AmbientTalk: Language Support for Mobile Computing

Jessie Dedecker Tom Van Cutsem Stijn Mostinckx Wolfgang De Meuter Theo D'Hondt

***Abstract*—An important aspect of system support for mobile computing involves alleviating the issues related to programming the underlying distributed system. Our approach to dealing with these issues is by means of programming language support. To this end, we have designed a high-level, distributed object-oriented programming language – called AmbientTalk – which has been designed specifically for mobile networks. We motivate the need for such dedicated language support based on the fundamental characteristics of mobile distributed systems. Subsequently, we describe the salient features of AmbientTalk with respect to distributed programming by means of an exemplar mobile computing application.**

***Index Terms*—ubiquitous computing, mobile computing, ambient-oriented programming, distributed programming languages, event-driven programming**

I. Introduction

THE past couple of years, pervasive and ubiquitous computing have received more and more attention from academia and industry alike. Wireless communication technology and mobile computing technology have reached a sufficient level of sophistication to support the development of a new breed of applications. Mobile computing enables applications running on mobile devices to spontaneously interact with devices in their proximity. Hence, realizing the vision of ubiquitous computing entails the construction of an open distributed system.

At the software-engineering level, we observe that, although there has been a lot of active research with respect to mobile computing middleware [1], we see little innovation in the field of programming language research. Although distributed programming languages are rare, they form a suitable development tool for encapsulating many of the complex issues engendered by distribution [2], [3]. The distributed programming languages developed to date have either been designed for high-performance computing (e.g. X10 [4]), for reliable distributed computing (e.g. Argus [5]) or for general-purpose distributed computing in fixed, stationary networks (e.g. Emerald [6], Obliq [7], E [8]). None of these languages has been explicitly designed for mobile networks. They lack the language support necessary to deal with the radically different network topology.

We take the position that a new breed of programming languages is needed to deal with and manage the complexity that arises from the novel hardware constellation used to realize the vision of ubiquitous computing. In previous work, we investigated the desirable properties of such novel programming languages based on four important hardware phenomena inherent to mobile networks [9]. Subsequently we designed one such language, named AmbientTalk, whose language features were specifically designed to aid the developer in dealing with the hardware phenomena of mobile networks [10]. Whereas previous work specifically targeted language designers by focusing on the implementation of novel language constructs, this paper covers AmbientTalk's novel distribution-related language features from an application developer point of view by demonstrating their concrete use in a small mobile computing application. Before introducing the example application and AmbientTalk's language constructs, we recapitulate the hardware phenomena of mobile networks and give a brief introduction to AmbientTalk.

Jessie Dedecker and Tom Van Cutsem are both research assistants of the Fund for Scientific Research Flanders, Belgium (F.W.O.), Stijn Mostinckx is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.
Authors are members of the Programming Technology Laboratory of the Department of Computer Science at the Vrije Universiteit Brussel, Belgium, email: {jededeck — tvcutsem — smostinc — wdmeuter — tjdhondt }@vub.ac.be

II. Motivation

In this section, we will motivate our claim that high-level yet general-purpose distributed programming languages are required which are specifically designed for developing software for mobile computing. We start this motivation by observing a number of phenomena which are derived from the characteristics inherent to mobile networks. Subsequently, we briefly discuss why existing approaches are insufficient to cope with these hardware phenomena.

A. Hardware Phenomena

Based on the fundamental characteristics of mobile hardware, we distill a number of phenomena which mobile networks exhibit. There are two discriminating properties of mobile networks: applications are deployed on *mobile* devices which are connected by *wireless* communication links with a limited communication range. The type of device and the type of wireless communication medium can vary, leading to a diverse set of envisaged applications. Devices might be as small as coins, embedded in material objects such as wrist watches, door handles, lamp posts, cars, etc. They may even be as lightweight as sensor nodes or they may be material objects "digitized" via an RFID tag[1]. Devices may also be as "heavyweight" as a cellular phone, a PDA or a car's on-board computer. All of these devices can in turn be interconnected by a diverse range of wireless networking technology, with ranges as wide as WiFi or as limited as IrDA.

