Pervasive Service Composition in the Home Network

André Bottaro[1,2], Anne Gérodolle[1], Philippe Lalanda[2]

[1]France Telecom R&D
28 chemin du vieux Chêne
38243 Meylan cedex, France
{firstname.lastname}@francetelecom.com

[2]LSR-IMAG, 220 rue de la Chimie
Domaine Universitaire, BP 53
38041 Grenoble, Cedex 9, France
Philippe.Lalanda@imag.fr

Abstract

The home environment becomes ready to host distributed devices dynamically adapting to service availability and reacting to user location and user activity. Sensors, high definition rendering systems, home gateways, wired and wireless controllable equipments are now available. Many protocols enable connectivity and interaction between devices. However, challenges remain: protocol heterogeneity, interface fragmentation and device composition static aspect make self-organization and dynamic reconfiguration hardly achievable. This paper describes attractive scenarios at home which lead to the definition of the pervasive service composition requirements. A software architecture facing the mentioned challenges is proposed over OSGi. It first enables developers to implement distributed plug-n-play applications like a local one. It also delivers a service-oriented middleware allowing spontaneous distributed service composition to occur at runtime.

Keywords: Home networks, Pervasive Computing, Service Orientation, Distributed Applications, OSGi.

1. Introduction

Mark Weiser [13] brought a great paradox more than a decade ago: Computers have to be numerous and interconnected in order to disappear from the user's awareness. The purpose of pervasive computing is to realize this vision of increasingly ubiquitous network-enable devices. Specifically, it aims at filling our environment with communication devices in order to assist us in our daily activities without our explicit intervention. Indeed, in the near future, human beings will engage interactions with a number of smart devices, faded into the environment, without being aware of their location or their precise nature and without going through complex, specific interfaces. They will simply express their needs or desires and the environment and the objects in it will configure themselves autonomously.

It seems that we are presently in a transitioning phase. The number and variety of smart communication devices are exploding: PDAs, smartphones, set-top-boxes, cameras, and electronic appliances can be found in many houses today. As envisioned by Moore's law, these devices are getting cheaper, smaller and are pervading every aspect of our life. The problem is that this invasion is chaotic: devices use a number of communication protocols and are rarely interoperable. Today, more than 50 candidate protocols, working groups and standard specifications for home networking already exist (see www.caba.org for an updated list). As a consequence, building consistent pervasive applications based on network enabled devices that spontaneously enter and leave the network turns to be a real challenge. In order to allow the massive deployment of pervasive services within houses, we believe that significant progress is needed along several dimensions. New architectures and techniques are actually needed in order to seamlessly integrate heterogeneous and changing devices and networks.

Service-oriented computing (SOC) appears as a very promising paradigm to deal with the inherent complexity and dynamism of pervasive computing. Loosely coupled service architectures provide the level of flexibility required to build pervasive applications. However, most available solutions focus on the technology allowing developers to publish and compose services and to make them communicate. In addition, these solutions are usually not interoperable. Also, programming dynamic service composition is a critical task that remains painful and error-prone for several reasons. First, the programmer has to deal explicitly with low level communication protocols and, once again, such protocols are numerous in the house context. Second, dynamic service availability is not addressed by current distributed middleware and, then, has to be directly managed by the programmers. Tools and techniques are necessary to hide the heterogeneity of the service protocols and to provide the developers with higher level programming primitives.

The purpose of this paper is twofold. First, it aims at presenting an open computing infrastructure for the development of home pervasive services. Then, it presents a declarative model to automate the mutual discovery and binding of services running on distributed and heterogeneous network nodes. A middleware hiding the multiplicity of service discovery and communication protocols is built according to this model.

This work is carried out within the ANSO[1] project which brings together major European actors interested in the development of pervasive home services, including telecommunication operators (France Telecom), video companies (Thomson) and home control solution providers (Schneider Electric). This work is demonstrated on a home application developed by France Telecom [1].

The paper is organized as follows. First, the targeted open computing infrastructure for homes is presented and the associated requirements as well. Then, in section 3, our computing platform hiding service and network heterogeneity is detailed. Section 4 describes a follow-up application and details the way it has been implemented. Section 5 compares our approach to related work. Finally, section 6 concludes this paper and provides perspectives.

