Ambient Clouds: Reactive Asynchronous Collections for Mobile Ad Hoc Network Applications

Kevin Pinte, Andoni Lombide Carreton,
Elisa Gonzalez Boix, and Wolfgang De Meuter

Software Languages Lab., Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussel, Belgium
{kpinte,alombide,egonzale,wdmeuter}@vub.ac.be
http://soft.vub.ac.be

Abstract. In MANET applications, a common pattern is to maintain and query time-varying collections of remote objects. Traditional approaches require programmers to manually track the connectivity state of these remote objects and adding or removing them from local collections on a per-object basis. Queries over these collections have to be manually recomputed whenever the collection or its elements change.

The code for maintaining these ad-hoc collections is scattered across the application code and leads to bugs hindering the application development process. In this paper, we propose an object-oriented abstraction called *ambient clouds*: a collection of objects whose contents are implicitly updated when changes occur. Ambient clouds can be queried and composed using *reactive* standard query operators. We show how ambient clouds ease the development of a collaborative peer-to-peer drawing application.

Keywords: collection, mobile ad hoc network, peer-to-peer application, language abstraction.

1 Introduction

The steep increase in popularity of mobile devices has yielded a market for applications running on mobile ad hoc networks (MANETs). MANET applications assume no fixed infrastructure and spontaneously engage in interaction when the devices they run on are in communication range. These applications communicate over wireless networks, for example Wi-Fi Direct or Bluetooth. Using mainstream programming languages such applications are usually conceived as distributed object-oriented applications, coordinating their actions by exchanging objects.

In MANETs, 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 *remote* objects which are *discovered* at runtime. For example,

J. Dowling and F. Taïani (Eds.): DAIS 2013, LNCS 7891, pp. 85–98, 2013.

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. In MANETs, the nature of the connection to the devices hosting these objects is volatile. Such a collection of remote objects is continuously fluctuating because of the volatile nature of the connections to the devices hosting these objects.

At the software level, we can identify two ad hoc ways commonly used to implement such collections. A first approach when using a distributed object-oriented programming language is to discover and store the remote objects in a local collection. Once discovered, the programmer must manually iterate over the collection's content and communicate with the stored remote objects by means of remote method invocations or asynchronous message passing. 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 mechanisms.

A second strategy is to employ an event-based distributed model such as a publish/subscribe architecture. In this case, the collections become groups of objects classified under a topic and potentially filtered on their content using predicates [1]. However, publish/subscribe middlewares abstract the network connectivity between the publisher and subscriber. This obliges programmers to bypass the middleware periodically to detect whether the publishers are still connected and verify that their published objects are still "alive" (i.e., in case of a crash). Additionally, the events signalled by the publish/subscribe middleware must still be manually converted to additions and removals on local collections. This hinders straightforward and efficient querying and composition of such collections.

None of both solutions provides adequate means to create, maintain and query collections of remote objects. This leads the programmer to write boilerplate code that is scattered throughout the actual application code. In this paper, we propose *ambient clouds*: a reactive asynchronous collection abstraction to maintain and query collections of remote objects in MANET applications. Additionally, ambient clouds provide *reactive* standard query operators. Querying or composing ambient clouds using these operators constructs a chain of dependent result collections. The operators observe the collections they were applied to. Changes in either the composition of the constituent elements are implicitly and incrementally reflected throughout the chain of dependent result collections.

In the remainder of this paper, we show the problems that led us to explore ambient clouds (section 2), how ambient clouds tackle these problems at a high level, and how programmers can use ambient clouds to quickly develop mobile applications (section 3). We explain how ambient clouds are implemented (section 4) and how they are used in the collaborative drawing application, called *weScribble* (section 5). Subsequently, we discuss related work (section 6) and, finally, conclude this paper and suggest how we intend to improve ambient clouds in the future (section 7).

2 Problem Statement

In what follows, we detail the issues that can be identified when developing applications that deal with collections of remote objects. We illustrate the problems using a chat application as a running example.

