Loosely-Coupled Distributed Reactive Programming in Mobile Ad Hoc Networks

Andoni Lombide Carreton*, Stijn Mostinckx,
Tom Van Cutsem**, and Wolfgang De Meuter

Software Languages Lab
Vrije Universiteit Brussel, Pleinlaan 2 1050 Brussel, Belgium
{alombide,smostinc,tvcutsem,wdmeuter}@vub.ac.be

Abstract. Pervasive applications running on mobile ad hoc networks have to be conceived as loosely-coupled event-driven architectures because of the dynamic nature of both the underlying network and the applications running on top of them. Such architectures can become tedious to develop and understand when the number of events and event handlers increases. The reason is that the control flow of the application is driven by event handlers or callbacks which are triggered independently and are scattered throughout the application code. In this paper, we propose a number of language constructs that reconcile the elegant processing of events of a reactive programming system with the loose coupling of a publish/subscribe system that is required to cope with the dynamic nature of mobile ad hoc networks.

Keywords: reactive programming, publish/subscribe, event-driven programming, mobile ad hoc networks.

1 Introduction

Pervasive applications running in mobile ad hoc networks cannot be structured as monolithic programs which accept a fixed input and compute it into some output. Instead, to allow responsiveness to changes in the dynamically changing mobile ad hoc network, programming paradigms targeting pervasive applications propose the adoption of event-driven architectures [1,2,3,4].

The traditional way of conceiving an event-driven system in a setting where producers and consumers change at runtime is by adopting a publish/subscribe architecture, where event producers publish events and event consumers subscribe and react to events, either using a topic-based or content-based subscription [5,6]. In mobile ad hoc networks, where devices dynamically join and leave, and where network connections can be broken at any point in time, the coupling between producers and consumers should be very loose to prevent that network

* Funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).
** Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

J. Vitek (Ed.): TOOLS 2010, LNCS 6141, pp. 41–60, 2010.

partitioning breaks the system when both parties are disconnected from one another. Furthermore, in such a setting without a fixed infrastructure such as naming servers, event producers should be able to notify event consumers without prior knowledge about their location and identity. This decoupling in space and time is what makes publish/subscribe so well suited to a dynamic environment such as a mobile ad hoc network [7,2] and both properties are needed to support application components that act as publishers and subscribers on *roaming devices*.

Eventually, the application layer should react to the events detected by the publish/subscribe layer. In most cases, this requires the programmer to bridge the gap between the underlying event notification system and the application layer. First, many event communication systems operate only on specific data types (e.g. event structs without methods) which lack some of the expressive power typically conveyed by high level programming languages. Consequently, application data types that correspond to events must adhere to additional criteria to allow mapping application data to events and vice versa. Furthermore, by adopting such an event-driven architecture, the application logic becomes scattered over different event handlers or callbacks which may be triggered independently [8]. The control of the application is no longer driven by an explicit control flow determined by the programmer, but by external events. This is a phenomenon known as *inversion of control* [9]. Control flow among event handlers has to be expressed implicitly through manipulation of shared state. E.g. unlike subsequent function calls, code triggered by different event handlers cannot use the runtime stack to make local variables visible to other executions (*stack ripping* [10]), such that these variables have to be made instance variables, global variables, etc. Finally, in more complex systems it is not always clear in which order different event handlers will be triggered, which can be critical in programming languages that allow side effects. In short, most publish/subscribe middleware lack a *seamless integration with a high level programming model* [11]. This is why in complex systems such an event-driven architecture can become hard to develop, understand and maintain [12,13].

In this paper, we propose a set of language constructs that enable the following:

1. **No inversion of control.** It should be possible to generate, combine and react to events all in the same programming language in an expressive way without inverting the control of the application and without introducing extra synchronization issues.
2. **Support for roaming.** Distributed application components notifying each other of events in an environment that exhibits all the characteristics of pervasive applications and mobile ad hoc networks mentioned above. This requires decoupling of event producers and consumers in both space and time.

Concretely, reactive programming techniques are used to build event-driven systems without inversion of control. Furthermore, we introduce *ambient behaviors*, a language abstraction built on top of publish/subscribe middleware which permits distributing event-driven applications over mobile ad hoc networks. The

expressive power of the language construct is subsequently illustrated by its use in a non-trivial collaborative application deployed on a mobile ad hoc network.

In the next section, the key technologies on which our approach is based, namely the programming language AmbientTalk and ambient references, are briefly explained. In section 3, we discuss reactive programming in AmbientTalk/R and introduce some event notification language constructs that support roaming and do not suffer from an inversion of control. Section 4 discusses a small pervasive application that we have implemented using our novel language constructs and subsequently in section 5 we point out the limitations of our approach. In section 6 we discuss some existing systems that permit building a similar kind of distributed event-driven systems by providing programming language support for one or more of the required features we pointed out above. Finally, section 7 concludes this paper.

2 Preliminaries

The *ambient-oriented* programming paradigm [14] is specifically aimed at pervasive applications running in mobile ad hoc networks. For this reason we chose to incorporate our language constructs in an existing ambient-oriented programming language. Ambient-oriented programming languages should explicitly incorporate potential network failures in the very heart of their computational model. Therefore, communication between distributed application components should happen without blocking the execution thread of the different components such that devices may continue doing useful work even when the connection with a communication partner is lost. Ambient-oriented languages also deal with the dynamically changing network topology in mobile ad hoc networks. The fact that in such networks devices spontaneously join with and disjoin from the networks means that the services these devices host cannot be discovered using a fixed, always available name server, but instead require dynamic service discovery protocols (e.g. broadcasting advertisements to discover nearby services).