2. Service oriented architecture

2.1 A platform centric vision

As previously introduced, we are working on an open service-oriented computing architecture for the home. This architecture comprises network-enabled devices and computing platforms connected through field buses.

Devices are pervasive elements integrated in the houses (screens, loud speakers, controllable shutters or heaters for instance) providing basic services to sense and act upon the environment. Many devices are today implemented as service providers and requesters using technologies like Jini or UPnP for instance – see www.jini.org and www.upnp.org. The smaller devices, in terms of computing resources, communicate through *ad-hoc* wired or wireless protocols. Proxies have to be created on a service platform to make them visible as services. Service oriented technologies are however numerous and often not compatible, which raises thorny integration issues.

Computing platforms often play the role of network gateways. Such gateways are already present in many houses (telecommunication Internet gateways, TV-connected set-top-boxes or utility service gateways) and it is likely that future homes will host several, heterogeneous such computing platforms. Gateways provide the resources needed to run higher level services making use of the connected devices. The purpose is to coordinate multiple devices and to ensure natural, sometimes invisible, interactions with the users. These interactions are obviously directed by high level goals that can be set by the user or by Internet services to which the user has subscribed. Some electronic devices can be exclusively connected to a given gateway. For instance, a home control gateway can coordinate the behaviors of specific devices like shutters or heaters. This approach meets proven market constraints which state that most manufacturers won't provide access to their devices to any operator or device (for safety, security and obviously business considerations).

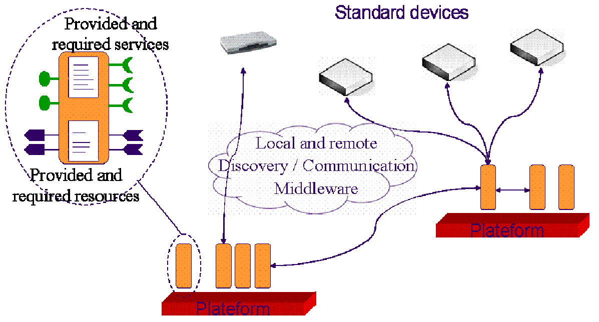

Provided and required services
Standard devices
Local and remote Discovery / Communication Middleware
Provided and required resources
Plateform
Plateform

Figure 1. Platform-centric architecture

Devices are dynamic, based on various protocols and provide information in heterogeneous ways. We believe that, in order to realize the vision of sophisticated pervasive services in the home, it is necessary to build an open computing framework to run user-related services on home gateways. Such framework should abstract the developer from low level details related, in particular, to service and communication technologies.

2.2 Requirements

A high level, service-oriented computing framework has to meet stringent requirements in order to be effective in the context of home gateways. In particular, it has to provide facilities regarding service description and bindings. In particular:

- **Automatic binding.** Service binding has to be automated according to required service availability.
- **Uniform treatment of services.** Remote and local services must be represented the same way on the service platform. In particular, they have to be accessed through the same mechanisms hiding network heterogeneity and specific service technologies.
- **Automatic service adaptation**: The service platform has to deal with the issue of interface fragmentation. In most current platforms, adapter

[1] ANSO is partially supported by the French Ministry of Industry under the European ITEA program.

generation is a tedious task that is mostly left to developers.

- **Dynamic service ranking.** Mechanisms should be provided to dynamically evaluate the interest of a service in a composition. For instance, context models should be included in the platforms.
- **Service continuity.** As soon as a potentially interesting service obtains a better ranking than a service involved in the current composition, the composer may take the decision to rebind to the better ranked service. Service continuity has to be maintained against service re-composition.

These requirements are successively addressed in the following section.

3 Service-oriented computing framework

3.1 OSGi and service dynamic availability

In order to meet the requirements presented here before, we have been working on top of the OSGi service platform standard (see www.osgi.org). OSGi is an open service platform defining a minimal component model, a small framework for administering components, and a set of standard services. Components are packaged in bundles, which are the deployment units in the OSGi model. The framework also defines mechanisms that facilitate the dynamic installation, start, stop, update, and removal of bundles.