Mobile networks composed of mobile devices and wireless communication links exhibit a number of phenomena

[1] Such tags can be regarded as tiny "computers" with an extremely small memory, able to respond to read and write requests.

which are rare in their fixed counterparts. In previous work, we have remarked that mobile networks exhibit the following phenomena [10]:

Volatile Connections. Mobile devices equipped with wireless communication media possess only a limited communication range, such that two communicating devices may move out of earshot unannounced. The resulting disconnections are not always permanent: the two devices may meet again, requiring their connection to be re-established. Quite often, such transient disconnections should not affect an application, allowing both parties to continue with their conversation where they left off. These volatile disconnections do expose applications to a much higher rate of partial failure than that which most distributed languages or middleware have been designed for.

Ambient Resources. In a mobile network, devices spontaneously join with and disjoin from the network. The same holds for the services or resources which they host. As a result, in contrast to stationary networks where applications usually know where to find their resources via URLs or similar designators, applications in mobile networks have to find their required resources dynamically in the environment. Moreover, applications have to face the fact that they may be deprived of the necessary resources or services for an extended period of time. In short, we say that resources are *ambient*: they have to be discovered on proximate devices.

Autonomous Devices. In mobile wireless networks, devices may encounter one another in locations where there is no access whatsoever to a shared infrastructure (such as a name server). Even in such circumstances, it is imperative that the two devices can discover one another in order to start a useful collaboration. Relying on a mobile device to act as infrastructure (e.g. as a name server) is undesirable as this device may move out of range without warning [11]. These observations lead to a setup where each device acts as an autonomous computing unit: a device must be capable of providing its own services to proximate devices. Devices should not be forced to resort to a priori known, centralized servers.

Natural Concurrency. Due to their close coupling to the physical world, most pervasive applications are also inherently event-driven. Writing correct event-based programs is far from trivial. There is the issue of concurrency control which is innate in such systems. Furthermore, from a software design point of view, event-based programs have very intricate, confusing control flow as they lack the structure provided by a call-return stack.

As the complexity of applications deployed on mobile networks increases, the above unavoidable phenomena cannot keep on being remedied using ad hoc solutions. Instead, they require more principled software development tools specifically designed to deal with the above phenomena. For some classes of applications – such as wireless sensor networks – such domain-specific development tools are emerging, as can be witnessed from the success of TinyOS [12] and its accompanying programming language nesC [13]. It is our conjecture that mobile computing applications would benefit similarly from a programming language designed to tackle the difficult distribution issues in their domain.

B. Evaluation

Before introducing the AmbientTalk programming language, we first evaluate related work with respect to its ability to deal with the hardware phenomena described above. We categorize related work into two categories: distributed programming languages and middleware.

B.1 Distributed Languages

To the best of our knowledge no distributed language has been designed that deals with all of the characteristics of the mobile hardware described above. The suitability of a distributed language for mobile computing largely depends on the nature of its remote communication mechanism. Languages that use synchronous communication such as Emerald [6] and Obliq [7] are the least suitable because synchronous message passing does not scale when performed over wireless, volatile connections. We elaborate upon the mismatch between synchronous communication and volatile connections in section IV. Languages like ABCL/f [14] and Argus [15] promote asynchronous send operations that return futures [16] (which are discussed in more detail below), but require processes to block upon accessing unresolved futures, which re-introduces the drawbacks of synchronous message passing. Languages based on the actor model, such as Janus [17], Salsa [18] and E [8] use pure asynchronous communication. However, these languages offer no support to discover ambient resources or to coordinate interactions among autonomous computing units in the face of volatile connections.

B.2 Distributed Middleware

An alternative to distributed languages is middleware. Over the past few years a lot of research has been invested in middleware for mobile networks [1], which can be categorized as follows:

RPC-based Middleware like Alice [19] and DOLMEN [20] are attempts that focus mainly on making ORBs suitable for lightweight devices and on improving the resilience of the CORBA IIOP protocol against volatile connections. Others deal with such connections by supporting temporary queuing of RPCs [21] or by rebinding [22]. However, these approaches remain variations of synchronous communication and are thus irreconcilable with the autonomy and connection volatility phenomena.

Data Sharing-oriented Middleware tries to maximize the autonomy of temporarily disconnected mobile devices using weak replica management (cf. Bayou [23], Rover [21] and XMiddle [24]). However, since replicas are not always synchronisable upon reconnection,

potential conflicts must be resolved at the application level. In spite of the fact that these approaches foster fruitful ideas to maximize the autonomy of mobile devices, to the best of our knowledge, they do not address the discovery of ambient resources.

Publish-subscribe Middleware adapts the publish-subscribe paradigm [25] to cope with the characteristics of mobile computing [26], [27]. Such middleware allows asynchronous communication, but has the disadvantage of requiring a program to be structured using callbacks to handle published events, which severely obfuscates the control flow of the code and requires careful concurrency control.

