Dynamic Service Composition and Its Applicability to E-Business Software Systems – The ICARIS Experience

Vladimir Tosic[1], David Mennie[2], Bernard Pagurek[1]

[1] Department of Systems and Computer Engineering, Carleton University
1125 Colonel By Drive, K1S 5B6, Ottawa, Ontario, Canada
{vladimir,bernie}@sce.carleton.ca
[2] The Bulldog Group Inc.
184 Front St. E., Suite 300, M5A 4N3, Toronto, Ontario, Canada
dmennie@bulldog.com

Abstract. This paper discusses dynamic service composition and its applicability to e-business software systems. Dynamic service composition is the process of creating new services at run-time from a set of service components. It supports business agility, flexibility, and availability – the features of critical importance in the modern business world and in e- and m-commerce software systems. However, dynamic service composition is challenging and requires addressing a number of issues. The paper summarizes experiences and uncovered issues from the development of the ICARIS architecture for dynamic service composition. ICARIS is a general-purpose dynamic service composition architecture based on Jini, JavaBeans, and XML. The paper also presents application of ICARIS to dynamic, on-demand establishment of security and trust relationships. Addition of security extensions to a B2C e-commerce system originally designed without appropriate security mechanisms illustrates dynamic service composition in ICARIS and its applicability to e-business software systems.

1. Introduction

Dynamic service composition is the process of creating new services at runtime from a set of service components. This process includes activities that must take place before the actual composition such as locating and selecting service components that will take part in the composition, and activities that must take place after the composition such as registering the new service with a service registry. A very important characteristic of dynamic service composition is that the new composite service need not be envisioned at design time. This feature, known as unanticipated dynamic composition [7], provides considerable flexibility for modifying and extending the operation of software systems during runtime. However, it also introduces a number of complications and problems for designing and operating software systems that support dynamic service composition. In this paper, we will describe our experiences with dynamic service composition and discuss how it can be used to improve the agility, flexibility, and availability of business software systems, particularly for e- and m-commerce systems. Before we proceed, let us first define the concepts of a service component and a service.

A service component is a self-contained unit of service provisioning and management that encapsulates some service functionality and appropriate data, and can be composed with other service components to form different services. A service component is relatively independent from other service components in a particular composition – it can be relatively easily detached and replaced with another appropriate service component. Also, it can be reused in many different service compositions. A service component has a well-defined interface, properties describing the component, and behavior. The properties specified for a service component may include: operational constraints, dependencies on other components or infrastructure, a list of operations (called composable methods) that can be reused or composed with other components, a description of the functionality of the component, a list of known relationships that can be formed with other components, and any other relevant information. The service 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 discovered indirectly through reflection and introspection facilities, assuming the programming language used to implement the underlying component has support for these features. It is important to note that a service component can provide not only software functionality and data, but also access to some hardware resources like memory, printing, network bandwidth, etc. It can eventually be implemented with one or more interacting software components, with one or many cooperating distributed objects, with some combination of software components and distributed objects, or with some other technologies, for example, mobile code. The given definition of a service component is intentionally quite broad and allows a wide range of components to fall within its scope. However, in this paper we are predominantly interested in software (a.k.a., algorithmic) service components.

The important characteristic that distinguishes a service from a service component is visibility to an external user (either human or an external software system). An external user can reference a service, not a service component. Only system infrastructure is permitted to interact directly with service components and to compose them into composite services. Note however that some units of service functionality can have both properties – visibility to an external user and being a part of composite services. We refer to such a unit of service functionality as a "service" in situations when it is used by an external user, and as a "service component" in situations when it is used as a part of a composite service.

2. Dynamic Service Composition

Component-based software engineering is widely recognized as a very important improvement in engineering of complex software systems. Among its advertised benefits are [12]: rapid software development, enhanced adaptability, scalability, and maintainability of resulting software systems. However, composition of software components during system design time and/or deployment time (a.k.a., static software composition) is not flexible and agile enough in cases when there are frequent runtime changes of requirements and/or operational circumstances that cannot be anticipated.