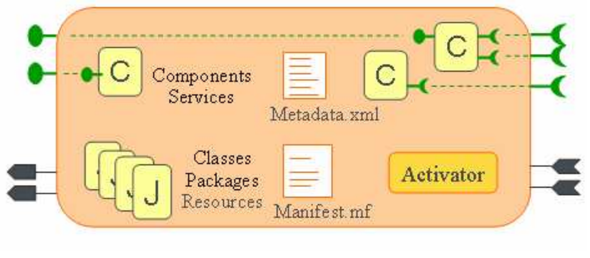


Figure 2Resources and components in an OSGi bundle

OSGi specifies a way to automate service binding between components under the Declarative Services specification which is derived from Service Binder project [4]. This specification allows developers to specify component dependencies in terms of provided and requested services in an XML file. The specification is implemented as a lightweight container in a bundle (Service Component Runtime – SCR) which manages bindings and service registrations. SCR attaches a component manager to every component at bundle starting time. This component manager uses OSGi service requests and OSGi service listeners to resolve service dependencies at runtime. When requested services are bounded, provided services are registered and the component is declared activated. The loss of a service binding leads to either a new service binding with another service provider or to the component deactivation. Service rebinding is kept simple.

OSGi brings the technical foundations to develop dynamic and autonomic home services. We believe nevertheless that it should be extended in order to facilitate the developers' task along, at least, two dimensions:

- the integration of various distributed service technologies,
- the specification of dynamic service dependencies.

Regarding the first point, although service technologies like Jini (www.jini.org) or UPnP (www.upnp.org) are handled by the OSGi specification, using them is not as transparent as it could be. For instance, the UPnP Device Service Specification provides APIs for interoperability between OSGi services and UPnP devices. Unfortunately, these APIs are specific to UPnP protocols and come in addition to OSGi core APIs to discover and bind services.

Regarding the second point, we believe that service composition above OSGi suffers from limitations in the Declarative Services model. On the service provider side, the Declarative Services specification enables the declaration of descriptive properties qualifying the provided features of the service. However, this declaration is static whereas services provided on dynamic networks are versatile. On the service requester side, the Declarative Services specification enables bundle developers to declare a service filter which shows the following limitations: The filter declaration is static and its expressivity is poor.

We have thus proposed several extensions to OSGi (see RFP 72: Extended Mapping for UPnP Discovery Transparency for instance) in order to support (i) the seamless integration of different heterogeneous and distributed service technologies and (ii) service description contextual dynamicity on the provider side and service ranking dynamicity on the composer side.

These extensions are presented here after.

3.2 Discovery drivers and protocol heterogeneity

In order to deal with heterogeneous and distributed service technologies, several issues have to be tackled:

- A common service description syntax has to be defined (and the mappings between this syntax and existing protocols have to be implemented).

- Common service discovery mechanisms have to be defined (and the mappings between these mechanisms and existing Service Discovery Protocols have to be implemented).
- Tedious stub code has to be statically or dynamically generated.
- Generic service events have to be defined in order to maintain dynamics.

We have tackled these issues through the development of bridges between technologies called Discovery Base Drivers and the generation of specific stubs called Refining Drivers. A Discovery Base Driver leverages its specific protocol stack to react to network events. Every available device is reified on the development framework as a programming language proxy whose dependency to the used protocols is hidden to developers. A Refining Driver turns the proxy into a more specific protocol-independent proxy.

3.2.1 Discovery Base Drivers

A particularity of the OSGi framework, which is exploited for our purposes, is the inclusion of the Service-Oriented pattern into the programming model. The local service orientation mirrors the mechanisms of the available plug-n-play protocol middlewares (see the list in related work).

The OSGi framework supports a service registry with an associated event notification system. Service providers are local Java objects which publish their interface and describe themselves with a set of properties. The Service requesters are notified by service registrations, modifications and unregistrations by the underlying event notification system. Protocol middlewares are based on the same logical mechanisms. Mechanisms are local on the OSGi platform whereas they are distributed in protocol middlewares.

