A Pattern-based Adaptation for Abstract Applications in Pervasive Environments

Imen Ben Lahmar*, Djamel Belaïd* and Hamid Mukhtar†
*Institut Telecom; Telecom SudParis, CNRS UMR SAMOVAR, Evry, France
Email: {imen.ben_lahmar, djamel.belaid}@it-sudparis.eu
†National University of Sciences and Technology, Islamabad, Pakistan
Email: hamid.mukhtar@seecs.edu.pk

***Abstract*—Using service-oriented architecture, applications can be defined as an assembly of abstract components that are mapped to a concrete level to fulfill their executions. However, several problems may be detected during their mapping as well as during their executions, which prevent them to be executed successfully. Thus, there is a need to adapt them according to the given contexts. In this article, we present some situational contexts that may trigger the adaptation of applications at init time or during their execution. Upon detection of certain changes in context, the applications are adapted accordingly. For this goal, we propose a set of adaptation patterns that provide an extra-functional behavior with respect to the functional behavior of the applications. These patterns are injected into abstract applications if a relevant context is sensed to ensure their mapping as well as their execution.**

***Keywords*-Adaptation patterns, mismatches, abstract applications, component model.**

I. Introduction

The proliferation of small devices and the increase in number of services created by various vendors for such devices have made Service-Oriented Architecture (SOA) a primary choice for mobile software developers. One particular approach for developing SOA-based applications is to use component-based application development.

Using this approach, an application is defined as an assembly of loosely-coupled components, requiring services from and providing services to each other. Complex applications can be crafted using an arbitrarily large number of software components. Specifically, when developing business applications using SOA, it becomes inevitable to implement the business functionality by a mix of self-contained, reusable and loosely connected components.

In such approach, it is possible to represent an application by an assembly of abstract components (i.e., services), which leads to automatic selection of services across various devices in the environment. At the time of execution, the services are mapped to concrete components, distributed across various devices.

To illustrate our point of view, let's consider a video player application that provides the functionality of displaying video to the user. The user is also able to control the playback of the application. The application is represented by an assembly of abstract components, which describe only the services required or provided by the application namely, controlling, decoding and displaying video. The application has to be mapped to the concrete components to achieve its realization.

The complexities involved in designing and realizing such applications have been identified and addressed by many previous approaches [2] [6] [15].

While the existing approaches may assume that a mapping from abstract to concrete application can be done effortlessly, many problems may arise at init time that prevent it to be achieved successfully for example the heterogeneity of interfaces of connections between devices. Furthermore, applications in pervasive environments are challenged by the dynamism of their execution environment due to, e.g., user and device mobility, which make them subjects to unforeseen failures.

These problems imply mismatches between the abstract application and the concrete level occurring at init time or during the execution of the application. That is, the application cannot be executed in the given context or in the new context if it changes. Therefore, applications have to be adapted in order to carry out their mapping, and subsequently, their execution.

In literature, we distinguish two main adaptive techniques, namely parametric and compositional mechanisms to adapt applications in pervasive environment [14]. The parametrization techniques aim at adjusting internal or global parameters, while the compositional adaptations allow the replacement of components implementations or the modification of the applications structure.

In our work, we are interested in the last category, i.e., the structural adaptation, to overcome the mismatches between the abstract application and the concrete level with respect to the functional behavior of the application.

Therefore, in our previous work [4], we have proposed to transform an abstract application to another one by injecting adaptation patterns into the abstract application, which provide an extra-functional behavior allowing the mapping and the execution of the application. To facilitate the description of the adaptation patterns, we have defined a generic adapter template that encapsulates the main features of an adapter.

In this paper, we make the following novel contributions, some of which extend our previous work [4]: 1) we identify some situational contexts according to which the application can be adapted and 2) we propose a set of adaptation patterns that define the adaptation behavior of an application given a certain context.

The rest of the article has been organized as follows. Section II presents the adaptation context that may trigger the adaptation of abstract applications. Section III describes the principle of our structural adaptation approach and the proposed set of adaptation patterns. In Section IV, we present an example scenario through which we give an architectural description of applications and patterns. In Section V, we present some implementation details, whereas, in Section VI, we provide an overview of existing related approaches as well as their limitations. Finally, Section VII concludes this article with some future directions.

II. Adaptation Contexts

A. Classification of Mismatches

A generalized notion of context has been proposed in [1] as *any information that can be used to characterize the situation of an entity (person, location, object, etc.)*. We consider adaptation context as any piece of information that may trigger the adaptation of the application.

We are interested in contexts that represent the mismatches between the abstract descriptions of applications and the concrete level. These mismatches imply that the current abstract description could not be realized in the given context, or in the new context, if it has changed.

We have classified the mismatches level, where they occur, into three categories: inter-components, intra-device and intra-devices mismatches (see Figure 1). This categorization covers not only the software mismatches but also the network and hardware problems.

At the inter-components level, we consider the mismatches that may arise at init time due to the non-satisfaction of the non-functional requirements of the components. These latter are system requirements which are not of a functional nature, but contribute decisively to the applicability of the system [9] like security, reliability, performance, etc. Thus, if they are not ensured at the concrete level, this may prevent the abstract application to be well mapped.

At this level, the mismatches may be also related to the heterogeneity of signatures, protocols or semantic [3]. In the present work, we do not focus on these mismatches as it has been previously studied [12] [13] [19]. However, it is possible to resolve them using our approach.

It is also possible to detect mismatches at intra-device level. These mismatches denote that the desired characteristics of devices, as specified by a service or a user, are not satisfied due to their reduced capacities like using a lower battery power, a slower CPU, etc. Thus, there is a need to adapt the abstract application to consider these requirements.

In case of a distributed environment, there is also a need to consider the mismatches occurring at inter-devices level. These mismatches may be related to the use of heterogeneous interfaces of connection, a lower bandwidth, etc. Thus, they have an impact in the communication between devices.