Static software composition is sufficient for constructing applications with well-defined specific requirements that are not likely to change frequently. If a software system has a loosely defined set of operations to carry out or it has to adapt to relatively frequent changes in the environment that might not even be predicted during design time, static software composition is too limited. Redesigning the system to accommodate the changes often requires considerable human involvement, which significantly slows down the overall reaction to change. Further, modifying or updating statically composed software usually requires disrupting its operation, which is not suitable for high-availability, mission-critical, and hard real-time systems. As will be discussed later in this paper, many business systems would benefit from greater runtime flexibility, agility, and availability of software systems.

Dynamic composition of software service components is an important step forward in achieving these goals. It enhances flexibility of software systems since it enables the runtime construction of new services, if they do not already exist, to address a specific problem. A number of useful services can be composed from a set of available service components. The set of potential service compositions grows exponentially with the size of the set of available service components. Some of these service compositions may not have been conceived of ahead of time. The services can be assembled based on the demands of the system or its users. The involvement of humans in the composition process is minimized. The users do not need to be interrupted during upgrades or the addition of new functionality into the system. To conclude, dynamic service composition supports rapid and autonomous (i.e., with minimal human involvement) adaptation even to some changes not envisioned during design time, while keeping the running software system constantly available to users.

Of course, dynamic service composition has certain limits. Not every new service can be realized as a combination of existing service components. Sometimes newly required functionality is significantly different from available service components. However, even in such cases dynamic service composition can be a useful element of the broader solution for dynamic system evolution. Encapsulation of the new functionality inside a new service component not only minimizes disruption of the operation of the running system and its users; it also enables a potential chain reaction of composing new services including the newly added service component.

In addition, dynamic service composition is a very challenging undertaking and there are a number of issues to take into consideration. It has some elements in common with static service composition but it also has some unique features. One of these features is the crucial nature of time limits. The dynamic service composition process often must complete within some specified, relatively short, time limits or it becomes impractical. Generally, it is an automated process with limited human involvement. There are many potential problems, exceptions, and errors that may occur during this process. The challenge lies in dealing with these unexpected issues in the limited time frame that is permitted for a particular composition. Also, it is not possible to precisely predict or test at design time what the exact environmental circumstances of operation will be at composition time and whether the process will be successful. While steps are taken to decrease the chance of a failed composition, it cannot always be avoided. Furthermore, even if the dynamic composition process seems successful, there is the potential for unexpected feature interactions that cannot be easily and rapidly discovered and recovered from. A feature interaction is the way a service

component (i.e., a feature) modifies or affects at runtime the behavior of other service components in a particular composition. 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. When unexpected feature interactions arise despite all measures taken to avoid them, it might be almost impossible for the composition infrastructure to correct the situation. Human (i.e., user) input is needed to determine if the side effects are neutral or service affecting. If the feature interactions cause the composite service to function incorrectly or behave erratically, the composite service can be terminated and never reassembled. However, in many situations it may be appropriate to simply ignore those feature interactions that do not seriously affect the operation of the composite service.

There is also a lack of support for dynamic composition techniques in programming languages and other development tools. The fundamental challenge in composing services at runtime is the design and implementation of an infrastructure that will support the process. Locating components at runtime requires a component library or code repository that is integrated with the software infrastructure that is actually performing the composition. The infrastructure should also support mechanisms to recover (e.g., rollback) from an unsuccessful composition and to discover and, if possible, recover from unexpected feature interactions. All these and other issues make the dynamic composition process inherently complex. Consequently, cost-benefit analysis must be taken into consideration before applying dynamic service composition techniques to a particular circumstance.

3. On E-Business Applications of Dynamic Service Composition

Unanticipated dynamic service composition has increasing relevance in software today because of the constant change and evolution of technologies, protocols, and standards. It can be a very important mechanism for use in mission-critical, high-availability, and hard real-time e-business systems and in other systems where there is a need to perform unanticipated changes to software without discontinuing its operation. Many e- and m-commerce systems fall into this category.

