Resilient Actors: A Runtime Partitioning Model for Pervasive Computing Services

Engineer Bainomugisha[1], Jorge Vallejos[1], Éric Tanter[2], Elisa Gonzalez Boix[1], Pascal Costanza[1], Wolfgang De Meuter[1], Theo D'Hondt[1]
[1]Programming Technology Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussels, Belgium
{ebainomu, jvallejo, egonzale, pascal.costanza, wdmeuter, tjdhondt}@vub.ac.be
[2]PLEIAD Laboratory, Computer Science Department (DCC), University of Chile,
Av. Blanco Encalada 2120, Santiago, Chile
etanter@dcc.uchile.cl

ABSTRACT

In pervasive computing, software applications vanish into the user's environment spreading their functionality to computers integrated into everyday devices. With the current state-of-the-art software tools, these characteristics put a great burden on programmers who have to enable the applications to dynamically partition across multiple devices, and to adapt such partitioning to frequent context changes such as network failures. This paper explores service partitioning techniques for development of pervasive computing applications. We propose a resilient actor model to structurally add service partitioning property to the pervasive applications. The service partitioning realised using resilient actor model happens at runtime, is user guided and the resulting partitioned application is retractable, and resilient to network failures.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.3.3 [**Language Constructs and Features**]: Modules, packages; Concurrent programming structures

General Terms

Design, Languages, Reliability

Keywords

runtime service partitioning, pervasive computing, resilient actors, programming language model

1. INTRODUCTION

In pervasive computing, software applications vanish into the users' environment, spreading their functionality in computers integrated into everyday devices. Rather than monolithic services, applications in this new paradigm are expected to offer services that maximise the use of the resources found in the user's surroundings [18]. Whereas this scenario is becoming ever more realistic from a hardware point of view, programming such applications remains notoriously difficult – due to the dynamic nature of pervasive computing environments. A pervasive computing environment consists of a variable number of stationary and mobile devices that become available or unavailable as the user moves about [3]. With the current state-of-the-art software tools, these characteristics put a great burden on programmers who have to enable the applications to dynamically partition across the devices, and to adapt such partitioning to frequent network disconnections.

This paper explores the use of *service partitioning* techniques for the development of pervasive computing applications [20, 8, 19]. Using these techniques, software applications can be decomposed into parts that can be distributed to different devices. However, thus far we observe that in most of the existing approaches the partitioning is a static operation performed by the programmers and cannot be changed by the end-user once the application is running. In this work we argue that partitioning of pervasive computing services should (1) occur at runtime, conducted by – non-technical – end users, (2) ensure that after the partitioning, the application can always come back to a local state, and (3) enable the partitioned application to be resilient to network disconnections.

To fulfill these requirements, we propose an actor-based service partitioning model called the *resilient actor model.* In this approach, the functionality of an application is decomposed into a set of *resilient actors* which are program units interconnected through *elastic* bindings. Partitioning such an application corresponds to moving the resilient actors to different devices, while reversing the partitioning is achieved by pulling back the resilient actors. Our model is built on top of the actor-based concurrency and distribution model of the AmbientTalk programming language [21], which is specially designed for developing pervasive computing applications. We validate our model by developing an ambient music player application.

The rest of the paper is structured as follows. Section 2 describes a scenario from which we derive requirements for partitioning of pervasive services. We present the resilient actor model in Section 3 and describe its language abstrac-

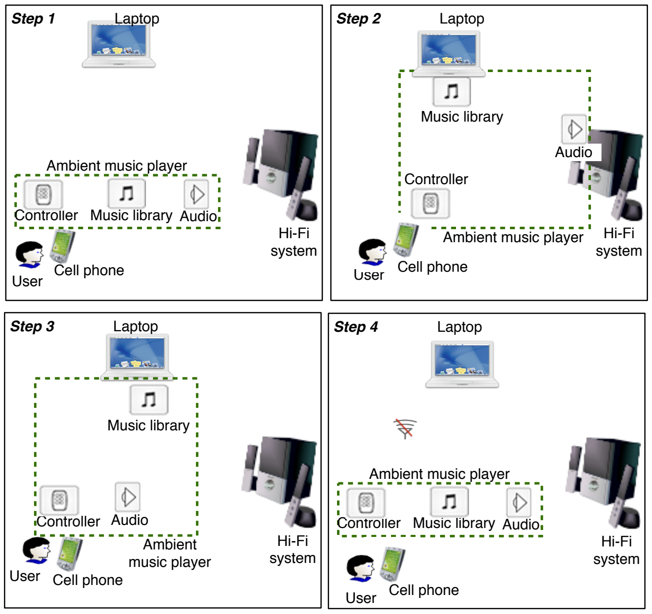

Figure 1: Scenario: Ambient music player in a pervasive computing environment

tions in Section 4. We discuss the results of our model in Section 5 and its implementation details in Section 6. We review the related work in Section 7, and conclude in Section 8.

2. PARTITIONING OF PERVASIVE COMPUTING SERVICES

In this section, we describe the requirements of service partitioning in pervasive computing environment. We derive these requirements from the analysis of a scenario of a music player application for pervasive computing environments.