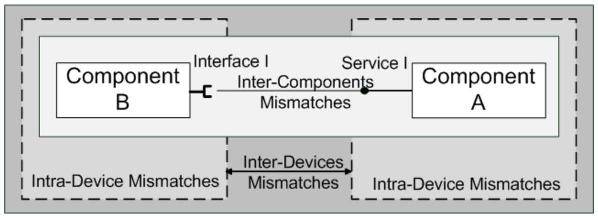

Fig. 1. Categorization of Mismatches levels

B. Detection of Mismatches

The application resolution and execution is ensured by our Middleware for Monitoring and Adaptation in heterogeneous environments (MonAdapter). The architectural design of MonAdapter is depicted in Figure 2.

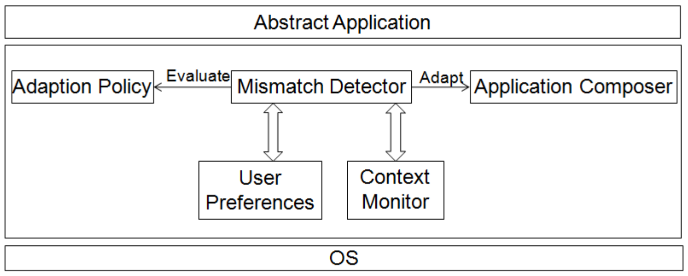

Fig. 2. Middleware for Monitoring and Adaptation

The middleware consists of a taskResolver component that maps the user task to the concrete level to identify components provided by the available devices. To that end, it relies on the device selection and component selection services to resolve the user task by mapping it to the concrete components based on the user preferences and some non-functional aspects [5].

A Mismatch Detector component is used to analyze the abstract description of the application compared to the capabilities of the selected devices, the user preferences and the extra-functional requirements of the components. For this goal, it relies on some monitor to capture the changes of the user preferences and the execution environment (network status, arrival and departure of devices or components, etc).

In case of the failure of the mapping or the application's execution due to the changes in the context or user preferences, the Mismatch Detector evaluates the application composition according to the adaptation policies provided by the Adaptation Policy component. If a policy for adaptation exist for that mismatch, the Application Composer is informed to adapt the application accordingly.

III. Structural Adaptation Approach

A. Principle of Our Approach

To overcome a captured mismatch between an abstract application and the concrete level, there is a need to adapt the abstract application to ensure its mapping and its execution.

Therefore, in our previous work [4], we have proposed a dynamic structural adaptation approach for abstract applications.

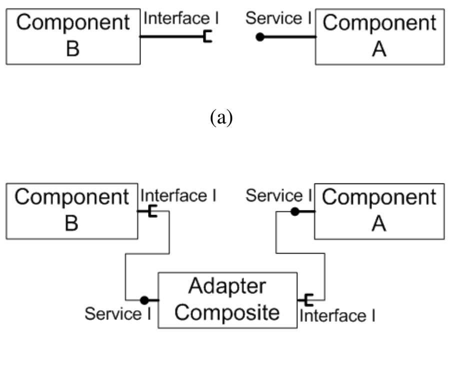

Fig. 3. Transforming abstract application using Adapter composite

Our approach consists of transforming an abstract application to another one that allows its mapping and execution. The transformation is ensured by injecting some extra-functional adapters into the abstract application without modifying its functional behavior.

For example, as shown in Figure 3, an adapter composite is injected to adapt the communication between components A and B. The adapter composite requires the service I of the component A and exposes a service implementing the interface I. This provided service will be used by the component B, since it corresponds to its required service. Thus, the abstract application is transformed by adding an extra-functional adapter to achieve its mapping or its execution.

B. Adaptation Patterns

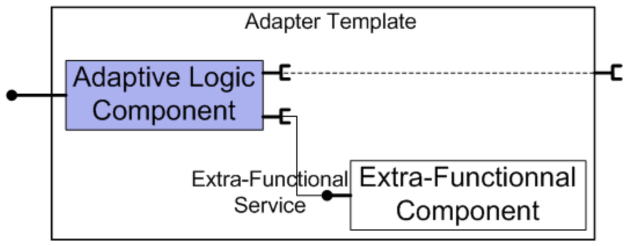

Fig. 4. Generic adapter template

As the basis for our approach, we have proposed to use adaptation patterns as adapter composites, which provide solutions for the detected mismatches between an abstract application and a concrete level and can be used in different contexts [4].

An adaptation pattern is defined following an adapter template, which consists of an adaptive logic and extra-functional components as shown in Figure 4. The extra-functional component provides transformation services allowing, e.g., encryption, compression, etc. However, the adaptive logic component encapsulates the adaptation logic and acts as an intermediate between the abstract and the extra-functional components. This component has a generated implementation as it depends to the interfaces of communicated components.

Using this approach allows us to separate the extra-functional logic from the business one, and hence, to add or remove adaptation patterns dynamically from the abstract application whenever there is a need.

In the following, we present a set of adaptation patterns that are defined following the adapter template. For each pattern, we present its description, the context in which, it will be used, and where it will be used to overcome that mismatch.

The list of the adaptation patterns is not exhaustive. However, it is possible to define such other patterns following our adapter template.

1) Encryption and Decryption patterns:

a) Description: we propose an encryption pattern that intercepts the interaction between components to encrypt messages transiting over a network in order to prevent the disclosure of information to unauthorized components. To use the original message, there is a need to restore it from the encrypted one. For this purpose, we have composed the encryption adapter with a decryption one that decrypts the received messages before using it by the target component.

The encryption and decryption adapters are defined following our adapter template. Each adapter consists of an adaptive logic component and an extra-functional one as shown in the Figure 5. The extra-functional component exposes a key property and provides a service ensuring encryption or decryption of a message following a symmetric key algorithm.

In case of a symmetric encryption and decryption, the transmitter and the receiver sides use the same key to exchange messages. However, if the extra-functional components implement an asymmetric algorithm, they use two different keys: public and private key.

b) Adaptation context: The encryption and decryption patterns will be injected at init time to ensure the security of the transferred messages, which may be sensitive to disclose as with credit card numbers and passwords.

