A Runtime Composite Service Creation and Deployment Infrastructure and its Applications in Internet Security, E-Commerce, and Software Provisioning

David Mennie[1] and Bernard Pagurek[2]
[1]*The Bulldog Group Inc., Toronto, ON, Canada*
[2]*Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada*
dmennie@bulldog.com, 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, a new common interface is constructed at runtime which allows the functionality of two or more components to be accessed from a single entity while the service components themselves remain distinct and potentially distributed within in a network. In the second approach, a new composite service is formed where all of the functionality of the service components is housed within a single new service. In the third approach, a new composite service is formed where all the functionality of the service components is extracted and re-assembled into the body of a single new service.

This paper describes the design of an infrastructure to support the runtime creation of composite services. An application to create user-defined security associations dynamically and deploy them between any two points in the Internet is presented to exemplify the need for dynamic service composition techniques. Some other potential applications in e-commerce and software provisioning are also discussed.

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 [3]. This can be done either statically at design-time or load-time, or dynamically at runtime. Statically 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.

One of the key definitions in dynamic service composition is that of a service component. Our definition of a *service component* is based on ideas from many different sources. The authors of the TINA (Telecommunications Information Networking Architecture) service architecture [7] state that a service component is a self-contained unit of service construction that provides an identifiable and distinct part of a service. They also state that a service component has an accessible interface. We extend this definition to include that a service component consists of a self-contained body of code with a well-defined interface, a set of attributes, and a behavior.

Service components are the basic elements or building blocks that can be used to construct services. A service component must have a name, properties, and an implementation. The properties include a description of the component which may include operational constraints, its dependencies (if any) on other components or infrastructure, a list of operations that can be reused or composed with other components, a description of the functionality of the component, a list of known relationships that it can form with other components, and any other relevant information. The specification may also contain a description of the behavior of the service component by annotating the contained operations or methods using a formal language or structured syntax. The interface used to access the component may be described directly in the specification or indirectly discovered through reflection and introspection assuming the programming language used to implement the underlying component has support for these features. This definition is quite broad and thus allows a wide range of components to fall within its scope.

A *service*, much like a service component, is an entity that has a well-defined interface and behavior. The important characteristic that distinguishes a service from a component is its visibility to the end-user. A service can be referenced by a user (i.e., it is visible) and a service component cannot be directly referenced by a user. In the service

composition architecture, defined later in this paper, only the system infrastructure is permitted to interact directly with the components based on the user's requests. Individual components may also be classified as services if they meet the requirements of both definitions and thus may provide functionality directly to a user. In general, services can be created by putting multiple components together using one or more mechanisms called composition methods. Such services are referred to as *composite services*.

Several recent industrial initiatives, most notably Microsoft's .NET and Sun's Open Net Environment (ONE), are based on the concept of a Web service. A Web service is more like a service component (according to our definition) since it communicates by means of XML-based standards. Our work is not bound to Web services because we want to research issues that are independent of the communication mechanisms used. We concentrate our efforts on researching issus that are currently not addressed by the industrial initiatives.

A *composition method* is the technique used for creating services from service components. The syntax and semantics of the detailed procedures needed to successfully form composite services are described in the composition method. The composition method also gives the requirements for the component's specification but not its implementation. Techniques for service composition, in particular dynamic service composition, will be examined in the next section.

2. Dynamic Service Composition

Composing services at runtime has some elements in common with static composition but it also has some unique objectives. Generally dynamic composition focuses on adapting running applications and changing their existing functionality either by adding new features or removing features. Locating components at runtime requires a component library or code repository that is integrated with the software infrastructure that is actually performing the composition. In other words, the system must be able to access this repository since dynamic composition is generally an automated process.

