

From Component-Based to Service-Based Distributed Applications Development and Life-Time Management

E. Grishikashvili, D. Reilly, N. Badr, A. Taleb-Bendiab

School of Computing and Mathematical Sciences

Liverpool John Moores University,

Byrom Street, Liverpool, L3 3AF, UK

Email: {cmsegris, cmsnbadr, cmsdreil, a.talebbendiab}@livjm.ac.uk

Abstract

Distributed applications are difficult to develop and manage due to their inherent dynamics and heterogeneity of component technologies and the possibility of different network protocols. Distributed component-based software applications often consist of a collection of software components that communicate via a distributed middleware. The distributed middleware, or simply middleware plays a crucial role by providing APIs and support functions to bridge the gap between the network operating system and distributed components and services. Based on an on-going research programme into self-adaptive software, this paper presents the early findings of our work, which is concerned with the development of a middleware-based framework for the “on-demand” software assembly for self-healing applications.

Keywords: Component-based software, Distributed Application, Service-Oriented Computing, Middleware

1. Introduction

Over recent years, component-based software development has received increasing interest from the distributed systems community as an emerging technology, which enables the modular development of new applications through the use of Commercial Off The Shelf (COTS) software components [1]. It is also accepted that component-based systems must be capable of coping with and adapting to changes that may include architectural or configuration changes or even changes in end-user requirements. This acceptance has led to a body of research work that propose compositional methods, which essentially separate the computational and compositional aspects present in component-based systems. For example, [2] presents a conceptual framework, which distinguishes between: components that adhere to an architectural style, scripts that specify

compositions, and glue that may be needed to adapt components’ interfaces and contracts.

Dynamic composition and configuration techniques provide useful mechanisms for design, maintenance and/or runtime adaptation of component-based systems, thereby reducing maintenance and/or development costs. An emerging trend in application development is that of a service-oriented abstraction, which regards an application as a federation of services, where a service represents a logical concept such as a printer or chat-room service. Services, which are provided and used by components, may be discovered dynamically and used according to a mutual agreed contract between provider and user. This relatively new approach has been adopted by the distributed middleware community, who are concerned with the development of dynamic and ubiquitous computing solutions that are capable of dealing with change and varying degrees of fault-tolerance.

Based on our on-going research into distributed component assembly and the management of self-healing software, this paper describes a middleware-based framework for on-demand service assembly and subsequent management of distributed applications.

The remainder of paper is structured as follow: section 2, describes the concepts of component-based software engineering and service-oriented development. In section 3 provides the review of current state of the art. Section 4 then goes on to describe on-going development of the framework architecture. Section 5 describes a case study, conducted to evaluate the architecture. Finally, section 5 draws conclusions and mentions future.

2. From Component-Based to Service-Oriented Distributed Applications

Component-Based Software Engineering (CBSE) has been widely documented and adopted in academia and industry, and many components standards have emerged and are currently in use including: JavaBeans, CORBA

and DCOM. There are several conceptual definitions of a software component, such as that of [3], namely:

“... A component can be considered as an independent replaceable part of the application that provides a clear distinct function ...”.

According to [4], a software component is a unit of composition, with pre-defined dependencies on other components. In the world of business systems, business components represent reusable conceptual artifacts that can be implemented and deployed in large business systems. Components may also be regarded as coherent packages of software that can be independently developed and delivered as a unit. The functionality of a component is accessed via an interface and interfaces may also be used to plug components together in order to construct a larger system [5].

For the purpose of our work, in conjunction with distributed object systems, we adopt the physical view of a component as a self-contained binary implementation, which consists of one or more objects. These objects occur as instances of the classes that make up a component. Components communicate with each other through connectors that are implemented via software interfaces, thereby providing a component-connector abstraction (Fig.1).