The security requirement can be expressed explicitly through the non-functional requirements of services. If the concrete components do not ensure this requirement as specified in the abstract level, there is a need to adapt the application in order to achieve its mapping.

For this purpose, the encryption and decryption patterns will be used to overcome the non-satisfaction of the non-functional requirements of services.

c) Where to use: The encryption and decryption patterns are used in distributed manner; the transmitter device will contain the encryption adapter to send encrypted messages, while the decryption pattern is used by the receiver side to restore the messages. For example, in Figure 5, an encryption pattern is used by a device B in order to encrypt the messages sent from a component B over the network. However, a decryption pattern was handled in a device A to restore the original message before using it by a component A.

2) Authentication and Integrity patterns:

a) Description: The authentication pattern is used to sign the transferred message between two components in order to ensure that a message has not been tampered with and that

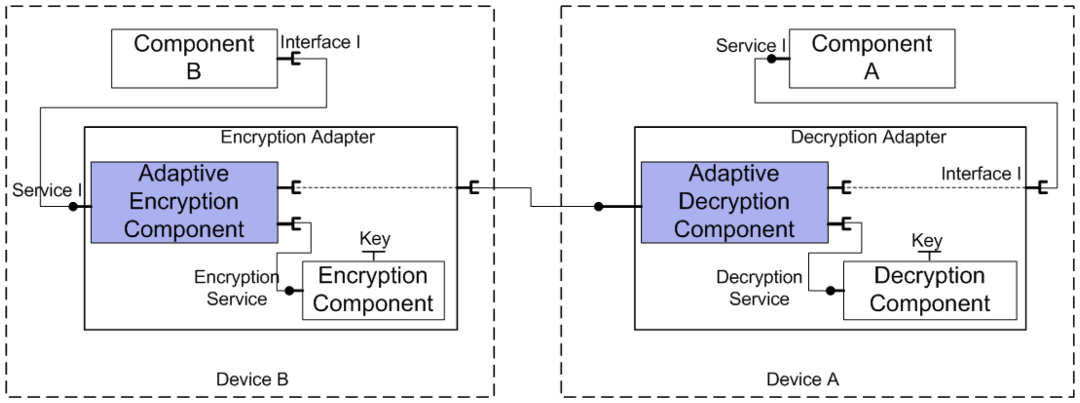

Fig. 5. Encryption and Decryption adaptation patterns

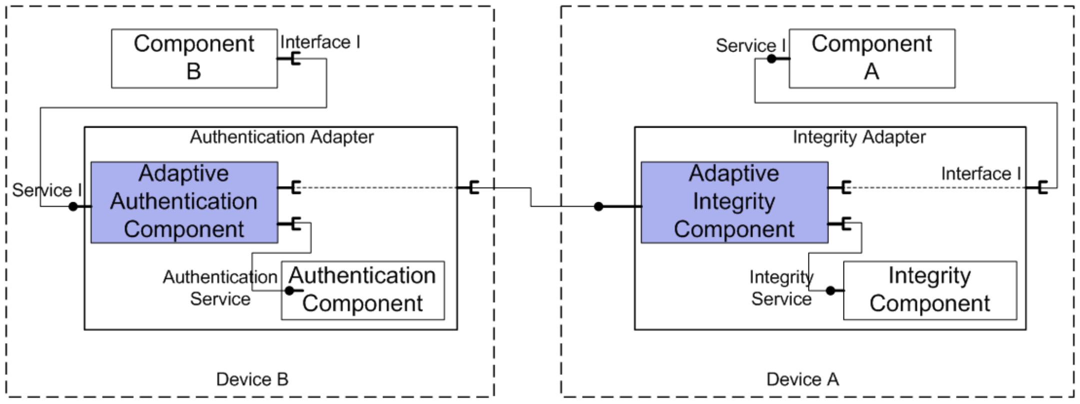

Fig. 6. Authentication and Integrity adaptation patterns

it originated from a certain component. The extra-functional component of the pattern generates a signature digest to add it at the end of the message before sending this later over the network.

In the receiver side, there is a need to validate the authentication of the message's signature to authorize component to invoke the requested service. Therefore, we have composed it with an integrity pattern that will prove the validity of a the transmitted message before forwarding it to the intended component.

The verification is done by comparing the received digest with a calculated one. If the message digest of the message matches the message digest from the signature, then integrity is also assured. Otherwise, the message is tampered with during its transfer. Thus, the extra-functional component of the integrity component, in Figure 6, returns a boolean result implying the validity of signed messages.

b) Adaptation context: An abstract component may specify at init time through its non-functional requirements the need to validate the received messages. If the concrete component does not ensure the authentication and the integrity of messages, this implies a mismatch between the abstract description and the concrete level.

c) Where to use: The authentication pattern is used by the transmitter side to send signed messages. In the receiver side, the integrity pattern will be used to check if the message is kept intact during its transfer over the network. Figure 6 shows an authentication pattern that is used by device B to send signed message from a component B to a component A. However, an integrity pattern is used by the device A, to validate the received message from the component B.

3) Splitting and Merging patterns:

a) Description: The splitting pattern is used to split a message into chunks. The pattern consists of an adaptive logic and an extra-functional component that returns a list of chunks to send over the network as shown in Figure 7.

To respond to the component's request, there is a need to merge the chunks beforehand. Therefore, we propose to compose the splitter pattern with a merger one to form the message from the received chunks. Hence, the extra-functional component of the merger pattern will construct messages from the received chunks, which are forwarded by the adaptive logic component to the intended component.

b) Adaptation context: The splitting and merging patterns are used to overcome a problem related to a lower network signal in order to have a decreased message delay. This may have an impact certainly for the transfer of bigger files or messages. In this case, the message is split into chunks for a quick transfer over a network and then merged to construct the original message.

