View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science 114 (2005) 47–64

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Towards Composition Management for Component-based Peer-to-Peer Architectures

Sascha Alda [1] Armin B. Cremers [1]

Institute for Applied Computer Science
University of Bonn
Bonn, Germany

Abstract

Recent peer-to-peer architectures do not fulfill the idea of a service-oriented architecture to allow the flexible composition of services towards concrete applications. This can be justified by the absence of flexible notations for the composition of services that incorporate the dynamic nature exposed by peer-to-peer architectures. In this work, the peer-to-peer architecture DeEvolve is presented that provides novel ways for the composition of services including the handling of exceptions such as the failure of peers. The intention of this approach is to facilitate even less-skilled end-users to compose and to maintain service-oriented applications.

Keywords: Component technology, composition languages, peer-to-peer architectures, exception handling.

1 Introduction

Software architectures represent software systems as a coherent set of high-level computational elements such as components, objects, or services together with their interactions and dependencies. The merit of an architecture-based development is to drift away from a low-level, code-based development towards a more flexible development focusing on the *composition* of self-contained building blocks. The structure of software architectures is often specified in a declarative manner by an external, formal notation. Prominent approaches for these notations are architecture description languages (ADLs) or workflow

[1] Email: {alda, abc}@cs.uni-bonn.de

1571-0661 © 2005 Published by Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.02.067

languages. Each approach comes along with appropriate assembly or adaptation tools in order to support the process of realizing software architectures even for less skilled end-users.

The notion of a software architecture makes at first no proposition concerning the actual organization, distribution, or availability of the constituting building blocks. Therefore, designers usually revert to so-called *architectural styles* to design software architectures on top of appreciated and established architectural organizations [5]. Well-known architecture styles for instance are the client-server, layered, or pipe and filters style. The selection of a style depends strongly on the intended use of the architecture.

In this work, the peer-to-peer paradigm [10] [2] is examined as another architectural style. Following this style, a peer-to-peer architecture constitutes a distributed architecture that consists of equal clients or so-called *peers*. Peers are capable not only of consuming, but also of providing computer resources that are encapsulated by *peer services*. In contrast to other architectures, peer-to-peer architectures assume an unstable, dynamic topology as an important constraint. Peers are solely responsible to affiliate to a peer-to-peer network. This degree of freedom permitted for peers could lead to unanticipated behavior or *exceptions* within the whole peer-to-peer architecture.

Though recent peer-to-peer architectures do already provide for various options for the interaction between peers, they exhibit two major drawbacks. First, neither architecture allows for the composition of peers (or their offered services) by means of appropriate notations. Moreover, current architectures are hardly resistant against unanticipated exceptions such as the failure of single peers, which is due to the absence of sophisticated models for exception detection and resolution. On the other hand, existing composition languages are not sufficiently flexible to describe compositions of services that reside within an unstable environment as typical for peer-to-peer architectures. There is also no explicit exception handling to describe alternative configurations in an exceptional case.

The major contribution of this work is a novel approach for the composition of peer services within a peer-to-peer architecture to overcome the drawbacks of existing architectures and notations. A notation called PeerCAT is presented for the composition of various peer services in a declarative manner. PeerCAT also allows to define *exception handlers* itemizing resolution plans in the case of exceptions caused for instance by the failure of peers. The structure of peer services is modelled by the composition of components. A component model does thereby prescribe the valid remote and local interaction primitives between services in a unified way. Both the component model and PeerCAT constitute the foundation for the component-based runtime environment DeE-

volve that allows to deploy component-based peer-to-peer applications.

This rest of this article is structured as follows. In the second chapter, the notion of the peer-to-peer architecture style as assumed in this work and a review of existing peer-to-peer architectures are pointed out. The third chapter presents the DeEvolve component architecture, covering the novel approach for composition management and exception handling. The related work presented in chapter 4 concentrates on the comparison of the illustrated solutions with other issue-related approaches. Chapter 5 finally concludes this paper and gives a brief outlook for future work.

2 Peer-to-Peer Architectures

The following sections present basic characteristics of the peer-to-peer paradigm and how this paradigm can be adopted as an architectural style.

2.1 Definition of Peer-to-Peer

Many definitions of peer-to-peer focus on the sharing of computer resources. The Peer-to-Peer Working Group for instance defines peer-to-peer as the sharing of computer resources and services by direct exchange of systems, where services and resources may include the exchange of information, processing cycles, and disk storage for files [10]. Peers thereby represent Internet-connected, unreliable, personal computers (PC) rather than high-end servers. Peer-to-peer architectures span a decentral network across existing network boundaries, without any central node or server, where data or services can be placed. This avoids a single-point-of-failure as typical for client-server systems: the failure of one individual peer does not affect the stability of the overall system.

