An Architecture to Support Dynamic Composition of Service Components

David Mennie, Bernard Pagurek
Systems and Computer Engineering
Carleton University
1125 Colonel By Drive, Ottawa, ON, Canada, K1S 5B6
{dmennie, bernie}@sce.carleton.ca

Abstract

The creation of composite services from service components at runtime can be achieved using several different techniques. In the first approach, two or more components collaborate while each component remains distinct, and potentially distributed, within a network. To facilitate this, a new common interface must be constructed at runtime which allows other services to interact with this set of collaborating service components as if it was a single service. The construction of this interface can be realized with the support of a service composition architecture. In the second approach, a new composite service is formed where all of the functionality of that service is contained in a single new component. This new service must be a valid service, capable of the basic set of operations that all other services can carry out. Our goal is to design an architecture to support the runtime creation of composite services, within a specific service domain, using existing technologies and without the need for a complex compositional language. We make extensive use of a modified Jini infrastructure to overcome many of the shortcomings of the JavaBeans component model. In this paper, we compare techniques for dynamic service composition and discuss the requirements of an infrastructure that would be needed to support these approaches.

Keywords: Dynamic service composition, runtime component assembly, component-based services

1 Introduction

One of the goals of component-oriented programming has traditionally been to facilitate the break up of cumbersome and often difficult to maintain applications into sets of smaller, more manageable components [7]. This can be done either statically at design-time or load-time, or dynamically at runtime. Selecting ready-made components to construct an application is sufficient for a relatively straightforward system with specific operations that are not likely to change frequently. However, if the system has a loosely defined set of operations to carry out, components must be able to be upgraded dynamically or composed at runtime. It is this need for dynamic software composition that we will examine in this paper.

2 Defining the Problem

Our research group has previously approached the problem of dynamic software composition as it relates to high-availability systems. Before we define the approach to runtime composition taken in this paper, we will briefly describe our previous experiences in the area of software hot-swapping.

2.1 Software Hot-swapping

Software hot-swapping is defined as the process of upgrading software components at runtime in systems which cannot be brought down easily, cannot be switched offline for long periods of time, or cannot wait for software to be recompiled once changes are made [3]. These systems include critical high-availability systems such as control systems and many less-critical but still soft real-time, data-oriented systems such as telecommunications systems and network management applications. An infrastructure that supports software hot-swapping must take into account many factors. There are synchronization and timing issues such as when an upgrade can occur and the maximum time window allowed for an upgrade. The size and definition of the incremental swap unit or module must be defined. The series of transactions required to carry out how that unit can be dynamically introduced into a running system must be defined. The state of the system must be known at all times. Placing the target system in a state where a swap can occur,
capturing the state prior to the swap, swapping the module, restoring the system state, and then switching the system over to the new swapped module must all be handled. A failure recovery mechanism must also be in place to rollback an unsuccessful swap without affecting the execution of the running system. System performance must not be compromised and additional side-effects of the swapping process must be minimized. We have developed a prototype infrastructure to support this approach to dynamic software upgrading which takes into account the aforementioned issues. It is described in more detail in another paper [3]. Projects such as SOFA/DCUP [9] have also provided infrastructures to support dynamic component updating in running applications.

2.2 Motivation

While component composition at runtime in high-availability systems poses some interesting challenges, it is focussed on a specific type of system. Many systems cannot be classified as high-availability. For this reason, a more generic composition infrastructure that is not over-complicated with the difficult timing, synchronization, and transactional concerns of a hot-swapping solution would be more appropriate.

The composition architecture described in this paper is dedicated to the creation of composite network services from service components. The research is motivated by that fact that in many areas of network computing the need for more complex services is rapidly increasing. As standardized means for service lookup and deployment become available, the ability to compose composite services out of service components becomes more realistic. The rest of this paper is devoted exclusively to defining and developing a tailored, dynamic service composition solution.