2.1 Scenario: Ambient Music Player

Figure 1 shows an Ambient Music Player (*AMP*) application that runs on devices such as cell phone, laptop, and Hi-Fi system. The *AMP* is composed of three services: the controller service for operating the music player, the music library service which contains playlists of songs, and the audio service for sound output. Assume that a user starts up the ambient music player on his cell phone (Figure 1 *Step 1*). As the ambient music player launches on the cell phone, a dialog box pops up notifying the presence of Hi-Fi system and laptop devices in the surroundings. The user decides to distribute the music player application by moving the audio service to the Hi-Fi system for better sound quality, using the music library service at the laptop, and maintaining the rest of the music player application (i.e the controller service) at the cell phone (Figure 1 Step 2). Furthermore, the user can decide to move the audio service back to the cell phone (Figure 1 *Step 3*). When there is a network disconnection between the cell phone and laptop, all the services come back to the cell phone (Figure 1 *Step 4*).

2.2 The Need for Resilient Service Partitioning

The *AMP* scenario reveals a number of issues that apply to applications that run in a pervasive computing environment. We refer to the action of distributing the music player application to run on multiple devices as *service partitioning*. The fact that the ambient music player runs on the laptop and the Hi-Fi system does not imply closing the application at the cell phone and starting up the application again at the laptop and the Hi-Fi system. This *AMP* scenario raises the following requirements for service partitioning in pervasive computing environment:

2.2.1 *Runtime Service Partitioning*

Traditionally, service partitioning is achieved statically, which requires knowing the devices at which application partitions will run at the development time. In a pervasive setting, this is impractical as users can decide to move services between devices as they become available or unavailable during the application execution. Therefore, there is the need for service partitioning at runtime. For instance, in the *AMP* scenario, the music player application initially running only at the cell phone device is partitioned at runtime to the cell phone, laptop, and Hi-Fi system devices (Figure 1 *Step 2*) when they become available.

2.2.2 *Retractable Service Partitioning*

Service partitioning should be retractable so that the availability of services is not affected as users move about. A partitioned application should be able to return to its local state. For instance in the *AMP* scenario, the user moves the audio service from the Hi-Fi system back to the cell phone (Figure 1 *Step 3*).

2.2.3 *Service Partitioning Resilient to Network Failures*

Pervasive environments are characterized by frequent network disconnections due to the volatile connections that interconnect the devices [13]. Therefore, a partitioned application running in such environment should be able to deal with network disconnections. For example, in the *AMP* scenario, a network disconnection may occur when the music library service is running on the laptop while the rest of the ambient music player is on the cell phone. In the face of such network disconnection, all the ambient music player services come back to the original device (i.e the cell phone) (Figure 1 *Step 4*).

To the best of our knowledge, no single existing approach for service partitioning addresses all the three requirements identified in this section. We observe that new trends of software applications such as [22] provide support (to some extent) of running amongst multiple devices. However, our main focus is to provide a programming language model for service partitioning that can be applied to any kind of application. We further discuss the related work in Section 7. In the following section, we introduce our *resilient actor model* for addressing these requirements.

3. THE RESILIENT ACTOR MODEL

In this work, we propose a programming language model for service partitioning, called the *resilient actor* model. It is built on top of the concurrency and distribution model of the AmbientTalk [21] programming language, which is an extension of the actor model [1], specially designed for pervasive computing environments. Originally, actors are defined as program units that encapsulate behaviour and communicate via asynchronous message passing. AmbientTalk model extends this definition by enabling the actor's behaviour to be represented as a container of objects which can be directly referenced from outside the actor. These references are specially provided with support for handling network failures. In the resilient actor model, we extend the AmbientTalk actors and references to provide support for service partitioning.

In the remainder of this section we introduce the main concepts of the resilient actor model, illustrate them using concrete *AMP* scenario, and further explain the model using definitions.

Resilient actor It is a program entity that encapsulates a set of objects and defines *elastic bindings* to other resilient actors. A resilient actor serves as a unit of service partitioning and represents an application functionality.

Elastic binding It is a unidirectional reference that interconnects two resilient actors. Each elastic binding supports two partitioning operations: *stretch* and *retract*. The stretch operation allows actors to be distributed to different devices. The retract operation is the "undo" to the stretch operation, i.e. it reverses the service partitioning caused by the stretching of an elastic binding. We propose two forms of retract: (1) *manual* retract which is initiated by the application end-user, and (2) *automatic* retract which is initiated by a network disconnection. An elastic binding maintains a history stack of the actor's previous states that we use to achieve retraction. The stretch and retract are high-level partitioning operations that are defined with different distribution and mobility techniques. We have designed these different techniques as *resilience strategies.*

Resilience strategy Resilience strategies specify different definitions of stretch and retract operations. A resilience strategy is applied to an elastic binding to specify the mobility policy for the resilient actor. Thus far, we provide four resilience strategies: (1) *move* which moves an actor to a new location, (2) *copy* which creates a copy of the actor and moves it to a new location, (3) *rebind* that changes elastic binding to reference a different actor providing the same service, and (4) *standstill* which makes an elastic binding to always reference the same actor during service partitioning. Retract operation under all these strategies, undoes the partitioning caused by the stretch operation. Resilience strategies are specified on both the elastic binding and the resilient actor. We further explain the stretch and retract operations under each strategy in Section 3.3.