The chat application presents the user with the option to create a chatroom or join an existing one. When the user enters a chatroom, he or she is presented with a list of users that joined this room and are currently in communication range. The application also shows the messages that belong to the chatroom and users can choose to ignore messages from certain other users.

P1. Volatile Collections. MANET applications discover other applications and services running in the environment to interact with them. The applications typically maintain a collection of "currently available" objects in which the application is interested. Since devices hosting MANET applications can appear and disappear at any moment in time, we regard these collections of remote objects as highly *volatile*. Therefore, specifying the contents of the collection *extensionally* (i.e., on a per-element basis) is problematic as the contents of the collections can change at any point in time. With current techniques, the programmer is left to manually synchronise the contents of these volatile collections of remote objects.

To interact with these collections, the programmer typically uses constructs such as indices and iterators. These constructs do not map well to volatile collections of remote objects because the collection changes underneath them. For example, the chat application maintains a list of currently co-located users. As users move about, the composition of this list changes dynamically.

P2. Querying and Composing Collections. A natural operation on collections of objects is to apply operators to compose them with other collections or query them. For example, the chat application applies a filter operation on the list of users to display only those that reside in the chosen chatroom.

Querying and composing volatile collections is not an *atomic action*: collections can grow or shrink several times while a composition or query is being computed. Furthermore, when employing asynchronous method invocation to communicate with remote objects, the results of a query over the elements of a collection may not be available instantaneously.

As the composition of a collection evolves, the initial result of applying an operator diverges from the current state. This means that programmers have to write additional code to ensure results from applying operators remain synchronised with the collection they were applied to. It also implies that compositions of collections resulting from queries over such volatile collections are themselves volatile. This requirement is a serious deviation from the traditional notion of querying collections, where the result of a query does not bear any relationship to the target collection.

P3. Propagating State Changes. When employing a distributed object-oriented model objects are either exchanged "by copy" or "by reference". This results in a collection of respectively local copies of the objects published by a remote device, or remote references to these objects. In the former case, whenever the owner changes an object, the changes should be propagated to all the copies spread across the network. In the latter case, whenever an object is changed, collections containing a reference to this object should be notified of this change. For example, when a user changes his nickname in the chat application, this should be reflected in the buddy list on the applications of other users.

This propagation of state changes can be accomplished through, for example, a publish/subscribe framework. However, changes made to an object can result in it being removed from or added to the result of a certain operation on the collection. In the chat application, a user that decides to move to another chat room changes the "current chat room" property. This change causes some collections to remove this user object from the old chat room, while other collections add the user to the new chat room. Thus, collections containing remote objects should have a means to subscribe to changes on properties of their constituent objects.

3 Ambient Clouds

In this section, we introduce a novel abstraction representing collections of remote objects named *ambient clouds*. Ambient clouds are an object-oriented abstraction that tackles the issues outlined above by combining event-driven interaction, based on a publish/subscribe model, and reactive programming.

We solve problem ***P1*** by allowing developers to specify an ambient cloud of a certain type of objects they want to interact with. The type of the objects acts as an initial filter in order to collect objects of interest. An *event-driven API* signals events whenever an object is added to or removed from the ambient cloud. To address problem ***P2*** we provide *reactive* standard query operators to query and compose ambient clouds. Any computation performed using such an operator is re-executed as the collection changes. The operators automatically handle asynchronous operations. This does not require breaking the abstraction of the collections by looking at their contents at a certain moment in time. To tackle ***P3*** we model object pass-by-copy and pass-by-reference semantics using *reactive objects* and *reactive isolates*. These are object-oriented reactive values of which the state changes over time. These changes are observed and the event-driven API *signals state modification events* to the collections in which they are contained.

3.1 AmbientClouds at Work

We have prototyped ambient clouds in the distributed programming language AmbientTalk [2]. AmbientTalk is an experimental programming language tailored towards developing peer-to-peer applications that operate in MANETs.