Discovery Base drivers react to network events and dynamically populate OSGi registry with local Java objects which represent remote services. Network events correspond to device publications and device departures. These events are reified on the platform through OSGi local event system.

The designed model of a discovery base driver defines low-level transparency in enabling the developer to import devices and export OSGi services without making them aware of the details of the protocols being involved. Base drivers populate the service registry with generic proxies whose API enables developers to send remote calls on the attached devices. However, the developer has to know this specific API mapping the protocol-specific syntactic description to programming language in order to send remote calls (see code example in the next section). An additional mapping between technology descriptions and OSGi service model could lead to specific stub generation and give higher-level transparency.

3.2.2 Refining Drivers

Specialized drivers are meant to provide the high-level programming language transparency needed by developers. Once again, these drivers benefit from the service orientation of the platform framework. Specialized drivers react to the presence of generic proxies in the service registry in instantiating proxies implementing specific interfaces and publishing these specific objects in the service registry.

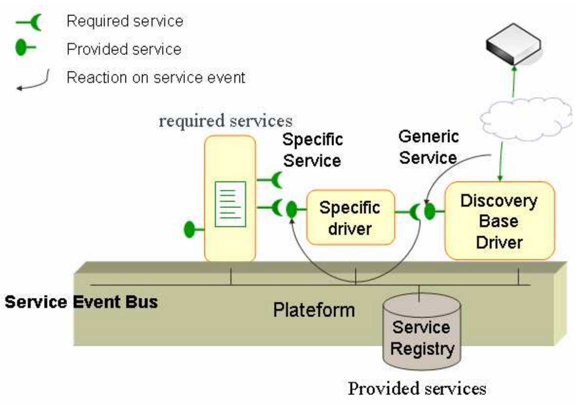


Figure 3Discovery and refining drivers

These specialized drivers are statically generated. For example, the UPnP XML description of a UPnP action called Play (operation to be called on a UPnP service) is depicted below. This action is taken from the description of a standard UPnP Media Renderer Device embedding an AV Transport Service:

```
<action>
  <name>Play</name>
  <argumentList>
    <argument>
      <name>InstanceID</name>
      <direction>in</direction>
      <relatedStateVariable>
        A_ARG_TYPE_InstanceID
      </relatedStateVariable>
    </argument>
    <argument>
      <name>Speed</name>
      <direction>in</direction>
      <relatedStateVariable>
        TransportPlaySpeed
      </relatedStateVariable>
    </argument>
  </argumentList>
```

```
</action>
```

The use of a generic driver on the OSGi Java programming model:

```
Dictionary dictionary = new Hastable();
Dictionary.put("InstanceID", "0");
Dictionary.put("Speed", "1");
upnpDevice.getDescription(null).
  getService("AVTransportService").
    getAction("play").invoke(dictionary);
```

The use of a specific interface generated by our UPnP Driver generator:

```
mediaRenderer.getAVTransportService().play(0,1);
```

3.2.3 Transparency issues

Masking differences between local service access and remote service access is a feature which is provided by some programming languages and associated technologies, e.g. Java RMI. However, the design of the overall application has to be thought with distribution awareness. At design time, the developer knows which the remote interfaces are in order not to make important mistakes. Some distribution issues can not be masked [10]: Latency, communication failure, distinct concurrency management, different memory access.

Potentially distributed interfaces must be clearly identified and their design has to address the intrinsic problems that are raised. Stub implementation (or generation) has to deal with these relevant problems and client implementation must also be performed with distribution awareness.

Finally, it becomes clear that handling distribution leads to tackling numerous tedious tasks. These are explicated in [2]. Thanks to this vision where every available remote service is automatically reified as a local service provider, protocol heterogeneity issue is turned into the interface fragmentation issue.

3.3 Reactive adapters and service adaptation

Interface fragmentation is a well-known limit of Service Oriented Architecture. Loose coupling in SOA is based on service interface contract between service providers and service requesters. However, the same semantic purposes may be specified with distinct syntactic interfaces by distinct actors on the service market.