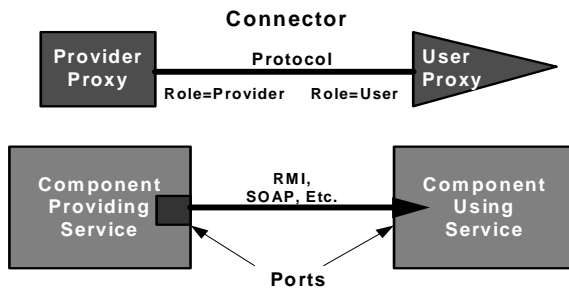


Figure 1: Diagram of Component connector, etc.

Components may provide services and/or use services provided by other components in a peer-to-peer like regime thereby providing a federation of services, which forms the basis of the alternate *service-oriented* abstraction, which we regard as:

“... A collection of application services spread over networked computers, which clients use remotely via distributed middleware services.” [6]

A service represents contractually defined behaviour that can be implemented and provided by any component for use by any other component, capable of meeting the contract.

Component models prescribe that programming problems can be seen as independently deployable black boxes that

communicate through contacts. The traditional client-server model often lacks well-defined public contracts that are independent of the client or server implementation, which renders the client-server model “brittle”. Through the service-oriented model, components may interchangeably provide and use services in to peer-to-peer manner, thereby eliminating the brittleness of the client-server model.

3. State of the Art Component-Based Service assembly and Management

The recent “state-of-the-art” developments in distributed application assembly adopt either component-based software concepts or service-oriented abstractions to provide dynamic assembly and management capabilities. The component-connector-port view of a distributed application is adopted by the Openwings community [7]. Openwings use an approach to assembly based on protocol specific connectors, which plug into ports, within the components, to facilitate synchronous and asynchronous communications between components.

The DARPA funded initiative for *Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA)*, [8], is actively investigating the use of software gauges to dynamically deduce component configurations and examine distributed applications as an assemblage of components. Reflective techniques are used by Diakov *et al.* in [9] to monitor distributed component interactions by combining CORBA’s *interceptor* mechanism together with Java’s thread API to “peek” into the implementation of CORBA components at runtime.

The Rio project [10], has made a significant contribution through an architecture that simplifies the development of Jini [11] federations by providing concepts and capabilities that extend Jini into the areas of Quality of Service (QoS), dynamic deployment and fault detection and recovery. Rio makes use of *Jini Service Beans (JSBs)*, *Monitor Services* and *Operational Strings*, were the latter are used to represent the collection of services and infrastructure components as an XML document. Also of interest in Rio is the *Watchable* framework, which provides a mechanism to collect and analyse programmer-defined metrics in distributed applications.

Mochizuki and Tokuda [12, 13] propose the *Improvised Assembly Mechanism (IAM)* that offers functionality to compose an application in an ad-hoc manner and achieve the application adaptation by adding, replacing, supplementing and relocating components at runtime. Through this approach, applications may be adapted to accommodate environmental changes such as the location changes of computing devices and user.

Our own approach has several similarities to the research work described above, but it concentrates specifically on the use of middleware services to provide *on-demand service assembly* and *assembly management services*. In particular, our approach makes use of a programming model that facilitates the assembly and subsequent management of distributed applications by extending core middleware services to provide assembly and management capabilities.

4. Impromptu Framework development

Typically, a distributed application is a composition of different components distributed over a network. The successful continuous operation of the application relies on its ability to make decision in the event of component failures. In the event of a failure, the application must then locate or discover another suitable component and replace the failed component “on-the-fly” at runtime, ideally without user interruption. These requirements raise needs for applications that are capable of self-management, self-healing and self-adaptation.

The following table lists the sequence of actions and processes that are organized into three categories in order to facilitate service assembly and provide self-adaptive capabilities. More detailed descriptions of these processes are provided later in this chapter.