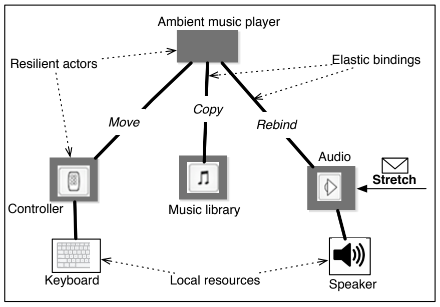

Figure 2: Ambient music player application constructed using resilient actors interconnected by elastic bindings

3.1 Resilient Decomposition of Services

Figure 2 depicts the *AMP* application introduced in Section 2.2 built using the resilient actor model. *AMP* services (controller, music library, and audio) are built as *resilient actors* interconnected by *elastic bindings*. Each service has a resilience strategy that specifies how it is distributed (i.e. *move* for the controller service, *copy* for the music library, *rebind* for the audio service). Assume that the application is initially running on one device. When a new device is discovered providing a better sound service, the user can simply move the audio service to the remote device. Internally, this partitioning is realised by applying the stretch operation on the audio service as illustrated in the Figure 2. Since this audio service is specified with the rebind strategy, then the stretch operation changes the elastic binding to reference the remote audio service. When the user wants to resume using the audio service at the original device, then a retract operation is initiated. The topmost (root) service represents a point of retraction where all *AMP* services will be retracted to, in case of a network disconnection.

We further explain the implementation of *AMP* application using our language abstractions in Section 4. In the remainder of this section we describe the resilient actor model definitions using equations.

3.2 Resilient Actor Model Definition

The following equation presents the definition of an actor that we use to explain our model:

$$a \triangleq (\{o_i\}_n, \{b_j\}_m) \quad (1)$$

An actor a encloses a set of objects $\{o_i\}_n$ and a number of bindings $\{b_j\}_m$ to other actors. We call the actor that contains the binding, the source actor s, and the referenced actor, the target actor t. The following equation shows the definition of a binding in terms of these two actors:

$$b \triangleq (s, t) \tag{2}$$

To support service partitioning, we extend the actor definition of Equation (1) to contain elastic bindings $\{eb_k\}_p$ to other actors and a resilience strategy st that defines the mobility policy of the actor. The equation below shows the definition of a resilient actor in our model:

$$a \triangleq (\{o_i\}_n, \{b_j\}_m, \{eb_k\}_p, st) \tag{3}$$

Each elastic binding eb has a resilience strategy st and a history stack h that stores the previous actor states and locations. The history stack is used to achieve retraction. The equation below shows the definition of an elastic binding between source s and target t actors at locations i and j, respectively.

$$eb \triangleq (s_{l_i}, t_{l_j}, st, h) \quad where\ i, j = (1, 2, ..n) \tag{4}$$

An elastic binding eb supports two partitioning operations: *stretch* and *retract*. The stretch operation can be applied to an elastic binding eb to move the target resilient actor to a desired location l_i as follows:

$$stretch(eb, l_i) \tag{5}$$

A stretched elastic binding can be restored to its previous state by applying a retraction operation as shown in the equations below:

$$retract(eb) \tag{6}$$

In the remainder of this section, we discuss stretch and retract operations under each resilience strategy.

3.3 Resilience Strategies

The stretch and retract operations are only high-level partitioning abstractions that have several implementations. We provide a number of resilience strategies that specify different behaviour for these partitioning operations. This section explains the four resilience strategies identified so far. For each resilience strategy we present the equation that corresponds to the stretch operation. The equation of the retract operation has similar semantics for all the strategies, and as such, it is described at the end of this section.

Move strategy.

The move strategy defines the stretch operation as moving the target actor to a new location. Existing references to the target actor are updated so that they point to the actor at the new location. A copy of the target actor is kept at the original location in order to achieve automatic retraction. The equation below illustrates the stretch operation under this strategy.

$$\frac{eb = (s_{l_1}, t_{l_1}, st, h) \quad st = move}{stretch(eb, l_2) \rightarrow (s_{l_1}, t_{l_2}, st, (t_{l_1}.h))} \tag{7}$$

We assume that the source and target actors are initially at the same location l_1. The operation *stretch(eb, l_2)* moves the target actor t from location l_1 to location l_2. The current state of the target actor and its location are stored on a history stack h, that we use to achieve retraction. Figure 3 illustrates the stretch and retract operations under this strategy.

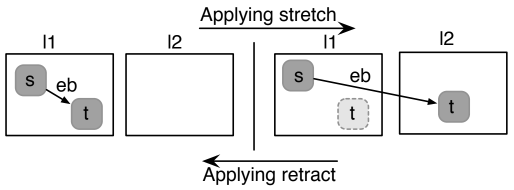

Figure 3: Before and after partitioning under the move strategy

Copy strategy.

The copy strategy specifies the stretch operation as creating a copy of the target actor and then moving it to a new location. Unlike the move strategy, under the copy strategy the original target actor is maintained at its location and can still be referenced by other elastic bindings. We further explain the stretch operation under this strategy using the following equation:

$$\frac{eb = (s_{l_1}, t_{l_1}, st, h) \quad st = copy}{stretch(eb, l_2) \rightarrow (s_{l_1}, t'_{l_2}, st, (t_{l_1}.h))} \tag{8}$$

In this case, a copy of the target actor t' is moved from location, l_1 to location l_2. These actions are depicted in the Figure 4.

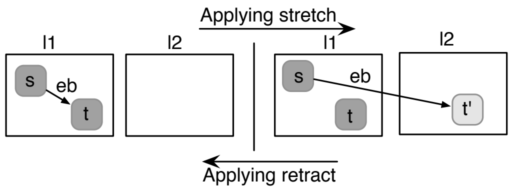

Figure 4: Before and after partitioning under the copy strategy

Rebind strategy.