Tuple Space based Middleware [28], [29] for mobile computing has been proposed more recently. A tuple space [30] is a shared data structure in which processes can asynchronously publish and query tuples. Most research on tuple spaces for mobile computing attempts to distribute a tuple space over a set of devices. Tuple spaces are an interesting communication paradigm for mobile computing. Unfortunately, because communication is achieved by placing data in a tuple space as opposed to sending messages to objects, it does not integrate well with the object-oriented paradigm.

In this section, we have motivated the need for a distributed language designed for mobile computing. We have described four hardware phenomena which unavoidably have to be dealt with when developing mobile computing applications. Distributed programming languages that are not designed for mobile computing lack essential features to deal with the phenomena. On the other hand, mobile computing middleware approaches address some phenomena, but fail to offer an integrated solution that helps the programmer develop his software expressively.

III. An Exemplar Mobile Computing Application

In order to get a feel for the kind of mobile computing applications we envisage, this section describes a small application which we will use as a running example throughout the paper baptised *mobiTunes*. The application is designed to run on PDAs or smartphones communicating via wireless technology such as WiFi or Bluetooth. The application presumes no communication infrastructure whatsoever (i.e. the PDAs spontaneously form ad hoc networks). The mobiTunes application allows users to share their music database with one another. More specifically, when two mobiTunes peers encounter one another, they exchange their music database (henceforth called their *library*). The mobiTunes application compares the incoming music library with the owner's library in terms of "similar music taste" (e.g. by searching for similar artists). This matching of libraries is used to implement mobiTunes' two main features.

First, when a user encounters a peer with a similar taste in music, mobiTunes can derive from that peer's library the songs which are not yet in the user's own library. In this way, mobiTunes can e.g. keep a record of "recommended albums" which can help the user when he next goes shopping. In other words, mobiTunes is used to "discover new music". Secondly, mobiTunes optionally acts as a "social match maker". When a user has enabled this feature, mobiTunes notifies him or her whenever another user with a similar taste in music (and which also has enabled this feature) is discovered in the proximity.

Applications like mobiTunes, which are deployed on mobile devices collaborating via ad hoc networks, are obliged to take the hardware phenomena described in the previous section into account. We describe how these phenomena influence the behaviour of mobiTunes:

Volatile Connections. Because of the mobility of users and the limited communication range of the ad hoc network, the connection between two mobiTunes peers is volatile. This has its repercussions on e.g. the exchange of the music libraries. If this exchange is done incrementally, rather than by sending the library in one piece, it will be much more resilient to intermittent disconnections.

Ambient Resources. A mobiTunes application requires other mobiTunes peers in order to provide useful functionality. These peers have to be discovered dynamically in the (nearby) environment. It is this ability that allows for the "social match maker" functionality, as this feature depends on two users being in one another's vicinity.

Autonomous Devices. There is no reliance on a central server to pair two mobiTunes peers. Although this makes discovery more expensive, the fact that no connection to a central server needs to be made increases availability (avoidance of a single point of failure) and involves no communication cost (ad hoc wireless communication is free of charge).

Natural Concurrency. mobiTunes peers clearly require proper concurrency control: multiple users can meet and exchange libraries simultaneously. Furthermore, mobiTunes peers should remain responsive to their environment even while engaged in communication with a particular peer.

In the following section, we introduce AmbientTalk, a high-level programming language designed specifically to develop applications such as the one described here. We focus on the language's support for distributed communication and discovery and describe how it aids the developer in dealing with the aforementioned hardware phenomena.

IV. AmbientTalk: an Ambient-Oriented Programming Language

AmbientTalk is a high-level object-oriented distributed programming language based on the ambient-oriented programming paradigm [9]. This paradigm has been specifically designed to deal with the hardware phenomena described above. In the following section we describe the language and describe how an ambient-oriented language like AmbientTalk differs from a standard object-oriented language. We introduce the features of the language by describing the implementation of the mobiTunes application piece by piece.