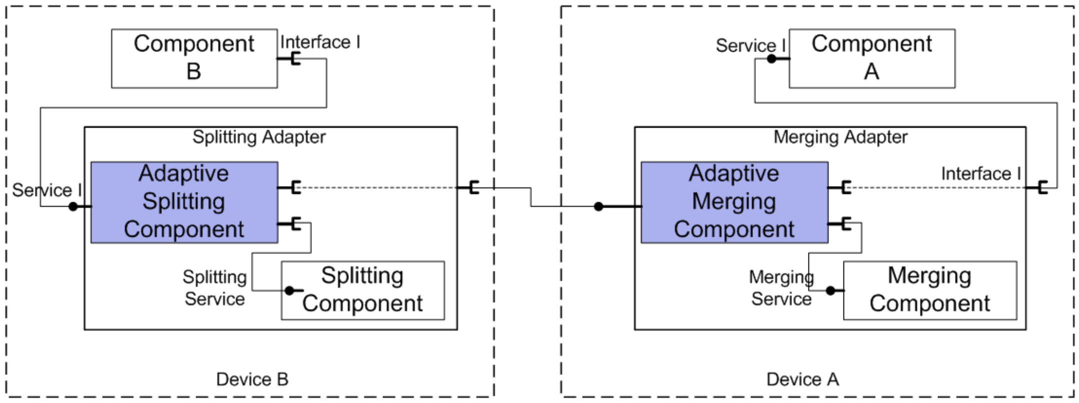

Fig. 7. Splitting and Merging adaptation patterns

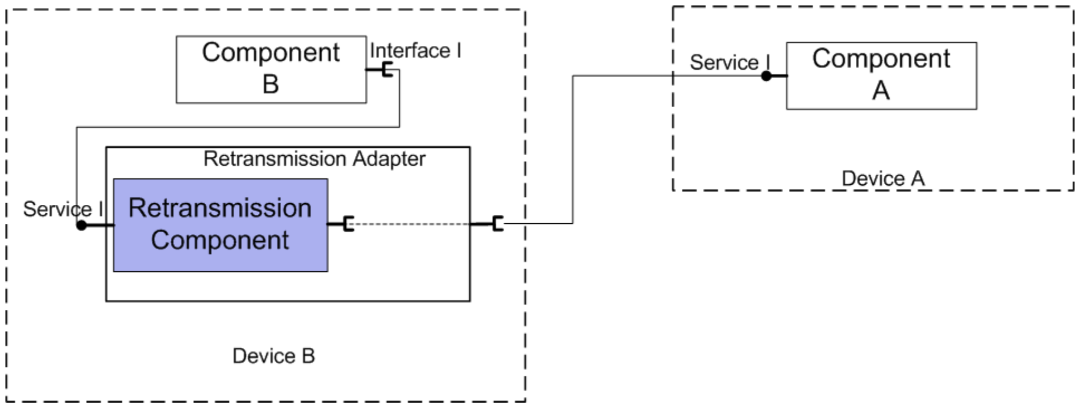

Fig. 8. Retransmition adaptation pattern

c) Where to use: The splitting pattern will be used by the transmitter device, while the merging pattern will be used by the receiver device to fulfill the interaction between devices. Thus, the component B in Figure 7 is able to send a message or a file into chunks to the component A for a quick transfer over a lower network signal.

4) Retransmission pattern:

a) Description: This pattern provides the functionality of retransmitting a failed call to a remote component. Its adaptive logic component, as shown in the Figure 8, attempts to retransmit message whenever a component does not receive a response to its calls. The component may use some error-detection codes or acknowledgments to achieve reliable message retransmission.

b) Adaptation context: The transmission in pervasive environment may be subject to failures because of e.g. network down. This can be captured by tracking the network work for a period of time. If the statistics shows that the system is not reliable, thus, the components' messages could get lost along the path. When it is not possible to deliver messages to remote components, the system should attempt to respond to the component request at the earliest possible opportunity by trying to retransmit the messages. For this reason, we propose a retransmission pattern that attempts to retransmit the messages to render the application reliable at init time.

The retransmission pattern is used also during the execution of the application, if the network is quick cut-off, to overcome the loss of messages. Thus, once the network is repaired, the retransmission pattern is established to retry the sending of calls. To detect this adaptation context, the middleware should monitor the status of the supported network, i.e., if it is activated or not.

c) Where to use: To ensure a reliable communication between devices, we propose to handle at init time a retransmission pattern in the sender side. For example, Figure 8 shows a retransmission pattern that is used by a device B to resend the failed call to a device A.

5) Compression and Decompression Patterns:

a) Description: The compression pattern is introduced between two components communicating with each other over a network in order to send compressed messages. However, in the receiver device, there is a need to decompress the message in order to be used by the target component. Therefore, we propose to compose the compression pattern with a decompression one to decompress the data before using it by the target component.

Each adapter consists of an adaptive logic component that relies on a non-functional component to compress or decompress message, as shown in Figure 9.