The issues related to the establishment of security and trust relationships in e- and m-commerce systems are becoming increasingly important. E-commerce requires security requirements that are above and beyond the requirements for traditional network security. It also requires mutual trust between a vendor, a customer, and all software and human parties involved. The issues related to the establishment of security and trust relationships have been proven to be major limiting factors to the growth of the Internet economy. As a very large number of different security mechanisms have been developed and new ones constantly emerge, the majority of businesses now employ one or more of these mechanisms, e.g., to secure the network link. However, the biggest challenge is now 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. Additionally, the challenge is how to introduce appropriate new security mechanisms into those business systems that were developed with inadequate security

or with old security mechanisms that are now deemed insufficient. Dynamic service composition can be used to achieve these goals and to alleviate four important issues. First, security is too often an "after-thought" during system development. Second, many e-commerce applications require specific security infrastructure components to be constantly available and running. Third, the security requirements of a system's users may change during runtime and can be quite different for different classes of user. Fourth, the broad range of applicable security protocols and algorithms, as well as their ever increasing variety, often makes it impractical or even impossible to build in comprehensive security support at design time. For example, it might not be financially justifiable for a smaller company to put a commercial Public Key Infrastructure (PKI) in place to meet the demands of a low volume of users with different security requirements. In such cases, using third-party pay-per-use security service components might be a more flexible and financially justifiable approach. The customer demanding the added security can be additionally charged to cover the costs.

Dynamic service composition can be also helpful in customizing services to various devices. This is a very important issue for emerging m-commerce systems. The variety of platforms for executing software, especially the lightweight platforms for mobile users, has increased during the last couple of years. Client-side software, for example m-commerce software, can now execute on cell phones, personal digital assistants (PDAs), digital pagers, laptops, personal computers, and many other devices. These devices differ in processing power, memory capacity, screen real-estate, graphics capabilities, networking capabilities, and many other features. Depending on which platform an application executes, different components of an m-commerce application may or may not be required. If the m-commerce application is component-based with different service components serving as service engines for the application, a user could move the application between devices and the application could scale to the device that it is running on at any given time. The composition of service engines could be performed dynamically using the appropriate infrastructure for dynamic service composition.

Dynamic service composition can also be used for establishing and managing value nets, the evolution of supply chains. A supply chain is defined as a sequential, pipeline, flow of value-adding activities from vendors to ultimate consumers necessary to produce a product or service effectively and efficiently. A value net bypasses this sequential structure by establishing a virtual community of supply chain partners that jointly mine the knowledge necessary for the development and marketing of their products. This enables delivering superior service and perfect customized orders that satisfy customers and differentiate involved companies from the competition. Dynamic service composition directly enables creating value nets for information-oriented products and services. But even more importantly, it provides agility, flexibility, and availability of e-business services useful in many value nets.

4. The ICARIS Architecture

We believe that the design and implementation of an appropriate infrastructure is the most important current challenge for the work on dynamic service composition. We