The rebind strategy specifies stretch operation as binding to the target actor at a remote location that provides the same service[1]. We use the following equation to illustrate the stretch operation under this strategy.

$$\frac{eb = (s_{l_1}, t_{l_1}, st, h) \quad st = rebind}{stretch(eb, l_2) \rightarrow (s_{l_1}, t^*_{l_2}, st, (t_{l_1}.h))} \tag{9}$$

Applying a stretch operation on eb, changes the binding from the target actor t at location l_1 to a different target actor t^* at location l_2. We further illustrate the this in the Figure 5.

[1]Our model relies on peer-to-peer publish/subscribe discovery mechanism to locate available services. We refer the reader to the dedicated literature [5] for further details

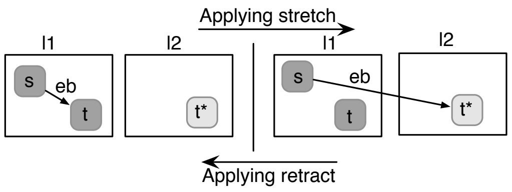

Figure 5: Before and after partitioning under the rebind strategy

Standstill strategy.
The standstill strategy specifies the stretch operation as always binding to the same target actor at one location. The standstill strategy may be useful to specify actors that represent services that must always be stationary at one device. We consider this strategy as the default for actors. We further illustrate the stretch operation under this strategy using the equation below.

$$\frac{eb = (s_{l_1}, t_{l_1}, st, h) \quad st = standstill}{stretch(eb, l_2) \rightarrow (s_{l_1}, t_{l_1}, st, (t_{l_1}.h))} \tag{10}$$

In this case, the target actor t remains at location l_1. We further illustrate stretch operation under this strategy in the Figure 6.

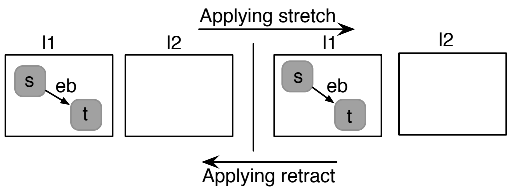

Figure 6: Before and after partitioning under the standstill strategy

In all strategies, the retract operation restores a stretched elastic binding to reference the previous target actor. We illustrate this in the equation below:

$$\frac{eb = (s_{l_1}, t_{l_j}, st, (t_{l_1}.h))}{retract(eb) \rightarrow (s_{l_1}, t_{l_1}, st, h)} \tag{11}$$

Applying *retract* operation restores the elastic binding from the target actor t at location j to the previous target actor t_{l_1}. For the copy, rebind and standstill strategies the target actor still exists at its previous location which can be bound to either by manual or automatic retraction. In the case of move strategy retraction involves moving the original target actor back to its first location and restoring the binding. However, moving back the original target actor is not possible in case of a network disconnection. Therefore, automatic retraction under the move strategy rebinds to the copy of target actor kept at the previous location.

3.4 Propagation and Resolution of Strategies

A source actor holds elastic bindings to target actors which in turn may hold elastic bindings to other actors and so on. This implies that a partitioning operation should be transmitted through all elastic bindings to other referenced actors and so on. We propose a *propagation* mechanism that enables stretch and retract operations to proceed from one actor to another via elastic binding.

Since we allow specification of a resilience strategy at elastic binding and resilient actor, strategy conflicts may arise. Strategy conflicts arise in case the elastic binding specifies a different resilience strategy from that of the resilient actor being referenced. In case of strategy conflicts, a single resilience strategy should be decided. We propose a *conflict resolution* mechanism which is based on case-by-case matching of the resilience strategies defined at elastic binding and resilient actor. Specifying an elastic binding with standstill or rebind strategy, the final strategy will always be rebind strategy, otherwise the resilience strategy defined on the resilient actor is considered. We allow specification of different resilience strategies mainly because different services may be implemented by different programmers.

In the following section, we demonstrate the resilient actor model language support that we provide for service partitioning by implementing the *AMP* application introduced in Section 2.2.

4. RESILIENT ACTORS IN AMBIENTTALK

In this section we demonstrate how the *AMP* application is implemented using our resilient actor model. We implement the resilient actor model using AmbientTalk [21], an actor language specially designed for pervasive computing environments. We extend AmbientTalk with two language constructs: `actor:resilientAs:` and `bindTo:resilientAs:` for defining a resilient actor and an elastic binding, respectively. We further provide two methods `stretch:` and `retract:` to support stretch and retract partitioning operations, respectively. We illustrate the usage of these language constructs by implementing the AMP application introduced in Section 2.

4.1 Implementation of an Ambient Music Player

Each AMP service is implemented as a resilient actor. The code snippet below shows the implementation of the music library service.

```
def musicLibrary := actor: {
 def myLib := Vector.new();
 def song := object: {
   def title;
   def artist;
   def init(aTitle,anArtist) {
      title := aTitle;
      artist := anArtist;
   }
 };
   def addSong(title , artist){
     myLib.add(song.new(title , artist));
   };
   def deleteSong(title){
     myLib.remove(title);
   };
   def getPlayList(){
     myLib.toArray();
   }
} resilientAs: [copy];
```

The `actor:resilientAs:` construct above creates a resilient actor which is bound to the `musicLibrary` variable