Standardization is a mean to agree on interaction models and precise syntactic interfaces. The issue in the Home Network is that many standardization committees rely on distinct protocols and define interfaces for the protocols they are attached to. For instance, the UPnP Forum is not only maintaining a set of protocols for plug-n-play generic mechanisms but is also specifying device and service description along with interaction models. Unfortunately, different committees do not usually share the same interaction models. For instance a LPR printer discovered with SLP discovery protocol may have an interaction model distinct from the printer description specified by the UPnP Forum.

Service adaptation is necessary to compose such semantically similar services. Service adaptation consists in providing adaptors, or proxies, to service clients. Adaptors implement the required interface and translate service calls to the service providers implementing a syntactically distinct interface.

Service adaptation can be done automatically in some specific cases. Most of the time, however, it is performed statically by developers. Several interesting works have been published in the literature of this domain. Adaptation is generically done through two successive phases:

- Service matching: it consists in indicating the services that match, then the methods that match, and finally operations on arguments in order to make the semantic meaning of used service output parameters and implemented service input parameters match. The required service may be implemented in using one or a series of several provided services to be successively called. Service matching may be statically done by developers [11] or automatically inferred thanks to ontological reasoning. The latter is a complex issue we have not tried to resolve yet.
- Service adapter generation: Given a service matching definition, a relevant adapter is easy to develop. This task may be done dynamically through the use of Java reflection [11] or through the use of byte code generation. The first approach is simpler to conceive but the latter gives faster adaptive feature at runtime.

We have built a repository storing mapping definitions for matching services. A reactive driver generator linked to this repository is running on the platform. This generator is listening to service events. Whenever a registered service provides a service type which is known as an input of a service adapting definition in the repository, the generator generates the byte code of the adapter, instantiates and registers the adapting object (Figure 4). The adapting object implements a required service type and uses the provided one to offer the required features. It is registered with the required service type in the service registry.

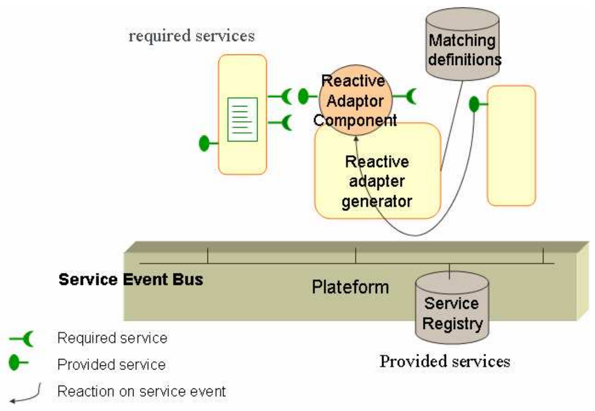


Figure 4Reactive Service adaptation

3.4 Dynamic contextual service ranking

Provided services are linked to contextual properties like device location, quality features, current available capacities, etc. Declarative Services only define static properties declaration at implementation time.

We have extended the OSGi model by adding a method declaration returning the dynamic properties to be attached to service registration (see *onregister* tag with *property-method* attribute in XML description below) in XML component metadata description. The extended component manager calls this method at service registration time in order to attach properties that can be defined by the component at runtime. A hook is also added for the component to warn the component manager to refresh service registration every time a significant contextual change occurs.

```
<!-- A component providing a service -->
<component name="service.implementation">
<implementation class="pack.PlayerImpl"/>
 <service>
    <provide interface="api.Player"/>
    <onregister
        property-method="getDynamicProperties"
    />
 </service>
</component>
```

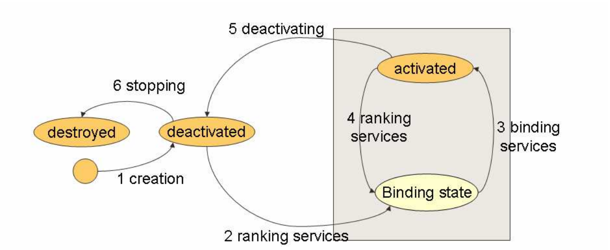
On the service requester side, the Declarative Services model enables bundle developers to declare a service filter. This approach suffers from the following limitations: (i) The declaration is static. (ii) The filter is expressed as an LDAP expression which only accepts simple comparative operators and simple types. (iii) No service sorting operation can be expressed.

