Event-driven Mobile Computing With Objects

Tom Van Cutsem and Wolfgang De Meuter
Programming Technology Lab, Vrije Universiteit Brussel, Belgium

ABSTRACT

We motivate why event-driven approaches are suitable to address the challenges of mobile and ubiquitous computing. In particular, we describe the beneficial properties of event-based communication in so-called mobile ad hoc networks. However, because contemporary programming languages feature no built-in support for event-driven programming, programmers are often forced to integrate event-driven concepts with a different programming paradigm. In particular, we study the difficulties in combining events with the object-oriented paradigm. We argue that these difficulties form the basis of what we call the object-event impedance mismatch. We highlight the various issues at the software engineering level and propose to resolve this mismatch by introducing a novel object-oriented programming language that supports event-driven abstractions from the ground up.

INTRODUCTION

This chapter focuses on programming abstractions for mobile computing (Mascolo, Capra, & Emmerich, 2002), a research domain that studies Weiser's vision of ubiquitous computing (Weiser, 1991) from a distributed systems' perspective. Mobile computing applications are deployed on mobile devices (e.g. cellular phones, PDAs, …) equipped with wireless communication technology (e.g. WiFi, Bluetooth,…). Such devices form so-called *mobile ad hoc networks*, which are characterized by the fact that connectivity between devices is often intermittent (connections drop and are restored as devices physically move about) and the fact that there is little or no fixed support infrastructure, such that devices can often communicate only with physically proximate devices.

Event-based coordination is a natural fit for such networks. Events can be disseminated to multiple nearby interested parties (subscribers) without necessarily knowing the exact identity of these subscribers. This key property of event-based systems is crucial in a ubiquitous computing context, where the identity and number of nearby devices is not known at development time.

Contemporary software is not built using event-driven abstractions from the ground up. Rather, software development is predominantly object-oriented. In this paradigm, distributed applications are expressed in terms of distributed objects sending messages to one another. Thus, an application programmer using a mainstream object-oriented language will be forced to implement his or her own event infrastructure on top of the object-oriented infrastructure.

We will describe that combining object technology with event-based technology is not without problems. In a nutshell, the most important differences are the following. Objects communicate by means of messages, not by means of events. They do so via *remote object references*, which can only refer to a single object throughout their lifetime. Using an event-based system, however, a publisher can send messages to an arbitrary number of subscribers. On the other hand, objects introduce useful abstractions such as request/response interactions that cannot be directly expressed using pure event-based publish/subscribe communication. We have named the combination of these and a number of other issues

the *object-event* impedance mismatch, by analogy with the *object-relational* impedance mismatch which describes the difficulties in combining objects with relational databases for the purpose of persistence (Carey & DeWitt, 1996).

We resolve the object-event impedance mismatch by means of a novel object-oriented programming language named *AmbientTalk* (Van Cutsem, Mostinckx, Gonzalez Boix, Dedecker, & De Meuter, 2007) that supports event-based programming from the ground up. For example, AmbientTalk provides asynchronous event notification between objects as a primitive operation. Rather than using a traditional multithreaded concurrency model, the language features a reactive event loop concurrency model (Miller, Tribble, & Shapiro, 2005). Also, the language supports referencing abstractions that allow a single object to directly refer to an entire group of distributed, proximate objects. Messages sent via such references are automatically treated as events that are broadcast to all objects subscribed to the group.

After having briefly introduced AmbientTalk's main concepts, we show how the language enables one to program in an object-oriented yet event-driven way thus overcoming the object-event impedance mismatch.

The objectives of this chapter are threefold. Our goal is to:

1. give a detailed understanding of why event-driven communication is highly suitable in mobile ad hoc networks.
2. discuss the key differences between event-driven and object-oriented programming, and the consequences thereof for the developer.
3. propose a novel programming model that is both object-oriented and event-driven, allowing developers to use the strengths of both models without unnecessary complications raised by the object-event impedance mismatch.

BACKGROUND

Many systems designed specifically for mobile ad hoc networks (MANETs) adopt an event-driven communication paradigm. We will support this claim by discussing a number of concrete event-driven middleware systems designed for MANETs. Before doing so, we first describe why event-driven communication is so useful in MANETs by showing how it promotes loose coupling between communicating parties.

Coupling Properties of Event-driven Communication

The advantages of event-driven communication lie in its loose coupling between communicating parties. In mobile ad hoc networks, such loose coupling is important because it allows communicating parties to abstract from the physical connectivity provided by the network, which is in constant flux because devices move in and out of communication range in unpredictable ways. By decoupling a communicating process from the underlying physical connectivity, communication is made more resilient in the face of temporary network disconnections, as will be explained later.

The following three properties are well-known in the literature, especially in the context of publish-subscribe architectures (P. Eugster, Felber, Guerraoui, & Kermarrec, 2003). They pertain to *decoupling* the communicating parties along three dimensions.

Decoupling in Time

Event-driven communication can be made decoupled in time, which implies that communicating processes do not necessarily need to be online simultaneously at the time an event is published. It is mostly achieved by buffering events in message queues, or by introducing a third-party "event broker"

which stores events on behalf of the publisher. This allows the publisher's events to be delivered to subscribers even after the publisher has gone offline.

Decoupling in time makes it possible for communicating parties to interact across intermittent connections, because events may be stored (by the publisher or by an event broker) while the network connection is down and transmitted when the connection is restored.

Decoupling in Space

Event-driven communication can decouple processes in space. This means that a process does not necessarily need to know the identity or the total number of processes with which it is communicating.

Decoupling in space has a number of advantages over communication that is tightly coupled in space (such as an RPC call or a remote method invocation), especially in mobile ad hoc networks. First, it allows event-driven communication to be anonymous, in the sense that publishers do not need to know the exact identity of the subscribers interested in their events. When using an event broker, the publisher need only know the identity of the broker. If the publisher directly communicates events to interested subscribers, communication can still be decoupled in space if the publisher can simply broadcast events to all nearby processes, allowing receivers to filter the event themselves based on the relevancy of the event's type or content.

A second advantage of space-decoupling is that it enables publishers to abstract from the total number of subscribers, and that it enables subscribers to abstract from the total number of publishers. This enables applications to adapt more gracefully to changes in the ad hoc network. Publishers and subscribers may be registered or unregistered with an event broker (e.g. because devices move in or out of communication range) without requiring changes in already registered publishers or subscribers.

Synchronization Decoupling

Event-driven communication decouples publishers and subscribers in synchronization. This implies that the control flow of publishers is not blocked (suspended) upon publishing events (i.e. a publisher does not have to wait for subscribers to process the event), or from the point of view of the subscribers, that subscribers do not block the control flow of the publisher while processing an event. Publishers publish events asynchronously, allowing their thread of control to continue processing, while the event broker takes care of delivering the events to registered subscribers.

Synchronization decoupling is important in mobile ad hoc networks, where high network latencies and frequent network disconnections render synchronous RPC-style communication impractical. Asynchronous communication is favorable when network latencies are high, and as previously noted, asynchronously published events can easily be buffered when the network connection with the event broker or subscribers is temporarily down.