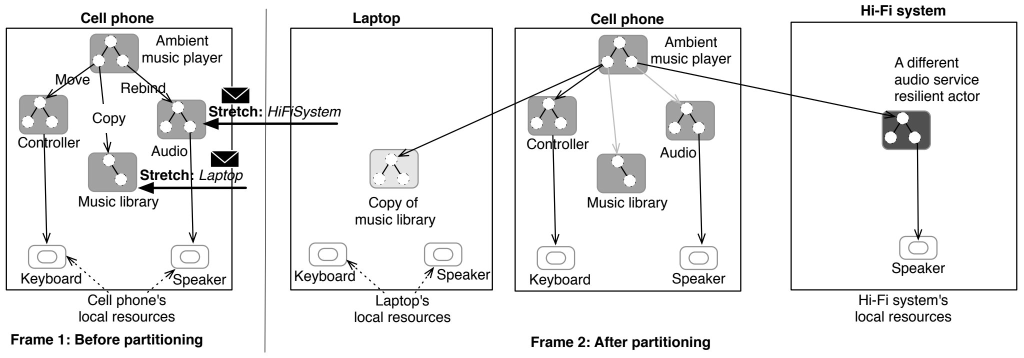

Figure 7: Partitioning of the Ambient Music Player Application

defined using the `def` keyword. The variable `myLib` represents the playlist of songs. The construct `object:` creates an object with two fields `title` and `artist` that is bound to the variable `song`. The `init` method plays the role of a constructor for creating an object from an existing one. The methods `addSong` and `deleteSong` add to, and remove songs from the playlist, respectively. The method `getPlayList` is responsible for retrieving a playlist from the music library as a table of songs. The argument to the `resilientAs:` is a table of resilience strategies that are specified for the `musicPlayerLibrary` resilient actor. In this case, `copy` strategy which implies that when the music player application is partitioned, the destination device will receive the copy of original music library. In case more than one strategy is specified, the resolution mechanism described in Section 3.4 is used at runtime to determine the final strategy.

We implement the audio service as follows

```
def  audio := actor: {
 def theSpeaker :=  /.at.devices.speaker;
 def play(audioContent) {
  theSpeaker <- receiveSound(audioContent);
 };
 def stop() {
    theSpeaker <- muteSound();
 };
} resilientAs: [move];
```

The `actor:resilientAs:` construct above creates a resilient actor which is bound to the `audio` variable. The variable `theSpeaker` holds a reference to the actor representing the speaker local resource. The `play` method sends the audio data to the `theSpeaker`. The expression `theSpeaker <- receiveSound(audioContent)` sends message `receiveSound` asynchronously[2] to the object `theSpeaker`. The `stop` method sends the `muteSound` message to the `theSpeaker` object. The argument to the `resilientAs:` specifies the resilience strategy applied to the `audio` resilient actor as `move`. This implies that the user can use any available sound device by moving the audio service from the current device.

The code fragment below demonstrates how the controller service is implemented.

```
def  controller := actor: {
 def theKeyboard := /.at.devices.keyboard;
 def getInput() {
       theKeyboard <- input();
 };
 def showcontrols() {
    //code for controller GUI
 };
} resilientAs: [move];
```

The `actor:resilientAs:` construct above creates a resilient actor which is bound to the `controller` variable. The variable `theKeyboard` holds a reference to the actor representing the keyboard local resource. The `getInput` method accepts user input from the keyboard. The `showControls` method displays the control menu for managing the music player application. The argument to the `resilientAs:` specifies the resilience strategy applied to that controller resilient actor as `move`. This implies that the user can manage the music player application from any device by moving the controller service to the preferred device.

The code fragment below shows the implementation of the ambient music player service.

```
def ambientMusicPlayer :=actor:{
   |controller , audio ,musicLibrary|
   def theController := bindTo: controller
                           resilientAs:[move];
    def theAudio := bindTo: audio
       resilientAs: [rebind(audioService)];
   def theMusicLib := bindTo: musicLibrary
     resilientAs: [copy];
   def play() {
      theAudio <- play(nextSong);
   };
   def stop() {
      theAudio <- stop();
   };
} resilientAs: [standstill];
```

The `actor:resilientAs:` construct above creates a resilient actor which is bound to the `ambientMusicPlayer` variable that represents ambient music player application as a whole. The `play` and `stop` method are responsible for forwarding the `play` and `stop` messages to the audio resilient actor. In the `actor:resilientAs:` construct, the argument

[2]AmbientTalk makes a distinction between sequential message sends (expressed as o.m()) and asynchronous message sends (expressed as o <- m()) [21]

to the `resilientAs:` specifies the resilience strategy to be applied to the music library resilient actor as `standstill`. This implies that as the user moves about, all services currently running at different devices always come back to the device on which the application was started in case of a network failure.

The `bindTo:resilientAs:` construct defines an elastic binding between two resilient actors. The `theController` represents the elastic binding to the `controller` resilient actor with the `move` strategy. The `theMusicLib` is an elastic binding to the `musicLibrary` service with the `rebind` strategy. In this case there is a conflict of strategies because `theMusicLib` elastic binding specifies `rebind` as a strategy while the `audio` resilient actor is defined with a `move` strategy. The final resolved strategy applied to the audio resilient actor is `rebind` based on the resolution mechanism explained in Section 3.4. The rebind strategy implies that when a different device is discovered providing a sound service, then the user can simply use the audio service at the remote location without moving the audio service at the current device.

