Handling membership dynamicity in service composition for ubiquitous computing

Jeppe Brønsted
University of Aarhus
Computer Science Department
Abogade 34, 8200, Århus N
Denmark
Email: jb@daimi.au.dk

Abstract—In ubiquitous computing, as more and more devices are introduced into the environment, new applications are made possible that exploit device capabilities in new ways. Currently, however, there is a mismatch between the effort involved in implementing these applications and the benefits they provide. Furthermore, there is a risk that the user loses the understanding of the system and although this is usually not a problem during normal use, it can be problematic if a breakdown occurs. A proposed solution that handles these problems is to use a service oriented architecture and implement applications as composite services and let information about the structure of the composites be available to the user at runtime.

As long as the set of services that constitute the composite is static, traditional techniques can be used to specify the composite. But if the member set is dynamic it is problematic to specify which nodes partake in the composite and how they interact. In this paper we introduce mechanisms for handling membership dynamicity in service composition specifications. We demonstrate how an application scenario developed in cooperation with users can be implemented using the mechanisms and sketch how a decentralised interpretation can be realised.

I. INTRODUCTION

As more and more devices are introduced into our daily lives the vision of ubiquitous computing approaches realisation. Devices interact in new ways to provide services and let users experiment with devices and services and let information about the structure of the composites be available to the user at runtime.

Currently, services incorporating multiple devices are typically implemented by device manufacturers to add increased value into their products and thus the possible applications are limited by what devices a particular company manufactures. Even when manufacturers cooperate to make their devices communicate using, e.g., open standards it is hard to predict which services the users want and which devices to combine. Furthermore, if a given service involves a particular set of devices, the service will only be available to users having that particular set of devices. Consider the following geotagger example mentioned in [1]:

A user has a GPS device, a digital camera, a mobile phone with GRPS, and a home server. When pictures are taken they are automatically tagged with positional information from the GPS and uploaded to the server using the mobile phone.

In this application is not clear which manufacturer would implement the application since all the devices involved can be used for multiple purposes and should, e.g., the phone manufacturer decide to implement the service, it would only be of use to the subset of customers having the particular relevant device constellation.

Another issue with applications consisting of multiple devices interacting is that if a fault occurs, it can be hard to determine the cause. In the above example, the GPS device could run out of battery and unless the application have been designed with that particular contingency in mind, the user has to check each of the devices to resolve the situation. Another problem could be that the GPS device is configured to send messages in a format that is not understood by the application. In case this the user is probably even worse off because all devices will appear to be working correctly.

If device capabilities are exposed through service interfaces, anyone with access to the interfaces can, in principle, develop applications exploiting the devices and thus the manufacturers will not have to try to anticipate every combination a particular device might be involved in. To achieve maximum interoperability the manufacturers have to agree on a format for service specifications or at least make available the formats used. Whether the manufacturers are interested in this is not the topic of this paper but if a standard was agreed upon the utility of the devices would be enhanced.

Service composition have previously been used to implement applications for ubiquitous computing environments [2]. Service orientation alone will not solve the problem of creating services exploiting the particular device set a user has. If applications are implemented by developers using a service composition framework [3][5] it is hard for the user to control which services are used, how they are connected, and how they interact. Another option is to let the user try to specify the task he wants to solve in an abstract way and let the middleware determine how services should be composed [6][9]. This has the benefit that the user does not have to know which services are offered by the devices but only has to be able to specify the task to be solved. On the other hand, if a failure occurs, it can be hard for the user to find the error because the user’s understanding of the system is rooted in the abstract task specification. A third option is to let the user experiment with devices and services and...
manually compose the application [9–13]. This requires that the composition language or tool is sufficiently understandable and at the same time complex enough to make it possible to express useful applications. Previously, both centralised [9–11] and decentralised [12, 13] algorithms have been used to govern the flow of service invocations in the composite.

When users compose services into applications, it is problematic to handle composites with a varying number of members. An example could be a chat application that dynamically includes new devices as they arrive. If the application was expressed in source code, this would typically be handled by collections and specialised logic specifying how the composite evolves over time. Since source code is not understandable by end users in general, we do not consider this to be an option. To the best of our knowledge, none of the previous approaches deal with composites with varying member sets.