Obviously, considering peer-to-peer as an interaction model between two equal nodes (or peers) is not new: it constitutes the original interaction model as apprehended in the ARPANET, the predecessor of the Internet. But for recent peer-to-peer architectures, however, the unexpected failure or unavailability of a peer is an important constraint to be taken into account. This constraint can be reasoned by the autonomous nature of a single peer: peers are free to decide, to what time and to what extend they offer their resources to other peers. The consequence of this autonomy could be the unavailability of peers and associated resources, which may in turn lead to failures of peers depending on these resources. While such an unstable environment for a DNS fragment would by no means be acceptable, peer-to-peer architectures have to put up with this circumstance.

Beyond the possibility of direct resource sharing, some authors point out the *self-organization* of peers into groups as another important aspect [2] [3].

Peer-to-peer computing should enable users to organize in groups without the assistance of a central authority. These self-governed communities can share, collaborate, and communicate, or participate in their own private web. Peer groups can thereby restrict the access of their computer resources to authorized peers. A peer first has to apply for group membership before it can join a group.

2.2 Peer-to-Peer as an Architecture Style

This section summarizes the different notions and characteristics of peer-to-peer architectures found in the literature and proposes a common *peer-to-peer architecture style* that is adopted throughout this work.

The peer-to-peer architecture style represents a pattern for distributed architectures consisting of equal nodes or peers, which do not exhibit a permanent affiliation to the topology of the architecture. Each peer thereby serves as a client and server at the same time, that is, it can not only revert to computer resources from other peers, but also provide resources to other remote peers. In the following, typical properties of the proposed style are elaborated based on a description catalogue for architecture styles provided in [5].

Design vocabulary. The fundamental design element is a *peer*. A peer is a node within a network topology that is not imposed to have a permanent affiliation to this topology. Peers can be logically grouped together to *peer groups*. A peer group is represented by a *super peer*. A super peer is the initiator of a group and is, from that point of time on, responsible to maintain the group. A single peer first has to *apply for a membership*, before it can *join* a distinct group. At any time, a membership can be *resigned* by both the peer or the super peer. Furthermore, each peer has the ability to provide computer resources such as data or services. Either type of resource is represented by a *peer service*. A peer service encapsulates resources and accomplishes the unified access to them through dedicated *ports*. A semantical description of a service and its belonging ports as well as an information how to contact a service are specified in terms of a *meta-description*. This meta-description is *published* by the peer providing a distinct peer service. Other peers can *discover* these descriptions, enabling them to contact the providing peer and, eventually, to use the particular peer service. If the consumer does not belong to the same group as the provider of a peer service, the peer first has to apply for membership as illustrated above.

Configuration rules. Conceptually, each peer is capable of being server and client at the same time, that is, capable of being provider and consumer of a peer service. Hence, there is a n:m relationship among all peers. The interaction between two peers takes place between two peer services or, more

exactly, between the ports of the services. A message-based event flow that can be designed based on the observer or publish-subscriber pattern [4] is assumed as a minimal interaction primitive.

Semantic interpretation. The underlying peer group concept has two semantic interpretations. The first intention is to subdivide peers into groups according to common interests or knowledge independent from any given organizational or network boundaries. Only peers that are interested or, moreover, are authorized to a group, are able to receive internal group messages. The second purpose is rather technically funded: the transmission of messages to a restricted number of (interested) peers instead of to all peers does actually reduce the network traffic.

Analyses. There are a plethora of possible analysis that can be conceived for this style. An important task is to schedule all connections or dependencies to other peers. The violation of a dependency to a peer during use time (for instance due to a failed peer) may not only result to local exceptional cases within a single peer, but also to global misbehavior within the entire peer-to-peer architecture. Exception handling therefore is a primary challenge for a peer-to-peer architecture. Exceptions can also occur during the start of an application, if remote peer services belonging to a previously composed application can not be resolved and integrated. This might lead to cases that the respective application cannot be started correctly. Besides, the adaptation of a peer service (for instance the deletion of single ports) can be erroneous, if dependencies from other peer services are violated.

Implementation Issues The adoption of the proposed style is not sufficient to obtain a practicable peer-to-peer architecture. For a concrete implementation, one has to incorporate a notation to describe the composition of peer services towards concrete applications. A composition language determines, which peer services constitute a composition and how these services interact with each other. Most notably, a language has to cope with the dynamic, unstable environment that is imposed on a peer-to-peer architecture. Assembly tools could not only enhance the process of service composition, but also the discovery and publication of peer service descriptions.