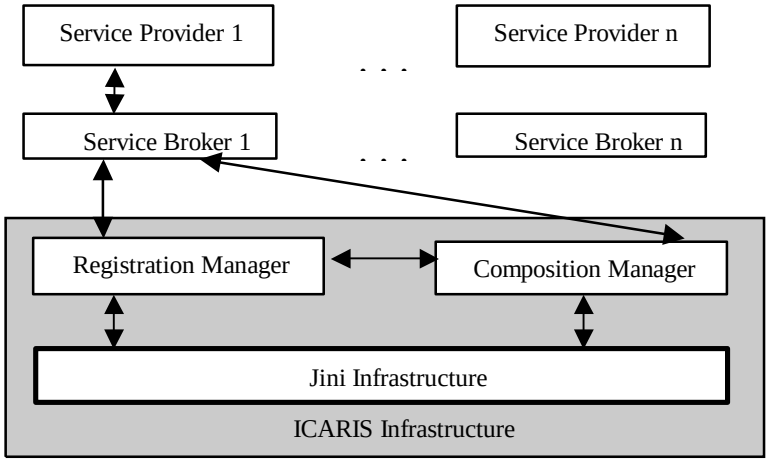

Fig. 1. High-Level Architecture of ICARIS

designed and implemented a general-purpose dynamic service composition architecture called the Infrastructure for Composability At Runtime of Internet Services (ICARIS) [11]. The architecture provides all of the required functionality to form composite services from two or more service components that have been designed for composability. A major contribution of this project was to design an architecture that makes use of the existing network computing technologies and carries out runtime assembly of services without the need for new compositional languages or infrastructures based on non-standard software. Our opinion is that dynamic composition techniques will not be embraced and widely deployed if they are based on software that is too specialized and without support of important market players. Therefore, for implementing the ICARIS architecture we have used proven and widely used technologies: the Java programming language, the Jini distributed computing technology, the JavaBeans component model, and the XML (eXtensible Markup Language) information specification standard. While these base technologies were not altered, extensions to the Jini infrastructure were made to enable the Jini Lookup Service (LS) to support service items with XML-based service attributes instead of just simple text-based attributes. Component composition is achieved at runtime by using an application of JavaBeans and the Extensible Runtime Containment and Services Protocol (ERCSP). As many features of JavaBeans, Jini, and XML were not used for ICARIS, the architecture is perhaps more heavyweight than necessary. However, many other technologies were evaluated before this Java-centric approach was selected for the final implementation. In the process of developing the architecture many problems were solved that are largely independent of the underlying implementation technologies.

There are three primary composition techniques that are supported by ICARIS [10, 11]. First, a composite service interface can be created based on several service components by extracting the signatures of the composable methods from each component and combining them together into a single interface. A composite service interface

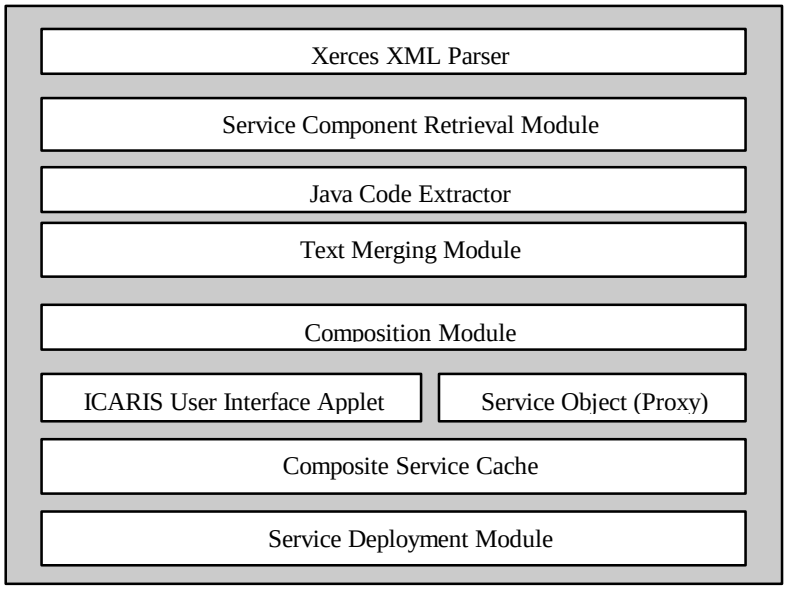

Fig. 2. Architecture of the ICARIS Configuration Manger

can be created quickly, but is not actually a new service. This composition technique provides a loose aggregation of service components using the Façade pattern. Service components involved in the composition remain distinct, while communicating with clients through the common composite service interface. The composite service interface redirects all incoming calls to the appropriate service component for execution.