2 Dynamic Service Composition

Dynamic service composition differs from other forms of software composition since it deals exclusively with network services. Network services are individual components, which can be distributed within a network environment, that provide a specific set of well-defined operations. The creation of composite services from a set of service components at runtime is feasible if its application is limited to a particular domain. Example domains include IP telephony, Internet security, e-commerce, and network management. While conceptual instances of cross-domain composite services exist, we prefer to limit our discussion to examples of concrete applications.

The first step in creating any composite service is to locate the service components that provide the functionality that is to be placed in the new service. To facilitate this process, all service components must be stored in a component directory that can be accessed at runtime. Searches of this directory must be tailored to the compositional attributes of the components. In other words, each component must have a clear description of the operations it can carry out, what methods (if any) can be extracted from it or used in the creation of a composite, and the input and output requirements of the component. Once the appropriate service components are located, we must determine the type of dynamic composition we will perform. There are various tradeoffs associated with the selection of a particular composition method. In many cases, more than one method is possible. However, the selection of the best method should be based on how the composite service will be used and the efficiency requirements of the resulting service. Another objective is to minimize unanticipated service behaviors once the composite is complete.

3.1 Creating a Composite Service Interface

The following approach can be used if a high-level of software performance for the service composite is not required. The idea is to create a new interface that will make a set of collaborating services appear as a single composite service. We have made use of an extended Facade design pattern [4] to help us create this interface. The Facade pattern is intended to provide a unified interface to a set of components and handle the delegation of incoming requests to the appropriate component. However, this is done statically at design-time. Our Dynamic Facade pattern facilitates the updating of the composite service interface as components are added at runtime. If the services taking part in the composite service are not co-located on the same network node and are instead distributed throughout the network, messages will need to be sent between the components via the interface. We are currently examining the potential for a distributed
Dynamic Facade pattern to delegate incoming requests to the appropriate service components even if those components are not located on the same network node. In a distributed composite service, we would expect a slight decrease in response time or operational performance. However, recent improvements in network bandwidth and transmission speed over network links will help to limit any performance decline. Figure 1a illustrates the realization of a composite service interface.

The primary advantage of this technique is the speed at which a composite can be created. This is due to the fact that a new component does not need to be constructed. In other words, no code needs to be moved or integrated from any of the components involved in order for the composite service to function. This technique is also referred to as *interface fusion* since the interfaces of each service component involved are merged into a single new interface. However, the interfaces of services 1 to n cannot simply be “glued” together. Modifications will be required to properly direct incoming messages to the appropriate components and in the proper sequence.

(a) Composite service interface  
(b) Stand-alone composite service

**Figure 1**: Composite services

### 3.2 Creating a Stand-alone Composite Service

If the performance of the composite service is more critical, creating a stand-alone composite service is a better solution than interface fusion. Performance may be improved since all of the code of the composite service is located on the same node (see Figure 1b). There are two primary means of creating a stand-alone composite service.

One approach leads to the dynamic assembly of service components in a way that is analogous to an assembly of pipes and filters. Jackson and Zave also adopt this technique in their distributed feature composition (DFC) architecture [6]. As in DFC, our architecture has the typical advantages of the pipe-and-filter architectural style. The main advantage is all service components can remain independent. This means they do not need to share state information and they are not aware or dependent on other service components. They behave compositionally and the set of service components making up a composite service can be changed at runtime.

(a) Serial chaining of service components  
(b) More complex component interconnection

**Figure 2**: Fig. 2a and 2b show various potential pipe-and-filter assemblies of service components within a stand-alone composite service. Fig. 2c illustrates an assembly of composable methods from several service components into a single new service containing a single body of code.
Figure 2a shows a basic configuration for a set of service components to be assembled. The input to the composite service is sent to the first service component, which in turn, sends its output to the input of the next service component in the chain. Obviously, each service component must be capable of handling the input it is given. A different result may be obtained if the components are re-ordered. The order in which components are assembled and the input requirements and output results of each component are specified in the service specification of each component. This service specification is physically stored with the component since the infrastructure will need to read it prior to determining if it will be required in a given composition scenario. Figure 2b shows the potential for more complex interconnections of service components. In this case, the operations performed by service component 2 are required several times in sequence. A loopback data flow can be used to achieve this without the need to chain several replications of the same component in sequence. Support for a loopback feature must be provided by the service component. This capability is also documented in the service specification of the component.

