* 35(2):114–131, 2003.
doi:http://doi.acm.org/10.1145/857076.857078

[EGS00] P. Eugster, R. Guerraoui, J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. *ECOOP 2000 – Object-Oriented Programming*, pp. 252–276, 2000.

[EM97] W. K. Edwards, E. D. Mynatt. Timewarp: techniques for autonomous collaboration. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '97, pp. 218–225. ACM, New York, NY, USA, 1997.

doi:http://doi.acm.org/10.1145/258549.258710
http://doi.acm.org/10.1145/258549.258710

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7(1):80–112, Jan 1985.

[GLS+11] E. Gonzalez Boix, A. Lombide Carreton, C. Scholliers, T. Van Cutsem, W. De Meuter, T. D'Hondt. Flocks: Enabling Dynamic Group Interactions in Mobile Social Networking Applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), Taichung, Taiwan, March 21–25, 2011*. Volume 1, pp. 425–432. ACM, 2011.

[KB02] A. Kaminsky, H.-P. Bischof. Many-to-Many Invocation: a new object oriented paradigm for ad hoc collaborative systems. In *OOPSLA 2002*. Pp. 72–73. ACM Press, 2002.
doi:http://doi.acm.org/10.1145/985072.985109

[MPR01] A. Murphy, G. Picco, G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*. Pp. 524–536. IEEE Computer Society, 2001.

[NKI04] Y. Ni, U. Kremer, L. Iftode. Spatial views: Space-aware programming for networks of embedded systems. *Languages and Compilers for Parallel Computing*, pp. 258–272, 2004.

[SGDD10] C. Scholliers, E. Gonzalez Boix, W. De Meuter, T. D'Hondt. Context-aware tuples for the ambient. *On the Move to Meaningful Internet Systems, OTM 2010*, pp. 745–763, 2010.

[TPST98] D. B. Terry, K. Petersen, M. J. Spreitzer, M. M. Theimer. The case for non-transparent replication: Examples from Bayou. *IEEE Data Engineering Bulletin* 21(4):12–20, december 1998.

[VMG+07] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter. AmbientTalk: object-oriented event-driven programming in Mobile Ad hoc Networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*. Pp. 3–12. IEEE Computer Society, 2007.