State of the art in event-driven middleware for MANETs

We previously argued that many systems designed for MANETs employ an event-driven communication paradigm. In this section, we demonstrate this by discussing a number of concrete systems. We divide related work into two broad categories: publish/subscribe systems and tuple space-based systems.

Publish/Subscribe Systems

In a publish/subscribe system, publishers and subscribers exchange data by means of event notifications. Subscribers may register to receive certain event notifications based on the type of the event (a.k.a. topic-based subscription) or on the event's contents directly (a.k.a. content-based subscription) (P. Eugster et al., 2003).

Many middleware systems for mobile ad hoc networks adapt the traditional Publish/Subscribe architecture with additional semantics that allow publishers and subscribers to define a physical range to scope the events they want to publish or receive. Two representative examples are Location-based Publish/Subscribe (LPS) (P Eugster, Garbinato, & Holzer, 2005) and Scalable Timed Events and Mobility (STEAM) (Meier & Cahill, 2003).

Location-based Publish/Subscribe

LPS is a content-based publish/subscribe architecture designed for nomadic networks (i.e. it is assumed that mobile devices can communicate via a shared infrastructure, e.g. a GSM or GPRS network). In order to scope interactions between devices, event dissemination and reception is bounded in physical space: a publisher defines a *publication range* and a subscriber defines a *subscription range*. Both are independent of the mobile devices' communication range. Only when the publication range of the publisher and the subscription range of the subscriber physically overlap is an event disseminated from the publisher to the subscriber. LPS decouples publishers and subscribers in time, space and synchronization. Decoupling in time is bounded by an event's *time-to-live*: after this timeout period has expired, an event is no longer published.

Scalable Timed Events and Mobility

STEAM is an event-based middleware designed for collaborative applications in mobile ad hoc networks. It shuns the use of centralized components such as lookup and naming services to avoid any dependencies of mobile devices on a common infrastructure. In STEAM, events can be filtered according to event type, event content and physical proximity. STEAM builds upon the observation that the physically closer an event consumer is located to an event producer, the more interested it may be in that producer's events. For example, in a Vehicular Ad hoc Network (VAN), cars can notify one another of accidents further down the road, traffic lights can automatically signal their status to cars near a road intersection or ambulances could signal their right of way to cars in front of them. To this end, STEAM allows events disseminated by producers to be filtered based on geographical location using *proximities* which are first-class representations of a physical range. Proximities may be absolute or relative (i.e. a relative proximity denotes an area surrounding a mobile node, changing as the node moves).

STEAM decouples publishers and subscribers in space and synchronization. It does not decouple them in time: published events are disseminated using multi-hop routing throughout their proximity, after which they disappear. Hence, if a subscriber is not in range at the time the event is disseminated, it will miss the event. Events must be made persistent by repeatedly publishing them.

One.world

One.world (Grimm et al., 2004) is a system architecture developed on top of Java, providing a common execution platform for pervasive computing applications. Again, an asynchronous publish/subscribe style of interaction is promoted because of its loose coupling, making communication decoupled in time, space and synchronization.

Epidemic Messaging Middleware for Ad Hoc Networks

Closely related to publish/subscribe systems are message queuing systems in which the event broker between publishers and subscribers is represented as an explicit queue. Popular message queuing systems, such as the Java Message Service (JMS) (Hapner, 2002) have been adapted for use in a mobile setting. One such adaptation is the Epidemic Messaging Middleware for Ad Hoc Networks (EMMA) (Musolesi, Mascolo, & Hailes, 2005) In JMS, Java components interact asynchronously by posting messages to and

reading messages from message queues. This can be used both for point-to-point and publish/subscribe interaction. In JMS, queues are often managed by central servers. EMMA replaces such central servers by a discovery mechanism that allows queues to be discovered in the local ad hoc network.

EMMA, like JMS, distinguishes between durable and non-durable subscriptions to message queues. A durable subscription remains valid upon disconnection. Non-durable subscriptions are cancelled upon disconnection and the subscription must be made anew upon reconnection. In JMS, the server buffers events while a durable subscriber is disconnected. In EMMA, events for disconnected subscribers are not buffered but rather sent using an asynchronous *epidemic routing protocol*. Using this protocol, messages are broadcast to each host in range, which in turn sends them to all hosts in its range, and so on. Epidemic routing does not guarantee message delivery, but the delivery ratio increases as the number of nodes in the ad hoc network increases. If a message is flagged as *persistent*, the sender is notified of successful delivery via an acknowledgement.

Communication in EMMA is naturally synchronization-decoupled using message queues. It is space decoupled thanks to the use of topics to describe publish/subscribe queues. Thanks to its automatic discovery management of queues, it is suitable for use in pure ad hoc networks.

Tuple Space-based Systems

Tuple spaces have originally been introduced in the coordination language Linda (Gelernter, 1985). In the tuple space model, processes communicate by inserting and removing tuples from a shared tuple space, which acts like a globally shared memory. Because tuples are anonymous, they are taken or copied from the tuple space by means of pattern matching on their content. Tuple space communication is decoupled in time because processes can insert and retract tuples independently. It is decoupled in space because the publisher of a tuple does not necessarily specify, or even know, which process will extract the tuple. This makes Linda ideal for coordinating loosely-coupled processes. The original Linda model has since been ported to contemporary languages such as Java resulting in artifacts such as IBM's TSpaces (Tobin et al., 2001) and Sun Microsystems' Javaspaces (Freeman, Arnold, & Hupfer, 1999).

Linda in a Mobile Environment

Tuple spaces have received renewed interest by researchers in the field of mobile computing. One shortcoming of the original tuple space model in light of mobile computing is the fact that synchronization decoupling is violated because there exist synchronous (blocking) operations to extract tuples from the tuple space. However, as the need for total synchronization decoupling became apparent for mobile networks, mobile computing middleware such as Linda in a Mobile Environment (LIME) (Murphy, Picco, & Roman, 2001) extends the basic model with *reactions* which are callbacks that trigger asynchronously when a matching tuple becomes available in the tuple space.

Naturally, a globally shared, centralized, tuple space does not fit the hardware characteristics of mobile ad hoc networks. Adaptations of tuple spaces for mobile computing, such as LIME, introduce agents which have their own, local *interface tuple space* (ITS). Whenever their host device encounters proximate devices, the ITS of the different agents is merged into a federated *transiently shared* tuple space, making tuples in a remote agent's tuple space accessible while the connection lasts.

Mobile Agent Reactive Spaces and Tuples on the Air

Other adaptations of the tuple space model for mobile ad hoc networks include Mobile Agent Reactive Spaces (MARS) (Cabri, Leonardi, & Zambonelli, 2000) and Tuples on the Air (TOTA) (Mamei & Zambonelli, 2004). These approaches circumvent the need for a globally shared tuple space by allowing

agents (in the case of MARS) or tuples (in the case of TOTA) to migrate between connected hosts. Using migration, agents and tuples can be co-located to ensure a stable communication.