2.1 AmbientTalk

AmbientTalk [15,16] is a distributed programming language embedded in Java[1]. The language is designed as a scripting language that can be used to compose Java components which are distributed across a mobile ad hoc network. The language is developed on top of the J2ME platform and runs on handheld devices such as smart phones and PDAs. Even though AmbientTalk is embedded in Java, it is a separate programming language. The embedding ensures that AmbientTalk applications can access Java objects running in the same JVM. These Java objects can also call back on AmbientTalk objects as if these were plain Java objects.

The most important difference between AmbientTalk and Java is the way in which they deal with concurrency and network programming. Java is multi-threaded, and provides either a low-level socket API or a high-level RPC API

[1] The language is available at `soft.vub.ac.be/amop`

(i.e. Java RMI) to enable distributed computing. In contrast, AmbientTalk is a fully event-driven programming language. It provides only event loop concurrency [17] and distributed objects communicate by means of asynchronous message passing. Event loops deal with concurrency similar to GUI frameworks (e.g. Java AWT or Swing): all concurrent activities are represented as events which are handled sequentially by an event loop thread.

AmbientTalk offers linguistic support to deal with the fluid topology of mobile ad hoc networks.

1. In an ad hoc network, objects must be able to discover one another without any infrastructure (such as a shared naming registry). Therefore, AmbientTalk has a service discovery engine that allows objects to discover one another in a peer-to-peer manner (by broadcasting UDP advertisements).
2. In an ad hoc network, objects may frequently disconnect and reconnect because of network partitions. Therefore, AmbientTalk provides fault-tolerant asynchronous message passing between objects: if a message is sent to a disconnected object, the message is buffered and resent later, when the object becomes reconnected. Other advantages of asynchronous message passing over standard RPC is that the asynchrony hides latency and that it keeps the application responsive (i.e. the event loop is not blocked during remote communication and is free to process other events).

2.2 Event-Driven Programming in AmbientTalk

AmbientTalk uses a classic event-handling style by relying on closures to function as event handlers. This has two advantages: closures can be used in-line and can be nested and closures have access to their enclosing lexical scope. Event handlers are (by convention) registered by a call to a function that starts with `when`. Events are always processed sequentially within the same event loop and event handler closures always run to completion before the next scheduled event handler is invoked, hence providing atomic execution of event handlers within the same event loop.

Throughout the rest of this paper we will use an example mobile application that assists visitors of a concert or other kind of event to trade tickets with other users. The following code snippet illustrates how AmbientTalk can be used to discover a `TicketVendor` object representing a user selling a ticket. Once discovered, the remote object is sent a message to retrieve its current location.

```
when: TicketVendor discovered: { |ticketVendor|
  when: ticketVendor<-getLocation() becomes: { |location|
    // Update user interface with the location.
  }}
```

The above code consists of two event handlers. The first event handler, registered by means of the `when:discovered:` control structure, is invoked when the language runtime discovers a `TicketVendor` object. Here, `TicketVendor` refers to a

Java interface. The discovered object is accessible via the `ticketVendor` variable, which denotes a remote AmbientTalk object that wraps a Java component implementing the ticket vendor. The syntax `obj<-msg()` denotes an asynchronous message send and is used here to query the `TicketVendor` object for its latest location. When the query message is processed by the remote `ticketVendor` object, that object's `getLocation` method is invoked. The return value of this method is used as the reply to the query. The caller is notified asynchronously when the reply has been computed. The `when:becomes:` control structure is used to install an event handler that can process this reply. The return value is passed to this event handler (cf. the `location` variable in the example).

As can be seen from the above example, service discovery and replies of remote queries are represented in AmbientTalk as events that trigger the appropriate event handlers, causing an inverted control flow. Furthermore, if the location of the ticket vendor has to be refreshed, the code shown above has to be executed periodically (e.g. in a loop). In the following section, we show how the events of discovering new and detecting lost services can be made implicit, by means of ambient references [18].

2.3 Language Abstractions for Roaming

When writing AmbientTalk code to query nearby services for data (e.g. all users selling a ticket) using the language features discussed in the previous section, one often writes a recurring pattern of code to deal with the discovery and loss of nearby services while a query is executing, and to deal with gathering the replies from all respondent services. To ease the writing of multicast queries, AmbientTalk introduces ambient references [18]. Ambient references represent a collection of nearby services of the same type. This collection is constantly kept up-to-date with the proximate physical environment: newly discovered services are added to the collection, while unreachable services are removed from it. This synchronization with the environment must no longer be done manually by the programmer, but is instead done by the ambient reference itself.