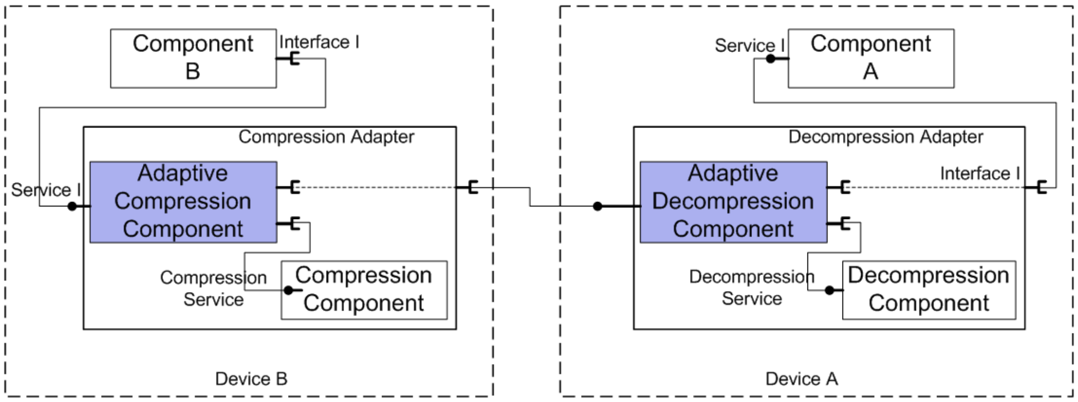

Fig. 9. Compression and Decompression adaptation patterns

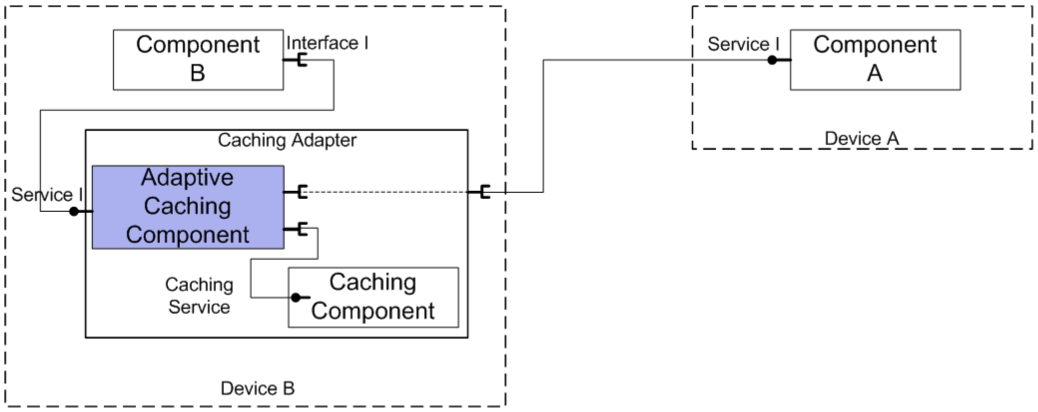

Fig. 10. Caching adaptation pattern

b) Adaptation Context: The compression and decompression patterns are introduced in response to a trigger generated by fluctuation in network QoS. For example, two components exchanging data may be adapted by using the compressor and decompressor adapters if the network latency or the throughput falls below a certain threshold. By using this adapter composite between the components, all the data will be compressed before transmission over network, allowing efficient transfer of data.

c) Where to use: We propose to use the compression pattern by the sender device in order to send compressed messages over network. However, the decompression pattern should be handled in the receiver side to decompress messages before using it as shown in Figure 9.

6) Caching Pattern:

a) Description: The caching pattern enables the application to cache messages in rapid memory. Figure 10 shows the main features of the caching pattern. It consists of an adaptive logic component that will first check the cache to see for example if the response of the component request can be found there. Failing to find the response in the cache, the adaptive caching component will forward the call to the target component. Once it receives the response, it will forward it to the caller component after storing it in the cache.

Thus, the caching service provides mainly methods for retrieving, updating and setting message in the cache. We assume also that the cache is already created else, the caching service should be able also to create a cache in a device.

b) Adaptation context: The caching pattern can be used during the execution of the application to avoid the congestion of a network by storing the responses to the services' requests. Therefore, the system should monitor the latency of the used network to identify if there is not a jitter. Otherwise, a caching pattern will be injected during the execution application to avoid the congestion for a further uses.

Moreover, some component may express through their non-functional requirements the need to have a decreased response time, for example the response time of service I is less than 50 ms. If the concrete component does not consider this requirement, there is a need to inject a caching pattern at init time in order to decrease the response time during the execution of an application.

c) Where to use: The caching pattern will be used either by the sender or the receiver side where the message will be stored. For example, in Figure 10, the pattern is used to decrease the response time to the requests of the component B by caching the call to service I in a cache of the device B.

7) Proxy Pattern:

a) Description: The proxy pattern allows components to access to services offered by others components. Figure 11

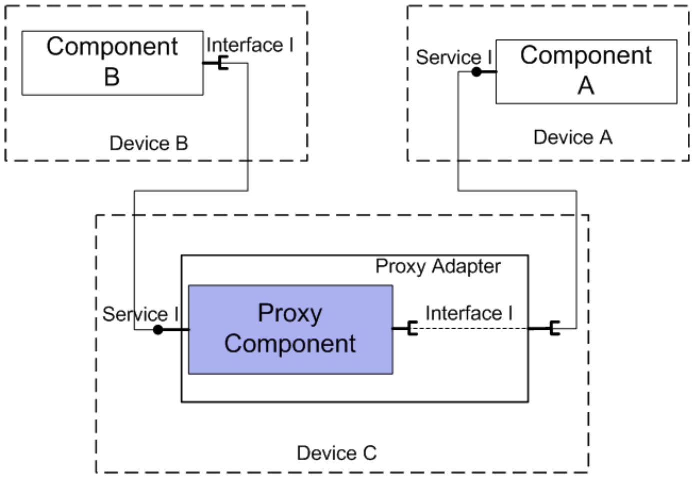

Fig. 11. Proxy adaptation pattern

shows a description of the proxy pattern following the adapter template. As it can be seen, the proxy pattern represents a specific case of the adapter template as it contains only an adaptive logic component that forwards the call of the service I to the component A.

b) Adaptation Context: The proxy pattern is useful to overcome the network factor related to the heterogeneity of network interfaces. For example, if two devices were selected to map an abstract application and they support two different connection interfaces, e.g., Bluetooth and Wifi, thus, the mapping will fail. Therefore, we propose to introduce the proxy pattern to act as an intermediate between the communicating components.

c) Where to use: To intermediate the communication between devices, we propose to generate the proxy in a third device. Thus, the components A and B, as shown in Figure 11, can communicate together via the proxy generated in a device C.