We overcome these limitations with the addition of a method declaration (see *sort-method* attribute in XML description below) in XML component requirements description. The method is called by the extended component manager at service dependency resolution time. It takes the available services that match the declared static filter as an input argument and returns the sorted list. This way, the sorting algorithm is expressed by the developer of the service requester with the expressivity of Java programming language. A hook method is also given in order for the component to force the component manager to re-evaluate this ranking method.

```
<!-- A component requiring a service -->
<component name="service.requester">
<implementation class="pack.ControlPoint"/>
   <reference name="PLAYER"
     interface="api.Player"
     target="(hifi=true)"
     cardinality="1..n"
     policy="dynamic"
     bind="bindPlayer"
     unbind="unbindPlayer"
     sort-method="sortPlayers"/>
</component>
```

Whenever this ranking method is called, the Component Manager takes the result into account and rebinds the requester to the best service requester if it has changed. The call to the ranking method is triggered

- by the container reacting to service events, i.e. whenever a service is registered (or modified) and fulfils the service filter and whenever a current bound service is modified or is unregistered.
- by the service requester when communication with the provider is unsatisfactory or when the service requester preferences have changed (user location and activity changed for example).



1. Creation
2. Ranking available filtered services
3. (Un,Re-) binding best services
4. Service ranking on following events: Bound service leaving or new service registered or requirements changed
5. Deactivation on following events: Mandatory service missing leading or stopping.
6. Stopping

Figure 5Component Lifecycle

Dynamic adaptation to contextual changes – dynamic service availability and user activity – leads to service re-composition. Ranking activity leads to a temporary component state called "Binding State" which is added to

Declarative Services component lifecycle. The extended lifecycle is depicted in Figure 5. In our architecture, re-composition is triggered and decided at the fine-grain component level.

3.5 Service continuity

Applications may naturally be stateful since its behaviour depends on a stored evolving context. Service rebinding may be harmful for the application if a part of the application state is stored on service provider side (service provider configuration, session ongoing results). In order to achieve service continuity, service requesters may avoid the loss of relevant information in avoiding service rebinding or in organizing a session state transfer. Service requesters avoid some service rebinding situations with a ranking algorithm which affects the best ranking score to the provided service already in use.

We deal with state transfer at the application level. It may be automatically managed in some specific applications. The service (or session) state is the result of successive internal or external operations called by the service. Storing the service state amounts to storing all the successive calls and responses and catching possible external artefacts the service rendered. Transferring it to the new service can be achieved in some cases by making the same sequence of calls on the new service and getting the new service take into account the external artefacts. The latter is not always achievable.

In Service-Oriented Architecture, services are recommended to be stateless according to the REST Architectural styles [5]. This is a reasonable approach in wide area networks where services are meant to handle several parallel sessions. Lying service implementations use common techniques to queue parallel calls and answer several calls at (approximately) the same time. However, the stateless predicate of the Rest architecture styles does not hide the hard task of session management on the client side. Moreover, this management infers that the relevant part of the stored session state may be passed at every service call.

In home networks, some services – e.g. a media content server – may act as a logical service on wide area network and serve many clients at the same time. On the contrary, many services are attached to devices and can not be used by many clients at the same time. For instance, the service provided by a washing machine is dedicated to successive use by home users. Thus, the washing machine may be considered as a service with a stateful behaviour.

In the audio-video follow-me application presented here after, media rendering devices provide stateful services. Re-composition occurs whenever a rendering device appears to be more adequate to the user location and activity. At re-composition time, the media composer attached to the user is responsible for rebinding the application to the *new* rendering device. At rebinding time, the media composer not only has to start another session on the new service as it was started on the previous one – setting screen and sound configuration, indicating the content to be played, playing the content – but also has to indicate the current position of the media playing session. The latter is specific to the media application and could not be inferred without application knowledge.