We now describe the language constructs provided to create and interact with ambient clouds in the context of the chat application.

Ambient clouds coarsely collect objects related to the application using a *type tag*. 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. Below we create the ambient cloud of all co-located users using the **`cloudOf:`** construct and passing it the `ChatUser` type tag as argument.

```
deftype ChatUser;
def users := cloudOf: ChatUser
```

We define an object representing a user of the chat application by means of the **`object:`** construct. A user has an identifier, a nickname and an attribute containing the name of the selected chatroom. To publish this object in the network as a `ChatUser` we use the **`export:as:`** construct of AmbientTalk.

```
def me := object: {
  def id := 123;
  def nickname := "Kevin";
  def currentChatroom := "purple" };
export: me as: ChatUser
```

Ambient clouds continuously synchronise their composition as devices move in and out of range. Users that leave communication range are automatically removed from the collection, users that appear in range are added automatically.

Note that users are represented as regular objects, they are published in the network by reference. The ambient cloud of users thus contains remote objects references that can be contacted by sending it asynchronous messages. Later we will show an example of an ambient cloud of remote objects passed by copy.

We further refine the ambient cloud of users using the reactive standard query operators we provide for ambient clouds:

```
def usersInRoom := users.where: { |user|
  equals(user←currentChatroom(), me.currentChatroom) };
def nicknames := usersInRoom.select: { |user| user←nickname() }
```

In this example we first filter the ambient cloud of users, selecting only users that reside in the same chatroom. To this end we use the **`where:`** method that takes a predicate as argument. After the filter operation we select the nicknames of the users using the **`select:`** operator, for example to display them in the GUI.

In both query operations, we send an asynchronous message to the remote user object (expressed by the ← operator in AmbientTalk). An asynchronous message send immediately returns a *future*, which is a placeholder for the actual return value of the message. In the above example the `equals` function waits

for the result of the message[1] to be available before computing the equality and in turn immediately returns a future. The **`where:`** and **`select:`** operators will automatically wait for futures to resolve with their values and process the results of asynchronous operations as they become available. If an element was removed before an asynchronous operation on that element was completed, the operation is cancelled and the result ignored.

The collections that result from applying operators to an ambient cloud transparently maintain a dependency relation to that ambient cloud. Any time the composition of the ambient cloud is changed or any time one of its constituents is changed, the operation *incrementally* updates the result collection. The result collection is never totally recomputed, rather dependent elements are either added to the result collection, removed from it or updated. Figure 1 shows the dependency chain that corresponds to the above code snippet.

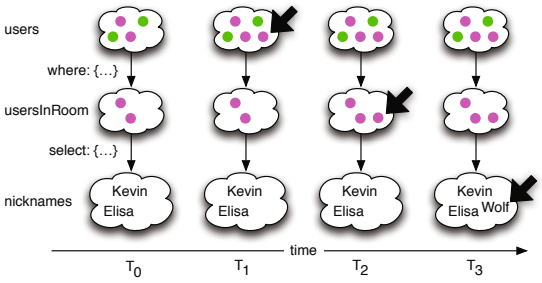

Fig. 1. Dependency tree of the ambient clouds in the chat application

Figure 1 shows the progression of events when the `users` ambient cloud changes. First, in step T_1, a new user is discovered and added to the `users` ambient cloud. In step T_2, the filter operation that generated the `usersInRoom` ambient cloud is then applied to this user object. Since the user has also joined the "purple" chatroom, the user is added to the resulting collection. In step T_3, the dependent `nicknames` ambient cloud is extended with the nickname of the user by applying the select operator to the added user object.

Aside from addition, two more events cause the operators to update their results. Figure 2 depicts the progression of events when a user is removed from the `users` ambient cloud in step T_4 to T_6. The user and nickname are subsequently removed from the dependent result collections. Starting from step T_7 a user switches from the "purple" to the "orange" chatroom. The filter operation is reapplied and causes the user and nickname to be automatically removed from the resulting collections.