2.1. Why Dynamic Composition is Inherently Difficult

Composing service components at runtime is a challenging undertaking because of all the subtleties of the procedure involved, the many exceptions to the compositional rules that can occur, and the potential for error. The challenge lies in dealing with the unexpected issues and incompatibilities that arise during assembly in a relatively short period of time so dynamic composition remains practical. Problems can be caused by issues related to non-functional requirements in addition to functional requirements. Before the actual composition is performed, it is important to ensure that the constructed software will satisfy the specified requirements [9]. In dynamic composition, as we have defined it, it is extremely difficult to predict beforehand the exact environmental conditions that will exist in a system at the time a composition is performed. We call this *unanticipated dynamic composition* [3], meaning that all potential compositions are not known and neither the service components nor the supporting composition infrastructure are aware if a particular composition will be successfully completed until it is actually carried out.

While steps are taken to decrease the chance of a failed composition, it cannot always be avoided. One of the measures taken to avoid complications is to bundle a service specification with each service component that describes the dependencies, constraints, or potential incompatibilities for the component. By looking at the specification for each component of interest before attempting to use them to create a composite service, failed attempts can be minimized or recovered from.

Despite these error handling mechanisms, potential behavioral interactions within the new composite service, between the operations extracted from the original components, may surface even if the structural composition is successful. The problem is similar to a program that compiles without errors but still fails to execute properly. Compilation is only one part of the successful execution of a program just as the composition process will not guarantee the composite service will function correctly. By making sure the operations of each component are well documented and accessible, runtime interactions can be minimized. When interactions arise despite a successful structural composition, it is almost impossible for the composition infrastructure to correct the situation. It is the responsibility of the user to determine if the side effects are neutral or service affecting. If the interactions cause the composite service to function incorrectly or behave erratically, the service can be terminated and never reassembled. However, if the interactions that do occur do not seriously affect the operation of the composite service, they can simply be ignored.

2.2. The Case for Dynamic Service Composition

Kniesel [3] provides an excellent example of the need to perform unanticipated, dynamic changes to a system without discontinuing its operation. He reminds us that the recent change from national currencies to the Euro, in many countries in the European Union, could not have been anticipated. The software used by banks, insurance

companies, and other financial institutions providing round-the-clock service had to be changed to the Euro while trying to limit customer impact. Had these software systems provided support for dynamic adaptation, these changes could have been made efficiently and with minimal service interruptions. However, many banks needed days to make the conversion and many were not able to switch over at all.

There are several key benefits to dynamic service composition. The most immediate is the fact that applications have greater flexibility since new services can be constructed to address specific problems if they do not already exist. Another benefit is users do not need to be interrupted during upgrades or the addition of new functionality into a system. In other words, a large set of services can be created from a set of basic service components. Also, services can be assembled based on the demands of an application or a user. For example, if a user requires an Internet search engine that will filter out advertising from the results returned for a particular query, the service can be assembled at runtime and sent to the user. This service may not have been designed or even conceived ahead of time.

2.3. The ICARIS Architecture

The fundamental challenge in composing services at runtime is the design and implementation of an infrastructure that will support the process. Recently, we designed and implemented an architecture called the **I**nfrastructure for **C**omposability **A**t **R**untime of **I**nternet **S**ervices (ICARIS) [5]. The architecture provides all of the required functionality to form composite services from two or more service components that have been designed for composability.

There are three primary composition techniques that are supported in the architecture [4, 5]. The first technique allows the creation of a composite service interface. The *composite service interface* allows a service component to remain distinct while providing some or all of its functionality through a common interface that it shares with other service components. In effect, clients can be presented with a unified interface that makes it appear as though they are communicating with a single service when in fact their method calls are just being intercepted by the composite service interface and redirected to the appropriate service component. To realize this composite interface, the architecture extracts the signatures for the composable methods from each component and combines them into a single interface. This interface will accept method calls for any operations provided by its service components and direct the calls to the appropriate component.

The second technique allows *stand-alone composite services* to be created. Here, a set of service components is interconnected to create a new stand-alone service. The service components are assembled using a pipe-and-filter architecture that basically chains the output of one service component to the input of the next.

The third technique facilitates the creation of a *stand-alone composite service with a single body of code*. This means the composable methods for each service component, involved in the composite service, are extracted and assembled into a new third component. The service specifications from each of the service components are also merged into a new composite service specification. One of the motivations for creating a new third component with a single specification and a single set of methods is to evaluate the performance of this structure. A composite service constructed using this technique often takes longer to create than using other techniques but it also tends to execute much faster. It is also the most challenging type of composite service to create so it is used to define extra requirements for the design of the system architecture. Further detail on the architecture can be found in [5].