2.3 Review of existing Peer-to-Peer Architectures

Most peer-to-peer architectures, today, provide an infrastructure to share documents among peers. Prominent file sharing architectures for example are Gnutella, Freenet, or Napster. However, none of these architectures feature the sharing of services among peers, nor the composition of services towards new applications. Neither architecture does employ the concept of self-organization of peer into peer groups. For this reason, these architectures do not fulfill en-

tirely the proposed peer-to-peer architecture style of Section 2.2.

In order to consolidate the efforts for the design of peer-to-peer architectures towards a unified architecture, Sun has announced the JXTA project. The result of this project is a generic set of open protocols providing a framework for the development of peer-to-peer architectures together with an open source reference implementation for the Java platform [15]. In contrast to the aforementioned file sharing architectures, JXTA relies on a service-oriented architecture (SOA) model. The common resource that is shared among peers are services, which can be discovered, published, and accessed through the interplay of the individual JXTA protocols described below. The central concept of JXTA is the notion of peer groups as a way for the self-organization of peers. Peers having joined a distinct group can use services, which are only available to authorized group members. The PEER MEMBERSHIP PROTOCOL (PMP) thereby regulates the application and the access to peer groups, as well as the exclusion from groups through dedicated authentification routines. Peer Groups are represented by super peers called *rendezvous peers*. The PEER DISCOVERY PROTOCOL (PDP) allows a peer to discover other peers, peer groups, and services from other peers. All discoverable resources are described by *advertisements*, which are XML-based meta-descriptions of the respective resources. The Discovery Protocol is also applied to advertise or to publish advertisements to other peers. The actual communication between two peer services is established by the PIPE BINDING PROTOCOL (PBP). This protocol is used by a peer to build up virtual channels or pipes between two services, which can then be used for a message-based communication between services. In addition to these three protocols, a couple of other, rather auxiliary protocols are supplied, which are implemented by these main protocols (see [15] for more information).

The JXTA framework constitutes the most sophisticated peer-to-peer architecture due to its generic nature. The framework fulfills the characteristics of a peer-to-peer architecture style remarkably. However, JXTA does not address most of the implementation issues and analysis recommendations as elucidated in Section 2.2:

- Though services can necessarily be retrieved and integrated from within an application, there is no formal notation nor a tool support that allows to compose services in a declarative and persistent manner. The discovery and publication of resources as well as the management of groups is only supported through a command line tool (JXTA Shell).
- The exception handling in JXTA is realized only to a minor degree. There are no protocols or mechanisms that can handle an exception such as the failure of peer. The only mechanism to detect a potential failure is to define

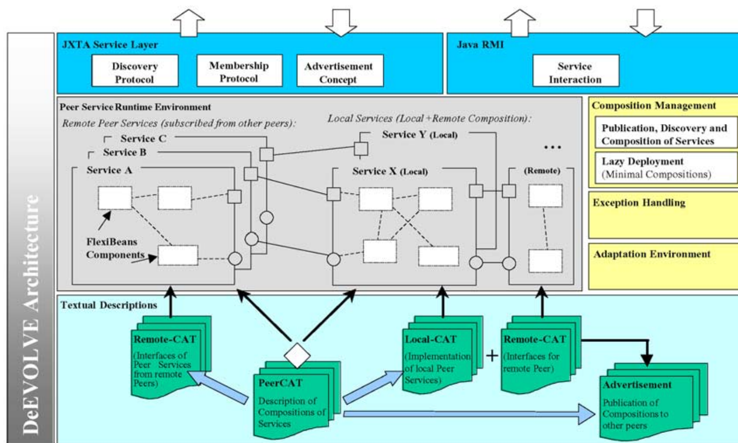

Fig. 1. The architecture of DeEvolve incorporating the JXTA peer-to-peer framework

a time-out for the discovery of peers and services.

In the next chapter, a novel approach for a peer-to-peer architecture is presented that incorporates the JXTA framework, and features additional concepts to overcome the mentioned drawbacks of JXTA.