A. AmbientTalk in a Nutshell

AmbientTalk distinguishes between passive and active objects. Passive objects behave like typical objects in standard object-oriented languages and are used to express local, sequential computations. The following code snippet shows how one could represent mobiTunes song objects in AmbientTalk:

```
Song : object({
  artist : "Elvis Presley";
  title  : "Just Because";
  album  : "Sunrise";

  new(anArtist, aTitle, anAlbum) :: copy({
    artist := anArtist;
    title  := aTitle;
    album  := anAlbum;
  });
  getArtist() :: { artist };
  getTitle() :: { title };
  getAlbum() :: { album };
  equals(aSong) :: {
    (getArtist() = aSong.getArtist()) &
      (getTitle() = aSong.getTitle()) &
        (getAlbum() = aSong.getAlbum())
  };
  toString() :: { artist + "-" + name + "["+ album +"]" }
});
```

AmbientTalk's object model is based on prototypes [31] rather than on classes. This means that, instead of creating classes which are instantiated at runtime into objects, the developer immediately creates objects. In the example above, the variable `Song` is bound to a prototypical song object using the `object` form. A song object contains the fields `artist`, `title` and `album`, defined using `:`. Furthermore, the song object defines methods (such as `new`, `getArtist`, ...) using `::`. For example, invoking `s.getArtist()` on a song `s` returns the song's artist. Next to creating objects "from scratch", objects can also be created by cloning existing objects. This is illustrated by the `new` method that returns a `copy` of the existing object. The `copy` form takes an expression as its argument that allows one to define how the clone should differ from the original object.

Passive objects are used for local, sequential computations only. Active objects, on the other hand, are the unit of concurrency and distribution. Active objects are based on an extension of the well-known actor model for distributed systems [32]. We will henceforth refer to active objects as *actors*. Each actor is associated with exactly *one* thread of execution and sequentially processes incoming messages that have been delivered to its `inbox` (its incoming message queue). An actor responds to messages according to a *behaviour*, which is a passive object that implements the methods corresponding to the incoming messages. Passive objects are never shared between actors such that no race conditions on the internal state of a passive object can exist. When passive objects are passed as arguments in messages to other actors, they are copied to preserve this sharing restriction.

All actors communicate exclusively via asynchronous message passing. AmbientTalk allows one to abstract over the volatile connections hardware phenomenon by buffering messages sent to actors currently unavailable for communication. The buffered messages are automatically transmitted whenever the connection is restored. The asynchronous message passing scheme is further detailed in section B.

As actors are the only objects capable of concurrent and distributed computation, the mobiTunes application is represented by an actor. The definition of the actor representing a mobiTunes peer is shown below:

```
MobiTunesActor :: actor(object({
   myLibrary: void;
   sessions : void;
   notificationsEnabled  : false;
   recommendedMusic : void;

   new(enableNotification) :: copy({ ... });
   init() :: { ... };

   isNotificationEnabled() :: { ... };
   addSong(song) :: { ... };
   showRecommendedSongs() :: { ... };

   requestLibraryExchange(senderPeer) :: { ... };
   sendSong(receiverPeer, idx) :: { ... };
   receiveSong(senderPeer, song) :: { ... };
   signalEndOfExchange(senderPeer) :: { ... };
}));
```

An active object is created using the `actor` form. This form takes a passive object as its argument that defines the behaviour of the actor. The `init` method is automatically invoked upon creation of an actor and is used to initialize it. The field `myLibrary` contains the owner's music library, the hashmap `sessions` maps connected mobiTunes peers to their exchanged music library, `notificationEnabled` holds a boolean value that indicates whether the owner is interested in being notified when another person with a similar taste in music is in the vicinity, `recommendedMusic` contains the list of songs suggested based on the matched libraries. The methods `requestLibraryExchange`, `sendSong`, `receiveSong` and `signalEndOfExchange` are part of the music library matching protocol and are further discussed in the following sections.

B. Non-blocking Communication

This section describes the design of the distributed communication facilities of AmbientTalk. We first motivate the need for non-blocking communication primitives in a distributed system designed for mobile computing and subsequently describe how these features are realised by means of asynchronous message passing between actors in AmbientTalk.

B.1 Motivation

Autonomous devices communicating over volatile connections necessitate non-blocking communication primitives. Blocking communication, in the guise of synchronous method invocations or blocking receive statements severly harm the autonomy of mobile devices. First, blocking communication is a known source of (distributed) deadlocks [33] which are extremely hard to resolve in mobile networks since not all parties are necessarily available for communication. Second, blocking communication primitives would

cause a program or device to block for a substantial amount of time when a communication partner is temporarily unavailable [1], [28]. As such, the availability of resources and the responsiveness of applications would be seriously diminished.