Second, a new stand-alone composite service can be created by means of using a pipe-and-filter architecture to interconnect service components. In essence, the pipe-and-filter architecture chains the output of one service component to the input of the next. While this is a fairly primitive connection scheme, some complex constructions are also possible. For example, the outputs of one component can be looped back into the inputs on the same component. Connection Services, retrieving the output from one component and sending it to the input of another, are automatically created by ICARIS. Note that one of the benefits of using a pipe-and-filter architecture is minimization of unexpected feature interactions. This composition technique requires longer time than a composite service interface, but it creates a single new composite service without altering the structure or logic of the composed components. The coupling between service components is stronger than with a composite service interface.

Third, the creation of a new stand-alone composite service with a single body of code is achieved by extracting and assembling the composable methods from software-based service components involved in the composition. The corresponding method signatures are also merged into a new composite service specification. This code composition technique creates a new fully functional and reusable service with a

strong coupling between the code of the composed service components. However, compared to the other two techniques, this is a very complex and lengthy technique that cannot be used for composing the majority of service components. It can be performed only with white-box software service components, not with other types of service components. Performing this composition technique might have sense when performance of the composite service is a key consideration. In theory, a composite service with a single body of code may take longer to create than the other types of composite services, but it should also execute faster. The gains in performance are particularly significant when the composed service components are distributed.

Note that other composition techniques, such as a stand-alone composite service with some service components processing the same input in parallel, are not supported by ICARIS because they could produce unexpected feature interactions. For many applications such composition techniques are not necessary. However, it seems that for general e-business dynamic service compositions parallelism of execution would be a powerful feature, if appropriate mechanisms for handling unexpected runtime feature interactions were used. This is one of the weaknesses of ICARIS.

ICARIS consists of the Jini infrastructure, the Registration Manager, and the Composition Manager (Figure 1). The Registration Manager is the entity that is responsible for managing registration and access rights. This includes registration of clients, servers, and service brokers. The Composition Manager is the entity that is responsible for the actual dynamic service composition in the system. Its architecture is given in Figure 2. Two other elements called the Service Broker and the Service Provider are also required but they are not considered to be part of the ICARIS infrastructure and they can be provided by third parties.

Service components are provided by a Service Provider and are stored with a Service Broker within a structure called a Service Item. Service Items could also be stored in a Jini LS for use by other clients and servers. A Service Item is made up of two major parts: a Service Specification and a Service Object (Proxy). The Service Object is a valid JavaBean. As every service component must be accompanied by semantic information, the Service Broker requires the Service Specification written in XML instead of a simple list of text attributes. After the Service Item is successfully located in the Service Broker's repository, the Service Object can be downloaded to the entity requesting it. To support composition of new stand-alone services with a single body of code, ICARIS also requires access to the raw source code for the Service Object as well. This cannot be bundled with the Service Component so it is stored in a separate repository in the Service Broker. The source code is uploaded to the Broker at the same time as the Service Item.

The Service Broker is used to store and retrieve Service Items. The Service Broker is an enhanced Jini LS that is able to parse and interpret an XML Service Specification with its embedded Xerces Java XML parser [1]. The Service Broker maintains the functionality of the original Jini LS but, apart from the Jini LS simple exact matching mechanism, it also supports a more advanced "fuzzy" matching mechanism. As it can be used independently from ICARIS, the Service Broker is not part of ICARIS. Namely, the Service Broker is capable of storing any Internet service, which may or may not be a composable service, provided by a Service Provider. On the other hand, ICARIS only retrieves a subset of these services that are service components designed for composability. The Service Providers are the sources of service

components in the network. The Service Provider uploads Service Items into the Service Broker. There can be multiple Service Providers registered with a single Service Broker. The Service Provider registers and interacts only directly with the Service Broker and not with the ICARIS infrastructure.