3 The DeEvolve Peer-to-Peer Architecture

This section presents DeEvolve, our notion of a component-based peer-to-peer architecture that fully implements the architecture style of Section 2.2. DeEvolve is a further development of FreEvolve, a client-server architecture aimed to deploy component-based groupware applications [12] [13]. The underlying idea of DeEvolve is to model the inner structure of a peer service as compositions of *software components*. Software components can be seen as self-contained units of composition with contractually specified interfaces, whereas the interaction among components takes place only through these interfaces [16]. A component model prescribes the valid interaction primitives for both local interaction of components within a service and remote interaction between distributed peer services in a unique way. Two composition languages called CAT and PeerCAT are supplied to describe the composition of components and the composition of services, respectively. DeEvolve is built on top of the JXTA framework (section 2.3) to realize basic operations like the advertisement or discovery of peer services. The overall architecture of DeEvolve is depicted in Fig.1. In the following section, the integral parts of this architecture are elaborated in more detail.

```
<peer-service id = "GrammarChecker“>
  <required_port type="TextInput_shared" id = "input" />
  <required_port type="StartProcess_event" id = "start" />

   <i_component id = "GrammerCheckerComponent">
    <required _port type="TextInput_shared" id = "input" />
    <required_port type="StartProcess_event" id = "start" />
   </i_component>

   <bind>
     <i_component id = "CheckerComponent">
      <required_port id = "input" />
     </i_component>
     <required_port id = "input"> /** Public port of peer service */

     <i_component id = "CheckerComponent">
       <required_port id = "start" />
     </i_component>
     <required_port id = "start"> /** Public port of peer service */
   </bind>
</peer-service>
```

Fig. 2. An example Remote-CAT description for a grammar checker

3.1 Component-based Peer Services

DeEvolve incorporates the FlexiBeans [13] component model to describe the structure as well as the valid interaction primitives for a peer service. This model is an extension of the conventional JavaBeans model [14]. FlexiBeans components are accomplished to interact remotely through the explicit integration of the RMI technology. There are two interaction primitives for components, event notification and interaction through a shared object. Shared objects serve as an abstraction for a data flow between two components. `FlexiBeans` allows to define *ports*, in order to discriminate between type-equal event or shared objects primitives. For the event flow primitive, a port can either function as the source (`provided_port`) or as the sink (`required_port`) of an event flow. The same distinction is made for a shared object: the port that initially provides the shared object is indicated as (`provided_port`).

The composition of FlexiBeans components to declare a single peer service is formulated in a language called CAT [12]. A peer service always consists of two different compositions, a local composition (Local-CAT) that implements the actual service and a remote composition (Remote-CAT) that constitutes the interface to this service (Fig. 1). The necessary remote interaction between these two compositions is described by a third description called DCAT. During deployment (start) of a peer service, these descriptions are parsed by DeEvolve. Based on the respective files, all specified components (correspond to concrete class-files) are then instantiated. In a third step, all bindings among the generated components are established. A remote peer can access the interface composition through a modified classloader. Each service can also be used by the local peer providing the service; here no remote access via classloader is necessary.

```
<!DOCTYPE jxta:MCA> <jxta:MCA xmlns:jxta="http://jxta.org">
 <MCID> urn:jxta:uuid-F374F3AF78944A249531B17A029F3F2205 </MCID>
   <Name> GrammarChecker </Name>
   <Desc> This service checks the grammar of your text. </Desc>
   <PeerGroups>
    <Group>
      <Name> Bonn </Name>
      <UUID> urn:jxta:uuid-4A85775B635C494185268447A091F98 </UUID>
    </Group>
    <Group>
      <Name> Employee </Name>
      <UUID> urn:jxta:uuid-08227743080E464BAFEA854BFE492D4 </UUID>
    </Group>
      <RuledForAll> true  </RuledForAll>
   </PeerGroups>
</jxta:MCA>
```

Fig. 3. Example for a class advertisement

An example for a Remote-CAT description is depicted in Fig. 2. This CAT description illustrates the interface for a grammar checker service that can be used by other peers. The actual grammar checking is executed on the originating peer providing this service. The remote part of this service consists of a shared object port to supply the text to the service and an event notification port to start the grammar check on the supplied text. Both ports are implemented by a single component (`i_component GrammarCheckerComponent`). The `bind` command connects the *internal* ports of the component to the corresponding *external* ports belonging to the service. The external ports represent the public interface to this service, which can be used by remote peers. The next section describes how peer services can be advertised and discovered within a peer-to-peer architecture.

3.2 Advertisement of Peer Services

The Discovery Service of JXTA has been used to publish the existence of a component-based peer services to all other peers within a given topology. Other peers can discover service advertisements and, thus, access a service. According to the advertisement concept in JXTA, three different advertisement will be provided for each peer service:

- *Class Advertisement* The class advertisement is used to advertise the existence of a peer service. It has the following structure as depicted in Fig. 3. This advertisement basically consists of an informal description for a peer service, providing other peers with the semantics or meaning of a service. The advertisement also determines the group affiliation of a service. A peer that wants to access a service has to be a member of at least one group listed in the advertisement. If the tag `ruledForAll` is set to "yes", then a peer must belong to all quoted groups. In the given example, the peer