Having discussed the implementation of different services of the application, we now explain how service partitioning occurs. Figure 7 *Frame 1* depicts the ambient music player developed using our language constructs. Assume that the user enters a room that has a Hi-Fi system and decides to move the audio service from the cell phone to Hi-Fi system. While the user interacts with the GUI of the music player to perform this action[3], internally the actor representing such service receives the stretch message as follows:

```
  audio <-  stretch: Hi-FiSystem;
```

In the above code snippet, `Hi-FiSystem` is the reference to the actor representing the Hi-Fi system. Other music player services can be moved to a desired location by sending the stretch message. For example, Figure 7 *Frame 2* shows the partitioned ambient music player application with music library at the laptop, audio service at the Hi-Fi system, and controller service at the cell phone. Note that the controller service is specified with a move strategy, and therefore the user can also move this service to any device in the surroundings. Applying a retract operation restores a service to its original location. For instance, the audio service can be retracted to the cell phone device by sending a retract message as follows:

```
  audio <-  retract;
```

5. DISCUSSION

In the previous section we demonstrated the resilient actor model language abstractions in action. In this section we discuss our resilient actor model solution in light of the requirements of service partitioning for pervasive computing environments identified in Section 2.2.

Runtime service partitioning Implementing each service as a resilient actor yields the application that is composed of interconnected resilient actors that can be distributed at runtime to different devices. The decision of which application services run on what device depends on the preference of the end-user.

[3]The discussion of the GUI implementation details is out of scope of this paper.

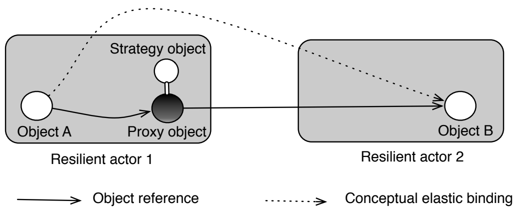

Figure 8: The Elastic Binding Implementation as a Proxy Object

Retractable service partitioning Because the application services are interconnected through elastic bindings, it is possible to move back a service to its original location even after the application is partitioned across multiple devices.

Service partitioning resilient to network failures The partitioned application distributed amongst multiple devices is resilient to network disconnections. In case of a network disconnection, an automatic retraction is initiated that moves the services to the available latest previous devices. This characteristic is particularly important in that the partitioned application is not immediately affected by the network failures.

The music player application is a simple experimentation example, however it demonstrates important issues that apply to any other application providing support for runtime service partitioning. Implementing runtime partitioning of pervasive services using traditional languages like Java requires the programmer to directly deal with low-level implementation details such as service discovery, socket-based communication, and serialization. The highly dynamic nature of pervasive environments (e.g. frequent network failures) makes these issues even more complex as they need to be tackled in combination with network reconfiguration (to manage references to services) and exception handling mechanisms (to deal with network failures). Whereas previous approaches such as automatic application partitioning [20, 19] and object migration [7, 15, 4] attempt to provide abstractions for dealing with these problems, none of these approaches addressees network failure handling. We further compare our work with the previous approaches to service partitioning in Section 7.

6. IMPLEMENTATION

Resilient actors have been implemented reflectively on top of the AmbientTalk language. We override the default Meta-Object Protocol [9] to intercept all asynchronous messages received by a resilient actor. We extend AmbientTalk actors with resilience strategies which have been implemented as objects. An elastic binding has been implemented as an extension to AmbientTalk's object reference. We implement an elastic binding as a proxy object whose behavior is represented by an object with a resilience strategy as its meta-level entity (Figure 8).

6.1 Resilience Strategy Object

A resilience strategy has been implemented as an object with two methods: `stretch` and `retract`. We consider the copy strategy to describe the implementation details of the resilience strategies. The code snippet below shows the implementation of the copy strategy:

```
def copyStrategy := object: {
   def isStretched;
   def proxyService;
   def prevService;
     def init(){
       isStretched := false;
       prevService := nil;
       proxyService := nil;
     };
   def stretch: location {
     isStretched := true;
     when: location disconnected: {
       retract ();
     };
   };
  def retract() {
    if: (isStretched) then: {
     isStretched := false;
     proxyService:= prevService;
    }
  };
  def getState(){
     proxyService.state();
   };
  def setState(newState){
     apply(proxySevice, newState);
   };
}
```

The variable `isStretched` is set to true or false depending on whether the stretch message has be received or not. The variable `proxyService` holds a reference to the object representing the current state of the actor. The variable `prevService` holds the state of the original resilient actor when the stretch message is received by the actor. The `stretch:` method implements the stretch partitioning operation. The argument to the stretch method is the desired location to which a resilient actor can be distributed.

In order to deal with network disconnections, we make use of the AmbientTalk's network failure handling mechanism with the `when:disconnected: {...}` construct. This construct places an observer on a remote reference which is triggered when a network disconnection occurs. The execution of the block closure under `when:disconnected:` construct sends a `retract` message to perform the retraction operation. We refer to this kind of retraction as automatic retraction. The `retract` method implements the retraction partitioning operation. The `getState` and `setState` methods are for changing and retrieving the current state of the actor, respectively.

6.2 Extensible Implementation

As explained above, a resilience strategy is implemented as an object which can be extended by the programmer to create custom strategies. To implement a new resilient strategy, a programmer basically defines an object with two methods `stretch:` and `retract`. In the remainder of this section, we explain a resilience strategy which extends the copy strategy with custom semantics.