Sending a message to an ambient reference causes this message to be multicast to all services in the collection. A message can also be annotated with an expiration period (in milliseconds). If a message has an expiration period, it will not only be multicast to all services in the ambient reference's collection at the time the message is sent, but also to any services discovered at a later point in time, until its expiration period has elapsed. Consider the following example query:

```
def werchterVendors :=
  ambient: TicketVendor where: { |tv| tv.event == "Rock␣Werchter" };

whenAll: werchterVendors<-getLocation()@Expires(5.seconds) becomes: {
  |locations|
  // Update the map GUI with the locations
}
```

The keyword `ambient:` allows one to create an ambient reference given a Java interface. Additionally, an optional `where:` clause can be specified that allows to filter on properties of the discovered objects to allow content-based publish/subscribe. The variable `werchterVendors` contains an ambient reference that refers to all nearby `TicketVendor` objects that are selling tickets for an event named `"Rock␣Werchter"`. The message `getLocation()` is asynchronously multicast to these services with an expiration period of 5 seconds. This implies that the message may be received by all proximate ticket vendors at the time it is sent, as well as to all additional ticket vendors discovered within the next 5 seconds. The `whenAll:becomes:` control structure allows the programmer to install an event handler that can be used to gather the results of the query. Within this event handler, `locations` refers to an array containing the locations of the ticket vendors that replied. The event handler is triggered when the message's expiration period has elapsed.

The above example shows how ambient references relieve the programmer from having to deal explicitly with the events of discovery and loss of nearby services: ambient references transform these events into additions to or removals from their encapsulated collection. The programmer only has to specify an intensional (topic-based or content-based) description of the services that have to be discovered. The programmer must still capture the replies to the query by means of the `whenAll:becomes:` callback, leading to an inverted control flow. Furthermore, if the user interface showing the locations of the ticket vendors has to be kept up to date with the physical environment, the query has to be repeatedly executed in a loop. Eliminating both phenomena is the topic of the next section.

3 Reactive Programming in AmbientTalk/R

Reactive programming is a programming paradigm employed for various purposes such as animation [19], real time systems [20] and robotics [21]. Originating in Haskell, it has been successfully introduced in various other languages such as Java [22] and Scheme [23]. Reactive programming revolves around the use of time-varying *reactive values* or *behaviors*. While evaluating a reactive program, the interpreter implicitly constructs a directed acyclic *dataflow graph* [24] which mirrors the call graph of the program. Reactive values form the nodes of the graph, while the directed edges represent data dependencies. In this section, we briefly describe AmbientTalk/R, an extension of AmbientTalk with support for reactive programming.

As a first introduction to reactive programming in AmbientTalk/R, consider the following example: assume that the user interface of the ticket trader application introduced in the previous section sports a user interface which includes a map that shows the position of ticket vendors. The user can use this map to seek out a vendor and purchase one of the tickets being offered. To facilitate navigation, the map must be centered on the user's current position. Hence, whenever the position of the user changes, the user interface should be updated. The code

excerpt below illustrates that using reactive programming, such behavior can be achieved without registering event handlers or suffering from inversion of control.

```
gui.centerOn(GPS_Location.latitude, GPS_Location.longitude);
```

In the above code excerpt, it is assumed that `GPS_Location` is a reactive value that represents the user's current location. Later in this section, we will illustrate how to construct such a reactive value by means of a built-in GPS location sensor. Given the `GPS_Location`, dependent reactive values are created implicitly when accessing its `latitude` and `longitude` fields respectively. These reactive values will be recomputed (i.e. the respective fields will be read anew) automatically whenever the `GPS_Location` is updated. In turn, the reactive values representing the user's current latitude and longitude are used as arguments to the invocation of the `centerOn` method. This method invocation will be lifted by the interpreter, resulting in the construction of a reactive value which depends on both reactive arguments. Hence, when either one of the arguments changes, the method will be invoked anew with the updated arguments. The dataflow graph that is constructed by evaluating this code snippet is shown in figure 1.

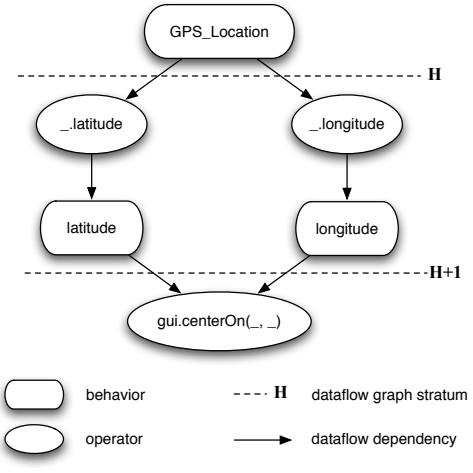

Fig. 1. Dataflow graph for centering the map in the ticket trader application

The graph shows the reactive value `GPS_Location` which acts as the *progenitor* for two dependent reactive values which represent the `latitude` and `longitude`. In turn, these reactive values are the progenitors for the reactive value that - as a side-effect - centers the user interface on the user's current location. Furthermore, figure 1 shows how the dataflow graph is partitioned into different layers or *strata* such that a reactive value only depends on reactive values situated in a lower stratum. This stratification (first proposed in [23]) is used when propagating the updates to ensure that a reactive value is recomputed only when all of its progenitors have been updated. For instance, when the user's current position

changes, the interpreter will first update both the `latitude` and the `longitude` reactive values before the `centerOn` method will be invoked anew.