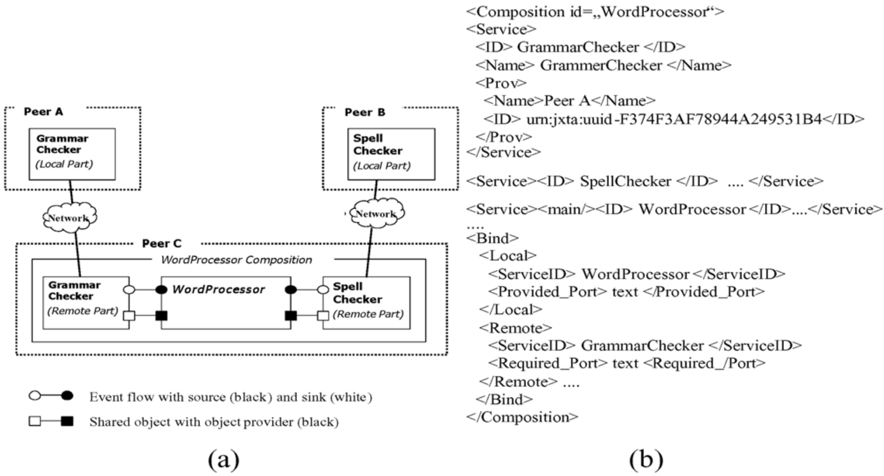

Fig. 4. Graphical (a) and declarative (b) representation of the Word Processor Composition (excerpt)

service `GrammarCheckerService` belongs to the groups `Bonn` and `Employee`.

- *Specification Advertisement* The purpose of the specification advertisement is to provide all information necessary to access a service. There can be multiple specification advertisements for a given class advertisement. The intention is to provide multiple versions of a given peer services.
- *Implementation Advertisement* This advertisement provides the Remote-CAT description of a peer services and, hence, a detailed specification, how to access the peer service. This advertisement is essential, if the peer services should be composed with other peer services (see Section 3.3).

So far, the concepts enables peers to advertise, discover, and to access peer servics. In the subsequent section, it is elucidated how one can compose services to yield a new application or even a new peer service.

3.3 Composition of Peer Services

The composition language PeerCAT is intended to fix compositions of peer services that constitute a new application. The main task of PeerCAT is to identify the services that belong to a composition and to bind the public ports of the respective peer services. Similar to the CAT language, port equality is assumed in two ways: (1) all binded connections must exhibit the same port type and (2) only provided and required ports can be connected. The type of a port is clearly determined by the name of a port and the class definition of the respective event or shared object. In Fig. 4 (a), an example composition representing a simple word processor application is depicted. This composi-

tion consists of three different peer services: a local service `WordProcessor` residing on the consuming peer and two remote services `GrammarChecker` and `SpellChecker`. The corresponding PeerCAT declaration of this composition is given in Fig. 4 (b). All service declarations consist of an address statement indicating the service providing peer. Exactly one service has to be declared as the main service of the composition (tag `main`). Within the `Bind` tag, the provided port `text` of the word processor is connected with both required ports `text` of service GrammarChecker and of service SpellChecker, respectively. The information concerning the ports of the remote peer services can be obtained by the corresponding implementation advertisements. Through these bindings, both remote services can refer to the text, to which they can pursue their checking procedures. A checked text annotated with corrections can be referenced by the word processor through the same shared object. All other bindings have been omitted for the sake of brevity. After the composition of these single services, the resulting application can be advertised and published as a peer service to other peers. Remote peers can thereby refer to the Remote-CAT description of the main service `Wordprocessor` to use the composition. This description is inserted in the implementation advertisement automatically.

During the deployment of this composition, the DeEvolve runtime environment tries to resolve the peer services by connecting the corresponding peers providing the service. Obviously, the application is incomplete and, in the first instance, not executable, if at least one peer service could not be resolved. In the next chapter, the notion of *minimal composition* is explained as a way to cope with this problem.

3.4 Minimal Compositions

In some application scenarios it is conceivable to indicate a composed application as executable, if a minimal subset of the overall composition has been resolved. A so-called minimal composition accomplishes to utilize an application in a minimal fashion. This assumes that the user of an application can abdicate from some services temporarily.