6.2.1 Towards Proactive Replication

The basic implementation of the copy strategy explained in Section 6.1 does not provide support for replication of the resilient actor state. This implementation may be extended such that the original resilient actor state is updated periodically with new state of its copy. In the code snippet below, we show our initial implementation towards proactive state replication for the copy strategy:

```
def copyStrategyExtension := extend:
 copyStrategy with: {
   def time := 10;
   def stretch: location {
    super^stretch: location;
     whenever: seconds(time) elapsed: {
      when: proxyService<- getState()
        becomes: { |mostRecentServiceState|
          setState(mostRecentServiceState);
        };
      };
   };
};
```

The expression `extend:with:` creates an object whose parent is `copyStrategy` and is bound to the variable `copyStrategyExension`. In the above example we override the default `stretch`: method with support to update the state of original resilient actor with the most recent state of the moved a copy resilient actor. The expression `super^stretch:` delegates the message `stretch:` the `copyStrategy` object. The block of code specified in `whenever:elapsed` construct will be executed after every 10 seconds to update the state of the original resilient actor. The `when:becomes` construct is used to obtain a return value from an asynchronous message send. The `becomes:` block of code is executed when the return value is resolved.

We also implement another extension of the default automatic retraction of the rebind strategy with the support to grant the use of a service for a certain period of time. For space reasons, we refer to [2] for more details.

7. RELATED WORK

The idea of service partitioning has been advocated as a mechanism for distributing a software application across available computing resources [10, 14, 17]. However, to the best of our knowledge no existing approach addresses all the requirements of service partitioning in a pervasive environment identified in Section 2.2. In this section, we discuss the closely related existing approaches for service partitioning. First, we discuss automatic service partitioning approaches based on objects (J-Orchestra [20], Addistant [19], JavaParty [15], and Doorastha [4]). Second we explore the service partitioning approaches based on agents and components (AdJava [7], Coign [8], and Hydra [17]).

7.1 Object-oriented Service Partitioning

J-Orchestra [20] is a system for transforming a centralized Java program into a distributed one. Service partitioning is achieved automatically by taking in as input a Java application in byte code format and a policy file with location information on which the partitions of the application will execute. J-Orchestra has been used to automatically partition realistic pervasive computing systems such as *Kimura* system [12, 23], but a number of drawbacks limit its applicability to pervasive applications [11]. First, J-orchestra achieves service partitioning at compile time and the location of services can not be changed once the application is started. Second, the resulting partitioned application is not resilient to network failures.

JavaParty [15] is a system that transforms a centralized Java program into a distributed JavaParty program that can be spread across a distributed environment. JavaParty achieves service partitioning through a runtime system that migrates objects from host to host based on load-balancing and network partitioning algorithms [6]. The programmer identifies potential distributable instances using the annotation `remote`. Unlike J-Orchestra, JavaParty does not involve rewriting of the Java program. The automatic distribution of objects achieved in JavaParty implies runtime service partitioning but no support for user controlled service partitioning. Rather, the runtime system migrates objects transparently based on the load balancing techniques. In addition, JavaParty does not support retractable service partitioning and no support for handling network failures.

Addistant [19] is a system for adapting a software application built to run on a single host for execution on multiple hosts in a distributed setting. More concretely, given a Java byte code, Addistant transforms it such that it can be executed on different Java Virtual Machines (JVMs). Like J-Orchestra, Addistant achieves service partitioning by translating a Java byte code based on the policy file provided by the programmer. The service partitioning achieved by Addistant happens at compile time and can not be dynamically changed at runtime. It does not provide support for retractable partitioning nor network failure handling.

Doorastha [4] is a system for adapting a centralized Java program for execution in a distributed environment. Doorastha achieves automatic service partitioning by use of code transformations and an additional runtime system on top of the Java Remote Method Invocation (Java RMI). The runtime system is responsible for object migration at runtime. The Doorastha system allows the programmer to annotate a Java program to turn it into a distributed one. Like JavaParty, service partitioning in Doorastha is realized through a runtime system that transparently migrates objects at runtime but no support is provided for user controlled object migration. In addition, Doorastha does not support retractable service partitioning and no mechanisms for handling network failures.

7.2 Component-oriented and Agent-oriented Service Partitioning

Coign [8] is an automatic partitioning system for applications built with Microsoft's Component Object Model (COM) components. The automatic service partitioning realised in Coign system is dependent on the minimal communication time between components. This service partitioning happens at compile time and cannot be altered at runtime. Moreover, the lift-to-front minimum-cut graph cutting algorithm [16] used by Coign can only produce two partitions of the application. This is impractical in a pervasive computing environment where multiple computing devices are available. Like J-Orchestra and Addistant, Coign does not provide support for retractable service partitioning nor network failure handling.

AdJava [7] is an automatic service partitioning tool that distributes an application across available resources in the network. AdJava is implemented as an agent-oriented system where the Java program is pre-processed to transform local objects into remote objects and the resulting code is compiled by the "regular" Java compiler. The programmer puts annotations into the program code to indicate the objects that can be distributed and a list of remote hosts on which agents are running and where objects are distributed. Like JavaParty, AdJava achieves runtime service partitioning based on load balancing technique but no support for user controlled service partitioning. AdJava does not provide support for retraction and no mechanisms for handling network failures.

Hydra [17] is a framework designed for pervasive computing environment for building applications that can be dynamically deployed at a computing device during execution of the application. Hydra achieves runtime service partitioning by building applications as mobile agents based on software components that can move from host to host. Since the components can be moved from host to host at runtime, some form of retraction can be realized. Hydra's main focus is on runtime partitioning of pervasive services. However, Hydra does not provide mechanisms for dealing with network disconnections.

