Group Communication Abstractions for Distributed Reactive Systems

Andoni Lombide Carreton, Stijn Mostinckx and Wolfgang De Meuter

Programming Technology Lab (PROG)
Vrije Universiteit Brussel, Pleinlaan 2 1050 Brussel, Belgium
{alombide} {smostinc} {wdmeuter}@vub.ac.be

Abstract. Pervasive computing in mobile ad hoc networks requires that applications react to a plethora of events fired by other devices in the mobile ad hoc network. Current context-aware and event-driven architectures require the programmer to react to these events via a carefully crafted network of observers and event handlers, while inherently introducing complex concurrency issues. This paper proposes the integration of group communication abstractions with Reactive Programming as an alternative technique that does not suffer from these problems.

1 Introduction

Because of the constant evolution of computing hardware, mobile computing devices with networking capabilities are becoming increasingly cheap, small and energy efficient. To reap the benefits of the resulting mobile networks, applications must be able to respond to changes in the environment in a timely fashion. State of the art context-aware applications are often conceived as event-driven architectures which consume events fired by a context-aware middleware framework. The events represent significant context changes which should percolate through the entire application, requiring a carefully crafted network of observers combined with complex synchronization code to deal with the inherent concurrency issues [1].
Recently, we have introduced *Reactive Context-aware Programming* [1] as an alternative technique, which considerably reduces the complexity of developing such context-aware applications by preserving a conventional programming style instead of requiring code to be structured around event handlers. Using this approach, objects on nearby devices are expressed as collections of references to these objects that dynamically reflect the state of the network. In this case however, communication with remote parts of the application boils down to iterating over such dynamic collections and manually encoding the appropriate communication scheme (e.g. sending a message to all objects joining the collection). This results in application logic polluted with recovery code for dealing with devices that disconnected while iterating over these collections. A second difficulty is that there is no easy way to meaningfully aggregate results of broadcast messages computed in parallel with the requesting application component. Our

position is that these problems can be solved by introducing group communication abstractions tailored towards a Reactive Programming environment.
In section 2, first Reactive Programming in AmbientTalk is explained. Subsequently, a motivating example is given where group communication abstractions are clearly desirable to be able to deal with the problems discussed above. In section 3, existing group communication techniques are discussed and it is shown why they cannot be used in a reactive environment. The last section is our position statement.

2 Reactive Context-aware Programming in AmbientTalk

Consider a shopping cart equipped with a RFID reader which allows it to detect all tagged products that are placed in the cart. The cart has a display which is used to display the total price of all items in the cart. We first define two ordinary functions. `calculateTotalCost` will iterate over a collection of prices to compute the total cost and `printCost` is used to show the price of all items in the shopping cart to the users.

Listing 1.1. `calculateTotalCost` and `printCost` functions.

```
def calculateTotalCost(aPriceList) {
  def sum := 0;
  aPriceList.each: { |price| sum := sum + price };
  sum
};

def printCost(aCost) {
  system.println("Total cost: " + aCost.asString());
};
```

We want to use these functions defined above to continuously display the price of the products in the cart. In Reactive Programming languages, a network of dependent computations is constructed implicitly by calling lifted functions which operate on time-varying values called *behaviors* [2]. Assume that the `RFIDTAGSINCART` variable shown in listing 1.2 denotes such a behavior that constantly represents the RFID tags within a user's shopping cart. The RFID tags are represented as objects which implement a `getPrice` method. A new behavior `pricesOfContainedProducts` is created by applying the `map` method to `RFIDTAGSINCART`. Whenever the `RFIDTAGSINCART` behavior changes, the `map` function is applied. The argument of `map` is a block closure which invokes `getPrice()` on every RFID tag. By passing the newly created behavior `pricesOfContainedProducts` to the (plain) `calculateTotalCost` function, the function is implicitly lifted and its result is a new behavior `totalCost`. This `totalCost` behavior is passed to `printCost`, which will print the cost on the screen each time the total cost changes.

Listing 1.2. Programming a reactive shopping cart in AmbientTalk.

```
def pricesOfContainedProducts := RFIDTAGSINCART.map: { |tag| tag.getPrice() };

def totalCost := calculateTotalCost(pricesOfContainedProducts);
```

```
printCost(totalCost);
```

2.1 Reactive Programming and Asynchronous Communication

Assume that the RFID tags from the example above are remote objects able to compute their price autonomously. Communication with remote objects is only possible by sending them asynchronous messages (using the `<-` operator). In listing 1.3, the list of RFID tags in our shopping cart is again represented by the `RFIDTAGSINCART` behavior, but this time it contains remote objects. Having this list of communication partners, the programmer is supposed to iterate over the individual objects, send them asynchronous messages and gather results from these messages. The result of an asynchronous invocation is a *future*, a placeholder for the result which is computed in parallel. Accessing the result requires installing an observer on the future using the `when:becomes:catch:using:` construct, as shown in the new version of the `calculateTotalCost` function. To deal with the fact that in a mobile ad hoc network communication partners may leave the network *during the iteration* over this list, the future observers must also specify how to handle exceptions.