The declaration of a minimal composition is also inserted in the PeerCAT description of a regular composition. For the composition of a word processor as exemplified in Section 3.3, the following minimal composition as provided in Fig. 5 could be reasonable. In this example, service `WordProcessor` and `SpellChecker` are regarded as a minimal composition. The justification for this constellation is that a user normally checks the spelling of a text at regular intervals, while the grammar of a text is checked only at the end of the text preparation. Thus, he can do without the grammar service for a short time,

```
<Composition id=„WordProcessor“>
  <Service> .. </Service>
   ...
  <Bind>... </Bind>
   ...
  <Minimal_Composition >
    <ServiceID> WordProcessor </ServiceID>
    <ServiceID> SpellChecker </ServiceID>
    <Deployment> timeout = "1000" </Deployment>
   <Accessable> yes </Accessable>
  </Minimal_Composition>
</Composition>
```

Fig. 5. PeerCAT declaration including the definition of a Minimal Composition for the Word Processor (excerpt)

without interrupting the entire preparation of his text.

During deployment of the minimal composition, the missing services are substituted by so-called *mock services*, which are generated by DeEvolve automatically. While running the minimal composition, DeEvolve still tries to resolve the missing services and to deploy them belatedly by replacing the respective mock services. This kind of deployment is indicated as *lazy deployment*. The `Deployment` tag indicates, how long the system is instructed to resolve the missing services. After this time, the user himself can use the DeEvolve console to discover an alternative service. The `Accessable` tag denotes, if a composition that has been advertised as peer service can be accessed by other peers, even if only a minimal composition has been resolved.

The concept of minimal compositions and lazy deployment facilitate the exception handling during the start-up or deployment of a composition. However, these concepts do yet not support the treatment of exceptions during the use time of an application. Exception handling during use time will be the topic of the next section.

3.5 Exception Handling during Use Time

PeerCAT facilitates the definition of exception handlers to react on unanticipated exceptions during the use time of an application. Thereby, two different levels for exception handling are supplied, exception handling on *architecture level* and on a *service level*.

The benefit of exception handling on an architectural level is that developers are not forced to extend or to adapt the code of single components to detect and to handle exceptions. Actually, only the formal PeerCAT notation of a composition has to be augmented by additional statements. This allows

```
<Exception_Handling>
 <Exception Type = "FAILURE">
 <Context>
   <ID> SpellChecker </ID>
</Context>
 <Handlers selection=„no“>
   <Initially>
       <notification text="The Spell Checker Service has failed!" />
   </Initially>
   <Handler order="1" >
       <Discover name="Spell Checker" timeout="1000" >
       <IntegrateService/>
   </Handler>
   <Handler order="2" >
         <Notification text=„Use the console to find a new service" />
   </Handler>
<Handlers>
</Exception_Handling>
```

Fig. 6. PeerCAT example for exception handling on architecture level

developers of peer services to purely concentrate on the development of the business code rather than on exception handling. An example of this type of exception handling is depicted in Fig. 6. This example handles the potential failure of peer service `MySpellChecker` belonging to the word processor example. For exception handling on architecture level mainly three parts have to be specified: the exception type (here: `FAILURE` denoting the failure of a service), the context in which this exception might occur (service `SpellChecker`) as well as an ordered list of exception handlers. If attribute `selection` of tag `handlers` is set to `true`, then the user himself can determine, which handler is to be executed. If `false`, the system tries to execute the first handler of the given list of handlers. In the example of Fig. 6 two handlers are defined. The first handler tries to discover an equivalent service with the same name. If the timeout is exceeded, the user himself can search for and integrate an alternative service. Alternatively, the application can be left as is without a new spell checking service. Initially, a notification text about the exception is displayed.

Exception handling on the level of a service is useful, if an exception should actually be handled by a single service. A service could for instance ensure that for the failure of a dependent service a distinct service has to be discovered and integrated. In order to forward the occurrence of an exeception, dedicated ports of a service can be bound to predefined system ports of DeEvolve. All necessary information of an exception (e.g. the context) is encapsulated by an event object and sent to this service. An example is illustrated in Fig. 7. In this example, the `WordProcessor` service can react on any failed service; the information, which service has failed can be referred from the event object. Based on this information, adequate exception handling routines can be

```
<Bind>
  <Local>
  <ServiceID> WordProcessor </ServiceID>
    <Required_Port> port_service_failure </Required_Port>
  </Local>
  <Local>
    <ServiceID> DEEVOLVE_EXCEPTION_SERVICE </ServiceID>
      <Provided_Port> port_service_failure  <Provided_/Port>
  </Local>
</Bind>
```

Fig. 7. PeerCAT example for exception handling on service level

invoked from within the service.

For both types of handling exceptions, DeEvolve keeps track to which remote peers a peer has established a connection to consume a service. These peers are pinged in regular intervals. If each ping follows a response from a peer, then the peer is said to be alive. If no response occurs, an exception is assumed. Besides the failure of service or peers, other types of exceptions are supported by DeEvolve, such as the loss of group access rights, the violation of constraints defined on public attributes of services, or the violation of a minimal composition during runtime.