In MARS, each device hosts a tuple space and agents can only access that local tuple space. To access another tuple space, agents can migrate between hosts. MARS features a metalevel tuple space that allows programs to register reactions: callbacks that trigger whenever agents perform a read and/or write operation on the baselevel tuple space.

In TOTA, rather than merging local tuple spaces upon network connection, tuples are equipped with a *propagation rule* that determines how the tuple migrates from one tuple space to another. Hence, in TOTA, agents can access one another's tuples because it are the tuples themselves that propagate through the network as connections are established. Like LIME, it augments the tuple space model with a form of event notification to notify agents when certain tuples arrive in their tuple space.

Tuple spaces act as a middle man between different processes. As a result, there is no notion of a reference to any particular process. Tuple space-based communication is necessarily global to all processes sharing the tuple space, which may lead to unexpected interactions between concurrently communicating processes.

In the following section, we discuss the difficulties of combining event-driven communication abstractions with the object-oriented programming paradigm. Subsequently, we show how these difficulties can be overcome, by introducing a novel programming language named AmbientTalk, which successfully combines objects with events.

THE OBJECT-EVENT IMPEDANCE MISMATCH

In the previous section, we have discussed that event-driven communication is effective in ad hoc networks because it minimizes the dependencies between publishers and subscribers in time, space and synchronization. Alternative communication mechanisms (e.g. RPC or remote method invocation) do not engender such loose coupling (P. Eugster et al., 2003). Traditional remote method invocations couple participants in time (to perform the invocation, the receiver object must be online), space (the receiver must be known, and there is only one receiver) and synchronization (the invocation is performed synchronously). Saif and Greaves (2001) provide additional arguments against the use of RPC-based communication in ubiquitous computing systems.

Given the above arguments, we are forced to conclude that:

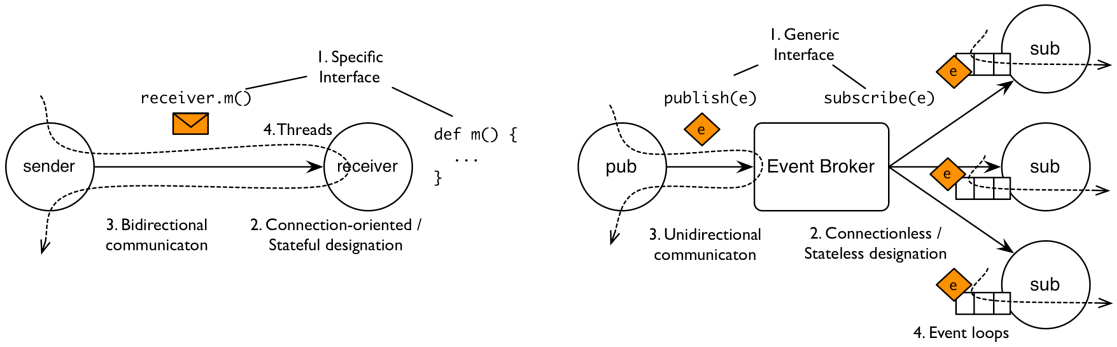
1. Object-oriented communication by means of traditional message passing schemes fails to provide a total decoupling of processes, which becomes a major obstacle when used in a distributed system connected by means of a mobile ad hoc network.
2. Publish/Subscribe and Tuple Space-based systems enable the best decoupling of processes, which makes them highly suitable for use in mobile ad hoc networks.

Combining these two facts, it appears that one is forced to abandon the object-oriented message passing abstraction if one wants to express scalable coordination between processes in a MANET. However, if the non-distributed part of an application is written in an object-oriented language, the overall application is then forced to combine two different paradigms: the object-oriented paradigm must interact with an alien communication paradigm (tuples or events).

A multi-paradigm approach is not necessarily a bad approach. However, its suitability depends on the difficulty of combining the paradigms involved. Below, we argue that combining objects with events is far from trivial. We claim that the lack of integration between the concepts from both paradigms leads to what we call the *object-event impedance mismatch*, analogous to the way object persistence suffers from the infamous *object-relational* impedance mismatch (Carey & DeWitt, 1996).

The object-relational impedance mismatch is caused by the fundamental differences between modeling data as objects and modeling data as tuples that are part of relations. For example, objects encapsulate their state, enabling operations to be polymorphic. Tuples expose state, enabling efficient and expressive filtering, querying and aggregation of data. Objects refer to one another via references, while tuples are associated with one another via foreign keys. Identity is fundamental to objects, while tuples lack any inherent form of identity, and so on. We discuss similar differences, but rather than contrasting objects with the relational model, we will contrast objects with event-driven communication models. ***Figure 1*** contrasts object-oriented with event-driven communication. The highlighted differences are discussed in each of the following sections.

Figure 1: Contrasting object-oriented communication (on the left) with event-driven communication (on the right).



Specific versus Generic Communication

In object-oriented programming languages, objects communicate by means of message passing. The interface of an object usually corresponds to the set of visible methods it (or its class) defines. Messages encapsulate a "selector", which usually corresponds to the name of the method to be invoked on the object to which the message is sent. In object-oriented programming, therefore, communication between objects is expressed in terms of very *specific* operations. The generic acts of *sending* and *receiving* these messages are entirely hidden by the language.

In a distributed object-oriented language, sending and receiving messages to and from *remote* objects is most often done implicitly by means of the regular message passing semantics already provided by the language. In an object-oriented language, remote communication can be succinctly expressed as `receiver.selector(arg)`. No explicit `send` operation is required. Likewise, it is often not necessary to introduce an explicit `receive` statement: received messages implicitly lead to the invocation of a method on the receiver object.

In event-driven systems and in tuple spaces, communication between processes is not in terms of specific operations, but rather in terms of very generic operations, whose arguments are the events or tuples to be communicated. This is very obvious in tuple spaces, where communication is performed in terms of explicit `in(tuple)` and `out(tuple)` operations. In an event-driven system, events are often dispatched to the event broker (the infrastructure between event producers and consumers) by means of a generic `broker.publish(event)` invocation. Likewise, event reception is often an explicit operation:

```
broker.subscribe(new Subscriber() {
  public void reactTo(Event e) {
    // handle incoming event
```

```
  }
})
```

One advantage of a generic communication interface is that it is much easier to express general patterns of communication. This is because such an interface already automatically abstracts from the details of the data being communicated. To express generic patterns via an object-oriented interface, one generally requires reflection to be able to intercept messages without reference to their specific selector and arguments (e.g. via Smalltalk's `doesNotUnderstand:` protocol (Foote & Johnson, 1989) or Java's dynamic proxy classes (Sun Microsystems, 1999)).

In short, if objects are to communicate with one another via events, they must abandon their otherwise specific communication interface in favor of the generic interface promoted by the event system.

Connection-oriented versus Connectionless Designation

In a distributed object-oriented program, objects communicate via point-to-point channels known as (remote) object references. This abstraction provides limited or no support for space-decoupled (anonymous and one-to-many) communication. However, it remains a useful communication abstraction with two important properties. First, a remote object reference is a *connection-oriented* communication channel, which allows the sender to know the identity of the object to which it sends messages. Second, it is a *stateful* communication channel ensuring that multiple messages sent via the same reference are processed by the *same* receiver. Later messages are sent via the reference in the understanding that messages sent earlier are processed first.

