AmbientTalk {Demo}

Modern Actors for Modern Networks

Tom Van Cutsem

Software Languages Lab,
Vrije Universiteit Brussel
tvcutsem@vub.ac.be

Abstract

The purpose of this demo is to showcase the AmbientTalk programming language. AmbientTalk is intended to be a "scripting language for mobile phones". It's a dynamic, object-oriented, distributed programming language with a focus on deployment in so-called *mobile ad hoc networks* - networks composed of mobile devices that communicate peer-to-peer using wireless communication technology. We discuss AmbientTalk's roots and devote special attention to its concurrent and distributed language features, which are founded on the actor model.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages

General Terms Design, Languages

Keywords Actor, MANET, Asynchrony, Events

1. Introduction

AmbientTalk is a modern (born 2006) actor-based programming language, designed specifically for a new class of computer networks, so-called *mobile ad hoc networks* [5]. These are networks populated by mostly mobile devices that communicate peer-to-peer using wireless communication technology, such as WiFi or Bluetooth. Thanks to the emergence of smartphone platforms such as iOS and Android, such networks have become omnipresent, and in this light AmbientTalk can best be summarized as "a scripting language for mobile phones".

The purpose of the demo is to give the workshop audience a brief overview of AmbientTalk, with special attention to its concurrent and distributed programming model, which is founded on actors. Our goal is to show how resilient mobile applications can be constructed with familiar building blocks like objects and messages, but also in what way these building blocks have to be adapted to fit the characteristics of mobile ad hoc networks.

2. AmbientTalk's Concurrency Model

In AmbientTalk, concurrency is spawned by actors: one AmbientTalk virtual machine may host multiple actors which execute concurrently. AmbientTalk's concurrency model is based on the communicating event loops model of the E programming language [4], which is itself an adaptation of the well-known actor model [1]. The E language combines actors and objects into a unified concurrency model. Unlike previous actor languages such as Act1 [3], ABCL [6] and Actalk [2], actors are not represented simply as "active objects", but rather as *vats* (containers).

Thus, actors are not represented as active objects, but rather as a collection of plain objects that share a single event loop. That event loop has a single message queue, containing messages addressed to its objects. The event loop perpetually takes a message from the message queue and invokes the corresponding method of the object denoted as the receiver of the message. Messages are processed serially to avoid race conditions on the state of the contained objects.

In AmbientTalk, each object is said to be *owned* by exactly one actor. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor may communicate using standard, sequential message passing or using asynchronous message passing. AmbientTalk borrows from the E language the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). It is possible for objects owned by one actor to refer directly to individual objects owned by another actor. Such references that span different actors are named *far references* (the terminology stems from E [4]) and only allow asynchronous access to the referenced object. Any messages sent via a far reference to an object are enqueued in the message queue of the owner of the object and processed by the owner itself.

Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the event loop processes of the actors which perpetually take messages from their message queue and synchronously execute the corresponding methods on the actor's owned objects. An event loop process never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. For example, when `A` sends a message to `B`, the message is enqueued in the message queue of `B`'s actor which eventually processes it.

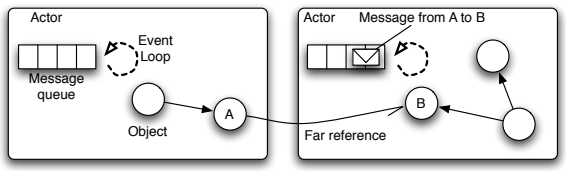

Figure 1. AmbientTalk actors as event loops

Asynchronous messages can be sent between objects owned by the same actor (via a local reference) or by different actors (via a

AGERE '11 Oct. 2011, Portland, OR, USA.

far reference). An asynchronous message send immediately returns a future, a placeholder for the actual return value.

AmbientTalk's concurrency model avoids low-level data races and deadlocks by design. It avoids low-level races because actors can only directly access their own objects. It avoids deadlocks because there is no blocking operation: message sending and reception is fully asynchronous.

3. Demo: a Simple Echo Service

During the demonstration, we will construct a simple echo service from scratch to showcase some of AmbientTalk's language features. The echo service simply accepts any incoming message and returns it immediately to the sender. We deliberately choose a trivially simple application to focus attention on the language rather than the application. A straightforward definition of such an echo service is defined in listing 1.

Listing 1. Definition of a Simple Echo Service

```
// the singleton echo service object
def s := object: {
  // method declaration
  def echo(msg) {
    system.println("Received: "+msg);
    msg // return value is the same message
  }
}

// define the type of the service (on the server)
deftype EchoService;

// advertise the service in the network
def pub := export: s as: EchoService;
```

The code in listing 1 defines a singleton echo service object `s`, and subsequently advertises this service in the local ad hoc network. The AmbientTalk VM takes care of broadcasting this advertisement to nearby listening VMs. On the client-side, a typical interaction with such an echo service is shown in listing 2.

Listing 2. A Simple Client Interaction

```
// define the type of the service (on the client)
deftype EchoService;

// discover the service
when: EchoService discovered: { |echoService|
  system.println("Discovered an echo service");

  // send an asynchronous message to the service
  def reply := echoService<-echo("test message")@TwoWay;
  // the following call does not block the actor:
  when: reply becomes: { |value|
    // react to the incoming reply
    system.println("Reply: " + value);
  }
}
```

Note that the client actor interacts with the echo service in a fully asynchronous and event-driven way: it first registers a callback to be triggered when an echo service was discovered by the underlying AmbientTalk VM. When this event occurs, the client sends an asynchronous `echo` message, and awaits the reply. Awaiting the reply is done by posting a callback, so that the underlying actor remains responsive to other events.

The above code only scratches the surface of AmbientTalk's language features. During the demo, we will additionally mention:

Futures Many callbacks can be avoided by using futures. We also discuss how futures can be chained together, leading to data-flow rather than control-flow synchronization.

Failures We discuss how AmbientTalk treats network failures, and how it allows applications to recover from them.

4. Summary

The purpose of the demo is to give a brief overview of AmbientTalk, a modern actor-based language. We focus specifically on AmbientTalk's concurrency and distribution model, founded on actors. Through the construction of a simple echo service, we will explain AmbientTalk's support for service discovery, asynchronous message passing and failure handling.

Availability

An open-source interpreter for AmbientTalk is available at `http://ambienttalk.googlecode.com`. The interpreter is written in Java and runs on any 1.4-compliant or later JVM. A special AmbientTalk distribution for Android phones is also available.

Acknowledgments

Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO). AmbientTalk is the product of a group effort. We thank Jessie Dedecker, Stijn Mostinckx, Wolfgang De Meuter, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Kevin Pinte and Dries Harnie.

References

[1] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986. ISBN 0-262-01092-5.

[2] J.-P. Briot. From objects to actors: study of a limited symbiosis in smalltalk-80. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 69–72, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-304-3. doi: http://doi.acm.org/10.1145/67386.67403.

[3] H. Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[4] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[5] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.

[6] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986. ISBN 0-89791-204-7. URL `http://doi.acm.org/10.1145/28697.28722`.