3.6 Prototypical Implementation and Evaluation

A first prototype of DeEvolve has been implemented in Java and is based on the Java reference implementation of JXTA as well as on the original FreEvolve platform. Basically, we have fully implemented the concepts concerning the management of compositions as explained in the beginning of this chapter. The underlying JXTA framework is accessed by the DeEvolve peer, which uses core JXTA services including the search service, the peer and peer group management, the publishing service, and the concepts of module specification. DeEvolve is already accompanied by a couple of useful tools. The DeEvolve console (Fig. 8 (a) supports the discovery and composition of peer services, the definition of minimal compositions. Another tool (Fig. 8 (b) supports the definition and advertisement of single peer services and the management of peer groups). Besides, the TailorClient [19] has been adopted by the original FreEvolve architecture for the adaptation of peer services by means of component-based adaptation mechanisms [13]. All other concepts - especially the exception handling in peer-to-peer architectures - are work in progress.

The evaluation of the presented concepts will be an important task for the nearer future. What basically needs to be proved is whether or not end-users perceive the tools and techniques for the composition of services as well

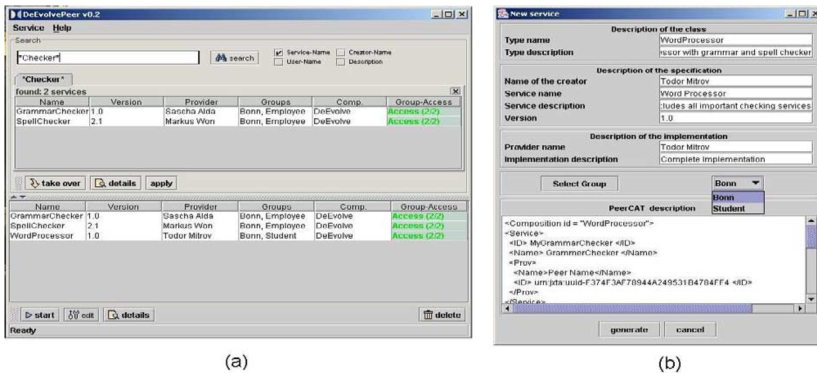

Fig. 8. The DeEvolve Console (a) and the Service Dialog (b) support the composition of peer services

as for the handling of exceptions as intuitive and easy to learn. The instrumentation of service compositions with exception handlers affords a relatively good understanding of the semantics of the complete composition, for instance the possible exceptions a composition may cause. Although the proposed instrumentation mechanisms are simple, end-user and, particularly, beginners could be swamped with the composition, instrumentation, and the selection of exception handlers. Similar observation were made in user tests for the first adaptation and compostion environments of the FreEvolve platform [19]. In order to improve the intuitiveness and the ease to learn of DeEvolve, one could think of further developments such as exploration environments, helping systems or wizards.

4 Related Work

In the literature, a plethora of architecture description languages (ADLs) plus accompanying toolsets and runtime environments can be found. In fact, most approaches presume a stable environment without the ability to change the topology dynamically, making them less practical for peer-to-peer architectures as supposed in this work. Though approaches like C2 [17] and Darwin [7] provide for dynamic modifications of compositions while the system is executing, it is not possible to define, when or under what condition (for instance due to an exception) configurations are to be carried out. Rapide [9] and Wright [1] are the only languages that support conditional re-configuration of architectures, but in a rather restricted way. Both notations provide a *where* clause to determine, under which conditions changes in the topology of an architectures are allowed. Components are thereby responsible to emit special control events to trigger these changes. The surplus value of PeerCAT is

that components (and services) do not have to emit control events to trigger changes. Exceptions are recognized by the DeEvolve runtime environment. Components only have to implement their business logic, but do not have to care about triggering events to signal an exception. PeerCAT also allows the interaction of an end-user to determine how an exception should be resolved. Although the language Wright admits the declaration of multiple alternative configurations, the involvement of an end-user is not designated. In addition, PeerCAT allows the meta-description (advertisement) of compositions. These descriptions improve the comprehension of a composition in particular for end-users, who are willing to integrate a (remote) composition in their own environment. The publication and discovery of compositions as well as the logical allocation of compositions into groups is not utilized by any approach.

Recently, rather workflow-based composition languages like WSFL [6] and XLANG [18] have emerged to describe the composition of web services. In contrast to the peer-to-peer architectures as aspired in this article, Web service architectures actually meet the conventional client-server architecture style. Exception handling, for instance to react on unavailable web servers, is yet missing in all these notations. Furthermore, the self-organization of web clients into groups is neither supported by these languages nor by any of the well-known standard protocols for web services like SOAP.