In this paper we investigate the problem of handling composites with dynamic membership. We present an extension of the PalCom assembly scripts that makes it possible to specify assemblies in which the member set varies over time and outline how a decentralised interpretation can be implemented. To evaluate the extensions we demonstrate how one of the PalCom prototype scenarios can be implemented using the mechanisms. The prototype scenario have been developed in cooperation with end users and represents realistic and relevant challenges.

The rest of the paper is structured as follows. In section II we briefly describe the PalCom architecture and the mechanism available for composing services into applications. In section III we present the prototype scenario and the problems involved in realising it using traditional techniques. In section IV we describe the extensions for handling membership dynamicity, describe how the prototype scenario can be realised using the extended assembly scripts, and outline how decentralised interpretation can be implemented. And, finally, in section V we conclude the paper and present future work.

II. THE PALCOM OPEN ARCHITECTURE

In the PalCom project [14] an open architecture [15] have been developed that supports users and application developers in making more understandable applications. Using the architecture and the runtime system, applications can be built by composing services through scripts [1]. The scripts can be defined by application developers or by users interacting with a service browser [15]. Services can be composed at runtime and the internal structure of the composite can be opened up and inspected in case of a failure or a misconfiguration. Being able to inspect the running system supports users in understanding how the application works and determine in which part of the system the failure is located. The composite can be altered at runtime by replacing services with alternates to resolve failures or to adapt the application to changing network conditions.

The design of the architecture and the scripting language has currently no support for composites where the set of members varies over time. Furthermore, a URN [17] for each of the involved services must be known when the composite is created. This implies that it is, e.g., not possible to specify a composite that will dynamically include new devices of a particular type as they arrive.

A. PalCom assemblies

In the PalCom architecture device functionality is encapsulated in services that can be remotely discovered and invoked. Each service has a set of commands which can be either in-going or out-going. In-going commands are similar to asynchronous methods with an optional number of parameters. They can be invoked from other services or by the user. Out-going commands makes it possible for the services to provide output. The output can be used as input for in-going commands or can be presented directly to the user. The output has an optional number of parameters. An example could be a service acting as an interface for a lamp. The service would have two in-going commands on and off and an out-going command state that is invoked every time the state of the lamp is changed. Services and commands are composed in assemblies described by assembly scripts where services and devices are declared along with description of which commands are connected. Variables that can hold state can also be declared.

The example given in the introduction can be implemented by the assembly script listed in figure 1. Lines 2–7 declares which devices take part in the assembly. Note that a unique name (URN) is given for each of the devices. Similarly, lines 8–13 declare the services in the assembly. In line 17–34 the flow of events through the assembly is described. E.g., lines 25–28 specify that when the out-going photo_taken command from the camera’s photo service is invoked, the tagger service’s tag_photo command on the mobile phone is invoked. In line 19 a variable is declared with a MIME-type [18] and in line 23 a value is assigned to it. The script in figure 1 can be created by using a text editor or by the user by interacting with a tool.

III. THE SITE STICKS SCENARIOS

Since the users are involved in creating the assembly scripts, an important design goal is that the script language is as simple as possible. One could easily use a general programming language to implement the assemblies, but this would defeat the goals of simplicity and understandability. The assembly
script language has to be as simple as possible while at the same time powerful enough to support relevant scenarios. In this section we present the PalCom scenario Site Sticks [19] where a composite service with a dynamic member set is required for realisation. The scenario has been developed in cooperation with landscape architects from Edinburgh.

When landscape architects try to visualise how a project will blend into the landscape at a building site, a typical approach is to place marker sticks that represent the shape of buildings, roads, gardens, etc. as outlined in the digital building plans. Looking at a site with hundreds of sticks (see e.g. figure 2) it can be hard to figure out which sticks represent a particular building. The challenge is to visualise the digital design combined with the physical reality.

In the site sticks scenario, each stick is equipped with an embedded system with wireless communication. When the stick is placed in the ground, the position and role in the design is registered in the stick using a GPS device and a PDA. Later, when the architect wishes to visualise a particular part of the design, he can select the part on the PDA and the corresponding sticks will light up with a distinct colour. The initial placement and the registration will not be dealt with in this paper.