Apart from the mentioned limitations of the supported composition techniques, the main limitations of ICARIS are consequences of the limitations of the used Java technologies. However, note that no other collection of technologies existing at the time of the ICARIS project was able to achieve the set goals. One limitation is the scalability of Jini. Jini was originally designed for resource sharing within a typical enterprise work group of about 10 to 100 people because people tend to collaborate with those they work closely. However, Jini does not scale to the level of the Internet. It would require very different performance and interaction characteristics to effectively handle a large number of users. The idea of a federation, or the ability for Jini communities to be linked together in larger groups, has been addressed in the Jini specifications but its practical scalability does not extend beyond about 1000 users or 10000 service components. JavaBeans is also not the ideal component model for dynamic service composition in an Internet environment. JavaBeans are primarily intended for use within a single address space. The mechanisms used for communication between Beans are based on direct method invocation and not on remote protocols. This limitation was removed by integrating JavaBeans into Jini services which are able to communicate across address spaces using an enhanced version of Java RMI (Remote Method Invocation). When a new JavaBean is added to a system, it is not suddenly recognized by other JavaBeans and used by them automatically. Making use of Jini, once again, removes this limitation by allowing service components to advertise the services they provide to a community of interested parties. Also, traditional JavaBeans must be explicitly linked to other JavaBeans in order to be used. Using the ERCSP protocol, JavaBeans can be introduced dynamically into the same BeanContext so they can be interconnected. However, if we wanted to create a composite service consisting of service components that remained distributed throughout the network, we could not use JavaBeans techonlogy directly. Another problem with the JavaBeans component model is that a JavaBean is only required to maintain a list of the registered Listener objects but no list of objects to which it listens itself. This means it knows about what components it is dependent on but not the components that are dependent on it for their functionality. This could be a problem if a service component in one composite service is needed by a component in another composite service at the same time. A “dependencies” section in the ICARIS XML service specification was designed to inform the system if a component had other components that it required in order to function properly. However, a component could still be used in two composite services at the same time, which could cause the problem described above. A major limitation of the ICARIS architecture is the need for the JavaBean source code to be available in the Service Broker for every service component to be able to create a stand-alone composite service with a single body of code. This limitation might be removed if modifications are made to the Java compiler.

Another issue is the proprietary nature of the used XML specification of service components. The language used for specification of service components is an important issue and we are researching it further in other projects. We are currently developing an extension of the WSDL (Web Services Description Language) standard.

5. The Composable Security Application

The Composable Security Application (CSA) is a Jini, JavaBeans, and XML-based implementation of the ICARIS architecture enabling dynamic, on-demand, construction and deployment of point-to-point security associations in order to introduce security services into applications that were not originally designed with security mechanisms. Apart from being based on the ICARIS architecture, CSA also uses functionality provided by IAIK-JCE [6] API (Application Programming Interface), which provides a re-implementation of the entire Java Cryptography Extension (JCE). IAIK-JCE comes with its own security provider, offering a great variety of cryptographic services and algorithms that are not supported in the default provider with JDK (Java Development Kit) 1.2. The IAIK-JCE Toolkit also provides a Certifying Authority (CA), so digital certificates can be used.

In the CSA, each major security algorithm supported by IAIK-JCE is contained within an individual service component. These service components can be composed at runtime to build composite security services based on the demands of the client and server for a particular security association. The client and corresponding server ends of the security association are then deployed using the support services provided by the composition infrastructure. Dynamic composition of security services using the ICARIS architecture is an accessible, robust, and flexible approach to provide many different types of security associations for many different applications. It is fast enough to assemble, deploy, and eventually adapt these associations at run time. CSA demonstrated how dynamic service composition could be used to increase the level of trust that users have towards their on-line business relationships. As already noted, an infrastructure that can simplify the selection and effective application of on-demand security is desperately needed. Security can only build trust if it is an interactive process and if all involved parties, including human users, are aware of the security measures being taken. This is where dynamic service composition can help.

To illustrate how the CSA works, we will examine how it could be used in a B2C (Business-to-Customer) e-commerce system that has been extended with the CSA so it can compose of security associations at runtime. The role of the CSA in this system is to enable a level of security that satisfies both the customer (human) and the service provider (the e-commerce software) in as transparent manner as possible, thus increasing the mutual feeling of trust. We will assume that all parties interacting with the CSA have previously registered with the CSA and trust all transactions they make with it. Further, we will assume that the Service Providers, which provide the security service components to the CSA, have uploaded all Service Items containing security services to a Service Broker registered with the CSA.