Exception handling on code-level is integrated in well-known programming languages like Java. The advantage of this method of exception handling is to handle a large number of different low-level types of exceptions for instance within a single component. However, developers of components cannot anticipate all situations, in which a distinct component will be deployed by third-parties and, thus, the possible exceptions it will cause. Hence, it is reasonable to integrate exception handlers after all components have been composed to a concrete application. The abstraction towards a more course-grained, declarative exception handling is intended to enable end-users the intuitive instrumentation of PeerCAT structures through appropriate tools.

The idea of minimal compositions have also been elucidated in [8]. The author thereby gives a formal definition for a minimal composition based on first order logic. However, no information is given how to resolve components into a deployed composition subsequently, as it is aspired with the lazy deployment concept outlined in this work.

5 Conclusions

This paper has presented a novel approach for the flexible composition of distributed services within software architectures that correspond to the peer-

to-peer architecture style. This style describes architectures consisting of equal peers without a permanent affiliation to the topology of a particular architecture. The dynamic nature of peer-to-peer architectures results in new problems and challenges that pertains in particular the composition of peer services. The DeEvolve peer-to-peer architecture addresses these issues by offering various options to handle exceptions that arise on an architectural level. The declaration of so-called exception handlers but also the composition of services is described by the composition language PeerCAT. Several accompanying tools support the discovery and advertisement as well as the composition of services. The further development of the existing tools towards graphical and yet more intuitive assembly tools is considered as the major future work. Besides, it would be reasonable to resign from the RMI technology for remote service interaction and to revert to the interaction facilities of JXTA. All existing interaction primitives of the current FlexiBeans model can be mapped by the Pipe mechanisms of JXTA. Admittedly, the integration of a new component model would not affect the presented approaches of composition management and exception handling

References

[1] Allen, R., Douence, R., Garlan, D., "Specifying Dynamism in Software Architectures", Proceedings of Foundations of Component-Based Systems Workshop, Zurich, Switzerland, September 1997.

[2] Barkai, D., *Peer-to-Peer Computing. Technologies for sharing and collaboration on the Net*, Intel Press, 2002.

[3] Brookshier, D., Govoni, D., and Krishnam, N., *JXTA: Java P2P Programming*, SAMS, Indianapolis, USA, 2002.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley, 1995.

[5] Garlan, D, Allen, R., and Ockerblum, "Exploiting Style in Architectural Design Environments", Proceedings of the ACM Symposium on Foundations of Software Engineering, New Orleans, USA, 1994.

[6] Leymann, F., *Web Services Flow Language (WSFL 1.0)*, May 2001.

[7] Magee, J., Dulay, N., Eisenbach, S. and Kramer J., " Speci-fying Distributed Software Architectures" Proceedings of 5th European Software Engineering Conference, LNCS 989, Springer, Barcelona, 1995.

[8] Millen, J.K., "Local Reconfiguration Policies" Proceeding of the IEEE Symposium on Security and Privacy, USA, 1999.

[9] Oreizy, P., Gorlick, M.M., Taylor, R.N., Medividovic, N., "An Architecture-based approach to Self-Adaptive Software", IEEE Intelligent Systems, May/June 1999.

[10] Peer-to-Peer Working Group, "What is Peer-to-Peer?", 2002, URL: http://www.p2pwg.org/whatis.

[11] Shirky, C. "Listening to Napster", in: Oram, A (ed.). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, Sebastopol, O'Reilly, 2001.

[12] Stiemerling, O., Hinken, R., and Cremers, A. B., "The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware", in: Proceedings of EDOC'99, IEEE Press, Mannheim. 1999.

[13] Stiemerling, O., *Component-Based Tailorability*, Dissertation, University of Bonn, 2000.

[14] Sun Microsystems Corp., *Java Beans Specification, V1.01*, 2000, URL: http://java.sun.com/products/javabeans/docs/spec.html

[15] Sun Microsystems Corp., *JXTA v2.0 Protocols Specification*, 2003, URL: http://spec.jxta.org/v2.0/

[16] Szyperski, C., Gruntz, D., and Murer, S., *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, London, 2002.

[17] Taylor, R.N., "A Component- and Message-Based Architectural Style for GUI Software", IEEE Transactions on Software Engineering, Vol. **22**, No. 6, 1996.

[18] Thatte, S., *XLANG: Web Services for Business Design*, Microsoft Corporation, 2001.

[19] Won, M., and Cremers, A.B., "Supporting End-User Tailoring of Component-Based Software - Checking integrity of compositions", in: Proceeding of CoLogNet 2002, Madrid, Spain, 2002.