Having illustrated how dependent reactive values are created implicitly by the interpreter in a reactive program, we now describe how to create new reactive values *ex nihilo*. For this, AmbientTalk/R introduces the `makeReactive` construct which creates a reactive value based on the object it is passed. In the code example given below, we define a `Coordinate` object[2] which represents a GPS location. In addition to its fields, `Coordinate` objects also define two methods, to wit `distanceTo` and `update`.

```
1  def Coordinate := isolate: {
2    def latitude := 0;
3    def longitude := 0;
4
5    def distanceTo(anotherCoordinate) {
6      /* Compute via the Haversine formula */
7    };
8
9    def @Mutator update(newLatitude, newLongitude) {
10     latitude := newLatitude;
11     longitude := newLongitude;
12   };
13 };
14
15 def GPS_Location := makeReactive(Coordinate.new());
16 GPS.addLocationObserver: { |lat, lng| GPS_Location.update(lat, lng) };
```

When passing an object to the `makeReactive` construct, a clear distinction should be made between accessor methods which only read the state encapsulated by the object (e.g. `distanceTo`) and mutator methods which can change the object's internal state (e.g. `update`). Therefore, all mutator methods must be explicitly tagged with an `@Mutator` annotation. This requirement stems from the fact that the semantics for invoking both kinds of methods on a reactive value differs significantly:

Accessor methods. When invoking an accessor method (or reading a field) on a reactive value, a dependent reactive value is created which depends on the receiver and additionally on any reactive values that were passed as arguments. Hence, if the reactive value is updated, these dependent computations will be performed anew.

Mutator methods. When invoking a mutator method (or writing a field), no dependency on the receiver is recorded. In other words, if none of the arguments

[2] The object is created by means of the `isolate:` AmbientTalk language construct. This simply constructs a special kind of object that has no surrounding lexical scope and thus can be easily copied over the network, instead of having it to pass by reference as is done for regular objects.

of the method invocation are reactive values, the method is simply performed once. If at least one reactive value was passed as an argument, a dependent reactive value is created which ensures that the mutator method is invoked anew whenever the reactive arguments change. However, changes to the receiver are simply disregarded. Furthermore, the interpreter ensures that whenever a mutator method has been invoked, all dependents of the receiver are notified that their progenitor has been updated.

This semantics is used in line 16 of the code example to register a location observer with the GPS device, which will be invoked whenever the user's position has to be updated. At this point in time, the mutator method `update` is invoked upon the reactive object `GPS_Location`. This mutator method is invoked once (since its arguments are ordinary numeric values), updating the coordinates to reflect the most recent sensor values. Afterwards, all reactive values which implicitly depend on `GPS_Location` will be notified that the location has changed. This may result for instance in an update of the user interface, such that the map is centered on the user's updated position. Note that mutator methods do not need to be atomic to guarantee correctness: the stratification explained earlier in this section of the dataflow graph constructed by the interpreter in combination with all updates that are scheduled in a single event loop according to this stratification prevent local concurrency control problems and ensure that the ordering of updates to reactive values mirrors the call graph of the program (which is critical when reactive updates trigger side effects).

3.1 Loosely-Coupled Distributed Reactive Programming

The reactive programming system described in the previous section only deals with events in a single, local event loop. In many cases, distributed application components are interested in events coming from other devices (and thus event loops) in the mobile ad hoc network. In this section, we introduce a language construct called *ambient behaviors* that allows the loosely-coupled propagation of events to reactive values hosted on different event loops by means of publish/subscribe. The transition from local reactive values to ambient behaviors needs some special consideration in order to uphold the ambient oriented programming characteristics mentioned in section 2. First of all, because of the dynamic nature of mobile ad hoc networks, one cannot assume a stable dataflow graph as is constructed on the local interpreter level, such as explained in the previous section. Instead, the dependencies between different distributed computations should be encoded in such a way that the ambient-oriented programming characteristics are upheld. When we rename behaviors to event producers and dependent computations to event consumers, this means that there should be a very loose coupling between event producers and event consumers. In classic publish/subscribe systems, event publishers do not have explicit knowledge about their subscribers and vice versa. In mobile ad hoc networks, the binding between consumers and producers must happen in the absence of any infrastructure, such as a centralized broker network. In this section, we describe a publish/subscribe system where event producers and consumers, denoting

reactive application components, find each other in the mobile ad hoc network by means of *intensional descriptions* that are broadcasted using UDP to allow *decentralized and spontaneous discovery*. The difference with an extensional approach (e.g. a list of registered subscribers) is that one merely states the conditions that the properties of a producer or consumer must satisfy to establish a loosely-coupled binding between the two.

Ambient Behaviors. We will continue the ticket trading example introduced earlier. Recall that ticket vendors have a behavior that denotes their current location by means of GPS coordinates. What we actually want to achieve is to discover ambient behaviors made available by other devices that signal the events in which we are interested, in this example a behavior that represents the GPS coordinates of the location of the ticket vendor. Publishing such a behavior happens as follows, on the ticket vendor's device:

```
exportBehavior: GPS_Location as: TicketVendorLocation
 to: { |buyer| buyer.interestedIn == "Rock␣Werchter" };
```

By exporting this behavior, applications running on other devices can subscribe themselves on the events that are signaled by this behavior. This happens as follows:

```
def vendorLocation := ambientBehavior: TicketVendorLocation
  where: { def interestedIn := "Rock␣Werchter" } @One;
```

The `ambientBehavior:` construct is used to create a local reactive value which is bound to one or more behaviors exported by other event loops. In the example given above, the `@One` annotation is used to indicate that `vendorLocation` should denote the location of a single ticket vendor, rather than a collection of vendor locations. Once an exported behavior can be found that matches the intensional descriptions given by the programmer (which can be either topic-based or content-based), the exported behavior will transparently start propagating update events to `vendorLocation`. Note that multiple applications could be subscribed to the `TicketVendorLocation` topic. The group communication required to notify all these subscribers is internally handled by the M2MI framework [1]. These events trigger an update in the reactive value which may result in further reactive computation in its own event loop. For instance, the `vendorLocation` could be used to update the location of the ticket vendor on the map in the graphical user interface:

```
GUI.showLocationOnMap(vendorLocation);
```