3. Applications of ICARIS

3.1. Composable Security Application

One major focus of the computer industry today is to establish a foundation for “trust” as it relates to the e-commerce environment. The primary goal of the prototype implementation described in this paper is to demonstrate how dynamic service composition technology can be used to increase the level of trust that users have towards their online relationships with other users, Internet service providers, vendors, and information sources.

3.1.1. Internet Security and Trust Issues in E-Commerce

Many Internet users are wary of conducting business or purchasing goods and services online. These users feel the absence of two people meeting and engaging in a vendor-to-customer relationship prevents them from properly evaluating how the business operates and judging if it is legitimate and safe. The constant threat of computer viruses, potential business scams, and fraud add to this lack of human interaction and are enough for users to be reluctant to embrace new uses of the Internet. It is for this reason that the parties involved in an online exchange of goods and services must trust each other implicitly before any substantial increases in Internet business will take place.

Security at the level of infrastructure is widely available today. The majority of businesses employ one or many security techniques in an attempt to gain the customer’s

trust. However, it is not the security of the network link that is the biggest challenge. The most important issue facing the Internet today is how to deploy security where it is needed, when it is needed, in the shortest time possible, and in as efficient and seamless a manner as possible. Because the majority of Internet users cannot possibly understand the intricacies of a security protocol, even the most robust and well-known security algorithms do little to increase a user's notion of trust. Infrastructure-level security provides the user with a finite level of online comfort that will not increase despite improvements in the underlying protocols used. It is only through easy to understand and interactive improvements in security that further trust can be obtained.

A software system that can quickly and easily deploy security software to both the originating and the receiving ends of a transaction in a manner that is seamless to the end users would allow a greater level of trust to be achieved in online business. It is this problem that our ICARIS-based prototype was developed to solve.

3.1.2. Dynamic Composition of Internet Security Services

The Composable Security Application (CSA) prototype is a Jini, JavaBeans, and XML-based implementation of the ICARIS architecture. The goal of the prototype is to allow the construction and deployment of security associations between a client and server in the network in order to allow security services to be introduced into applications that do not already have access to security.

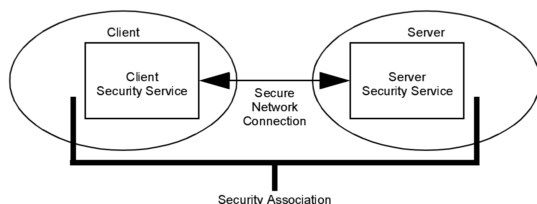

Figure 1. Security Association

A *security association* is a relationship between two network devices that allows the devices to exchange secured information. For example, after two network nodes form a security association, the units can send encrypted information to each other and decrypt each other's messages. This is illustrated in Figure 1.

In the CSA, the specification of security associations is carried out at runtime. This means the composite client and server security services can be constructed dynamically. The composite services are made up of service components that each contain a cryptographic algorithm or related piece of security infrastructure. These service components are assembled by ICARIS to construct the appropriate composite security services based on the demands of the client and server for a particular security association. The CSA is an accessible, robust infrastructure that supports pluggable security for any application. It is able to establish many types of security associations. Additionally, it is both fast enough to assemble and deploy these associations at runtime and flexible enough to add or remove secure services to meet the applications it serves.

3.1.3. Secure Software Provisioning

In this section, we provide a scenario for how the CSA could be used for secure software provisioning over the Internet. In our example, a customer wishes to purchase software from a major vendor. To increase the customer's trust in the vendor, the customer can use the CSA to choose the security measures he wishes to put in place between the vendor and himself.

The CSA has a set of predefined security standards for different types of transactions stored locally. It updates the customer's user interface with a set of security choices based on the type of transaction selected and the security requirements of the vendor. In this case, the customer requests that a digital envelope be deployed and this requirement is sent to the CSA. The digital envelope is an example of how a symmetric key cryptography algorithm and a public key algorithm can be combined into a single service.