[1] By annotating the message send with `Due(t)`, a timeout `t` (in milliseconds) can be specified for the resulting future.

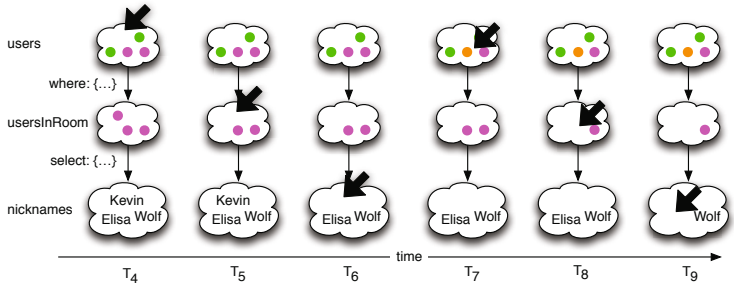

Fig. 2. Removing a user from the ambient cloud and changing the chatroom attribute of a user

3.2 Reactive Standard Query Operators

Our ambient clouds support over 20 operators, including the operators defined by the Standard Query Operation (SQO) [3] API of the Language Integrated Query (LINQ) framework for .NET [4].

The reactive standard query operators are defined as methods on ambient clouds. We also provide a language extension that provides syntactic sugar for writing queries and turns them into first-class language constructs. In the example below we illustrate how to group chat messages with their authors.

```
1  deftype ChatMessage;
2  def messages := cloudOf: ChatMessage
3          groupBy: { |msg| msg.userId }
4          join: usersInRoom
5            on: { |msg, user|
6            equals(msg.userId, user←id()) }
```

In the example above we define the ambient cloud of chat messages starting from line 2. In line 3 we group the ambient cloud based on a `userId` attribute using the **`groupBy:`** operator. This results in an ambient cloud of groups of messages identified by the `userId`. These groups of messages are again ambient clouds which in turn can be queried. Starting from line 4 we associate users with a group of messages using the **`join:on:`** operator. The join is performed based on matching the user's identifier with the identifier attribute in the message. This can be used to ignore the messages of a certain user in the chatroom.

4 Implementation

As mentioned before, ambient clouds are implemented in AmbientTalk and available to the programmer as a library. Ambient clouds are built on top of the Java Collections Framework[2] and combine AmbientTalk's service discovery mechanism

[2] AmbientTalk is entirely implemented in Java and runs on top of the JVM. Java classes and objects can be accessed from within AmbientTalk and vice versa [5].

(based on IP multicasting) with *reactive sets* containing *reactive values*. Note that AmbientTalk is an actor-based language. Execution (e.g., updating ambient clouds) within an actor is sequential and actors communicate by sending asynchronous messages that are processed in sequence by the receiving actor.

4.1 Reactive Asynchronous Collections

The implementation of ambient clouds is based on an reactive asynchronous collection framework that employs local Java collections. Reactive asynchronous collections are conceived as the combination of observable collections with reactive asynchronous operators. Our model consists out of the following collection types:

- **set**: no ordering, no duplicates (similar to Java HashSet)
- **list**: ordering, duplicates allowed (similar to Java ArrayList)
- **sorted set**: sorting, no duplicates (similar to Java TreeSet)

The programmer uses an event-driven API to install event handlers in order to observe a collection. These event handlers are executed when elements are either added or removed from the collection.

```
1 def s := ObservableSet.new();
2 whenever: s extended: { |el| system.println("added " + el) };
3 whenever: s reduced: { |el| system.println("removed " + el) }
```

This example creates a new reactive set and registers two event handlers that write a message to the standard output every time an element is added (line 2) or removed (line 3) from the collection.