In order to use the CSA, the customer needs to first locate it through the Jini LS. When the Jini LS finds the CSA, it returns the appropriate Service Object to the customer. This Service Object contains a user interface so the customer can interact with the CSA, e.g., to request that a security association be established between his node and the e-commerce system. When the CSA receives the request, it asks the e-commerce system if it has permission to establish a secure association with it. If the CSA is not granted permission, the customer is informed and the CSA will terminate the setup. But, if the e-commerce system accepts the request, which we will hereafter assume, the CSA sets up secure channels between itself and the customer and be-

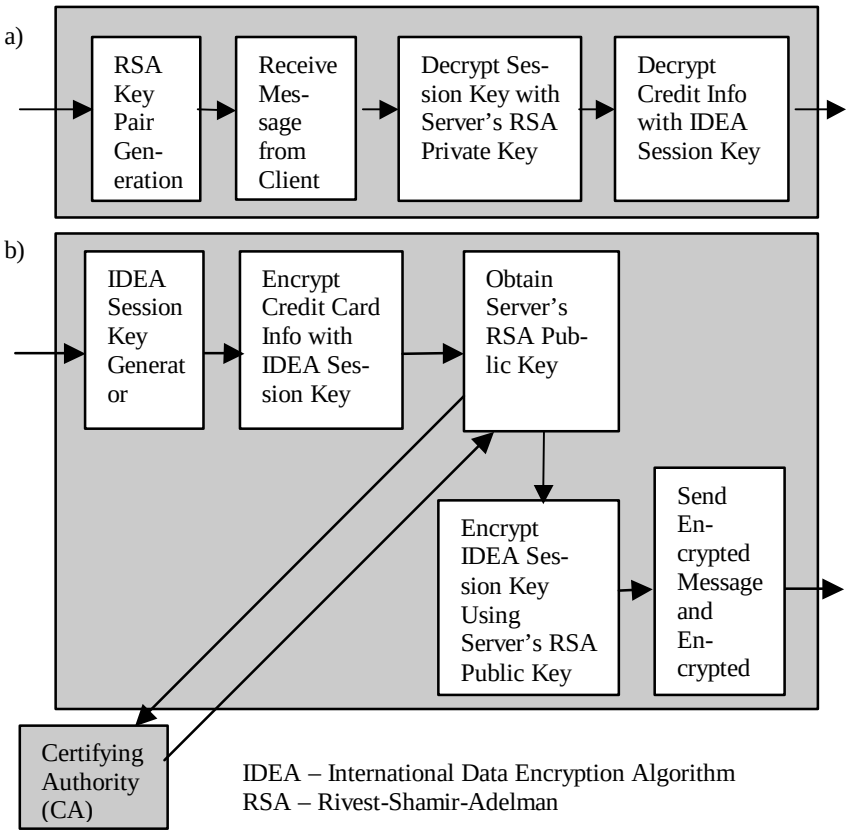

Fig. 3. An Example of a CSA Composite Service: a) Server Side; b) Client Side

tween itself and the e-commerce system. These channels ensure that a party not involved in the association cannot determine the security preferences of the association.

Once the secure channels are established, the CSA asks the customer to specify the particular type of transaction she/he wishes to perform. After receiving the transaction information from the customer, the CSA asks the e-commerce system for an XML description of its requirements for this particular type of transaction. These requirements, which have been previously defined by an administrator of the e-commerce system, become the minimum security requirements for the security association. The customer cannot lower them, but she/he can raise the level of security used. Based on the type of transaction and the security requirements specified, the CSA sends a list of potential security mechanisms that could be used to implement a secure association for the transaction. If the customer does not care what particular algorithm is used to provide the specified level of security, the CSA will use a default algorithm. We will assume that the customer wants a security association with a digital envelope based