IV. Example Scenario Using Adaptation Patterns

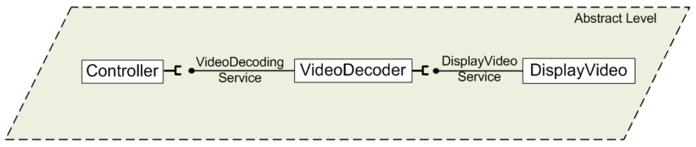

Fig. 12. Video Player application

Referring back to the video player application described in the introductory section, Figure 12 shows an abstract description of the Video player application that consists of three components: a *VideoDecoder* component, a *DisplayVideo* component and a *Controller* Component. The *Controller* component sends a command to the *VideoDecoder* component to decode a stored video. The *VideoDecoder* component decodes a video into appropriate format. Once the video is decoded, it is passed to the *DisplayVideo* component to play it. This is done using the service provided by the *DisplayVideo* component through an appropriate programming interface. The description of an application can be done with the help of an Architecture Description Language (ADL). For this purpose, we have used Service Component Architecture (SCA) [18] to describe abstract applications.

SCA provides a programming model for building applications and systems based on a Service Oriented Architecture. The main idea behind SCA is to be able to build distributed applications, which are independent of particular technology, protocol, and implementation. SCA applications are deployed as composites. An SCA composite describes an assembly of heterogeneous components, which offer functionality as services and require functionality from other components in the form of references. Along with services and references, a component can also define one or more properties.

Figure 13 shows an SCA description of the VideoPlayer application shown previously in Figure 12. It provide *DisplayVideoService* and consists of the controller, videoDecoder and DisplayVideo abstract components.

Using SCA, it is also possible to specify the services' requirements abstractly and the components' implementations provide the corresponding concrete policies [17]. Abstract resource requirements can be specified by using `@requires` attribute, while the `@policySets` attribute is used to specify the concrete resources. The policies are applied to implementation and contain the requirements that should be fulfilled before selecting the components to which the policy sets are attached.

For example, the controller component requires that its messages sent to the *DecodingVideoService*, should be authenticated. Therefore, its reference is marked with an intent "authentication" (line 4 in Figure 13). However, the *DecodingVideoService* is marked with the "'integrity"' intent to check the validity of the messages received from the controller component (line 7 in Figure 13). Figure 14 shows a description of the authentication abstract intent that is applied to the component binding.

```
<composite name="VideoPlayer">
 <service name="DisplayVideoService" promote="DisplayVideoComponent/DisplayVideoService" />
 <component name="ControllerComponent">
     <reference name=" DecodingVideoService" target="VideoDecoderComponent" requires="authentication"/>
 </component>
 <component name="VideoDecoderComponent">
     <service name="DecodingVideoService" requires="integrity">
        <interface.java interface="eu.tsp.iaria-example.VideoDecoderInterface" >
     </service>
      <reference name="DisplayVideoService" target="DisplayVideoComponent "/>
 </component>
 <component name="DisplayVideoComponent">
      <service name="DisplayVideoService">
        <interface.java interface="eu.tsp.iaria-example.DisplayVideoInterface" />
      </service>
 </component>
</composite>
```

Fig. 13. SCA description of the Video Player Application

```
<intent name="authentication" constrains="sca:binding">
<description>
Communication through this binding must be authenticated.
</description>
</intent>
```

Fig. 14. SCA policy intent of an authentication requirement

The resolution of the abstract description of the video player application into respective concrete components is required for the realization of the task. We assume that the execution environment consists of three devices: a Smartphone (SP), a flat-screen (FS) and a laptop device (LP).

Following the matching algorithm [16], the application composer of the middleware has identified a *Controller* and the *VideoDecoder* components in SP and a *DisplayVideo* component in FS. Thus, the LP device is eliminated.

However, the concrete components of the videoDecoder and the controller services do not support the policies related to the authentication and integrity intents as specified in Figure 13. Thus, there is a mismatch between the given abstract description of the video player application and the concrete level.

To resolve this mismatch, we propose to inject the authentication and integrity patterns into the abstract application as shown in Figure 15. Thus, the controller component is able to send authenticated commands to the VideoDecoder component. These commands will be validated at first by the integrity pattern before forwarding it to the videoDecoder component. As a result, the application is transformed, as shown in figure 15, to contain the authentication, and the integrity patterns in addition to its own components.

Figure 16 represents the SCA description of the integrity pattern. It consists of an adaptive logic component representing the integrity intent. To this end, we have extended the SCA description by the `@type` attribute (line 6) to specify what is the intent represented by the adaptive logic component if it is either a proxy or a splitting or a compression patterns. Moreover, the implementation of the adaptive logic component should be generated dynamically since this component depends to the *decodingVideoService* of the videoDecoder component (line 9). For this purpose, we have extended SCA by a new attribute `@generated` that specifies if the implementation is generated or not (line 6). Furthermore, the adaptive logic component relies on the integrity extra-functional component to check the validity of the received message before forwarding it to the VideoDecoder component.

In another case, we assume that during the execution of the application, the bandwidth of the supported network becomes weak. This may have an impact on the quality of video that requires a high QoS. Towards this change of context, we propose to adapt the abstract video player application by splitting the frame into chunks and then compress them for a quick transfer.

For this purpose, we have composed together the splitting, compression, decompression and merging patterns, as shown in Figure 15 for a quick transfer of messages with a lower bandwidth. The splitter adapter will split a frame sent from the VideoDecoder component to the DisplayVideo one into chunks. These latter will be compressed before their transfer over the network. Once the chunks are received by a device, there is a need at first to decompress them and then to merge them before forwarding it to the DisplayVideo component. Hence, the abstract application is adapted by injecting a composite of adapters to overcome a mismatch triggered by a low bandwidth.

V. IMPLEMENTATION

In order to validate our approach, we have implemented a prototype in Java. To that end, we have used SCA [18] to describe applications in abstract way and then map them to the concrete components.