Additionally, the collections can be observed for changes to their constituents. When a reactive value is added to a collection, the collection installs an event handler to observe the state of the value. When the state of the reactive value changes, the collection in turn notifies its own observers. The example below shows an event handler that writes a message to the standard output whenever the state of a constituent reactive value changes.

```
whenever: s changed: { |el| system.println("changed " + el) }
```

Ambient clouds are created by connecting the AmbientTalk discovery protocol to reactive sets. Below we show the skeleton code to manually construct the ambient cloud of users in the chat application.

```
1 def users := ReactiveSet.new();
2 whenever: ChatUser discovered: { |user|
3   users.add(user);
4   whenever: user disconnected: { users.remove(user) };
5   whenever: user changed: { users.notifyChangeObservers(user) }}
```

In line 1 we first create an empty reactive set. In line 2 we install an event handler to discover objects of type `ChatUser`, using the built-in **`whenever:discovered:`** construct of AmbientTalk. When an object of this type is discovered, it is added to the set in line 3. Two more event handlers are installed in lines 4 and 5. The first event handler is installed using AmbientTalk's built-in **`when:disconnected:`** construct and removes the element from the set when a disconnection occurs. The last event handler is executed when the state of the `user` object is altered and notifies the set of this change.

4.2 Reactive Objects and Isolates

In the implementation of ambient clouds we modelled objects as *reactive values*. A reactive value (also *behavior*) is a value that changes over time [6]. We regard objects as composite values of which the state can be altered over time using field assignment.

Programmers can publish objects in the network either by reference or by copy, as *reactive objects* or as *reactive isolates*. Reactive objects are transferred by reference, remote peers obtain an *observable remote object reference*. An observable remote object reference consists of a proxy object and a reference to the remote object. Reactive collections can install observers on the proxy object which are notified when the state of the object is modified locally. This causes operations applied on the object to be recomputed. Reactive 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 ambient clouds of remote peers. The underlying implementation locally observes the state of an isolate and implicitly synchronises state across the copies on different devices. Note that race conditions are prevented by the actor system of AmbientTalk, as explained in section 4. Of course, the programmer should bear in mind that the remote peer may be disconnected and that the copy is temporary out of sync. Any time the state is synchronised, the observers registered by reactive collections are notified.

In subsection 3.1 we showed an example of a user represented by a regular object. The example below shows the creation of a chat message represented by an isolate.

```
def aMessage := isolate: {
  def text := "Hello!"
  def userId := 123 }
```

Reactive objects and isolates are key in the design of ambient clouds since changes to their state triggers re-computation of dependent results.

4.3 Reactive Standard Query Operators

The reactive standard query operators rely heavily on the observable features of the collections to incrementally update their results. They update their results based on three kinds of events: the insertion and removal of elements in

a collection and state changes in the elements. When an operator is applied to a collection, the necessary observers are installed and the operator is applied on all elements already contained in the collection. For example, consider the implementation of the **where:** operator below.

```
1  def where: predicate {
2    def result := self.new(); // self refers to the current object
3    self.each: { |e|
4      if: (predicate(e)) then: { result.add(e) } };
5    whenever: self extended: { |e|
6      if: (predicate(e)) then: { result.add(e) } };
7    whenever: self reduced: { |e| result.remove(e) };
8    whenever: self changed: { |e|
9      if: (!predicate(e)) then: { result.remove(e) } };
10   result.parent := self;
11   result };
```

The operator takes a predicate as argument. In line 2 we create a new collection that contains the results of applying the operator. In lines 3 and 4 we first apply the predicate to all elements already contained in the collection and add them if that application succeeds. In lines 5 and 6 we install a handler that applies the predicate to elements added to the collection and add them to the result collection if necessary. In line 7 we install a handler that removes elements from the result as they are removed from the collection. The handler in lines 8 and 9 reapplies the predicate to an element if its state was changed, possibly removing the element from the result if applying the predicate no longer succeeds.

This code clearly shows that the automatic updates of the result are incremental. The execution of the event handlers in lines 5, 7 and 8 concern the addition, removal or update of a single element in the result. In line 10 we register the parent of the result to be the collection over which we applied the operator. In line 11 we finally return the resulting collection.