The second approach to creating a new stand-alone service is shown in Figure 2c. Here, the service logic, or code, of each service component is assembled within a new composite service. In general, all of the code from each component cannot be reused since certain methods are specific to an individual service or are not useful in the context of a composite service. For this reason, composable methods are identified in the service specification of each component. The appropriate sections of the service specifications from each component involved are also assembled to form the service specification of the composite service. The runtime creation of a new and functional service specification ensures that this new composite service has all of the basic attributes of any other service. This upholds the widely accepted principal that the composite should itself be composable [13].

The primary advantage of a stand-alone composite service is it can be reused and composed easily with other services. Reuse of a composite service interface is more difficult since the service components providing the functionality are not contained in a single entity. Another advantage is the new composite service will execute at a higher level of performance with regard to internal message transmission since all of the code is executing in the same location.

Constructing a stand-alone composite service at runtime is a very complex undertaking. While many of the processes common to both forms of dynamic service composition are still present (refer back to section 3), other challenges exist. The largest of these is to create a new functional service and successfully deploy it in a relatively short period of time. While the process of combining runtime services could be performed prior to when the service is actually needed, with the composite stored in a library for future use, we are more interested in determining to what extent the runtime construction of a composite service for immediate use is feasible.

Now that we are familiar with the terminology and processes of dynamic service composition, we can determine the requirements of a service composition architecture that will support the creation of such services.

3.3 Architectural Assumptions

We stated earlier that our goal is to design an architecture that makes use of existing technologies. We can justify this choice, over implementing a proprietary solution, since our prototype will not support components that were not originally designed to function as part of a composite service. This does not mean that all possible compositions have to be envisioned before the component can be designed. We still allow the content of the composite service to be determined at runtime. We simply define a set of requirements that each service component must satisfy in order for it to be used in our architecture.

3.4 Requirements of a Composable Service

A composable service component is both a valid service and a valid component. This means it has the properties of a component, as defined by a well-known component model, but it is optimized for use as a stand-alone service in a network environment. It has also been designed specifically to support composability.

The most critical element of a composable service component is the service specification. The service specification contains the inputs, outputs, dependencies and constraints of the service, in addition to a detailed description of the operations it performs. Another important feature is the composable service component must be easy to locate and retrieve. This is also facilitated by the service specification, which is
examined by the infrastructure during service lookup and retrieval. Finally, the code of each service component can be fully reusable or partially reusable. As we mentioned earlier, all composable methods must be clearly specified in the service specification of the component. It will be impossible for the infrastructure to determine at runtime if code within a service component can be reused unless it has been explicitly labeled as reusable.

3.5 Requirements of a Service Composition Architecture

An architecture supporting dynamic service composition must have a repository or library of composable service components. This library must allow a service, matching a well-defined set of attributes, to be retrieved in a relatively short period of time. To achieve this, the architecture must have a means of examining the service specification of a service component. This will require that the service specification be written using a well-structured description language to allow for straightforward parsing of the specification file. Finally, the architecture will require a valid component model in order to support component composition.

4 Proposed Architecture

Technologies are currently available that can be used to create an architecture to support dynamic service composition within a service domain. However, many of these technologies provide only a partial solution and must be extended and customized to work under the conditions outlined in this paper – conditions which were not necessarily anticipated during the design of these technologies.