Publish/subscribe systems and tuple spaces feature stateless one-to-many communication by default. In a mobile ad hoc network, this allows publishers to simply broadcast events to nearby interested subscribers without knowing their identity or their total number. At the same time it becomes difficult to express connection-oriented communication, as an event or tuple becomes accessible to *all* registered subscribers, or stateful communication, because multiple consecutive events may be received by a *different* set of subscribers. There exist ways of building connection-oriented communication on top of a publish/subscribe system (e.g. private event topics encoding a connection), but these abstractions are second-class, just like connectionless one-to-many communication is a second-class abstraction in an object-oriented programming language.

Note that the notion of a connection-oriented communication channel does not necessarily imply a *secure* communication channel. The above discussion deals with scoping and coupling issues on a software engineering level, not with issues such as one process intentionally "eavesdropping" on a communication channel between other processes.

In short, if objects are to communicate with one another via events, they must abandon the remote referencing abstraction in favor of connectionless communication via the event broker. This makes the receiver(s) of a message anonymous, which enables space-decoupled communication, but at the same time may lead to interference between communicating objects.

Bidirectional versus Unidirectional Communication

Object-oriented programs communicate via message passing which fosters a request/reply style of communication. Even though this request/reply style is often implemented by means of synchronous (remote) method invocation, much research in concurrent object-oriented programming has been devoted to maintain the request/reply interaction pattern while relaxing the synchronization constraints (e.g.

future-type message passing in ABCL (Yonezawa, Briot, & Shibayama, 1986), wait-by-necessity in ProActive (Baduel et al., 2006)).

Publish/subscribe systems and tuple spaces decouple processes by essentially introducing pure asynchronous one-way operations (e.g. publishing an event, writing a tuple). Request/response interaction can of course be built on top of such systems, e.g. by manually correlating request and response events/tuples by means of an identifier. Again, these are second-class abstractions.

Conversely - and perhaps less obviously - an asynchronous, unidirectional event notification also has to be represented by means of second-class abstractions in an object-oriented language. Signaling an asynchronous event is often performed by *synchronously* invoking a "notification" method that returns no result. The first-class asynchronous notification of an event system is thus represented confusingly as a synchronous method invocation in an object-oriented system.

In short, if objects are to communicate with one another via events, they must abandon the bidirectional request/response message passing abstraction in favor of unidirectional event notification.

Threads versus Event Loops

In an event-driven system, events are usually delivered to an application by an *event loop* which is an infinite loop that accepts incoming events and dispatches them to the appropriate event handler. However, the integration of event delivery with multithreaded object-oriented languages often leaves much to be desired. The archetypical integration represents event handlers as "listener" or "observer" objects whose methods are invoked directly by the event loop. Because of the synchronous method invocation semantics predominant in object-orientation, it is the thread of control of the event loop itself that executes the method. The following code snippet depicts a canonical example of event notification in Java:

```
// code executed by application thread
broker.subscribe(new Subscriber() {
  public void reactTo(Event e) {
    // code executed by event notification thread
  }
});
```

This style of event notification has two important consequences:

- Event handler objects must be made multiple thread-safe, i.e. they require synchronization constructs to prevent data races when they concurrently access state manipulated by application-specific threads. Also, because thread management lies outside the control of the application, it is entirely implicit in the code whether multiple events are signaled to registered event handlers concurrently or sequentially. A change in the thread management may thus introduce race conditions into the application.
- If the event handler object uses the event loop's thread of control to perform application-level computation, it can make the event loop unresponsive. A testament to this is the documentation of the event-driven Java GUI construction framework Swing that advises developers to structure their applications such that listener methods relinquish control to the framework as soon as possible (Sun Microsystems, 2008).

Event loops provide the programmer with the ability to consider the handling of a single event as the unit of concurrent interleaving. The major strength of this model is that it significantly raises the level of abstraction for the developer: rather than having to consider the possible interleaving of each *basic instruction* in each method body, the programmer need only consider the interleaving of each distinct event. It has been argued even in the context of thread-based object-oriented concurrency models that complete mutual exclusion of each method of a concurrently accessible object ought to be the norm (Meyer, 1993).

Event loops also have their drawbacks. Event delivery is an asynchronous process, and most event-driven systems cannot succinctly express the overall control flow of an application. Rather, the control flow is dispersed across many different event handlers, a phenomenon known as *inversion of control* (Haller & Odersky, 2006) or *stack ripping* (Atul, Jon, Marvin, William, & John, 2002) in the literature. This often results in code and data that is fragmented across calls and callbacks.

Choosing between thread-based or event-based concurrency is the topic of long-standing debates in the literature, and we are certainly not the first to contrast these two systems. A well-known talk by Ousterhout provides a more general discussion comparing the assets and drawbacks of threads and events (Ousterhout, 1996). Miller (2006) studies the concurrency control properties of both models in detail.

In short, if objects are to communicate with one another via events, the event broker becomes an additional source of concurrency in the object-oriented program. In multithreaded languages, dealing with this additional source of concurrency is non-trivial, as it requires the programmer to carefully insert additional locks to ensure overall thread-safety. What is needed is an alternative model of concurrency control between objects that allows one to easily compose new sources of events with an existing application.

Reconciling Objects with Events

In this section, we have contrasted the communication properties of object-oriented and event-based publish/subscribe models. The simplest solution to resolve the object-event impedance mismatch is to discard either objects or events and to resort to a single-paradigm solution where only one of both is used. However, it should be clear from the above discussion that both paradigms have their merits. Neither does one solve the object-relational impedance mismatch by discarding objects or relational databases.

To resolve the object-event impedance mismatch, we will develop a novel programming model that tries to *unify* as much concepts as possible in the event-driven domain with concepts in the object-oriented domain. In the unified model, the programmer can seamlessly use event-driven communication in an object-oriented program. Our unified model does need to make tradeoffs, and will combine aforementioned properties of objects and events as follows:

- Objects should be able to communicate via events by means of the familiar, *specific* communication interface afforded by message passing. This makes communication more concise because there is no need for explicit `publish` and `subscribe` operations.
- Objects should be able to communicate via events by means of the familiar object referencing mechanism. While sometimes stateful connection-oriented communication is still required, object references must be augmented such that they can directly express stateless, *connectionless* communication.
- Objects should be able to communicate via events and still retain the ability to perform the *bidirectional* interactions afforded by method invocation.
- Objects should be equipped with an *event loop* concurrency model that can adequately cope with the additional sources of concurrency introduced by event brokers.

In the next section, we show how the AmbientTalk programming language achieves the above requirements. Afterwards, we revisit the object-event impedance mismatch and how it is resolved by AmbientTalk by unifying object-oriented with event-driven communication.