B.2 Realisation

AmbientTalk actors communicate with one another via asynchronous message passing. Actors do not wait for messages to be transmitted to the recipient actor. The following code snippet shows how one mobiTunes actor transmits its library to another mobiTunes actor stored in the variable `receiverPeer`. Note that the library is transmitted incrementally, song by song, such that an actor does not need to resend the complete library in case of a disconnection. After all songs have been transmitted, a message is sent to the `receiverPeer` to signal the end of the exchange.

```
sendSong(receiverPeer, idx) :: {
  if(myLibrary.numSongs() >= idx, {
    ...
    receiverPeer#receiveSong(thisActor, currentSong)
    ...
  }, {
    receiverPeer#signalEndOfExchange(thisActor)
  }) };
```

Asynchronous message sends are denoted using a `#` instead of a dot. Also note that the variable `thisActor` refers to the current actor, similar to how `this` represents the current passive object. Asynchronous message sends are not easily reconcilable with return values. It requires the use of either "callback" methods or a program written in continuation-passing style in order to process results. Such programming idioms clutter the code which is why AmbientTalk adopts the use of *futures* or *promises*, a frequently recurring abstraction in concurrent and distributed languages (e.g. in Multilisp [16], ABCL [34] and Argus [15]). An asynchronous message send always immediately returns a future object, which is a placeholder for the real return value. Once the real value is computed, it "replaces" the future object; the future is said to be *resolved* with the value.

Most languages, including the ones listed above, make a process block on an unresolved promise or future (either implicitly by using its value in an expression or explicitly via e.g. a `touch` or `claim` operator). One notable exception is the language E [8], which disallows waiting for a promise to be resolved. Instead, E provides a `when`-construct which registers a closure, parameterized with the determined value, with the promise. The promise schedules this closure for execution when it has been resolved with a value. AmbientTalk adopts this `when`-construct from the language E.

The code excerpt below illustrates the use of futures and the `when`-construct in the mobiTunes example application. Recall from the description of the application above that a user may optionally specify that he wants to be notified by the application whenever another user with similar taste in music has been detected in the environment. This notification should only trigger when that user has also enabled the notification function. Hence, after exchanging libraries, the remote peer is queried for its notification setting before the user can be notified:

```
signalEndOfExchange(senderPeer) :: {
  senderLibrary: sessions.get(senderPeer);
  matchPercentage: compare(myLibrary, senderLibrary);
  if(matchPercentage >= THRESHOLD, {
    recommendedMusic.union(senderLibrary.difference(myLibrary));
    when(senderPeer#isNotificationEnabled(),
         lambda(notificationForPeer) -> {
           if(notificationsEnabled & notificationForPeer, {
             notifyUser(senderPeer, matchPercentage)
           })
    })
  }) };
```

The `lambda` keyword denotes the construction of a lexically scoped closure whose body is executed asynchronously at a later point in time. When the future returned by the asynchronous invocation `isNotificationEnabled()` is resolved with a value, the deferred `when` code block is scheduled for execution with that value bound to `notificationForPeer`. The block will eventually be executed by the actor's single thread when it is not processing messages. This is important to guarantee that the code block executes atomically without interfering with other message processing code in the same actor. The `when` code block enables the processing of return values in the same scope of the original message send without having to resort to callbacks.

Message reception is implicit in AmbientTalk. Received messages are buffered in an actor's inbox and processed sequentially. Processing a message entails invoking its corresponding method. Hence, message reception is aligned with method invocation. The example below shows how a mobiTunes actor can receive the `isNotificationEnabled` message by implementing a method with the same name:

```
isNotificationEnabled() :: { notificationsEnabled }
```

The return value of the method is the value used to resolve the future associated with the message that gave rise to the method invocation.

C. Peer-to-Peer Service Discovery

Mobile computing applications require a service discovery mechanism to detect useful software services available in the (wireless) network. For example, in the mobiTunes application, peers must discover one another in the nearby environment in order to synchronize their music library. In this section, we motivate why AmbientTalk has built-in support for service discovery at the language level, and subsequently demonstrate how that support is realised.

C.1 Motivation

In section II, it was observed that (ad hoc) mobile networks consist of autonomous devices where resources are "ambient": they have to be discovered in the nearby environment as devices are roaming. This hardware topology implies that devices do not necessarily rely on a third party

to start an interaction with each other. This is in contrast to client-server communication models where clients interact through the mediation of a well-known server (e.g. chat servers or white boards). In mobile networks, a device cannot acquire an explicit reference to a remote service beforehand as the address of the device hosting the service is unknown. This address can only be discovered at runtime, as devices spontaneously encounter one another. What is needed is an extension to the object-oriented paradigm that allows objects to create references to remote objects, not on the basis of that object's address (or similarly based on static URLs), but on the basis of an intensional description of the object's provided services.