Note that this code was was simplified for demonstration purposes. It does not deal with the possibility that an asynchronous operation is performed in the predicate application.

5 The weScribble Application

In this section, we validate ambient clouds in the implementation of a collaborative drawing application for the Android platform, called weScribble[3]. The application allows users to dynamically participate in drawing sessions with other people co-located. Aside from mobile Android devices and wireless ad hoc connections between these devices, no other infrastructure is assumed.

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 either join a session or choose to create a new

[3] WeScribble is available from Google Play at `http://bit.ly/eOxpLg`.

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 synchronises the canvas with the existing participants to fetch the shapes that were already drawn and displays them on the screen. The user can then draw on the canvas of that session and the changes to the canvas are propagated to the other participants of the session whose canvas is updated accordingly. If a user temporarily disconnects from the network, he or she can keep drawing. Upon reconnection, those changes are synchronised with the other users in the session.

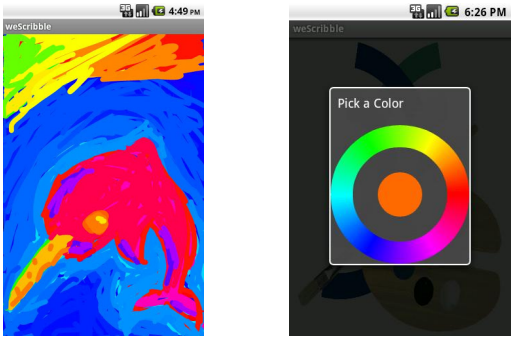

Fig. 3. The weScribble Android application

5.1 Ambient Clouds in weScribble

In this section, we illustrate the use of ambient clouds in the implementation of weScribble. weScribble uses ambient clouds for two different purposes: A drawing session consists of an ambient cloud that contains the shapes drawn by users. The users themselves are also contained in an ambient cloud.

First we collect the users that participate in our drawing session and extract their user names:

```
deftype Painter;
def painters := cloudOf: Painter
                where: {|p| equals(p←session, currentSession)};
names := painters.select: { |p| p←name }
```

We display the names of all discovered users in the GUI. Using the event-driven API we install two event handlers to show and hide names as they are added to or removed from the ambient cloud.

```
names.each: { |n| GUI.showInList(n) };
whenever: names extended: { |n| GUI.showInList(n) };
whenever: names reduced:  { |n| GUI.removeFromList(n) }
```

Users can choose to ignore shapes from other users by enabling a toggle in the GUI. We continue by defining an observable set containing the users that we choose to ignore.

```
def ignoredPainters := ObservableSet.new();
def ignore(painter) { ignoredPainters.add(painter) }
```

We now obtain the ambient cloud of shapes (represented by isolates) to display by joining the ambient cloud of all shapes with the difference of the ambient cloud of users and the users to ignore.

```
deftype Shapes;
def shapes := cloudOf: Shapes
                join: (painters.except: ignoredPainters)
                  on: { |shape, painter| s.painterId == p.id }
```

Note here that adding users to the `ignoredPainters` set causes its shapes to disappear from the `shapes` ambient cloud. Finally we draw the relevant shapes and install event handlers to draw and hide shapes as the ambient cloud is updated.

```
shapes.each: { |s| GUI.draw(s) };
whenever: shapes extended: { |s| GUI.drawShape(s) };
whenever: shapes reduced: { |s| GUI.removeShape(s) };
whenever: shapes changed: { |s| GUI.redrawShape(s) };
```

5.2 Discussion

In the implementation of weScribble ambient clouds tackle the issues outlined in section 2 as follows.

- ***P1:*** Ambient clouds automatically maintain collections of co-located users and the shapes they created. The programmer is relieved from manually synchronising the contents of these collections with the network situation.
- ***P2:*** We used reactive standard query operators over ambient clouds to filter users based on the drawing session. We also associate users with their shapes, filtering out shapes of users we wish to ignore without having to manually update these results when users appear or disappear.
- ***P3:*** When users change the colour of their shapes, these changes are automatically propagated to the applications of other users by means of reactive isolates. If users change their nickname this is automatically reflected in the user lists of the other users using reactive objects.