The open source software JAVA programming ASSISTant (Javassist) library [11] is used to generate the byte codes of the

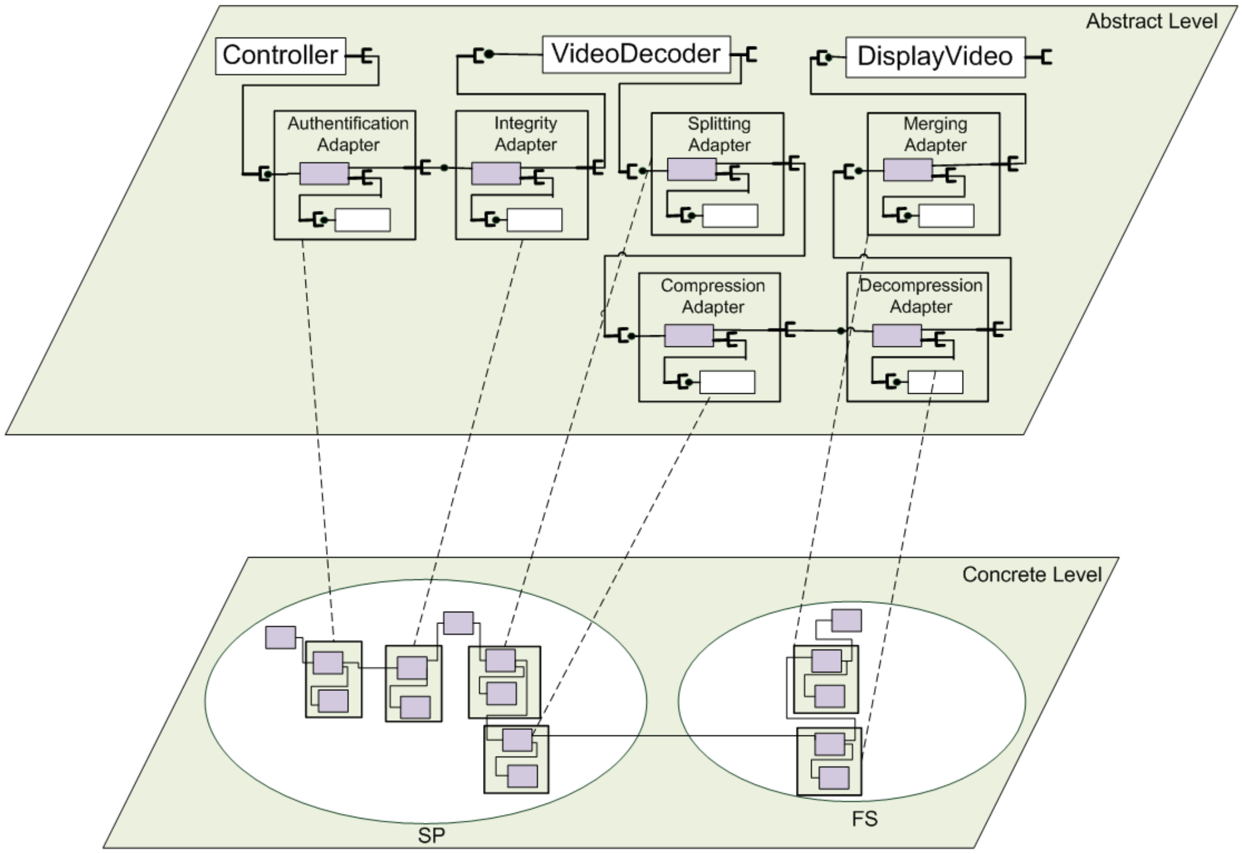

Fig. 15. Adaptation of the Video Player application

adaptive logic component of the adaptation patterns. Indeed, it enables Java programs to define a new class at runtime and to modify a class file when the Java Virtual Machine (JVM) loads it.

Moreover, the java API java.lang.reflect is used to obtain reflective information about classes and objects. This information is used by the Javassist library to allow the adaptive logic component to implement the required service.

VI. Related Work

A lot of work has been devoted to structural adaptation of component-based applications due to mismatches captured at the concrete level. In the following, we detail some of the existing approaches as well as their limitations.

Spalazzese and Inverardi [19] considers a mediator concept to cope with the heterogeneity of the application-layer protocols. The approach first abstracts the behavioral description of the mismatching protocols highlighting some structural characteristics. This is done by using ontology. Then, it checks the possibility for the two protocols to communicate. If the two protocols are not complementary, the framework should find out the suitable mediating connector using some basic mediators connectors patterns. However, these mediators connectors are limited to the protocol level. Moreover, the mediators are specified only by the framework and their concrete realizations remain a challenge for the authors.

In the same context, Fuentes et al. propose to use aspectual connectors that provide support to describe and to weave aspects to components [8]. However, these connectors are described at design time, which limits the possibility to extend applications with new aspects. Moreover, the specification of connector template relies on the used aspects as well as the functional behavior of application.

In [13], Li and al. tackle the behavioral mismatches and propose to use the mediation patterns. They categorize the mismatching levels and focus their work on signature and protocol mediations using six basic patterns. The identification of pattern is done following some rules predefined by the designer.

The major drawback of this approach is that is limited to the behavioral mismatches, thus, they do not consider the mismatches related to hardware, network characteristics of devices. Moreover, the generation of the mediator pattern is done by pseudo codes predefined by the designer. However, in our work, the adaptive logic component of the adaptation pattern is generated dynamically by specifying only it's required, provided interfaces and the extra-functional service.