Listing 1.3. Reactive Programming and asynchronous communication.

```
// This is now a list of futures
def pricesOfContainedProducts := RFIDTAGSINCART.map: { |tag|
  tag<-getPrice()        // Returns a future
};

def sum;

def calculateTotalCost(aFutureList) {
  sum := 0;
  aFutureList.each: { |priceFuture|
    when(priceFuture) becomes: { |price|
      sum := sum + price;
    } catch: TimeoutException using: { |exc|
      system.println("Connection with tag lost!")
    }
  }
};

calculateTotalCost(pricesOfContainedProducts);

printCost(sum);
```

One possible solution is splitting up the computation in multiple future observers and using a mechanism external to the reactive dataflow to aggregate the results (e.g. a shared variable or data structure containing results, `sum` in this example). This is essentially writing the application in a classic event-driven style where events are generated by the resolutions of futures, introducing the same code structuring problems as mentioned in section 1. Also note that by setting `sum` to `0` in the `calculateTotalCost` function, event handlers from previous iterations can lead to incorrect sums when they are triggered with a slight delay. Furthermore, the `sum` variable is not a behavior, which means that the display of the shopping cart will not be automatically updated anymore.

The alternative is accumulating all the replies from the asynchronous invocations in a list by waiting for each reply in the `map` loop. However, by introducing such a synchronization point, we may render the application unresponsive and increase the chances of unreachable communication partners while the loop is still executing [3].
In both approaches it is impossible to write a straight forward implementation such as in listing 1.2. These problems, that even show up when simply adding some numbers like in the example above, stem from the fact there is not enough abstraction for communicating with actual states of (parts of) the network. We intend to solve them by introducing group communication abstractions that allow us to communicate with volatile sets of objects by returning a reactive group of results.

3 Group Communication Abstractions for a Distributed Reactive Environment

Given the properties of the pervasive applications in mobile ad hoc networks that we envision, we have identified the requirements that we will impose on group communication abstractions for a distributed reactive environment:

- **Broadcasting of messages:** It should be possible to broadcast a single message to all objects in the group without explicitly referencing the objects. By holding explicit references to distributed objects, code has to be structured around recovery mechanisms to deal with the frequent disconnections.
- **Sustained communication:** Objects should not be known beforehand to be able to join the group and messages should be transparently sent when new objects join the group. This is necessary to reflect the dynamic nature of the network.
- **Reactive aggregation of results of asynchronous invocations:** It should be possible to specify how to aggregate the results of the broadcasted messages into a *reactive* value (e.g. one result only, list of incoming results, application-specific accumulation of results...). The fact that this value must be reactive (e.g. a behavior) is the key to be able to connect the results of asynchronous message sends to the dataflow of the sequential reactive program in a meaningful way.

3.1 Existing Group Communication Abstractions

Group communication abstractions are often used to provide transparent replication in fault tolerant systems. An example of this are Concurrent Aggregates [4]. Concurrent Aggregates are abstractions behaving like a single distributed object, but actually consist of a group of objects. Messages sent to an Aggregate are executed by a non-deterministically selected object in the group. Hence, Concurrent Aggregates do not support broadcasting of messages.
ActorSpaces [5] are also groups of distributed objects, but can be defined using

an intensional description (i.e. the objects do not have to be known beforehand). Messages can both be sent to a single object in an ActorSpace, or can be broadcasted to all objects in the group. However, once the ActorSpace is created, the objects it consists of are fixed and no objects can join or leave the group. Therefore, ActorSpaces do not support sustained communication.
M2MI handles [6] are anonymous references to remote objects exported by means of a Java interface type designed specically for ad hoc wireless proximal networks. M2MI distinguishes between uni, multi and omnihandles. Unihandles refer to one specific proximate object, multihandles to a specific group of proximate objects and omnihandles to all proximate objects of the handles interface type. M2MI handles only support the asynchronous invocation of methods that do not return a value or raise no declared exceptions. Moreover, messages are delivered only to receivers currently in range, and otherwise immediately discarded, which means that M2MI handles are not capable of sustained message sending to new objects.
AmbientTalk has its own construct for referencing a volatile set of remote objects: Ambient References [7]. Remote objects designated by an Ambient Reference do not have to be known beforehand, but can be dynamically discovered. The arity of the communication, the time to keep broadcasting the message to newly discovered objects and the time to wait for replies can be specified. If a one-to-many communication scheme is used, sending a message to an Ambient Reference returns a *multifuture*. Multifutures behave like normal futures, with the difference that they can be resolved multiple times. Different observer constructs are available for gathering results from one-to-many invocations (i.e. trigger on the first returned value, trigger on each new value and trigger when no more new values will be returned). Because future observers have to be used instead of a reactive value to gather the results of broadcasted asynchronous messages, we cannot use Ambient References in a reactive system.