<i>Phase</i>	<i>Sequences</i>	<i>Details</i>
Visual Assembly And Testing	Requirements	<i>List of required services</i>
	Services Assembly	S1 -> S3 -> S2 <i>Check Dependency</i>
	Execution	<i>Run the service</i>
Deployment	Configuration	<i>Give the place to live</i>
Application Management	Monitoring	<i>Check service constrains</i>
	Diagnose	<i>Identify the conflict</i>
	Solution Strategy.	<i>Notify, handle exceptions</i>
	System Controller	<i>Select plan, make decision (go back to assembly process)</i>

Table 1: Application assembly and lifetime management

Starting from the assembly and lifetime management model of table 1, we developed the *Impromptu* framework for *impromptu on-demand distributed service assembly*. The framework combines a variety of services for *runtime discovery, binding and executing, assembling and monitoring* distributed components that may each offer different services. The framework provides the user with capabilities to create new application services on-demand at runtime. This capability is provided through meta-representation of an application service in the form of an XML document. The framework also makes use of self-adaptive, self-healing paradigms that provide capabilities for detecting component failures and for applying subsequent remedial action in the form of “repair” strategies. The current version of the IMPROMPTU-framework has been implemented using Jini, which is a Java-based middleware technology developed by Sun Microsystems [11].

4.1 Jini Middleware Technology

The service-oriented abstraction, mentioned previously, is used extensively in Jini, which is based on Java’s Remote Method Invocation (RMI). Jini enables distributed applications to be developed as a series of clients that interact remotely with application components and their services through the proxies. The service object downloaded to client is supplied by the component that provides the service. A client can use any service-providing component that implements the interface instead of being statically configured to communicate with a particular component. Jini was chosen as the middleware technology for the framework, due to its rich support for service-oriented development, but many of the principles apply equally to other middleware technologies such as CORBA and Web Services.

4.2 Impromptu Framework Architecture

IMPROMPTU is a service-oriented framework for developing distributed applications by locating and assembling the distributed service-providing components required by an application. The Figure 2 illustrates the IMPROMPTU architecture.

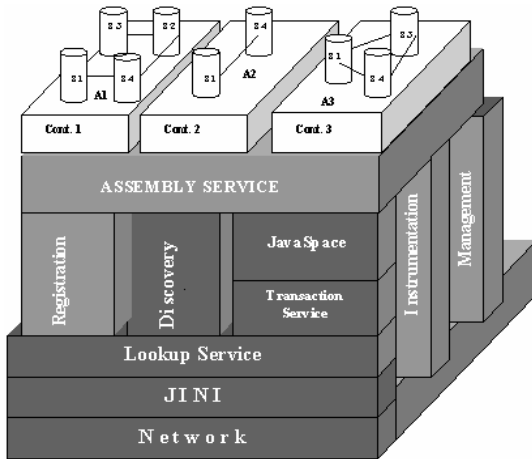


Figure 2: Framework architecture

The architecture is built upon Jini's core middleware services, which include: lookup, JavaSpace, transaction and discovery services and are shaded dark in Figure 2. The framework services (lighter shading) extend the core middleware services to provide: registration, configuration, assembly, operation, instrumentation and management services.

- Registration Service: registers application services provided by software components.
- Configuration Service: places services into containers (e.g. jar files) and auto-generates service proxies (i.e. Java RMI stubs).
- Assembly Service: locates the required components and invokes the registration and configuration services in order to assemble applications in accordance with the user specified XML document. Assembly service is considered further in next section.
- Operational Service: maintains the support environment required by certain application services (e.g. Java Servlets require a web server is running).
- Instrumentation Services: monitor and record client access of application services including method invocations. Instrumentation services consist of a series of monitor services, gauge services, probe services and logger and analyser utilities. Instrumentation services are considered further in [14].
- Management Services: examine information provided by instrumentation services. Management services use control rules, activated by conflicts, to identify the cause of a conflict and select a conflict resolution solution strategy. Management services are considered further in [15].

The top layer of the architecture contains the application-level service-providing components, which are

assembled, configured, instrumented and controlled, at runtime, by the framework services. For the remainder of this chapter, we describe the Assembly Services only as consideration of the other framework services exceeds to scope of the paper.

