AmbientTalk
Seminar presentation

Axel von Engel

Institute of Operating Systems and Computer Networks
Technische Universität Braunschweig

July 13th, 2007
Outline

Motivation
- Ambient Resources
- Ambient-Oriented Programming

AmbientTalk
- Basic Language Features
- Advanced Language Features

Example Application
- Scenario Setup
- Source Code

Conclusion
Ambient Resources

An ambient resource is
- a networked device
- dynamically (un)available
- existence not known a-priori, require discovery

Examples:
- PDAs discovering each other when in communication range
- WLAN hot-spot
Hardware phenomena

As constraints like processing power and power drain become less important to mobile computing, new constraints arise:

- Connection Volatility
- Ambient Resources
- Autonomy
- Natural Concurrency
Ambient-Oriented Programming

The following concepts have been found crucial to address the hardware phenomena:

- Classless Object Model
- Non-blocking Communication
- Reified Communication Traces
- Ambient Acquaintance Management
Outline

Motivation

Ambient Resources
Ambient-Oriented Programming

AmbientTalk

Basic Language Features
Advanced Language Features

Example Application

Scenario Setup
Source Code

Conclusion
AmbientTalk

AmbientTalk is developed at the Programming Technology Lab of the Vrije Universiteit Brussel. Used as a language laboratory to experiment with ambient-oriented programming languages. Basic language features, explicitly designed to cover all ambient-oriented programming concepts:

- double layered object model
- first-class mailboxes
- reflectively extensible kernel
Double Layered Object Model

- active objects (\textit{actors})
  - single thread of execution
  - passed as reference
  - non-blocking communication
- passive (normal) objects
  - passed by copy
  - synchronous message passing
**First-class Mailboxes**

- Reified communication (right)
- Reified environment (left)
- Observers can be attached to a mailbox
Reflectively Extensible Kernel

AmbientTalk is built of three layers:

- Base AmbientTalk interpreted by
- Metacircular AmbientTalk interpreted by
- Native AmbientTalk written in Java

*Mirrors* and *Mirages* (German: Illusion) are used to extend AmbientTalk’s features using Metacircular AmbientTalk.
Advanced Language Features

- Stripes – annotate an object with a type
- Futures – allowing non-blocking calls to return sth.
- Symbiotic Programming with Java

and many more ...
Outline

Motivation
Ambient Resources
Ambient-Oriented Programming

AmbientTalk
Basic Language Features
Advanced Language Features

Example Application
Scenario Setup
Source Code

Conclusion
Example Application

Scenario:
An instant-messenger application running on hand-held mobile devices. The application will be able to handle disconnects transparently due to AmbientTalk’s communication concept.

Core application consists of:
- an interface used by local application
- an interface addressed by remote messengers
- event handler code
local interface

```java
1    def buddyList := jlobby.java.util.HashMap.new();
2
3    def localInterface := object: {
4        def sendTextMessage(to, string) {
5            def buddy := buddyList.get(to);
6            if: (nil == buddy) then: {
7                listener<-unknownBuddy(to);
8            } else: {
9                def msg := TextMessage.new(username, string);
10               buddy<-receiveTextMessage(msg,
11                   singleCallLease: seconds(30) for: ( object: {
12                   def resolve() {
13                       listener<-printMessage(msg);
14                   }
15                   }));
16               }
17            }
18        }
```
remote interface

20 def remoteInterface := object: {
21   def receiveTextMessage(textMessage, future) {
22     listener<-printMessage(textMessage);
23     future<-resolve();
24   }
25   def getName(future) { future<-resolve(username) };
26 
27 }
28 export: remoteInterface as: InstantMessenger;
event handler

30 whenever: InstantMessenger discovered: { |messenger|
31 messenger<-getName(singleCallLease: seconds(30) for: (object:{
32    def resolve(name) {
33        if: (nil == buddyList.get(name)) then: {
34            buddyList.put(name, messenger);
35            listener<-addBuddy(name);
36            when: messenger disconnected: {
37                listener<-buddyOffline(name);
38            }
39            when: messenger reconnected: {
40                listener<-buddyOnline(name);
41            }
42        }
43    }
44 });
45 }});
**Outline**

<table>
<thead>
<tr>
<th>Motivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambient Resources</td>
</tr>
<tr>
<td>Ambient-Oriented Programming</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AmbientTalk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Language Features</td>
</tr>
<tr>
<td>Advanced Language Features</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario Setup</td>
</tr>
<tr>
<td>Source Code</td>
</tr>
</tbody>
</table>

| Conclusion                |
Conclusion

benefits:
- superior concurrency and distribution abstractions
- reflectively extensible
- java library access

drawbacks:
- frequently changing
- interpreted language
  - source code open
  - slower than compiled code
Questions?