AMBIENTTALK

AmbientTalk is a programming language embedded in Java. The language is designed as a distributed scripting language that can be used to compose Java components that are distributed across a mobile ad hoc network. The language is developed on top of the Java 2 Micro Edition (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 multithreaded, and provides either a low-level socket API or a high-level RPC API (i.e. Java RMI) to enable distributed computing. In contrast, AmbientTalk is a fully event-driven programming language. We discuss this is more detail below. Furthermore, network programming in AmbientTalk is only possible via an asynchronous API based on message passing. AmbientTalk is designed particularly for ad hoc networks:

- 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.
- 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).

Below, we first discuss AmbientTalk's event loop concurrency model in more detail. Subsequently, we explain AmbientTalk's support for publish/subscribe interaction by means of a new kind of object references named ambient references.

Event Loop Concurrency

In AmbientTalk, code is not executed by threads but rather by event loops. Each event loop perpetually processes events from an event queue and dispatches these events to appropriate event handlers. This works similar to how GUI frameworks (e.g. Java AWT or Swing) operate. All concurrent activities in the system are represented as events that are asynchronously handled by event loops. Unlike threads, event loops have no mutable shared state and communicate strictly by means of events. Because of this, event loops never have to lock state such that locks become unnecessary. Because there are no locks, event loops can never suspend on one. Therefore event loops cannot deadlock one another.

AmbientTalk maps the above concepts from event-driven programming onto concepts from object-oriented programming. The mapping is based upon the model of communicating event loops introduced in the E programming language (Miller et al., 2005). This model is itself based on the well-known actor model of computation (Agha, 1986). In AmbientTalk, event loops are known as *actors*. The event queue of an actor is called a *mailbox*. Events themselves are represented as *messages*. Firing an event is done by

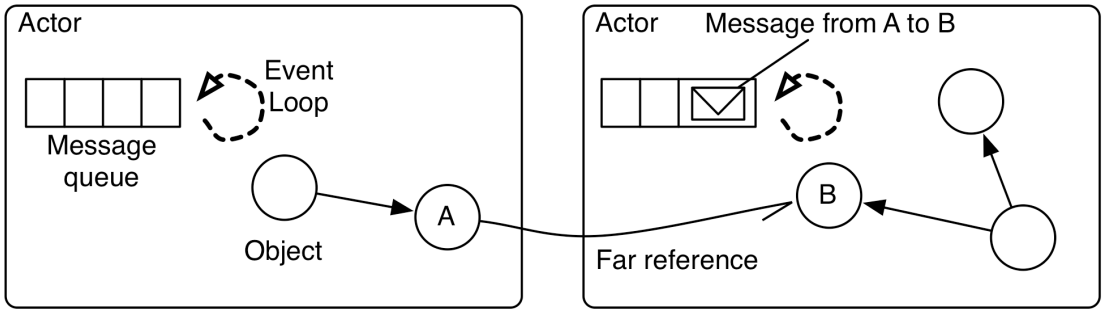
asynchronously sending a message to an object. The message is then enqueued in the mailbox of its actor. The actor handles the message by invoking the corresponding method on the receiver object. Hence, event handlers are represented as *methods* of the objects contained in the actor. ***Table 1*** summarizes the mapping of event-driven onto object-oriented concepts in AmbientTalk.

Table 1: Mapping event-driven concepts onto object-oriented concepts

	Event-driven Programs	AmbientTalk
Unit of concurrency	Event Loops	Actors
Communication via	Events	Messages
To send	Fire an event	Send a message asynchronously
To receive	Register a callback	Define a method
To handle	Invoke a callback	Invoke a method

In AmbientTalk, objects can communicate either by synchronous message sending (syntax: `obj.m()`) or by asynchronous message sending (syntax: `obj<-m()`). A single actor may contain multiple regular objects, and it is possible for objects contained in one actor to refer to objects contained in other actors. Such references that span different actors are named *far references* and only allow asynchronous access to the referenced object. Synchronous access to an object via a far reference raises a runtime exception. Any messages sent via a far reference to an object are enqueued in the message queue of the actor owning the object and processed by the owner itself. This is illustrated in ***Figure 2***. The dotted lines represent an actor's event loop that perpetually takes messages from its mailbox and synchronously executes the corresponding methods on its objects. The control flow of an actor's event loop never "escapes" its actor boundary. When object A sends a message to object B via a far reference, the message is enqueued in the message queue of B's actor, which eventually processes it.

Figure 2: AmbientTalk actors as event loops.



The following code snippet illustrates how AmbientTalk can be used to query a `WeatherService` object representing a weather service in the ad hoc network to retrieve weather information for a given city:

```
when: weatherSvc<-getWeather("Brussels, Belgium") becomes: { |info|
  // update weather information in the user interface
}
```

```
// code hereafter is executed immediately after the message send
```

The above code consists of an asynchronous message send and an event handler to process the reply. We assume that the weather service object is accessible via the `weatherSvc` variable, which denotes a remote AmbientTalk object that wraps a Java component implementing the weather service.

When the `getWeather` message is received by the remote `weatherSvc` object, that object's `getWeather` method is invoked. The return value of this method is used as the reply to the message. This reply is signaled asynchronously to the caller. 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 `info` variable in the example). Note that code following the `when:becomes:` control structure is executed immediately after sending the message, before the event handler code is fired.

To summarize, AmbientTalk is a scripting language embedded within Java that allows programmers to easily write distributed event-driven programs. As illustrated by means of the above example, event-driven concepts are represented in AmbientTalk by means of object-oriented concepts such as messages and method invocations. However, what is still lacking is a means to perform a publish/subscribe style of communication with remote objects. This is the subject of the following section.

Ambient References

In the weather service example, we assumed that the `weatherSvc` variable contained a reference to the remote weather service object. In this section, we discuss how such a reference can be easily constructed by means of *ambient references*. An ambient reference is a reference to one ore more service objects of a given type that are available in the ad hoc network. Usually this type corresponds to a Java interface that the service(s) implement. For example, a reference to a nearby weather service can be constructed as follows, assuming that `WeatherService` represents a Java interface:

```
def weatherSvc := ambient: WeatherService;
```

The variable `weatherSvc` contains an ambient reference, which is a proxy for any nearby object exported (made available) via the `WeatherService` interface. The AmbientTalk runtime automatically starts a service discovery request and keeps track of the available exported objects that can be referenced by the ambient reference.

Ambient references decouple sending and receiving objects in synchronization, space and time. Decoupling in synchronization is achieved because one may only send messages asynchronously via an ambient reference. The sender object does not wait until recipient service objects receive the message.

Decoupling in space is achieved because remote service objects are addressed by means of types (e.g. Java interfaces). A sender object does not need to know the identity of the service object to which the ambient reference refers, only its type. Ambient references can also be used to send a message to not just one, but to *all* available matching services in the ad hoc network. To broadcast a message, it must be annotated with an `@All` annotation, and results must be processed as follows:

```
def sensors := ambient: TemperatureSensor;
whenAll: sensors<-getTemperature()@All becomes: { |values|
  // values is an array of temperature sensor readouts
}
```

In this example, all nearby `TemperatureSensor` services are asked to sense the current temperature. Results from a broadcasted message may then be processed by means of the `whenAll:becomes:` control structure.