4.3 On-Demand Service Assembly Model

Fig. 3 shows or model of service assembly, which is explained further below.

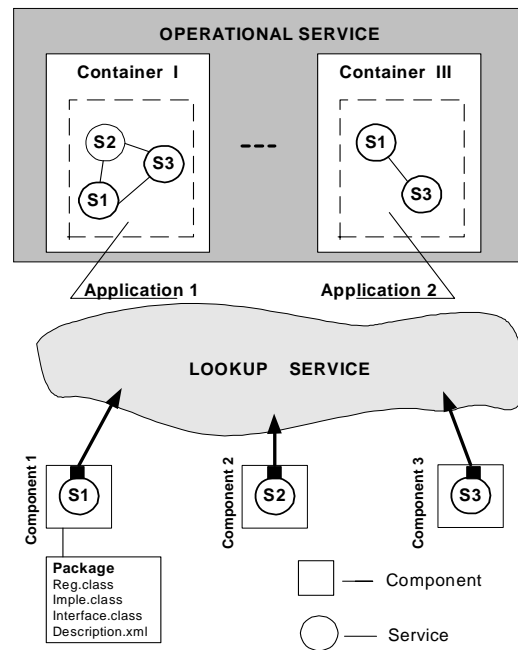


Figure 3: Service Assembly

The *Operational Service* is a Virtual Space containing one or more containers. A *Service Container* is a Virtual environment where services are executed and assembled ready for incorporation into a new application and each *Container* may contain one or more services. A *Component* is a set of all necessary class files necessary to allow service publication and service use via remote method invocation (components is typically stored in a jar file)

A *Service* is a logical concept that may be network service or a software service. Service can be implemented and provided by a component for use by any other component. Services may be distributed across several different machines and accessed through Jini's discovery/lookup mechanism and used through Java's RMI protocol.

In Fig. 3 *Component 1* offers *Service 1* by first registering it on a *Lookup Service* with a *Service Description* attribute, where the *Service Description* is an XML document and contains all information about the service. This information is needed by other components that may

either implement the service or may need to invoke the methods of the service. On the first invocation of a service method, the service comes to life within the Virtual Service Container, which may contain a number of services. When the user needs two, three or more services to work together as an application, s/he makes a request for these services and the framework finds a suitable component capable of providing these services.

By using Jini's lookup service with a service name and attributes it is possible to find and parse the XML document describing the service (Service Description attribute). The XML document specifies the signatures of the methods implemented by the service. Any component that wants to use the service may parse the XML document component wanting to obtain the method signatures and invoke the service methods as required. The reconfiguration service is responsible for saving the information about different versions/ states of the services. Each different combination of services creates a new distributed application and the system creates and saves meta-information about each application. Service combinations are stored in XML format and the Service Manager oversees the deployment and subsequent operation of each new application through the use of monitor and control services, which together provide self-adaptive capabilities.

5. Case Study

As mentioned previously, the development of the framework is ongoing. However, to date, the basic architecture underlying the framework has already been implemented and tested on an existing Jini application, using different use case scenarios. In order to illustrate the ideas above, we describe the following scenario:

“Denis is on the train coming back from his business trip and has a meeting scheduled for 9:00 am on the following morning with his company manager. For this meeting he must produce a report of the trip meeting and summarize his recommendations and of course complete his expenses forms. His main concern is the arrival time of 01:00 am the following morning, which would leave little time for him to prepare for the meeting. So his solution is to produce the report on train using his laptop. He starts the report, but the TextEditor on his laptop does not include spell checker nor a calculator or currency converter. Fortunately, the IMPROMPTU framework comes to his rescue by first locating the necessary service-providing components and then invoking these services without him having to worry about the whereabouts or configuration of the services.”