The point here is that while the ticket vendor roams the environment and his GPS device signals updates to all subscribed behaviors, the maps on the user interfaces of the (reachable) interested parties are transparently updated with

the new locations without resorting to callbacks. Furthermore, if an ambient behavior is disconnected from the exported behavior it was bound to, the ambient behavior will attempt to match with another exported behavior in the ad hoc network. Finally, since ambient behaviors are treated as regular behaviors by the interpreter, they can be used in local reactive code as if they were behaviors that depend solely on local changes. On the other hand, applications that export the `GPS_Location` do not have explicit knowledge to which event consumers they propagate events, nor do they keep an explicit list of event consumers. This loose coupling between event producers and consumers is necessary to reflect the dynamic nature of mobile ad hoc networks and to support roaming of devices.

Reactive Queries in Mobile Ad Hoc Networks. The mechanism described above can only be used if there are ambient behaviors published in the network. Otherwise, one has to obtain ambient behaviors by querying the network for relevant information oneself. For this the programmer is provided with an abstraction that allows creating a behavior that autonomously queries the network to update itself. This abstraction is an integration of ambient references (which allow querying the network by sending messages) with the reactive programming language facilities of AmbientTalk/R (which allow reacting to and processing events without inversion of control). The example below shows the creation of a behavior by querying the network using a reactive ambient reference.

```
def werchterVendors :=
  ambient: TicketVendor where: { |tv| tv.event == "Rock␣Werchter" };

def locations := werchterVendors<-getLocation()@Refresh(5.seconds);
```

Note that the `getLocation()` message is annotated with `@Refresh`, which implies that the result of the message is accumulated in a reactive value. Hence, the `locations` variable contains a reactive value which initially denotes an empty array. The `@Refresh` annotation implies that the annotated `getLocation` message is sent every 5 seconds to all nearby ticket vendors offering a ticket for Rock Werchter[3]. The resulting `locations` behavior is updated every five seconds and contains an array of all responses from the ticket vendors in range. Since `locations` is a behavior, it can be passed on to other functions or methods as a normal value, as done below to update the map in the user interface of the user with *all* locations:

```
locations.each: { |coordinates| GUI.showLocationOnMap(coordinates) };
```

[3] In addition to the `@Refresh` annotation, one can add annotations to the message that specify the message sending semantics. By varying these annotations, one can decide to send the message to all objects in range like in the example, which will result in a changing array of results, or send the message to just one of the objects in range, resulting in a behavior containing a single value.

Note that by making use of a reactive query, the programmer does not have to explicitly poll the environment in a loop any more.

To conclude, integrating ambient references with reactive programming allows the results of queries over the network to be collected into a behavior that is automatically synchronized with the environment. Ambient references provide an abstraction over the events of appearance and disappearance of services in the network, while the reactive programming system provides an abstraction over the events generated by the reception of results of asynchronous queries. Reactive queries can be regarded as the dual language construct of ambient behaviors, offering pull-based instead of push-based communication.

4 The Ticket Trader Application

In previous sections, we have used the dynamic discovery of ticket vendors and their location as a running example to explain the various features of AmbientTalk/R and ambient behaviors. This section presents a slightly more elaborate version of the application, which matches ticket vendors with prospective clients. In publish/subscribe terminology, ticket vendors *publish* the offers for tickets they are willing to sell, while their prospective clients *subscribe* to events concerning tickets being offered in their vicinity. Clients are able to identify which ticket offers are relevant to them by specifying which events they want to attend, the price they are willing to pay for the ticket and the maximal allowed distance between themselves and the ticket vendor. The latter filter requires that both vendors and clients have access to a GPS device, such that their GPS coordinates can be used to compute the distance.

Note that different ticket vendors can offer tickets for the same event (possibly for a different price) and that different clients can be interested in the same ticket. Furthermore, both vendors and clients roam the environment, can cancel their offers, change the price of their offers, and announce new offers. The different instances of the application on the different devices should all respond to these changes.

Before turning our attention to the implementation of both parties in the system, we will show the implementation of a very simple object representing a ticket offer:

```
def TicketOffer := isolate: {
  def eventName := nil;
  def price    := 0;
  def location := nil;

  // Constructor
  def init(anEventName, aPrice) {
    eventName := anEventName;
    price     := aPrice;
  };
};
```

The object contains three slots: the event the ticket provides access to, the price at which it is currently being offered and the vendor's current location. During the course of the application, the latter two values may change: the vendor may roam and decide to adjust the price at which the ticket is being offered. Typically, the price can be reduced if interest is low or if the event is about to start.