6 Related Work

The problems around group abstractions for mobile applications is well established. However, most of the research focuses on group communication which is

not the focus of this work. In this section, we discuss related work that focuses on organising remote objects in intermittently connected peer-to-peer applications.

Distributed Asynchronous Collections (DACs) [7] 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 publish objects, so the programmer still has to track this manually.

Tuple spaces [8] allow distributed parties to publish tuples to a conceptually shared memory. LIME [9] even allows 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. 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 [10] 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 programmers to react on objects, joining, leaving or being modified in the group. Additionally, they can not be composed, nor queried.

M2MI [11] introduces *handles* that denote a dynamic group of remote Java objects of the same interface and *omnihandles* that refer to all proximate objects of a certain interface. However, is not possible to react to state changes in the objects referred to by an omnihandle, nor is there support for querying.

Microsoft's Reactive Extensions for .NET [12] allows processing events by modelling them as streams of values on which standard query operators are defined. The difference with our work is that we define these standard query operators on object-oriented collections instead of event streams.

Reactive programming [6] is a paradigm that represents a program as a data flow graph based on the notion of *time-varying values*, which form nodes in the graph. If an operation is applied to a time-varying value, the operation is inserted as a node in the graph with a dependency edge to the time-varying value. When a time-varying value changes, dependent computations are automatically re-executed by propagating the change through the graph. We discuss the relation between reactive programming and our work in section 7.

7 Conclusion and Future Work

In this paper, we introduced ambient clouds: an object-oriented abstraction that automatically maintains collections of remote objects. These ambient clouds support reactive standard query operators that implicitly update their results

as changes in the underlying ambient clouds occur. Together they relieve the programmer from re-implementing the common patterns when dealing with **(1)** volatile collections of remote objects, **(2)** querying and composing these collections and **(3)** propagating state changes to derived collections.

We have implemented a MANET application called weScribble using ambient clouds to illustrate how ambient clouds circumvent these problems in a fully functional collaborative drawing application.

As possible avenues for future work, we wish to further extend our reactive collection framework with other collection types such as a hash map or tree structures. Additionally, unlike our model, in reactive programming languages, there are no special reactive operations. Every operation that depends on a reactive value is implicitly lifted to the reactive level. Currently, on our reactive asynchronous collections, special query operators are used to process changes. We are integrating our collections into a reactive programming language to reduce manual lifting.

References

1. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Computing Survey 35, 114–131 (2003)
2. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: AmbientTalk: object-oriented event-driven programming in mobile ad hoc networks. In: SCCC 2007, pp. 3–12. IEEE Computer Society (2007)
3. Microsoft Corporation: The .NET standard query operators. Technical Specification (2006)
4. Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework, 706–706 (2006)
5. Cutsem, T.V., Mostinckx, S., Meuter, W.D.: Linguistic symbiosis between event loop actors and threads. Computer Languages, Systems & Structures 35, 80–98 (2009)
6. Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A survey on reactive programming. ACM Computing Surveys (2012) (to appear)
7. Eugster, P.T., Guerraoui, R., Sventek, J.: Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 252–276. Springer, Heidelberg (2000)
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7, 80–112 (1985)
9. Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the 21st International Conference on Distributed Computing Systems, pp. 524–536. IEEE Computer Society (2001)
10. Van Cutsem, T., Dedecker, J., Mostinckx, S., Gonzalez Boix, E., D'Hondt, T., De Meuter, W.: Ambient references: addressing objects in mobile networks. In: OOPSLA 2006, pp. 986–997. ACM Press (2006)
11. Kaminsky, A., Bischof, H.P.: Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In: OOPSLA 2002, pp. 72–73. ACM Press (2002)
12. Microsoft Corporation: The reactive extensions for .NET (2013), `http://msdn.microsoft.com/en-us/data/gg577609`