The service requirements for a digital envelope are placed into an XML-based service template by the CSA. This template is sent to a service broker to retrieve the required security services. If the service broker locates one or more service components that satisfy the service template, the service objects (proxies) for these components are returned.

For the digital envelope, in this example, the following service components are required for the client:

- A service component that can generate an IDEA (International Data Encryption Algorithm) session key. IDEA is a commonly used symmetric key cryptography algorithm.
- A service component that can encrypt the user's credit card information using the IDEA algorithm and the IDEA session key
- A service component that is able to obtain the vendor's RSA (Rivest-Shamir-Adelman) public key. Named after its creators, RSA is a commonly used public key cryptography algorithm.
- A service component that can encrypt the session key using the RSA algorithm and the vendor's RSA public key

- A service component that can send the encrypted credit card information to the vendor's composite security service

The following service components are required for the vendor:

- A service component that can generate a RSA key pair (public key and private key)
- A service component that can receive an encrypted message from a client
- A service component that can decrypt the session key using the RSA algorithm and the vendor's RSA private key
- A service component that can decrypt the message using the IDEA algorithm and the session key

Once this minimum set of services is obtained, the CSA will form a client and server pair of composite services from these service components. The client and server services usually differ in the order the service components are assembled. For example, if the client service encrypts some data with one algorithm (Alg1) and then a second algorithm (Alg2), the server service must decrypt that data in the reverse order (Alg2 then Alg1). For this reason, the service components must be assembled in the opposite order for the server service.

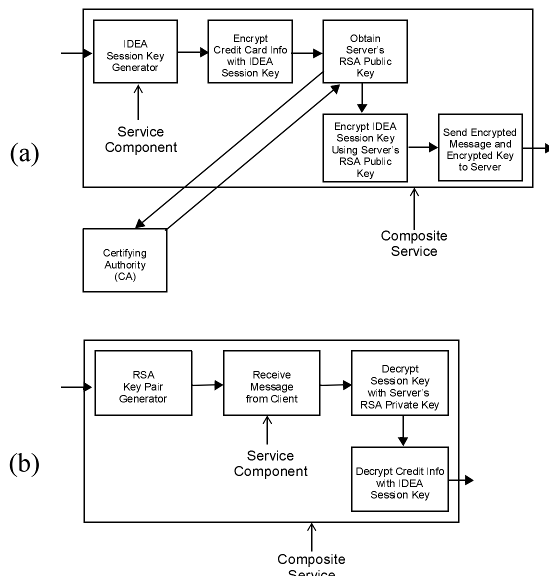

Figure 2. Client (a) and Server (b) Composite Security Services for a Digital Envelope

Figure 2a shows a logical view of how the client composite security service will be constructed. It will be assembled as a stand-alone composite service. Figure 2b shows a logical view of how the server composite security service will be constructed. It will also be assembled as a stand-alone composite service. Note that the RSA Key Pair Generator in the server security service must generate a public and private key before the client can use the server's public key to encrypt the session key. Also, note that the service components in the server service are assembled in the reverse order to their complements in the client service. This is needed because the session key must be decrypted before it can be used to decrypt the message.

Finally, the composite services are deployed from the CSA to the client and server nodes. The services are instantiated and the setup phase begins. At this stage, the client service establishes a connection with the server service directly using the protocol agreed upon in the requirements definition stage. At this point, the customer is informed that the security association is established, and business can proceed.

3.2. Other Applications

There are many other applications that could take advantage of the ICARIS architecture. A brief description of some of these applications will be provided in this section.

The idea of constructing security associations dynamically could be extended to virtual private networks (VPNs) or other communication channels. The ICARIS architecture could be used to deploy client and server services to the front and back ends of the communication link that could consist of many service components including QoS components, security components, network management components, and billing components. Custom networking services that can be created and deployed at runtime would be quite useful since VPNs often need to be set up temporarily and on short notice.