In this example, service continuity is achievable. At service rebinding time, the media content has to be played on the next rendering device before being stopped on the previous one. Superposition of the two streams requires that the streaming server allows this feature (multicast stream or multiple unicast streams).

4. Application

4.1 Background

We are currently working on an audio-video system which follows the user and adapt to his activity in the home. The purpose of the system is to provide appropriate information on the adequate rendering device in the user's vicinity. A screen is in the vicinity of the user if the user is located in a defined zone in front of it. Levels of adequacy may be defined according to user distance to the screen, the orientation of the screen, user visual capacity and moreover, user activity.

When the system is activated and media content is asked to be played, the system computes the location of the user thanks to a location management system based on sensors distributed in the home. A list of the rendering devices available in the vicinity of the user is then computed. The system selects the best one using potentially sophisticated rules about the user attention demanded by the device, the current needs of the user, the presence of other persons and their activities, etc.

Let us take two scenarios to illustrate this application. First, when the user is watching an action movie for the first time and he is going to bed in the middle of the movie, the system infers that he would like to keep watching the film on the available screen and loud speakers with the best quality. If a screen and loud speakers are available in the room, the movie is displayed on these devices. Otherwise, the movie is stopped when the user leaves the living-room. A timeframe during which both rendering systems are simultaneously playing the movie can be defined.

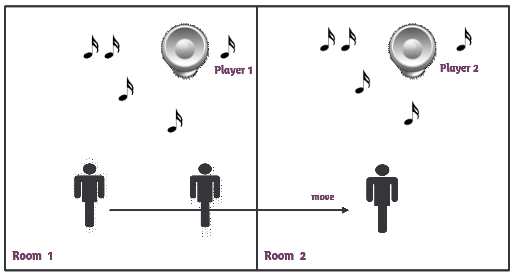


Figure 6 Audio stream follow-me application

A second example can include a change of activity. The phone is ringing when the user is watching a video on the high-definition screen in the living-room. If the user picks up the phone, the loud speakers in the living room may be chosen by the system to render the phone call. If the phone is sophisticated, the screen may render an image or the video of the remote person who is talking on the phone. After the end of the call, the source of the displayed video is again the movie player.

4.2 Realization

Our framework has been applied to build the mentioned contextual audio streaming follow-me application. Distributed entities are depicted in Figure 7. Streaming source may be an Internet site or a UPnP Media Server. Players are UPnP Media Renderers and devices with ad hoc SLP interfaces. Yellow pages listing all the known streaming resource may be a UPnP Media Server. A smart service composer using the extended SCR is available as a remote control on a PDA, the control point is on a Home Gateway and the location server is installed on a PC.

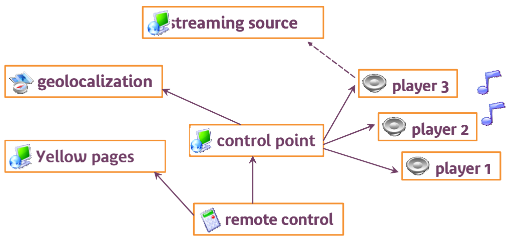


Figure 7 Architecture of the follow-me application. The plain arrows represent established bindings.

The developer of the control point defines three methods (*bindPlayer, unbindPlayer, sortPlayers*). As the user moves across rooms, the extended SCR is notified by the application when the user moves or begins a well-defined activity. Discovery drivers are service events source. A service event is sent whenever a service is registered, modified or unregistered. Any time such an event occurs and is related to the current service composition, it automatically calls the sorting method provided by the service requester and calls the binding method with the most adequate available player and then calls the unbinding method to unbind the previous player.

5. Related work

Several service–oriented middleware technologies are standardized and available today: UPnP, DPWS [9], Bonjour (www.apple.com/macosx/features/bonjour), IGRS (www.igrs.org), Jini, SLP [8], Web Services [7], CORBA, etc. These standards define protocols and mechanisms for service discovery and interaction. All these technologies may be used as distributed plug-n-play protocols in our architecture.

Indiss [3] enables interoperability between legacy service clients and providers developed with specific protocols. The Indiss architecture acts as a bridge between topologic networks and between discovery technologies. The approach is fundamentally different in that Indiss acts directly on network communication whereas our one adapts the service platform according to the network.