C.2 Realisation

AmbientTalk achieves peer-to-peer service discovery by the introduction of *ambient references*. An ambient reference is a local actor that represents a remote service (i.e. it is a proxy to a remote actor). Unlike traditional proxies, which are acquired via a name server or created via a URL, ambient references describe the actors they denote using an intensional description of the services they provide. This description takes the form of a *service type*. Such a service type describes the services to which the ambient reference can bind. For example, in the mobiTunes application, every mobiTunes actor identifies itself by declaring that it provides the `MobiTunesPeer` service type, as follows:

```
serviceType(MobiTunesPeer); // declare new service type
MobiTunesActor :: actor(object({
  init() :: { provide(MobiTunesPeer); ... };
  ...
}))
```

A mobiTunes actor engages in service discovery during initialization by declaring an ambient reference to other mobiTunes peers in the environment:

```
init() :: {
  provide(MobiTunesPeer);
  sensor : ambient(MobiTunesPeer);
  ...
};
```

The variable `sensor` is an ambient reference that refers to any actor providing the `MobiTunesPeer` service type. Once an ambient reference has been declared, code blocks may be attached to it which will be executed whenever a new peer is discovered in the environment. When a mobiTunes peer wants to start an interaction with a newly discovered peer, it can do so by registering a block of code with the ambient reference, which is triggered whenever a new peer is discovered:

```
when_found(sensor, lambda(remotePeer) -> {
  sessions.put(remotePeer, MusicLibrary.new());
  remotePeer#requestLibraryExchange(thisActor)
});
```

The `when_found` language construct executes the given closure whenever a new peer is discovered (e.g. when another user's PDA joins the ad hoc network). A reference to the newly discovered peer is passed as an argument to that closure. When a peer is discovered, a new library is created to store its songs during the exchange of the music library. After this library is created, the discovered peer is asked to transmit its music library to the discovering peer.

D. Failure Handling in the face of Volatile Connections

We have shown in the preceding sections that the asynchrony of remote communication between actors in combination with the buffering of messages sent to disconnected actors allows the programmer to abstract over network disconnections. This is a good property when such failures are only transient, caused by intermittent network disconnections or by people moving temporarily out of one another's communication range. However, it is clear that not all network failures can be masked in this way. Some broken connections will never be restored and require proper failure handling.

D.1 Motivation

When designing a failure handling mechanism for an ambient-oriented programming language, it is important not to lose the property of being able to abstract over temporary network disconnections. After all, the paradigm prescribes that network disconnections are omnipresent and mostly caused by the volatility of wireless connections. That is why failure handling in AmbientTalk is based on timeouts: actors are signalled as being "unavailable" only when they have been disconnected for longer than a given timeout period. This timeout period effectively allows the programmer to define which network failures the application should regard as transient (and can be disregarded) and which failures must be treated as fatal (and should be acted upon).

D.2 Realisation

AmbientTalk allows programmers to deal with unavailable actors via ambient references. Just like it is possible to trigger code whenever a new actor is discovered, it is possible to trigger code whenever a discovered actor is unavailable for longer than a given timeout period.

In the mobiTunes application, two peers may encounter and start exchanging their music library, but can disconnect before the exchange is complete and never encounter one another again. Any resources held due to this partial exchange should eventually be freed. When a peer has been disconnected for longer than five minutes, any resources held by the current exchange session are discarded. These resources include the partially exchanged library of the unavailable peer and – less obviously – all messages sent to the unavailable peer which are still buffered for transmission. The following code snippet shows the failure handling source code:

```
TIMEOUT_PERIOD :: 5*60*1000; // 5 minutes
init() :: {
  provide(MobiTunesPeer);
  sensor : ambient(TIMEOUT_PERIOD, MobiTunesPeer);
  when_found(sensor, lambda(remotePeer) -> { ... });
  when_lost(sensor, lambda(remotePeer) -> {
    sessions.delete(remotePeer);
    discardMessagesSentTo(remotePeer)
  }) }
```

First, the ambient reference described previously is annotated with the timeout period to discriminate between transient and fatal disconnections. Second, just like one can use `when_found` to trigger code when a peer is discovered, one can use `when_lost` to react to lost peers. As shown in the code, the partial library is removed from the `sessions` map and all messages destined for the lost peer are simply discarded. For the purposes of this paper, `discardMessagesSentTo` may be regarded as a primitive method. In the actual implementation, however, this method is implemented in AmbientTalk itself because an actor can access and modify its own *outbox* which stores its outgoing buffered messages. The details of such reflective code are outside the scope of this paper but can be found in previous work [10].