By default, when sending a message to an ambient reference for which no matching services are available, the message is not received by any service and is discarded. To introduce time decoupling, messages may be annotated with an `@Expires(timeout)` annotation. Here, `timeout` is a timeout period (in milliseconds), describing how long the message should remain available to be received by remote services. The ambient reference will then buffer the message until it expires.

How is it that ambient references combine publish/subscribe communication with object-oriented communication? Objects communicate via object references. In publish/subscribe systems, components communicate via event brokers (the mediator between publishers and subscribers). Ambient references *represent the event broker as an object reference*. Therefore, an ambient reference can be regarded as a little publish/subscribe engine of its own. In a publish/subscribe system, publishers send events to an event broker, which is responsible for delivering those events to interested subscribers. With ambient references, the act of sending a message to an ambient reference represents the publication of an event for consumption by nearby interested objects, which implicitly subscribe to these messages by being exported under a universally agreed upon type.

Previous and Related Work

The AmbientTalk language discussed in this chapter is more accurately named AmbientTalk/2, because it is a revised version of an earlier language with the same name, which we shall refer to as AmbientTalk/1 (Dedecker, Van Cutsem, Mostinckx, D'Hondt, & De Meuter, 2006). AmbientTalk/1 is a programming language that distinguishes between active and passive objects. Active objects are the unit of concurrent and distributed computing and resemble AmbientTalk/2's actors. They contain regular (a.k.a. passive) objects. Unlike in AmbientTalk/2, in AmbientTalk/1 passive objects cannot be referred to from within another active object. Only active objects can be addressed remotely. Passive objects also cannot be passed by (far) reference between active objects, they are always passed by copy.

A novelty of AmbientTalk/1 in comparison to similar languages based on active objects is that the language introduces additional mailboxes to store outgoing messages and previously sent or received messages (each active object has an "inbox", an "outbox", a "sentbox" and a "receivedbox") and that these mailboxes are made accessible to the programmer. By manipulating these mailboxes, an application can monitor and change messages received in the past and to be processed in the future. This allows one to express among others synchronization, failure handling and replication strategies. AmbientTalk/2 does not feature these mailboxes explicitly, but if required they can be reconstructed by the programmer. AmbientTalk/1 features no embedding with Java, such that it cannot be used to compose Java objects across an ad hoc network.

A detailed explanation of ambient references is beyond the scope of this chapter. An extensive explanation of their design and implementation can be found elsewhere (Van Cutsem, 2008). The ambient reference abstraction is not the first one that tries to unify object-orientation with a publish/subscribe interaction style. The ActorSpace model (Callsen & Agha, 1994) was an early attempt at augmenting actors with the space-decoupling properties of the tuple space model. Distributed Asynchronous Collections (P. T. Eugster, Guerraoui, & Sventek, 2000) combine objects with events by representing the event broker as an object-oriented *collection*, rather than as an object *reference.* Many to many invocations (M2MI) is a novel paradigm for communication in wireless ad hoc networks (Kaminsky & Bischof, 2002). M2MI introduces *handles*, special references akin to ambient references that enable one to unicast, multicast or broadcast invocations to proximate objects. As is the case with ambient references,

Java interfaces are used to anonymously designate the receivers of the invocation. M2MI handles have no equivalent for the `@Expires` annotation supported by ambient references, and thus do not feature any time decoupling.

Limitations

The main limitation of our language-centric approach using AmbientTalk is its lack of interoperability and its platform-dependence. AmbientTalk objects can only communicate with remote objects that are also written in AmbientTalk, using a protocol and message format specific to the language. Furthermore, the implementation is very dependent upon Java technology (although the language itself is technology-neutral) and difficult to configure (e.g. it is tied to a specific network protocol stack). In addition, the current implementation lacks extensive performance optimizations.

Ambient references currently lack a number of features common in commercial event queuing systems. Such features include ranking available services according to their properties, fuzzy matching of service descriptions, persistent events and transactions, and encrypted communication to ensure privacy. Also, the matching based on types assumes that all devices in the network know about and agree upon the names of these types. This assumption may not always hold in open, non-administered, ad hoc networks.

THE OBJECT-EVENT IMPEDANCE MISMATCH REVISITED

We now turn our attention once more to the issues in combining objects with events, which we previously named the object-event impedance mismatch. In each of the following sections, we revisit the problems described previously and show how they are dealt with in AmbientTalk.

Specific versus Generic Communication

Previously, we contrasted the specific communication interface of objects with the generic communication interface that is often provided by publish/subscribe architectures. Ambient references maintain the specific interface of objects by representing events as asynchronously sent messages. This has both drawbacks and advantages. A drawback is that this makes the representation of events explicit in the code. In the weather service example, the event `getWeather(city)` is represented as a message send of which the message selector identifies the kind of event and the message arguments constitute the event data. Both sender and receiver need to be fully aware of the structure (selector and arguments) of the message.

Representing events as messages has two major advantages. First, event publication can be unified with object-oriented message sending. Rather than having to explicitly construct an event as an object of a certain type and then invoking a generic `publish(Event)` method, a message is asynchronously sent to an object representing the subscribers and the event's type becomes the selector of that message. Second, event handling can be unified with object-oriented method invocation. Rather than having to represent event notification by subscribing a generic `reactTo(Event)` callback method to an event type, it is represented by having an event loop invoke a method whose selector corresponds to the kind of event.

In short, AmbientTalk resolves the impedance mismatch by representing events as asynchronously sent messages, thus maintaining the specific communication interface of object-oriented message passing.

Connection-oriented versus Connectionless Designation

Recall from our previous discussion that while object-oriented referencing abstractions provide connectionless communication but no space decoupling, pure event systems provide space decoupling but do not cater to any connection-oriented communication.

AmbientTalk naturally supports connection-oriented designation by means of its far references. A far reference to an object provides a stateful communication channel to that object. However, this channel is not decoupled in space: the identity of the receiver object must be known to the far reference.

While far references cater to connection-oriented designation, ambient references enable connectionless designation: they designate any number of service objects anonymously by means of their type. Also, by means of the `@All` annotation, they provide direct support for one-to-many communication. As a result, one-to-many messages act as event notifications to nearby interested service objects (subscribers). Time decoupling can be introduced in the event system by means of the `@Expires` annotation. Communication across an ambient reference is stateless, so subsequent messages may be received by different service objects. This enables a mobile device to transparently access different service objects as the user roams.

One difference between ambient references and publish/subscribe systems is that in the latter systems, it is generally the subscriber that specifies what kind of *events* it wants to accept. Ambient references invert this relationship. Using ambient references, it is the sender of a message (the publisher) that specifies what kind of *receiver* (subscriber) can accept the message. The type of an ambient reference thus delimits what services are eligible to receive its messages. The downside is that, if a receiver object (subscriber) places additional constraints on the message contents, it will have to filter messages explicitly by means of conditional tests in its invoked method. In contrast, a content-based publish/subscribe system can perform some such conditional tests within the event broker. This introduces less overhead because the broker can filter out certain events before notifying the subscriber.

In short, AmbientTalk resolves the impedance mismatch by providing ambient references, which introduce a form of stateless and connectionless designation in the object-oriented paradigm. If stateful communication is required, far references are the abstraction of choice.