4.1 Selection of Available Technologies and Required Extensions

We have purposely avoided developing or using a compositional language similar to Lava [7] or CHAIMS [1] since these languages are more appropriate for solving the problem of generic software composition. A compositional language facilitates the following activities: it allows components to be defined within existing non-segmented code, it allows the state and behavior of a component to be inherited by another component, it allows components to be dynamically adapted, or it allows components to be modified at runtime [2]. However, we are not interested in “componentizing” a monolithic software system. This is a completely different body of research with challenges that we feel will hinder our particular interests in dynamic service composition of composable components. Our approach does not facilitate the restructuring of a previously designed system into smaller, more manageable components but rather allows a new system to be designed so it can exploit the advantages of a dynamic component-based architecture.

We will now describe the support that several identified technologies provide for dynamic service composition and the extensions that must be made to them to arrive at a workable solution.

4.1.1 Jini

We have chosen to use Jini connection technology [12], developed by Sun Microsystems, as our service repository and retrieval system. We chose Jini because it is a distributed computing technology that facilitates the lookup and deployment of service components by providing the necessary networking infrastructure and distributed programming facilities. We have created a Jini service called the Composition Manager to oversee all aspects of dynamic service composition.

Jini (see Figure 3) is based on the Java language. This makes it ideal for use in our architecture since we are making extensive use of the JavaBeans component model, which we will discuss in the next section. It allows us to create a federation of software components in a network environment, where the components can dynamically connect to share services without any prior knowledge of the existing network environment. The core runtime infrastructure provides mechanisms for adding, withdrawing, locating and accessing services on the network. Service components will use this infrastructure to advertise themselves when they join the network, whereas the CM and other Jini services can use this infrastructure to locate and contact services.

The infrastructure of Jini consists of several parts. The Lookup Service (LS) is a bridge between services and clients and acts as a service broker component [10]. There are three protocols used by Jini:
Discovery, Join and Lookup. Discovery is used by a service component to locate the LS. Once a service component has discovered the required LS, it can register or advertise its services to the LS using the Join protocol. Jini stores each valid service component in the LS as a service item that consists of a service proxy object and several associated service attributes used to describe the available service. When a client locates a service matching its criteria using the Lookup protocol, it is the service object (proxy) part of the service item that actually gets downloaded to the client from a LS. The service object implements the various service interfaces that are used by a LS to identify and match a service component at the time of lookup. The Composition Manager (CM) uses these interfaces to interact with the actual service after a successful lookup. The CM specifies its requirement for a service using a service template. A service template specifies the type of service required and a set of service attributes. The LS returns service objects (proxies) to the CM that have correctly matched the service template.

Figure 3: Jini Infrastructure with Composition Manager

Jini technology uses the code mobility features of the enhanced version of Remote Method Invocation (RMI) for passing the service code to the CM. All objects are serialized and the location (URL) of its codebase is annotated to tell the CM where the object’s class files are located and how they should be retrieved.

4.1.2 JavaBeans

We assume the reader is familiar with the basic concepts in the JavaBeans component model. However, we will briefly describe the key features of the Extensible Runtime Containment and Services Protocol (ERCSP) for JavaBeans [11] since it provides the facilities we will use for dynamic composition.

The ERCSP standard extension provides an API that enables Beans to interconnect at runtime. It enables a Bean to interrogate its environment for certain capabilities and available services. This allows the Bean to dynamically adjust its behavior to the container or context in which it finds itself. The API consists of two parts: a logical containment hierarchy for Beans components and a method of discovering the services that are provided by Beans within such a hierarchy.

Figure 4: Fig. 4a shows a service component. Fig. 4b shows how BeanContexts can be nested at runtime. Fig. 4c shows Beans combining into a single composite service.
The containment hierarchy enables grouping of Beans in a logical manner which can easily be navigated. This grouping is established through the use of a BeanContext container. A BeanContext can of course contain other BeanContexts thus allowing for any arbitrary grouping of components. The Services API within the ERCSP gives Beans a standard mechanism to discover which services other Beans may provide and to connect to these Beans to make use of those services. Beans can use introspection to find each other’s capabilities.