From a conceptual point of view it’s natural to express the application that makes a subset of the sticks light up as a composite service consisting of the sticks and a PDA. However, the scripting language presented in section II-A provides only limited support for the description of the assembly. One limitation is that the script will be big because all devices and services have to be declared explicitly. Another is, that the assembly cannot be expanded to include more sticks without changing the script. In the following section we propose extensions to the assembly scripts that will make it possible to express the application described more naturally.

IV. HANDLING MEMBERSHIP DYNAMICITY

The extensions of the assembly scripts we propose can be divided into two parts: Selection is about selecting which devices should participate in the assembly and naming deals with how to represent the services and devices in the specification of the flow of events.

A. Selection

Given a set of nodes we need a method for selecting those that should be part of the assembly. In the unextended version of the scripts this is done using URNs but for assemblies with dynamic membership this is, as mentioned previously, not enough. Instead, we propose to use a simple wildcard pattern on the device URN so that a single line in the device declaration part of the script (lines 3–8 in figure 1) can declare multiple devices. Lines not including a wildcard character (‘*’) will be interpreted as before. As an example, the line:

\[ \text{stick} = \text{urn:palcom://stick*} \]

matches all devices with a URN with the prefix \text{urn:palcom://stick}. Hereby, all the sticks in the site sticks scenario can be declared in a single line.

While wildcards provides an easily understandable way of specifying a lot of devices, it has the drawback that semantic information has to be encoded in the device URN. In situations where devices with similar URNs have different service sets, it can be a problem only to include a subset of the devices in the assembly. Therefore, as an additional method for selection, we also use the information already present in the service declaration part of the script (lines 9–15 in figure 1) to exclude irrelevant devices. For example, line 13 in figure 1:

\[ \text{photo_db on server} = \text{/photo_db}; \]

specifies that the service with the name \text{/photo_db} on the device \text{server} will be used under the name \text{photo_db} in the rest of the script. We propose that this statement will also imply that all devices matching the line that declares the device \text{server} must also have a service with the name \text{/photo_db} to be allowed into the assembly.
B. Naming

With the extension mentioned above, each line in the device and service declaration parts of the script potentially declares multiple devices/services and associates a name. This name is used in the eventhandler part of the script to specify the flow of events. We extend the semantics of the eventhandler part so that the line:

```
when photo_tagged from tagger on mobile {
```

is used when the out-going command `photo_tagged` is invoked on *any* mobile device. In the case that mobile only denotes a single device, the interpretation is unaltered. Similarly, the interpretation of the invoke part of the eventhandler is changed so that the line:

```
send activate()
```

will invoke the `store_photo` in-going command on *all* the devices denoted by `server`. Again, if `server` only denotes a single device, the interpretation is unaltered.

To allow for local flow of events on devices declared with a wildcard, the device name can be prepended with the keyword "particular" in the eventhandler. Assume for example that the device is declared using a wildcard and the eventhandler includes the following lines:

```
when my-out-command from my-service-a
    on particular my-device {
    send my-in-command()
    to my-service-b on my-device;
}
```

Then, when the `my-out-command` is invoked on a particular device, the `my-in-command` will only be invoked on the same device.

The modifications above have the property that if no device is declared using a wildcard, then there is no modifications of the interpretation of the assembly scripts. This implies that the changes are backwards compatible.

C. Implementation of the Site Sticks scenario

We claim that the mechanisms for selection and naming described above can be used to implement the site sticks scenario in a simple and natural way.

Assume that each of the sticks is equipped with two services, blink and building. The blink service has a single in-going command `isPartOf` that will make the stick light up. The building service has an in-going command `isPartOf` and an out-going command `isPartOfTrue`. If `isPartOf` is invoked with a building identifier that the stick represents a part of, the `isPartOfTrue` will be invoked.

The PDA has a `stick-gui` service where the user can specify which building he wants to see. The `stick-gui` service has a single out-going command `select` that will be invoked with a building identifier when the user selects a building.