Bidirectional versus Unidirectional Communication

Publish/subscribe systems are good at broadcasting information from publishers to subscribers. However, if subscribers need to pass information to event publishers, this can only be accomplished by turning the subscribers themselves into publishers and by turning event publishers into subscribers explicitly to gather the replies. AmbientTalk avoids this pattern by means of in-line event handlers via the `when:becomes:` and `whenAll:becomes:` control structures. The return value of an asynchronously invoked method can naturally serve as an implicit reply from receiver (subscriber) to sender (publisher).

In short, ambient references resolve the impedance mismatch by using event handlers to express bidirectional communication without giving up on the full synchronization decoupling afforded by event brokers.

Threads versus Event Loops

Previously, we noted that event-driven frameworks are mostly incorporated into (multithreaded) object-oriented languages by means of listeners and their callback methods. However, because a callback method is invoked synchronously by a thread that is not managed by the application, the application developer must be aware of the resulting concurrency control issues. By unifying event notification with the asynchronous invocation of a receiver's method, such issues are avoided. In particular:

- because the AmbientTalk language ensures that incoming messages are processed serially by an actor, the receiver object does not need to guard against race conditions on its data. While the serial execution of incoming messages conservatively limits the overall concurrency of the system (i.e. some methods are safe to execute in parallel), the resulting system becomes safer and

more compositional (additional sources of events and additional event types can be added without requiring any additional concurrency control in existing code).

- because methods are processed asynchronously by the actor owning the receiver object, the thread of control of the event broker remains responsive. Hence, a method that takes a long time to complete does not monopolize the resources of the entire event delivery subsystem.

The major drawback of event-based systems is that they suffer from an inversion of control. This drawback is mitigated to some extent in AmbientTalk because of its support for in-line event handlers. This allows the continuation of an asynchronous message send to be specified at the point where the send is performed, leading to less code fragmentation because the reply can be processed in the same computational context as the one in which the message was sent.

In short, because of the event loop architecture of AmbientTalk, service objects must not take any additional synchronization precautions when being designated by one or more ambient or far references. This is in contrast to multithreaded object-oriented programs where explicitly subscribing to an event broker introduces concurrency control issues.

Reconciling Objects with Events

Previously, we explicitly stated which properties of objects and events needed to be combined. Here, we summarize how AmbientTalk achieves a unification of objects and events:

- AmbientTalk maintains the *specific* communication interface of object-oriented message passing by representing events as (asynchronously sent) messages.
- Ambient references, like event brokers, provide *connectionless* designation, catering to anonymous interactions with groups of proximate objects. Far references provide *connection-oriented* designation, but are not decoupled in space.
- AmbientTalk makes use of in-line event handlers to retain the *bidirectional* communication of message passing, without sacrificing the time and synchronization decoupling afforded by event-driven communication.
- AmbientTalk's *event loop* concurrency ensures that receivers of messages (subscribers) do not need to take any synchronization precautions when handling messages (events).

The combination of AmbientTalk's event loops, far references, in-line event handlers and ambient references together effectively bridge the gap between event-driven and object-oriented abstractions.

Alternative Approaches to Resolving the Impedance Mismatch

We already mentioned Distributed Asynchronous Collections (DACs) (P. T. Eugster et al., 2000), which combine objects with events by representing the event broker as a *collection*, rather than as an object *reference.* While this approach allows for connectionless communication in an object-oriented system, it still provides a publish/subscribe interface to the programmer. We go further in integrating publish/subscribe with object-orientation than DACs. For example, we unify events with messages and event delivery with method invocation. No such unification is provided by DACs. Also, DACs provide no support for dealing with replies to events. Adding an element to a collection is a unidirectional operation.

Eugster et al. (2001) propose another object-oriented language extension with support for events. Again, this language extension does not attempt to fully integrate the publish/subscribe style with object-oriented concepts. Rather, this extension provides *both* object-oriented and event-driven concepts, with little integration between the two. Events are represented as objects, but the extension still distinguishes event notification from message passing and objects maintain their thread-based execution model. Similarly, Matsuoka and Kawai (1988) have studied the incorporation of tuple space communication in an

object-oriented language. Again, their system does not make any attempt at unifying concepts of tuple space-based communication with object-oriented language features. They represent tuples as objects, but they still distinguish interaction with a tuple space from sending messages to objects.

Summary

We have discussed the differences between event-driven and object-oriented communication. Because of these differences, event-driven communication is not always easy to integrate in an object-oriented application. We have named this phenomenon the object-event impedance mismatch.

Subsequently, we introduced AmbientTalk, a programming language intended for composing software components across a mobile ad hoc network. AmbientTalk is an object-oriented programming language but provides first-class abstractions to represent event-driven communication in terms of asynchronous messages. A publish/subscribe style of interaction is made possible by means of ambient references, which allow one to anycast or broadcast a message to available objects of a certain type in the ad hoc network.

Finally, we have shown how the abstractions provided by AmbientTalk resolve the object-event impedance mismatch. Objects still communicate by message passing, can still perform bidirectional communication and can engage in connectionless communication via ambient references. Multithreading is replaced with event loop concurrency, enabling additional event sources to be added to an application without change in terms of concurrency control.

FUTURE TRENDS

The emerging fields of mobile and ubiquitous computing form an ideal application domain for advanced distributed event-based systems. Applications in this domain have to deal with a high number of context changes, including changes in the physical environment (such as the availability or unavailability of particular services). The current state of the art to incorporate events into an application is via middleware, requiring the application developer to interact with the middleware via a traditional library API.

In the future, we hope to see this state of the art evolve into systems where the actual programming language in which applications are built is augmented with standard support for distributed event-based communication. Doing this requires language designers to think about alternative models of computing where event-based concepts can be mapped onto concepts that are already present in the language or the language's paradigm.

AmbientTalk is our first step towards such an event-based programming language. Our current prototype implementation serves as a proof-of-concept. Future work focuses in part on a more efficient implementation and on more elaborate applications. So far, AmbientTalk has been used to build among others a peer-to-peer instant messaging client, a matchmaking application that allows proximate buyers and sellers to trade items with one another, a multiplayer version for ad hoc networks of the well-known video game Pong and a mobile social networking application that allows users to share their profiles and keep each other up to date of what they are or have been doing.

CONCLUSION

Like in many other application domains, the loose coupling afforded by event-driven communication has turned event-based techniques into a de facto standard for the design of applications that need to be deployed on highly volatile mobile ad hoc networks. Such ad hoc networks form the hardware substrate on top of which pervasive and ubiquitous computing applications are deployed.

While event-based programming is a crucial technique, it remains only a second-class abstraction in mainstream programming languages. This exposes a need for new models that combine event-driven communication with the first-class abstractions already present in today's programming languages.