Our evaluation of closely related approaches reveals that no single approach addresses all the three service partitioning requirements identified in Section 2.2 . For example, JavaParty [15] and AdJava [7] attempt to achieve runtime service partitioning through a runtime system that performs automatic object distribution. However, their service partitioning is controlled by the load-balancing techniques and no mechanisms for user controlled service partitioning is supported. J-Orchestra [20], Addistant [19], and Coign [8] systems achieve the service partitioning at compile time and can not be changed at runtime. Of all these approaches, Hydra [17] is only the system that was designed for pervasive computing environments with main focus on supporting runtime service partitioning. Although these approaches address some issues of services partitioning, none of these systems provides support for handling network failures.

8. CONCLUSIONS AND FUTURE WORK

This paper discusses service partitioning in the domain of pervasive computing. We have identified requirements for service partitioning in a pervasive computing environment: (1) Runtime service partitioning, (2) Retractable service partitioning, and (3) Service partitioning that is resilient to network failures. We subsequently propose a *resilient actor model* that addresses these requirements. Using our resilient actor model application services are represented as resilient actors interconnected by elastic bindings. These elastic bindings support stretch and retract operations for service partitioning and retraction respectively. We have discussed a set of resilient strategies (*copy, move, rebind and standstill*) that can be applied to the resilient actors to specify the mobility policies of application services. We have described an extensible implementation of the resilient actor model which can be customised to provide different implementations of the stretch and retract operations. We have evaluated the resilient actor model by applying it to the construction of an ambient music player application. The resulting application can be partitioned at runtime by the end-user to run on multiple devices. The partitioned application is retractable and resilient to network failures.

In this paper we have described a conflict resolution mechanism of resilience strategies that is based on performing match between the set of strategies specified on the definition of the elastic binding and the set of strategies specified on the resilient actor. We are investigating on a resolution mechanism that is based on context information. For example an extension implementation can be provided such that a resilient strategy chosen is dependent on the user location, the CPU load or computational power of the device.

9. ACKNOWLEDGMENTS

This work was partially funded by SAFE-IS project in the context of the Research Foundation - Flanders (FWO), the MoVES project, and the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB).

10. REFERENCES

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] E. Bainomugisha. Resilient Service Partitioning for Pervasive Computing Services. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, September 2008.

[3] J. Bohn, V. Coroama, M. Langheinrich, F. Mattern, and M. Rohs. Disappearing computers everywhere – living in a world of smart everyday objects. In *Proc. of New Media, Technology and Everyday Life in Europe Conference*, London, UK, Apr. 2003.

[4] M. Dahm. Doorastha - a step towards distribution transparency. In *JIT, 2000. See http://www.inf.fu-berlin.de/ dahm/doorastha.*

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. m. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[6] D. J. Evans and W. U. N. Butt. Load balancing with network partitioning using host groups. *Parallel Computing*, 20(3):325–345, 1994.

[7] M. M. Fuad and M. J. Oudshoorn. Adjava - automatic distribution of java applications. In M. J. Oudshoorn, editor, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia, 2002. ACS.

[8] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 187–200, Berkeley, CA, USA, 1999. USENIX Association.

[9] G. Kiczales, M. J. Ashley, L. Rodriguez, A. Vahdat, and D. G. Bobrow. Metaobject protocols: Why we want them and what else they can do. In *MIT Press*, pages 101–118. Cambridge, MA, USA, 1993.

[10] N. King. Partitioning applications. In *DBMS and Internet Systems magazine, 1997. See http://www.dbmsmag.com/9705d13.html.*

[11] N. Liogkas, B. MacIntyre, E. D. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida. Automatic partitioning: Prototyping ubiquitous-computing applications. *IEEE Pervasive Computing*, 3(3):40–47, 2004.

[12] B. MacIntyre, E. D. Mynatt, S. Voida, K. M. Hansen, J. Tullio, and G. M. Corso. Support for multitasking and background awareness using interactive peripheral displays. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 41–50, New York, NY, USA, 2001. ACM.

[13] C. Mascolo, L. Capra, and W. Emmerich. Mobile computing middleware. In *In Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.

[14] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 43, Washington, DC, USA, 2002. IEEE Computer Society.

[15] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, Nov. 1997.

[16] R. L. Rivest and C. E. Leiserson. *Introduction to Algorithms.* McGraw-Hill, Inc., New York, NY, USA, 1990.

[17] I. Satoh. Dynamic federation of partitioned applications in ubiquitous computing environments. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 356, Washington, DC, USA, 2004. IEEE Computer Society.

[18] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17, 2001.

[19] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. *Lecture Notes in Computer Science*, 2072:236–255, 2001.

[20] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic Java application partitioning. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 178–204, London, UK, 2002. Springer-Verlag.

[21] T. Van Cutsem, S. Mostinckx, Gonzalez, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Computer Science Society*, pages 222–248, November 2007.

[22] A. Voida, R. E. Grinter, N. Ducheneaut, K. K. Edwards, and M. W. Newman. Listening in: practices surrounding itunes music sharing. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 191–200, New York, NY, USA, 2005. ACM Press.

[23] S. Voida, E. D. Mynatt, B. MacIntyre, and G. M. Corso. Integrating virtual and physical context to support knowledge workers. *IEEE Pervasive Computing*, 1(3):73–79, 2002.