To determine the vendor's current position, we reuse the `GPS_location` abstraction, which was defined previously as follows:

```
def GPS_Location := makeReactive(Coordinate.new());
GPS.addLocationObserver: { |lat, lng| GPS_Location.update(lat, lng) };
```

The following code excerpt defines an AmbientTalk type (which corresponds to a Java interface type) that will be used as the topic under which ticket offers are published.

```
deftype TicketOfferT;
```

Having described the necessary abstractions, we can now describe how ticket offers are published by the vendor:

```
1  def TicketVendor := object: {
2    def offeredTickets := HashMap.new();
3
4    // Offer a new ticket.
5    def offerTicket(eventName, price) {
6      def ticketOffer := makeReactive(TicketOffer.new(eventName, price));
7      ticketOffer.location := GPS_Location;
8      offeredTickets.put(eventName, ticketOffer);
9      exportBehavior: ticketOffer as: TicketOfferT;
10   };
11
12   // Change the price of a ticket on offer.
13   def setTicketPrice(eventName, newPrice) {
14     (offeredTickets.get(eventName)).price := newPrice;
15   };
16 };
```

Tickets are offered to all nearby prospective clients by invoking the `offerTicket` method. In it, a reactive `TicketOffer` object is created (line 6). Because the object is reactive, the vendor is guaranteed that whenever the offer changes, these changes will be automatically propagated to all prospective clients. One way in which the offer might change is if the location of the vendor changes. Note that in line 7, the location associated with the offer is set to the vendor's `GPS_Location`. Due to the fact that `GPS_Location` is a reactive value and due to the semantics of mutating reactive values (see section 3), the `location` field of the `ticketOffer` will be set anew whenever the `GPS_Location` is updated. In turn, this update will be propagated to reactive values which depend on `ticketOffer`.

In this particular case, this includes all prospective clients that are currently in range. The fact that these prospective clients can detect the offer, stems from the fact that it is published using the `exportBehavior:as:` construct in line 9.

Additionally, vendors keep track of the various events for which they offer tickets in the `offeredTickets` map. This map is used to update the price at which tickets are being offered. The `setTicketPrice` method uses the mapping to find a particular reactive ticket offer, in order to update its price. Due to the semantics of mutating reactive values, setting the price causes an update to be propagated to all prospective clients in reach.

The following code excerpt shows the `findOffers` function, which permits prospective clients to detect ticket offers that have been exported by nearby vendors by means of the `ambientBehavior:` construct.

```
1  def findOffers(event, maximumPrice, maximumDistance) {
2    // Subscribe to TicketOffers
3    def allNearbyOffers := ambientBehavior: TicketOfferT @All;
4
5    // Filter out interesting TicketOffers
6    allNearbyOffers.filter: { |offer|
7      (offer.eventName == event).and: {
8        (offer.price <= maximumPrice).and: {
9          GPS_Location.distanceTo(offer.location) <= maximumDistance }}};
10 };
11
12 def werchterTicketVendors := findOffers("Rock␣Werchter", 200, 500);
13 gui.updateWithOffers(werchterTicketVendors)
```

The ambient behavior `allNearbyOffers` will be bound to a collection that contains all ticket offers made by nearby vendors. This semantics is due to the fact that the `@All` annotation is used, rather than the `@One` annotation that was showcased earlier. In other words, `allNearbyOffers` is a reactive value denoting an array of exported ticket offers. This size of this array evolves as ticket vendors go in and out of range.

The reactive collection `allNearbyOffers` is subsequently filtered to produce a selection of ticket offers that are relevant to the client (lines 6-9). An offer is deemed relevant if it provides access to the correct event, its price does not exceed the maximum set by the client and if the distance to the client's current location does not exceed a given maximum. Since `allNearbyOffers` is an ambient behavior, the invocation of its `filter:` method (an accessor method) creates a dependent reactive value, which is the return value of the function. This reactive value is updated when vendors go in and out of range, but also if one of the previously detected offers change (i.e. the vendor has moved or the price has been updated). Furthermore, it is important to note that the condition to determine whether an offer is relevant also depends on the location of the prospective client. In line 9, the vendor's location is compared to the current location of the client. This implies that an offer can suddenly become relevant as the client is roaming.

In the code snippet, the `findOffers` function is called to find ticket offers to attend Rock Werchter, which cost less than 200 euro and whose vendor is less than 500 meters away. The resulting collection is passed as an argument to the `updateWithOffers` method of the user interface. This method expects an array of ticket offers (which all contain their last location) and draws them on the map. Since the `werchterTicketVendors` collection is a reactive value, this method will be invoked anew whenever the collection is updated.

Evaluation. Notice that ticket vendors and their prospective clients are loosely coupled to one another. They discover one another by means of a topic-based publish/subscribe architecture (which uses a common Java interface to denote the type of events that are exchanged). Parties that are disconnected from each other (e.g. by network partitioning of the mobile ad hoc network) are automatically discarded while newly connected parties dynamically discover each other and start exchanging events.

Furthermore, the publication of new events is integrated closely with the imperative object-oriented programming style of the host language. Provided that mutator methods are properly identified, any object can be used to create a reactive value. Once such a reactive value has been published, it suffices to write one of the object's fields or invoke one of its mutator methods to implicitly emit events notifying all reachable subscribers of the change.

Finally, the subscriber can trivially indicate which events it is interested in receiving (by means of a topic-based subscription) and can handle incoming events without resorting to a complex network of event handlers. In the ticket trader example, two sources of events are considered, to wit `allNearbyOffers` and `GPS_Location`. Changes to the former are the result of the appearance and disappearance of vendors (which result in the addition and removal of certain offers) as well as updates to the offers themselves (i.e. price and/or location updates). Changes to the latter are the result of roaming clients, and affect the number of offers reported to the user as they affect the distance between the vendor and the prospective client. The interplay between these different event sources are handled implicitly by the AmbientTalk/R interpreter.