Given the services and devices described above, the site sticks scenario can be implemented by the script in figure 3.

```
assembly SiteSticks {
  this = global-service-name;
  devices {
    stick = urn:palcom://Stick;    
    pda = urn:palcom://PocketLoox;
  }
  services {
    blink on stick = /blink;    
    building on stick = /building;    
    stick-gui on pda = /stick-gui;
  }
  connections {
    ...
  }
  script {
    eventhandler {
      when isPartOfTrue from building
        stick = urn:palcom://Stick *;
        to blink on stick;
        send isPartOf(thisevent.building)
    }
    when select from stick-gui on pda {
      send store_photo(thisevent.Image)
      to photo_db on server;
    }
    when photo_tagged from tagger on mobile {
      isPartOfTrue
      from building
      to blink on stick;
    }
  }
}
```

Fig. 3. SiteSticks script

The stick represents part of the building specified, the stick will be told to blink in lines 21–25.

D. Decentralised interpretation

One way to interpret the assembly scripts is to let a central node handle the eventhandler part (e.g. lines 16–26 in figure 3) of the scripts. Every time an out-going command is invoked, a message is sent to the central node which then determines which in-going commands should be invoked. As long as the assemblies only include a few nodes all connected to the same network, centralised interpretation is a viable option. However, for assemblies like the site sticks assembly it is not a good idea because the building service’s invocation of the blink service (lines 21–25 in figure 3) would have to go through the central node. Therefore, we argue, it is necessary to interpret the scripts in a decentralised manner.

Decentralised interpretation can be implemented by distributing the script to all nodes in the assembly and let each node handle a part of the eventhandler script. When the member set of assemblies varies over time it is important that the distribution of the interpretation of the script is done in such a way that nodes leaving the network have as little influence as possible on the execution of the assembly. This means that if an assembly requires that a service S1 should invoke another service S2, the connection between the services should be handled by the nodes hosting S1 or S2. Hereby, that part of the assembly only fails if S1 or S2 fail.

We propose to divide the handling of the eventhandler part according to the following simple rule: An eventhandler clause is handled by the node originating the out-going command in the clause. As an example, the eventhandler clause in line 17–20 in figure 3 is handled by the pda node because the out-going command invocation comes from the pda. Similarly, lines 21–25 should be handled by each of the sticks.
Only nodes that participate in the assembly should receive the script. This requires that the distribution of the script itself depends upon the script. One way to accomplish this could be to flood the network in a n-hop radius with the script and let nodes not matching the device declarations ignore the script. Unlike its centralised counterpart, decentralised interpretation requires that all nodes are able to interpret parts of the scripts and this excludes a class of devices with very limited resources. An alternative to completely decentralised interpretation is to relieve some nodes of the interpretation responsibility. This, however, requires a method for selecting the nodes taking over the responsibility and is left as future work.

V. CONCLUSION

In this paper we investigated the problem of handling composite services with dynamic membership. We proposed extensions of the PalCom assembly scripts that make it possible to specify compositions with dynamic member sets and showed how a prototype scenario from the domain of landscape architects could be implemented using the proposed extensions. Finally, we outlined how the scripts can be interpreted in a decentralised manner.

We argue that the extended script language is simple and understandable. The idea behind the language is a basic case construct that specifies the flow of service invocations. A visual representation of the language might further support the user in understanding the scripts. Simplicity and understandability are relative to the user and therefore it is important also to make studies of end users working with the language. At present, none of the extensions have been implemented and therefore such studies are left as future work.

The suggested decentralised interpretation requires that all participating nodes are able to interpret the script and that can be a problem for very resource constrained devices. More powerful devices acting as proxies for the limited devices could be one way of handling this. In contrast to its centralised counterpart, the decentralised interpretation have the potential to scale better since no node have the sole responsibility for all communication in the assembly. Since no implementation exists at present, performance measurements are left as future work.

ACKNOWLEDGEMENTS

The author acknowledges the work done by the participants of the PalCom project to develop prototype scenarios and design and implement the PalCom open architecture and associated tools. The work presented in this paper has been supported by ISIS Katrinebjerg (http://www.isis.alexandra.dk).

REFERENCES