Our approach is close to REMMOC's [6], in that it allows protocol adapters to be plugged in at deployment time and hides protocol heterogeneity to SOA developers. However, service binding automation is not addressed by the REMMOC project.

A. Ketfi [11] promotes a semi-automatic service adaptation through the use of generic adapters. The matching algorithm is a graphic interface which enables a skilled user to indicate which services and methods are to be matched at runtime. Dynamicity relies on java reflection used to link provided to required methods. Java reflection slows down the following communication. Byte code generation offers better performance.

A follow-me application is depicted in [12]. It describes how to implement a simple location-aware audio application with Java Media Framework and Jini.

6. Conclusion

Our framework, built on top of OSGi, allows developers to focus on functional code. It relieves application developers from the tedious and error-prone programming tasks related to complex issues: service heterogeneity, distribution and dynamic service availability.

Based on the service-oriented computing paradigm, the architecture also offers an extensible framework where new communication or new service discovery protocols can be added by writing the corresponding discovery base drivers and refined driver generators. Declarative Services, Service Distribution and Service Adaptation, are dealt as separate concerns. This separation of concerns relies on the extensive use of the service registry.

Thanks to this architecture, spontaneous composition occurs at runtime. Every component declares its provided and required services. The framework maintains the list of available requested services, maintains the registration of provided services with their current attached contextual properties, and links the adequate components according to dynamic contextual requirements. Local or remote service providers are bound in an undistinguishable way.

Future work will tackle the issue of representing the ranking operators and mechanisms in the service composition description. Composition contextual behaviour may be dealt as a non-functional need.

References

1 André Bottaro, Johann Bourcier, Clément Escoffier, Didier Donsez, Philippe Lalanda, "A Multi-Protocol Service-Oriented Platform for Home Control Applications", Demonstration at IEEE Consumer Communications and Networking Conference (CCNC 2007), Las Vegas, 2007

2 André Bottaro, Anne Gérodolle, Philippe Lalanda, "Pervasive Spontaneous Composition", First IEEE International Workshop on Service Integration in Pervasive Environments, Lyon, France, 2006.

3 Yérom-David Bromberg, Valérie Issarny, "Indiss: Interoperable Discovery System for Networked Services", 6th International Middleware Conference, Grenoble, France, 2005.

4 Humberto Cervantes, Richard S. Hall, “Automating Service Dependency Management in a Service-Oriented Component Model”, Workshop on Component Based Software Engineering, May 2003

5 Roy Thomas Fielding, PhD thesis, University of California, Irvine, 2000

6 Paul Grace, Gordon S. Blair, Sam Samuel, "ReMMoC, "A Reflective Middleware to Support Mobile Client Interoperability", Proceedings of International Symposium on Distributed Objects and Applications, November 2003

7 Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, Peter Brittenham, Yuichi Nakamura, Paul Fremantle, Dieter Koenig, Claudia Zentner, “Building Web Services with Java”, Sams Publishing, 2004

8 Erik Guttman, Charles Perkins, John Veizades, Michael Day, “Service Location Protocol, Version 2”, RFC 2608, June 1999

9 François Jammes, Antoine Mensch, Harm Smit, “Service-oriented device communications using the devices profile for web services”, Proc. 3rd international workshop on Middleware for pervasive and ad-hoc computing, Grenoble, France, Nov. 2005

10 Samuel C. Kendall, Jim Waldo, Ann Wollrath, Geoff Wyant, “A Note on Distributed Computing”, Sun Microsystems Technical Reports, November 1994

11 Abdelmadjid Ketfi, Noureddine Belkhatir," Dynamic Interface Adaptability in Service Oriented Software", 8th International Workshop on Component-Oriented Programming (WCOP), Darmstadt, Germany, July 2003.

12 Robin Kirk, Jan Newmarch, "A Location-aware, Service-based Audio System", IEEE Consumer Communications and Networking Conference, 2005

13 Mark Weiser, “The computer for the 21st century”, Scientific American, 265(3):66-75, September 1991