E. Summary

This section has given an overview of AmbientTalk's most important language constructs to aid the programmer in dealing with the complexities of software development for mobile computing. We have introduced these constructs by means of a simple yet typical mobile computing application. It is important to note that the code snippets shown throughout the paper represent nearly all of the essential code that an application programmer needs to write in order to develop a running mobiTunes application. Much of the difficulties related to distribution are either tackled by the language runtime or by reflective code which is reusable by a large number of programs in the form of language constructs.

V. Validation

In this section, we take a step back and reconsider how AmbientTalk's language constructs aid the programmer in dealing with the hardware phenomena described in section II:

Volatile Connections. Because messages are sent asynchronously and because they are automatically buffered when sent to a disconnected recipient, AmbientTalk's message passing scheme is inherently suitable for communicating over the volatile connections induced by wireless communication media. Furthermore, the use of futures and the `when`-construct enable the expression of asynchronous reply processing without fragmenting the source code. Finally, because all send and receive abstractions in the language are non-blocking, actors remain responsive to incoming events even while communicating with disconnected parties.

Ambient Resources. Ambient references are used to discover nearby actors based on the services those actors provide. The `when_found` and `when_lost` language constructs enable the programmer to intervene when actors become available or unavailable.

Autonomous Devices. Ambient references discover actors based on a broadcasting protocol. No shared infrastructure is required to support the discovery. References to other actors are created based on service types describing those actors, rather than based on fixed addresses or URLs.

Natural Concurrency. The concurrent nature of the physical world is controlled thanks to an actor's property of sequential message processing. Concurrency within a single application is still possible by creating multiple actors. Within a single actor, however, method bodies and closures deferred using `when`, `when_found` and `when_lost` run atomically and are free of race conditions. Race conditions may of course still exist at the message-passing level and require additional concurrency control which we will not further describe here.

An interpreter for the AmbientTalk language has been implemented in Java. Two or more AmbientTalk interpreters communicate via sockets over WiFi. An interpreter has been implemented for resource-constrained devices on top of J2ME. Currently, only CDC (Connected Device Configuration) is supported, allowing AmbientTalk to be run on PDAs and smartphones. We are planning on porting AmbientTalk to CLDC (Connected Limited Device Configuration) for embedded devices in the near future.

VI. Conclusion

The last couple of years, a lot of research effort has been invested in middleware and libraries to aid developers in writing software to be deployed on mobile networks. In the programming language community, however, no programming language has been designed that specifically targets the development of mobile computing applications. Our position statement is that programming languages are a good medium to abstract much of the difficulties associated with software development for mobile computing. It is *not* our goal to paper over difficult distribution issues by *hiding* them in a language. Rather, we want to make the essential issues related to distribution *explicit* but more manageable in the form of dedicated programming language constructs.

This paper has introduced the programming language AmbientTalk, a high-level object-oriented distributed programming language which has been designed specifically for writing mobile computing applications. AmbientTalk has been designed bottom up, taking into account the unescapable hardware phenomena engendered by the wirelessly communicating mobile hardware. At its core, AmbientTalk is based on the actor model. The asynchrony of the communication between actors closely matches the event-driven nature of mobile computing systems. AmbientTalk elaborates on the actor model by providing more high-level support for e.g. dealing with return values and by introducing a built-in peer discovery mechanism through special actors known as ambient references.

References

[1] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich, "Mobile Computing Middleware," in *Advanced lectures on networking*, pp. 20–58. Springer-Verlag New York, Inc., 2002.

[2] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 261–322, 1989.

[3] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr, "Concurrency and distribution in object-oriented programming," *ACM Computing Surveys*, vol. 30, no. 3, pp. 291–329, 1998.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, New York, NY, USA, 2005, pp. 519–538, ACM Press.

[5] Barbara Liskov, "Distributed programming in Argus," *Communications Of The ACM*, vol. 31, no. 3, pp. 300–312, 1988.

[6] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109–133, February 1988.

[7] Luca Cardelli, "A Language with Distributed Scope," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1995, pp. 286–297, ACM Press.