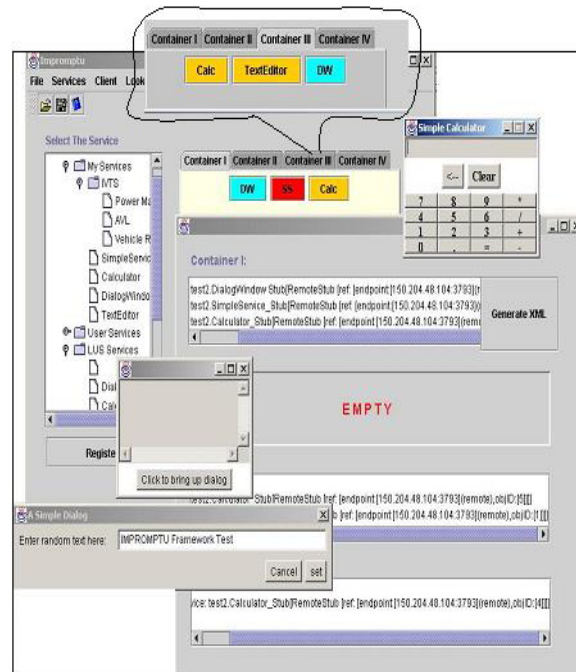


Figure 4: Case-study application

Figure 4, shows a screen-shot of the GUI of the application that was developed to test different case-study scenarios. The screen-shot illustrates the software environment developed to support the runtime discovery of published software and the assembly utility to compose new user applications.

The service tree presents the user with a list of all the services that are available on a lookup service at a particular instant in time. The user may choose the services that s/he needs but in order to invoke methods on the services, but any client-side component that needs to invoke service methods needs to know the method signatures. Unfortunately, these are only known by the service developer, who is responsible for the implementation of the service. To address this dilemma, we use XML-based service descriptions that specify method signatures as mentioned previously.

To explain the above use of XML, we reconsider the calculator that may specify *add*, *subtract*, *divide* and *multiply* methods and in the scenario described above the calculator has a single method *getCalcInterface*, that displays the calculator GUI by returning a Java *JFrame* object, but still this method is not known to the user. One of the novel aspects of our approach is that of the XML meta-representation, which is attached to a service as an attribute and passed to a client component via a lookup service. An Assembly Service parses the XML file to acquire specific information about the components and their properties. The service-providing components provide they can be solely software components or simple hardware components (e.g. a

printer) or a complex combination of the two. In our simple case above, the only a single method of invocation is specified in the XML file to provide a calculator GUI (i.e.

```
<MethodInvocation>
getCalcInterface</MethodInvocation>
```

but in a more complex case there may be several different methods (e.g. for a calculator with no GUI the methods would include *add*, *subtract*, *divide*, *multiply*). An Assembly Service gets the information about the location of actual class file using Java's reflection API, which is used to acquire the structural contents of a Java class including the list of methods implemented by the class