We conclude this chapter with a summary of our objectives. We have:

1. discussed why event-driven communication is suitable in mobile ad hoc networks by describing its support for decoupling in time, space and synchronization.
2. highlighted the key differences between objects and events, the combination of which we have named the object-event impedance mismatch.
3. proposed a novel programming model to resolve the mismatch. This model is validated by means of a programming language named AmbientTalk. The language's event loop model allows event-driven concepts (events, event handlers, …) to be mapped almost one-to-one onto object-oriented concepts (messages, methods, …). The language supports publish/subscribe communication in an object-oriented manner by means of ambient references, which represent a traditional event broker as a special kind of object reference.

REFERENCES

Agha, G. (1986). Actors: a Model of Concurrent Computation in Distributed Systems: MIT Press.

Atul, A., Jon, H., Marvin, T., William, J. B., & John, R. D. (2002). Cooperative Task Management Without Manual Stack Management. Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, 289-302.

Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., et al. (2006). Programming, Deploying, Composing, for the Grid. In J. Cunha & O. Rana (Eds.), Grid Computing: Software Environments and Tools: Springer-Verlag.

Cabri, G., Leonardi, L., & Zambonelli, F. (2000). MARS: A Programmable Coordination Architecture for Mobile Agents. IEEE Internet Computing, 4(4), 26-35.

Callsen, C. J., & Agha, G. (1994). Open Heterogeneous Computing in ActorSpace. Journal of Parallel and Distributed Computing, 21(3), 289-300.

Carey, M. J., & DeWitt, D. J. (1996). Of Objects and Databases: A Decade of Turmoil. VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, 3-14.

Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., & De Meuter, W. (2006). Ambient-oriented Programming in AmbientTalk. Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP), 4067, 230-254.

Eugster, P., Felber, P., Guerraoui, R., & Kermarrec, A. (2003). The many faces of publish/subscribe. ACM Comput. Surv., 35(2), 114-131.

Eugster, P., Garbinato, B., & Holzer, A. (2005). Location-based Publish/Subscribe. Fourth IEEE International Symposium on Network Computing and Applications, 279-282.

Eugster, P. T., Guerraoui, R., & Damm, C. H. (2001). On objects and events. OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, 254-269.

Eugster, P. T., Guerraoui, R., & Sventek, J. (2000). Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming, 252-276.

Foote, B., & Johnson, R. (1989). Reflective Facilities in Smalltalk-80. 4th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 89), 327-335.

Freeman, E., Arnold, K., & Hupfer, S. (1999). JavaSpaces Principles, Patterns, and Practice. Essex, UK: Addison-Wesley Longman Ltd.

Gelernter, D. (1985). Generative communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1), 80-112.

Grimm, R., Davis, J., Lemar, E., Macbeth, A., Swanson, S., Anderson, T., et al. (2004). System support for pervasive applications. ACM Trans. Comput. Syst., 22(4), 421-486.

Haller, P., & Odersky, M. (2006). Event-Based Programming without Inversion of Control. Proc. Joint Modular Languages Conference, 4228, 4-22.

Hapner, M. (2002). Java Message Service Specification (Version 1.1): Sun Microsystems, Inc.o. Document Number)

Kaminsky, A., & Bischof, H.-P. (2002). Many-to-Many Invocation: a new object oriented paradigm for ad hoc collaborative systems. OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 72-73.

Mamei, M., & Zambonelli, F. (2004). Programming Pervasive and Mobile Computing Applications with the TOTA Middleware. PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications, 263-276.

Mascolo, C., Capra, L., & Emmerich, W. (2002). Mobile Computing Middleware. In Advanced lectures on networking (pp. 20-58): Springer-Verlag New York, Inc.

Matsuoka, S., & Kawai, S. (1988). Using tuple space communication in distributed object-oriented languages. SIGPLAN Not. Special issue: 'OOPSLA 88 Conference Proceedings, 23(11), 276-284.

Meier, R., & Cahill, V. (2003). Exploiting Proximity in Event-Based Middleware for Collaborative Mobile Applications. Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03), 285-296.

Meyer, B. (1993). Systematic concurrent object-oriented programming. Communications of the ACM, 36(9), 56-80.

Miller, M. (2006). Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. John Hopkins University, Baltimore, Maryland, USA.

Miller, M., Tribble, E. D., & Shapiro, J. (2005). Concurrency among strangers: Programming in E as plan coordination. Symposium on Trustworthy Global Computing, 3705, 195-229.

Murphy, A., Picco, G., & Roman, G.-C. (2001). LIME: A Middleware for Physical and Logical Mobility. Proceedings of the The 21st International Conference on Distributed Computing Systems, 524-536.

Musolesi, M., Mascolo, C., & Hailes, S. (2005). EMMA: Epidemic Messaging Middleware for Ad hoc networks. Personal Ubiquitous Comput., 10(1), 28-36.

Ousterhout, J. (1996). Why Threads Are A Bad Idea (for most purposes), Presentation given at the 1996 Usenix Annual Technical Conference.

Saif, U., & Greaves, D. J. (2001). Communication primitives for ubiquitous systems or RPC considered harmful. International Conference on Distributed Computing Systems, 240-245.

Sun Microsystems. (1999). Dynamic Proxy Classes. Retrieved August 25, 2008, 2008, from http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

Sun Microsystems. (2008, February 14th 2008). Concurrency in Swing. Retrieved August 12th 2008, 2008, from http://java.sun.com/docs/books/tutorial/uiswing/concurrency

Tobin, J. L., Alex, C., Yuhong, X., Jonathan, G., Venu, V., Sean, L., et al. (2001). Hitting the distributed computing sweet spot with TSpaces. Comput. Netw., 35(4), 457-472.

Van Cutsem, T. (2008). Ambient References: Object Designation in Mobile Ad Hoc Networks. Programming Technology Lab, Faculty of Sciences, Vrije Universiteit Brussel.

Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., & De Meuter, W. (2007). AmbientTalk: object-oriented event-driven programming in Mobile Ad hoc Networks. Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007), 3-12.

Weiser, M. (1991). The computer for the twenty-first century. Scientific American, 94-100.

Yonezawa, A., Briot, J.-P., & Shibayama, E. (1986). Object-oriented concurrent programming in ABCL/1. Conference proceedings on Object-oriented programming systems, languages and applications, 258-268.

KEY TERMS AND DEFINITIONS

Mobile ad hoc network: a computer network without any fixed infrastructure, consisting of mobile nodes that communicate by means of wireless links.
Decoupling in Time: two or more processes are decoupled in time if they do not need to be online at the same time while communicating.
Decoupling in Space: two or more processes are decoupled in space if they can communicate without needing to know one another's identity.
Decoupling in Synchronization: two or more processes are decoupled in synchronization if their control flow is not blocked upon sending or receiving information.
Event broker: the middleman in a publish/subscribe architecture that decouples subscribers from publishers. Both publishers and subscribers register with the event broker, whose task it is to forward published events to the appropriate subscribers.
Connection-oriented communication channel: a channel in which the sender process knows about the identity of the unique receiver process, and in which it is assumed that messages arrive in order. Example: communication via TCP.
Connectionless communication channel: a channel in which the sender process does not know about the identity and the number of receivers, and in which no assumptions are made that messages arrive in order. Example: communication via UDP.
Event Loop: a perpetual loop that accepts events from one or more event sources and dispatches these events (usually sequentially) to the appropriate event handlers.