3.2 Our Approach

A technique not specifically aimed at distributed and concurrent applications is Higher Order Messaging [8]. Higher order messages are messages sent to an object that take other messages as arguments. The higher order message decides how to deliver the argument messages to the receiver and what to do with the results of the argument messages. Higher Order Messaging may be an interesting way of implementing different group communication strategies in a reactive environment, especially in languages where messages are first class objects, such as AmbientTalk. As an example, we adapt the code from section 2:

Listing 1.4. Reactive Programming and Higher Order Messages.

```
def pricesOfContainedProducts := RFIDTAGSINCART.collectResultsOf: <−getPrice();

def calculateTotalCost(aPriceList) {
  def sum := 0;
  aPriceList.each: { |price| sum := sum + price };
  sum
};
```

```
def printCost(aCost) {
  system.println("Total cost: " + aCost.asString());
};

def totalCost := calculateTotalCost(pricesOfContainedProducts);

printCost(totalCost);
```

Assume that `collectResultsOf:` is a higher order message defined on the `RFIDTAGSINCART` behavior that broadcasts the first class `getPrice` argument message (constructed using the `<-` operator) to all RFID tags in the behavior (including to newly discovered tags). Replies from the RFID tags are collected into a new behavior which contains a list that is maintained by the Reactive Programming system to contain the results of the message sends to currently connected (i.e. present in the shopping cart) RFID tags. Newly received responses are automatically collected and responses of (disconnected) tags that are taken out of the shopping cart are automatically discarded. With such a construct that exploits the Reactive Programming system, the rest of the code from the example in section 2 can remain unchanged.

4 Position Statement

Writing context-aware applications involves reacting to changes in the environment to adopt or fire the correct behavior. Using state of the art event-driven techniques, applications are required to register event handlers which will be triggered when context changes occur. Relying on explicit event handlers has effects which percolate throughout the entire design of the application, as access to shared data needs to be protected from race conditions and context dependencies in the program need to be encoded manually by registering observers.
Using the Reactive Programming paradigm to implicitly reflect the context of an application is a novel approach that avoids these problems. However, current Reactive Programming mechanisms are not integrated with asynchronous communication, which we deem necessary for distributed applications in a mobile ad hoc network [9]. In this case, the application still has to be written in a classic event-driven style to bridge the gap between sequential (local) code and (remote) requests executed in parallel. We propose the use of group communication abstractions to abstract away the current state of the network. To integrate them in a Reactive Programming language, we impose a number of requirements such that asynchronous group communication can be connected to sequential code without losing the power of implicitly reflecting the state of the network.

References

1. Mostinckx, S., Lombide Carreton, A., De Meuter, W.: Reactive context-aware programming. In: Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2008). Volume 10 of Electronic Communications of the EASST., DisCoTec (June 2008)

2. Cooper, G.H., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In Sestoft, P., ed.: ESOP. Volume 3924 of Lecture Notes in Computer Science., Springer (2006) 294–308
3. Friedman, R.: Fuzzy group membership. In Schiper, A., Shvartsman, A.A., Weatherspoon, H., Zhao, B.Y., eds.: Future Directions in Distributed Computing. Volume 2584 of Lecture Notes in Computer Science., Springer (2003) 114–118
4. Chien, A.A., Dally, W.J.: Concurrent aggregates (ca). In: PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming, New York, NY, USA, ACM Press (1990) 187–196
5. Agha, G., Callsen, C.J.: Actorspace: an open distributed programming paradigm. In: PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM Press (1993) 23–32
6. 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, New York, NY, USA, ACM (2002) 72–73
7. Van Cutsem, T., Dedecker, J., Mostinckx, S., Gonzalez, E., D'Hondt, T., De Meuter, W.: Ambient references: addressing objects in mobile networks. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM (2006) 986–997
8. Weiher, M., Ducasse, S.: Higher order messaging. In: DLS '05: Proceedings of the 2005 conference on Dynamic languages symposium, New York, NY, USA, ACM Press (2005) 23–34
9. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented Programming in Ambienttalk. In Thomas, D., ed.: Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP). Lecture Notes in Computer Science, Springer (2006) 230–254