When one component requires a service provided by another component in order to deliver it's prescribed functionality then there is a dependency relationship between the two components [15]. Conceptually, a dependency is a directed relationship between a set of components, that can be represented using the ideas of [5] who defines the relationship based on two roles: the dependent component (a client) and the free or independent component(s) (service provider(s)). Using this representation we are able to maintain the dependencies that may exist for each service within a virtual container. Dependencies may also be dynamic, changing in accordance with different user tasks and these dependencies are represented within the XML descriptions with each application service as shown in Fig. 5.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
New Application Description
-->
<!DOCTYPE NewApplication (View Source for full doctype...)>
<Container>
  <Service ServiceID="s1">
    <ServiceName>Calculator</ServiceName>
    <Port>4061</Port>
    <Host>cmsegris</Host>
    <ServiceLocation>http://cmsegris:8080/userdoc/Calculator</ServiceLocation>
    <InterfaceName>CalcInterface</InterfaceName>
    <InterfaceLocation>http://cmsegris:8080/userdoc/Calc.Interface</InterfaceLocation>
    <Method>getCalcInterface</Method>
    <Dependency>Antecedent</Dependency>
  </Service>
  <Service ServiceID="s2">
    <ServiceName>DialogWindow</ServiceName>
    <Port>4061</Port>
    <Host>cmsegris</Host>
    <ServiceLocation>http://cmsegris:8080/userdoc/DialogWindow</ServiceLocation>
    <InterfaceName>DWIn</InterfaceName>
    <InterfaceLocation>http://cmsegris:8080/userdoc/DialogWindow</InterfaceLocation>
    <Method>getDWInt</Method>
    <Dependency>Dependent</Dependency>
  </Service>
</Container>
```

Figure 5: XML description for combined application services

6. Conclusions

In this paper we have described a framework, based on component/service-oriented abstractions. We have

described the connection between the two related paradigms in terms of services being provided and used by components. We have described an implementation of the framework based on Jini middleware technology and mentioned Jini's suitability for component based software development. We have also described our on-demand service assembly model, which makes use of an XML-based meta-representation, which specifies method signatures that allow remote client components to use services provided by other components. We have demonstrated the framework through a case study scenario based that considers how applications can be created as an assembly of components with minimum effort on the users behalf. In our future work, we intend to extend the framework to integrate monitoring and management services that respond to and correct inconsistencies and failure by adapting the systems behaviour.

References

1. Crnkovic I., Larsson M., "A Case Study: Demands on Component-based Development", In Proceedings of 22nd International Conference on Software Engineering, ACM Press, 2000.
2. Piccola – A Small Composition Language. <http://scgwiki.iam.unibe.ch:8080/SCG/10>
3. Zeynep Kiziltan, Torsten Jonsson, and Brahim Hnich. "On the Definitions of the Concepts in Component Based Software Development", Uppsala University Department of Information Science.
4. Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley and ACM Press, 1998, ISBN 0-201-17888-5
5. Alan W. Brown, Kurt C. Wallnau. "An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering", ICSE 98, Japan 1998
6. Emmerich, W, "Engineering Distributed Objects", John Wiley & Sons Ltd., ISBN: 0-471-98657-7, 2000.
7. Openwings Community, "Openwings Overview, Alpha v0.7", <<http://www.openwings.org>> (accessed January 2002).
8. Garlan, D. (Carnegie Mellon University) and Stratton, R.. (Air Force research Laboratory), "Architecture-based Adaptation of Complex Systems", DASADA Project List, DARPA (Program Sponsor), <<http://schafercorp-ballston.com/dasada/projectlist.html>> (accessed January 2002).
9. Diakov, N.K.; Batteram, H.J.; Zandbelt, H.; Sinderen, M.J., "Monitoring of Distributed Component Interactions", RM'2000: Workshop on Reflective Middleware, New York, 2000.

10. Jini Community, "Rio Architecture Overview", Rio Project, <<http://www.jini.org/projects/rio>> (accessed January 2002).
11. Jini Community, <<http://www.jini.org>> (accessed January 2002).
12. Mochizuki M, Tokuda H. "*Improvised Assembly Mechanism for Component-Based Mobile Applications*" IEICE TRANS. COMMUN., VOL.E84-B, NO.0 2001
13. Mochizuki M, "*A Middleware Architecture for the Improvised Assembly of Component-Based Applications*". Ph.D Thesis, Keio University, 2000.
14. Reilly, D. and A. Taleb-Bendiab, "*Dynamic Software Instrumentation for Jini Applications, in Proceedings of the 3rd International Workshop on Software Engineering Middleware*" SEM 2002 (ICSE 2002), Orlando, Florida, USA, Springer-Verlag, 2002.
15. Badr, N.; Reilly, D.; Taleb-Bendiab A., "A Conflict Resolution Control Architecture for Self-Adaptive". Proceedings of International Workshop on Architecting Dependable Systems WADS 2002 (ICSE 2002), Orlando, Florida, May 2002.
16. Keller A, Blumenthal U, Kar G. "*Classification and Computation of Dependencies for Distributed Management*". Proceedings of the *Fifth IEEE Symposium on Computers and Communications (ISCC 2000)* France, 2000.