[8] M. Miller, E. D. Tribble, and J. Shapiro, "Concurrency among strangers: Programming in E as plan coordination," in *Symposium on Trustworthy Global Computing*, R. De Nicola and D. Sangiorgi, Eds. April 2005, vol. 3705 of *LNCS*, pp. 195–229, Springer.

[9] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter, "Ambient-Oriented Programming," in *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, ACM Press.

[10] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented Programming in Ambienttalk," in *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, Dave Thomas, Ed. 2006, Lecture Notes in Computer Science, pp. 230–254, Springer.

[11] Alan Kaminsky and Hans-Peter Bischof, "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, 2002, pp. 72–73, ACM Press.

[12] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A. Brewer, and David E. Culler, "The emergence of networking abstractions and techniques in TinyOS," in *Proceedings of the first Symposium on Networked Systems Design and Implementation (NSDI 2004)*. March 29-31 2004, pp. 1–14, USENIX.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: a holistic approach to networked embedded systems," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[14] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa, "Abcl/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation," in *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, G. Blelloch, M. Chandy, and S. Jagannathan, Eds. 1994, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pp. 275–292, American Mathematical Society.

[15] B. Liskov and L. Shrira, "Promises: linguistic support for efficient asynchronous procedure calls in distributed systems," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 1988, pp. 260–267, ACM Press.

[16] Robert H. Halstead, Jr., "Multilisp: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, 1985.

[17] K. Kahn and Vijay A. Saraswat, "Actors as a special case of concurrent constraint (logic) programming," in *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, New York, NY, USA, 1990, pp. 57–66, ACM Press.

[18] Carlos Varela and Gul Agha, "Programming dynamically reconfigurable open systems with SALSA," *SIGPLAN Not.*, vol. 36, no. 12, pp. 20–34, 2001.

[19] Mads Haahr, Raymond Cunningham, and Vinny Cahill, "Supporting corba applications in a mobile environment," in *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, New York, NY, USA, 1999, pp. 36–47, ACM Press.

[20] P. Reynolds and R. Brangeon, "DOLMEN - service machine development for an open long-term mobile and fixed network environment," 1996.

[21] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek, "Mobile computing with the rover toolkit," *IEEE Transactions on Computers*, vol. 46, no. 3, pp. 337–352, 1997.

[22] A. Schill, B. Bellmann, W. Bohmak, and S. Kummel, "Infrastructure support for cooperative mobile environments," *Proceedings of the Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. WET ICE '95*, pp. 171–178, 1995.

[23] D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer, "The case for non-transparent replication: Examples from Bayou," *IEEE Data Engineering Bulletin*, vol. 21, no. 4, pp. 12–20, december 1998.

[24] Stefanos Zachariadis, Licia Capra, Cecilia Mascolo, and Wolfgang Emmerich, "Xmiddle: information sharing middleware for a mobile environment," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, New York, NY, USA, 2002, pp. 712–712, ACM Press.

[25] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[26] Gianpaolo Cugola and H.-Arno Jacobsen, "Using publish/subscribe middleware for mobile systems," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 4, pp. 25–33, 2002.

[27] Mauro Caporuscio, Antonio Carzaniga, and Alexander L. Wolf, "Design and evaluation of a support service for mobile, wireless publish/subscribe applications," *IEEE Transactions on Software Engineering*, vol. 29, no. 12, pp. 1059–1071, December 2003.

[28] A. Murphy, G. Picco, and G.-C. Roman, "Lime: A middleware for physical and logical mobility," in *Proceedings of the The 21st International Conference on Distributed Computing Systems*. 2001, pp. 524–536, IEEE Computer Society.

[29] Marco Mamei and Franco Zambonelli, "Programming pervasive and mobile computing applications with the tota middleware," in *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, Washington, DC, USA, 2004, p. 263, IEEE Computer Society.

[30] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan 1985.

[31] Henry Lieberman, "Using prototypical objects to implement shared behavior in object-oriented systems," in *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*. 1986, pp. 214–223, ACM Press.

[32] Gul Agha, *Actors: a Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[33] Carlos A. Varela and Gul A. Agha, "What after java? from objects to actors," in *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, Amsterdam, The Netherlands, The Netherlands, 1998, pp. 573–577, Elsevier Science Publishers B. V.

[34] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama, "Object-oriented concurrent programming in ABCL/1," in *Conference proceedings on Object-oriented programming systems, languages and applications*. 1986, pp. 258–268, ACM Press.