5 Limitations and Future Work

As a first issue, it is clear that the reactive programming system of AmbientTalk/R comes with an overhead in terms of computational resources compared to the plain AmbientTalk interpreter. More (dataflow) events are scheduled in the different event loops of an application and more memory is being consumed to keep track of the different dataflow dependencies in a local application. Although in the near future we will investigate to which extent this is the case, we also observe that, with respect to pure processing power, AmbientTalk and AmbientTalk/R are targeted towards highly networked applications where the network will be the performance bottleneck instead of the interpreter.

Some of the limitations stem directly from the hardware characteristics of mobile ad hoc networks. Because of both the unreliability of the connections between the different devices and the unpredictable delays on the arrival of messages at remote parties, our system cannot provide real-time guarantees on the processing and reacting to events. Furthermore, when a message is sent from one device before a different message is sent from another device, the underlying AmbientTalk virtual machines do not guarantee that these messages will arrive at their destination in that same order. Providing such guarantees would involve keeping a global clock over all the distributed virtual machines (an assumption that is for example made in the GEM event monitoring language [25]) to time-stamp events, which is impractical in this setting. The programmer has to take this into consideration if causality between events has to be inferred. However, our system focuses on applications that work with a human time scale (e.g. seconds, minutes), so slightly drifting distributed clocks are tolerable.

Currently, there is no way for an event consumer to tell its event producer to limit the events that it wants to receive: an ambient behavior publication or subscription can only be cancelled. Afterwards, the subscription can be re-established. We have to investigate whether this can lead to network congestion or performance issues on the device that acts as event consumer. We might look for inspiration in some of the systems mentioned in section 6 which incorporate load balancing.

Finally, the naming and discovery of services happens via Java interfaces. We make the underlying assumption that the name of such Java interfaces represents a unique service and is known by all participating services. This discovery mechanism also does not take versioning into account explicitly. For example, if the `TicketVendor` from the example in section 2.1 is updated, older clients may discover the updated service, and clients that want to use only the updated service may still discover older versions. Clients and services are thus themselves responsible to check versioning constraints.

6 Related Work

Solar [26] is a graph-based abstraction for collecting, aggregating and disseminating context information targeting mobile, pervasive applications. The abstraction models context information as events, which are produced by *sources*, flow through a directed acyclic graph of event-processing *operators*, and are delivered to subscribing applications. Applications describe their desired event stream as a tree of operators that aggregate low-level context information published by existing sources into the high-level context information needed by the application. The *operator graph* is thus the dynamic combination of all applications' subscription trees. Solar assumes centralized, reliable components to process the subscription requests from applications (which may dynamically join and leave the network) and deploys operators onto appropriate nodes as necessary. These centralized components render Solar unsuitable for mobile ad hoc network applications.

Flask [27] is a functional reactive programming language embedded in Haskell that uses Haskell as a meta-language to generate node-level code fragments in a subset of Haskell called Red. Red is intended to run on resource-constrained sensor nodes and is stripped from language features such as closures and recursive data types to eliminate arbitrary allocation. Just like our approach, Flask constructs a distributed dataflow graph, but in this case at compile-time instead of run-time using the Haskell meta-language, which causes Red code fragments to be deployed on the distributed nodes. To cope with a dynamically changing network topology in mobile ad hoc networks we require the nodes to be deployed at runtime instead of at compile-time.

Opis [28] is a functional reactive extension to Objective Caml for developing event-based distributed systems. An Opis protocol description consists of a reactive function (called event function) describing the behavior of a distributed system node. Opis is very related to our approach in the sense that it both applies reactive programming as a paradigm to implement event-based distributed applications and uses a peer-to-peer overlay protocol to disseminate the events between distributed application components. However, Opis focuses on wide-area networks where nodes are fixed and hence does not support roaming.

SpatialViews [29] is an extension to Java designed to query wireless sensor networks. SpatialViews allows the specification of virtual networks of which the nodes are discovered dynamically with user-specified (physical) location and time constraints and execute mobile code that constitutes to the global query. SpatialViews might as well be a suitable building block to implement the language constructs presented in this paper and the location and time constraints that can be placed on nodes can enhance these constructs with similar features.

A similar idea exists in Location-based Publish/Subscribe (LPS) [30] in which publishers and subscribers are not only bound by means of a topic-based or content-based subscription, but also by taking into account external context such as the physical location of the different parties. However, such external context is provided in LPS by centralized infrastructure, whereas our approach does not assume any infrastructure. The ticket trader example application used in section 4 is strongly expired by the examples used to illustrate LPS.

Finally, the language constructs proposed in this paper are integrated in a distributed, imperative object-oriented scripting language. There exist dedicated languages for event processing, such as Aurora [31]. Aurora is a centralized stream processor that uses the popular *boxes* and *arrows* paradigm found in most process flow and workflow systems. Tuples flow through a loop-free, directed graph of processing operators (i.e., boxes) which the programmer has to specify using a graphical user interface. Aurora was afterwards extended to Aurora* and Medusa, which make decentralized event processing possible and additionally allow high level load balancing and load shedding policies to be expressed using specialized middleware. These systems however, focus on internet-scale application running in a reliable network.