Figure 4a shows how a JavaBean can be nested within a Jini service to create a service component. In this way we can use Jini for component storage and retrieval while taking advantage of the compositional features of the JavaBeans component model. We have shown the service specification in this diagram to highlight the enhancements we have made to the JavaBean. Figure 4b shows how a BeanContext can be introduced from one service component into another service component at runtime to create a stand-alone composite service. Figure 4c shows that a more reusable stand-alone composite service can be created where all of the code is contained within a single JavaBean.

4.1.3 eXtensible Markup Language (XML)

The eXtensible Markup Language (XML) is used to format data into structured information containing both content and semantic meaning [5]. A markup language is a mechanism to identify structures within a document in an easy and consistent way. XML uses a system of tags which are directives to the XML processor – the application that will eventually parse and display the document. We can use XML to describe how to format the data as well as the data itself (the semantics of the data). In other words, XML is most often used as a data description language. Another primary feature of XML, is it allows a user to create new tags, a feature which is not supported in HTML.

XML provides a convenient and highly effective way to encode a service specification in such a way that the Composition Manager can quickly determine the attributes of that service and the operations it can perform. Figure 5a shows how we have enhanced the Jini Lookup Service to include an XML parser. This will allow us to read the service specification stored in each service component at runtime. A limited example of an XML service specification is shown in Figure 5b.

```
<?XML version="1.0" ?>
<SERVICE>
  <DESCRIPTION>
    <NAME>Call Forward Unconditional</NAME>
    <VENDOR>Carleton University</VENDOR>
    <VERSION>1.3.2</VERSION>
    <PROTOCOL>H.323</PROTOCOL>
  </DESCRIPTION>
  <PROPERTIES>
    <COMPOSABLE>Yes</COMPOSABLE>
    <INPUTS>Caller, Call Agent</INPUTS>
    <OUTPUTS>Callee, Gatekeeper</OUTPUTS>
    <CHAINING_ORDER>First</CHAINING_ORDER>
    <COMPOSABLE_METHODS>Forward, Log Call Info</COMPOSABLE_METHODS>
  </PROPERTIES>
</SERVICE>
```

Figure 5: Fig. 5a shows the addition of an XML parsing facility to the standard Jini Lookup Service needed to interpret service specifications. Fig. 5b shows a simple XML service specification.

5 Conclusions and Future Work

This paper presents two approaches to dynamic composition of service components and a proposed software architecture to support these techniques. A composite service interface can be created if the composition needs to be carried out in a relatively short period of time. However, there is limited reuse potential for this service since the service components involved in the composite service may be distributed on several network nodes. The advantage of interface fusion is that it is relatively straightforward to assemble the interface and deploy the composite service.

The second method is to create a stand-alone composite service. This is considerably more difficult since code must be physically moved from one component into another. It takes a much longer time to
construct a stand-alone composite service since the service specification must be created from the service specifications of the member components and a completely new component must be assembled. The advantage, however, is that this new service can be stored in a service component directory for future use. It is a valid service component just like its member service components and therefore has reuse potential.

Currently, a prototype of our composable service architecture is under development. We are instrumenting our system with performance metrics and plan to carry out a scalability analysis in an effort to quantify the limitations of our proposed solution. We realize that scalability is already an issue with Jini since the technology is targeted to a network (or workgroup) of one thousand nodes in size. Therefore, the applicability of Jini in larger networks is doubtful. However, this problem can be handled to some extent with a hierarchical infrastructure organization. Once our prototype implementation is complete, we will focus on developing applications of dynamic service composition such as IP telephony and multimedia service composition and the runtime assembly of new network management services.

Acknowledgements

The authors would like to Babak Esfandiari for his helpful comments. The research in this paper is supported by Communication Information Technology Ontario (CITO).

References