Other related work in this area [7][12] have also investigated matching of Web service interfaces by providing a classification of common mismatches between service interfaces and business protocols, and introducing mismatch patterns. These patterns are used to formalize the recurring problems

```
<composite name="IntegrityAdapter">
  <service name = "DecodingVideoService" promote= "AdaptiveLogicComponent/DecodingVideoService" >
  <component name="AdaptiveLogicComponent" >
     <service name="DecodingVideoService">
     <interface.java interface="eu.tsp.iaria-example.VideoDecoderInterface" />
     <implementation.java  type ="Integrity"  generated="True" />
     </service>
     <reference name="IntegrityComponent"/>
     <reference name=" DecodingVideoService" target="VideoDecoderComponent" />
  </component>
  <component name="IntegrityComponent">
       <service name="IntegrityService">
      <interface.java interface="eu.tsp.iaria-example.IntegrityInterface" />
       </service>
    </component>
<composite>
```

Fig. 16. SCA description of the integrity pattern

related to the interactions between services. The mismatch patterns include a template of adaptation logic that resolves the detected mismatch. Developers can instantiate the proposed template to develop adapters. For this purpose, they have to specify the different transformation functions.

We can identify two important limitations compared to our approach. First, the mismatch patterns are limited to the interfaces and protocols mismatches. Second, the specification of adapters supplies some pseudo code predefined by the designer. However, in our approach, we are able to specify dynamically the different components of a used pattern; by generating the implementation of its adaptive logic component and mapping the extra-functional one following our matching algorithm

In [10], Cao et al. propose an approach to component adaptation dealing with non-functional mismatches. Their adaptation framework includes extra-functional adapters which mediate the mismatching behaviors between the client and server. Therefore, they propose to use adapters presented that provide extra-functional interfaces customized by the user.

Compared to our approach, the specification of adapters is done with the help of the user, whereas in our work, the adapters are specified using our template. Moreover, the specified adapters depend to the adapted application. Hence, it can be used on other context. However, our patterns may be used by any applications as their description is independent of the functional behavior of the application.

VII. Conclusion and Future Work

In this paper, we have identified some situational contexts at init time and during the execution of the application, according to which the application can be adapted. These contexts represent mismatches between an abstract application and the concrete level and they may arise at inter-components, intra-device or inter-devices levels.

Towards these mismatches, we have proposed a set of adaptation patterns that are injected into an abstract application to ensure its mapping or its execution. These adapters provide an extra-functional behavior with respect to the functional behavior of the application. The list of the adaptation patterns is not exhaustive. However, it is possible to define such other patterns following our adapter template.

We are looking forward to identify rules for the use of adaptation pattern describing where and when the adapter will be used.

References

[1] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *the 1st international symposium on Handheld and Ubiquitous Computing*, HUC' 99, pages 304–307, Karlsruhe, Germany, 1999.

[2] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. Pcom - a component system for pervasive computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, PerCom'04, page 67, Orlando, FL, USA, 2004.

[3] Steffen Becker, Antonio Brogi, Sven Overhage, Er Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In *Springer-Verlang, LNCS*, page 2006. Springer, 2006.

[4] Imen Ben Lahmar, Djamel Belaïd, and Hamid Mukhtar. Adapting abstract component applications using adaptation patterns. In *Proceedings of the Second International Conference on Adaptive and Self-adaptive Systems and Applications*, ADAPTIVE, pages 170–175, Lisbon Portugal, 2010.

[5] Imen Ben Lahmar, Djamel Belaïd, Hamid Mukhtar, and Sami Chaudhary. Automatic task resolution and adaptation in pervasive environments. In *Proceedings of the Second International Conference on Adaptive and Intelligent Systems*, ICAIS, pages 131–144, Klagenfurt, Austria, 2011.

[6] Sonia Ben Mokhtar, Nikolaos Georgantas, and Valérie Issarny. Cocoa: Conversation-based service composition in pervasive computing environments with qos support. *Journal Of System and Software*, vol 80, no 12:1941–1955, 2007.

[7] Boualem Benatallah, Fabio Casati, Daniela Grigori, H. R. Motahari Nezhad, and Farouk Toumani. Developing adapters for web services integration. In *Proceedings of the International Conference on Advanced Information Systems Engineering*, CAiSE, pages 415–429, Porto, Portugal, 2005.

[8] Lidia Fuentes, Nadia Gámez, Mónica Pinto, and Juan A. Valenzuela. Using connectors to model crosscutting influences in software architectures. In *The First European Conference on Software Architecture*, ECSA'07, pages 292–295, Madrid, Spain, 2007.

[9] Matthias Galster and Eva Bucherer. A taxonomy for identifying and specifying non-functional requirements in service-oriented development. In *IEEE Congress on Services*, SERVICES '08, pages 345–352, Honolulu, Hawaii, USA, 2008.

[10] Jingang Xie Guorong Cao, Qingping Tan. A new approach to component adaptation dealing with extra-functional mismatches. In *International Conference on Information Engineering and Computer Science*, Wuhan,China, 2009.

[11] JAVA programming Assistant. http://www.csg.is.titech.ac.jp/ chiba/javassist/.

[12] Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, pages 94–107, 2009.

[13] Xitong Li, Yushun Fan, Stuart Madnick, and Quan Z. Sheng. A pattern-based approach to protocol mediation for web services composition. *Information and Software Technology (IST)*, 52:304–323, March 2010.

[14] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *Journal of IEEE Computer*, 37:56–64, 2004.

[15] Hamid Mukhtar, Djamel Belaïd, and Guy Bernard. A graph-based approach for ad hoc task composition considering user preferences and device capabilities. In *Workshop on Service Discovery and Composition in Ubiquitous and Pervasive Environments*, New Orleans, LA, USA, dec 2007.

[16] Hamid Mukhtar, Djamel Belaïd, and Guy Bernard. User preferences-based automatic device selection for multimedia user tasks in pervasive environments. In *the 5th International Conference on Networking and Services*, ICNS' 09, pages 43–48, Valencia, Spain, 2009.

[17] Open SOA Collaboration. Sca policy framework v1.00 specifications. http://www.osoa.org/, 2007.

[18] Open SOA Collaboration. Service component architecture (sca): Sca assembly model v1.00 specifications. http://www.osoa.org/, 2007.

[19] Romina Spalazzese and Paola Inverardi. Mediating connector patterns for components interoperability. In *The fourth European Conference on Software Architecture*, ECSA'10, pages 335–343, Copenhagen, Denmark, 2010.