7 Conclusion

We have presented a number of language constructs that reconcile the loose coupling of a distributed publish/subscribe architecture and the elegant event processing of a reactive programming language, AmbientTalk/R. Concretely, the dataflow graphs constructed by the AmbientTalk/R interpreter to keep track of dataflow dependencies can now be seamlessly distributed by means of ambient behaviors, which is a new language construct added to the language. The distributed dataflow dependencies are implemented on top of a decentralized publish/subscribe architecture to achieve a very loose coupling between the dependents (event consumers) and their progenitors (event producers). Hence, event producers can be dynamically replaced at run-time when they become unreachable due to network partitions. By adopting the reactive programming paradigm, the reception of events can be represented as (external) updates to a reactive value. Such updates are propagated implicitly to all relevant parts of the application. Hence, it is possible to react trivially to external events without the inversion of control that would result from having to resort to the use of explicit callbacks.

References

1. Kaminsky, A., Bischof, H.P.: Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In: OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 72–73. ACM, New York (2002)
2. Meier, R., Cahill, V.: Steam: Event-based middleware for wireless ad hoc networks. In: ICDCSW '02: 22nd International Conference on Distributed Computing Systems, Washington, DC, USA, pp. 639–644. IEEE Computer Society, Los Alamitos (2002)
3. Murphy, A., Picco, G., Roman, G.C.: 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, Los Alamitos (2001)
4. Grimm, R.: One world: Experiences with a pervasive computing architecture. IEEE Pervasive Computing 3(3), 22–30 (2004)
5. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Achieving scalability and expressiveness in an internet-scale event notification service. In: PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, pp. 219–227. ACM Press, New York (2000)
6. Cugola, G., Nitto, E.D., Fuggetta, A.: The jedi event-based infrastructure and its application to the development of the opss wfms. IEEE Trans. Softw. Eng. 27(9), 827–850 (2001)
7. Meier, R., Cahill, V.: Taxonomy of distributed event-based programming systems. In: ICDCSW '02: 22nd International Conference on Distributed Computing Systems, Washington, DC, USA, pp. 585–588. IEEE Computer Society, Los Alamitos (2002)
8. Chin, B., Millstein, T.: Responders: Language support for interactive applications. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 255–278. Springer, Heidelberg (2006)

9. Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Lightfoot, D.E., Szyperski, C. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 4–22. Springer, Heidelberg (2006)
10. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: USENIX Annual Technical Conference, pp. 289–302. USENIX Association, Berkeley (2002)
11. Verissimo, P., Casimiro, A.: Event-driven support of real-time sentient objects. In: 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, Guadalajara, Mexico (January 2003)
12. Levis, P., Culler, D.: Mate: A tiny virtual machine for sensor networks. In: International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA (October 2002)
13. Kasten, O., Römer, K.: Beyond event handlers: programming wireless sensors with attributed state machines. In: IPSN '05: 4th international symposium on Information processing in sensor networks, Piscataway, NJ, USA. IEEE Press, Los Alamitos (2005)
14. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-Oriented Programming. In: OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM Press, New York (2005)
15. 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: Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), pp. 3–12. IEEE Computer Society, Los Alamitos (2007)
16. Van Cutsem, T., Mostinckx, S., De Meuter, W.: Linguistic symbiosis between event loop actors and threads. Computer Languages Systems & Structures 35(1) (2008)
17. Miller, M., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 195–229. Springer, Heidelberg (2005)
18. Van Cutsem, T.: Ambient References: Object Designation in Mobile Ad Hoc Networks. PhD thesis, Vrije Universiteit Brussel, Software Languages Lab (May 2008)
19. Elliott, C., Hudak, P.: Functional reactive animation. In: ACM SIGPLAN International Conf. on Functional Programming, vol. 32(8), pp. 263–273 (1997)
20. Wan, Z., Taha, W., Hudak, P.: Real-time FRP. In: International Conference on Functional Programming, ICFP'01 (2001)
21. Peterson, J., Hudak, P., Elliott, C.: Lambda in motion: Controlling robots with haskell. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, p. 91. Springer, Heidelberg (1999)
22. Courtney, A.: Frappé: Functional reactive programming in Java. In: Third International Symposium on Pratical Aspects of Declarative Languages (March 2001)
23. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 294–308. Springer, Heidelberg (2006)
24. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. 36(1), 1–34 (2004)
25. Mansouri-samani, M., Sloman, M.: Gem: A generalized event monitoring language for distributed systems. IEE/IOP/BCS Distributed Systems Engineering Journal 4, 96–108 (1997)

26. Chen, G., Kotz, D.: Context aggregation and dissemination in ubiquitous computing systems. In: WMCSA '02: Fourth IEEE Workshop on Mobile Computing Systems and Applications, Washington, DC, USA. IEEE Computer Society, Los Alamitos (2002)
27. Mainland, G., Morrisett, G., Welsh, M.: Flask: staged functional programming for sensor networks. In: 13th ACM SIGPLAN international conference on Functional programming, pp. 335–346. ACM, New York (2008)
28. Dagand, P.E., Kostić, D., Kuncak, V.: Opis: reliable distributed systems in ocaml. In: TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation, pp. 65–78. ACM, New York (2009)
29. Ni, Y., Kremer, U., Stere, A., Iftode, L.: Programming ad-hoc networks of mobile and resource-constrained devices. In: ACM SIGPLAN conf. on Programming language design and implementation, pp. 249–260. ACM, New York (2005)
30. Eugster, P.T., Garbinato, B., Holzer, A.: Location-based publish/subscribe. In: NCA '05: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications, Washington, DC, USA, pp. 279–282. IEEE Computer Society, Los Alamitos (2005)
31. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S.: Scalable distributed stream processing. In: CIDR (2003)