Network management services could also be deployed to a problem node based on the requirements that are needed at a particular time. For example, if a network outage is detected, a remote maintenance service could be constructed out of various service components that would perform a battery of network diagnostic tests. These tests could be selected by a network operator based on the problem at hand. Other service components that could be used in network management could include services that gather network metrics (throughput, traffic density, congestion, delay, response time), locate faults (i.e., pinpoint exactly where the network is failing), diagnose faults (i.e., determine the cause of the problem in the network), or reconfigure a network agent such as an SNMP agent.

4. Conclusions and Future Work

In this paper, we summarized our experiences with the

design and implementation of an architecture (ICARIS) to support the dynamic composition of service components. We discussed our Composable Security Application and its potential impact on e-commerce. We also suggested several other potential uses for ICARIS.

One area of future research is to incorporate behavioral service specifications into ICARIS service components. A behavioral specification is the formal description of what is supposed to happen when software executes. The ICARIS architecture currently makes use of an XML-based service specification that mainly captures structural information about a component but not behavioral information. The architecture could employ a behavioral specification to verify, statically or at runtime, that the software meets it requirements. This would also allow the architecture to make more intelligent choices about which components it should select for a particular composite service. Interactions could be minimized between components because the behavior from each component could be understood by the infrastructure.

Currently, behavioral specifications and formal methods are not widely used by programmers because the development tools are immature and difficult to use. Also, most programmers do not have a sufficient mathematical background to be comfortable with using the notations required in most behavioral specification languages. If behavioral specification was contract-based, however, it would be more accessible to programmers. A design by contract model views the relationship between a class and its clients as a contract that takes the form of assertions such as boolean invariants, preconditions, and postconditions. Boolean invariants and preconditions document the contractual obligations a client must satisfy before calling an operation in a class. When the client fulfills its obligations, boolean invariants and postconditions document the class supplier's contractual obligations for how the operation must behave and what it must return.

As we continue to move toward component-based software engineering, software development will become less creative and more routine. The task of the developer will shift from coding to designing and integrating [8]. There is an opportunity for the integration of components to be done at runtime and potentially automatically with the help of a composition infrastructure such as ICARIS, assuming it is practical. However, a full quantitative evaluation of the ICARIS architecture must be carried out in order to accurately assess the feasibility of the various composition techniques, as well as, their performance and scalability characteristics. Many additional challenges will need to be overcome before we can reap the full benefits of dynamic service composition.

Acknowledgments

The authors would like to thank Vladimir Tosic for his helpful suggestions and comments. The research in this paper is supported by Communication Information Technology Ontario (CITO).

References

[1] Feng, N., G. Ao, T. White, and B. Pagurek, "Dynamic Evolution of Network Management Software by Software Hot-Swapping", Accepted for the *Seventh IFIP/IEEE International Symposium on Integrated Network Management - IM 2001*, Seattle, Washington, USA, May 14-18, 2001.

[2] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[3] Kniesel, G., "Type-Safe Delegation for Run-Time Component Adaptation", In R. Guerraoui (Ed.), *Proceedings of the 13th European Conference on Object-Oriented Programming - ECOOP '99*, Springer, Lisbon, Portugal, June 1999.

[4] Mennie, D., and B. Pagurek, "An Architecture to Support Dynamic Composition of Service Components", Presented at the *Fifth International Workshop on Component-Oriented Programming - WCOP 2000*, held in conjunction with *ECOOP 2000*, Sophia Antipolis, France, June 2000.

[5] Mennie, D. W., "An Architecture to Support Dynamic Composition of Service Components and Its Applicability to Internet Security", M.Eng. thesis, Carleton University, Ottawa, Canada, October 2000.

[6] Oreizy, P., M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp.54-62.

[7] *TINA Service Architecture Annex*, Kristiansen, L. (Ed.), Version 5.0, June 16, 1997.

[8] Voas, J., "Maintaining Component-Based Systems", *IEEE Software*, Vol. 15, No. 4, July/August, 1998, pp. 22-27.

[9] Yau, S. S. and F. Karim, "Component Customization for Object-Oriented Distributed Real-time Software Development", *Proceedings of 3rd IEEE International Symposium on Object-oriented Real-time Distributed Computing*, March 15-17, 2000.