on IDEA (International Data Encryption Algorithm) and RSA (Rivest-Shamir-Adelman) encryption and that these requirements are acceptable for the e-commerce system. The customer's selection is sent over the secure channel to the CSA. The CSA then translates the requirements for this security association into an XML-based service template. This service template is sent to the Service Broker to retrieve the required security service components. If the Service Broker locates one or more service components that satisfy the service template, the appropriate Service Object(s) are returned. When the minimum set of service components is obtained, the CSA will form a client and server pair of composite services from these service components. In most cases, a stand-alone composite service will be constructed because the security service components used by the CSA are all very suitable for pipe-and-filter assembly. The order of the service components along the pipe is determined based on a predefined algorithm stored with the Composition Manager and also by examining the Service Component Specification stored with each service component. Due to the nature of the supported security algorithms, the client and server usually have the reverse order of assembled service components.

Figure 3.a) shows a logical view of how the stand-alone server composite security service is constructed, while Figure 3.b) shows the client-side complement for the same security association. Note that the automatically generated Connection Services are not shown on these diagrams in order to simplify them. Also 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.

The composite services are deployed via the secure channels from the CSA to the customer and the e-commerce system nodes. The services are installed and the setup phase begins. At this stage, the customer establishes, using the agreed upon protocol, a secure channel directly with the e-commerce system. When the customer is informed that the security association is established, the secure information exchange can proceed. All data emerging from the customer node now passes through the composite client security service. It is sent along the secure channel to the server security service where it is decrypted and passed to the e-commerce system.

6. Related Work

Two projects, dynamic software upgrading with minimal disruption to consumers [5] and dynamic adaptation of service components with multiple classes of service [13] that we are currently working on are closely related to our work on dynamic service composition. These two projects work on different aspects of dynamic adaptation of service components. They further support the goals of business system agility, flexibility, and availability. Dynamic software upgrading with minimal disruption to consumers is very important feature for high-availability, mission-critical, and hard real-time systems. It is very beneficial, or even essential, for many business systems, like on-line stock trading software. The concepts that we are working on in the project on dynamic adaptation of service components with multiple classes of service support the flexibility and adaptability of business systems in several ways. Multiple classes of service support working with consumers that have different characteristics. They

also enable a service component to provide to every consumer an appropriate level of service and QoS (Quality of Service) and to better balance limited underlying resources. As advocated in telecommunications service management, one advantage of having a relatively limited number of classes of service over other types of service customization is manageability. Note that for complex business systems manageability is a very important issue. The dynamic adaptation mechanisms that we are developing in this project have limited power compared to finding alternative service components, but they enable faster and simpler adaptation and enhance robustness of the relationship between a service component and its consumer. They enable a service component provider to retain existing consumers and also do not require establishment of new trust relationships between service components. In this project we are also working on a new language, an extension of WSDL, for comprehensively describing service components and provided classes of service.

Several recent industrial initiatives—for example, from Microsoft (.NET), IBM (Dynamic e-business), HP (Web Services Platform), and Sun (ONE – Open Net Environment)—are based on the concept of a Web service. A Web service is a service or a service component (in our definition) that communicates by means of XML-based standards. Our work is not bound to Web services because we want to research issues that independent of the communication mechanisms used. We concentrate our efforts on researching issues that are currently not addressed by the industrial initiatives. However, the experiences from our research could be very useful for the emerging industrial approaches to dynamic composition of Web services, like IBM's WDFL (Web Services Flow Language) [8], and also for HP's eFlow [3].

Several different approaches – for example, [7], [14], and [2] – to dynamic software composition and closely related issues were presented in the literature. The majority of such research has focused on techniques for creating new applications at runtime on a single node or in a distributed system. Our research is concerned solely with the creation of new network services from a set of service components that have been designed for composability. Another key difference is that the composite services that we create do not have to execute on the computer where they are originally constructed – they can be deployed to where they are needed using mobile code. Also, while the past research on dynamic software composition mainly explores composition of service components that remain distinct, our work additionally allows construction and deployment of new stand-alone services with a single body of code. Note again that a lot of related work suggests proprietary solutions (e.g., modifying the Java Virtual Machine – JVM), while we on the contrary wanted to maximize the reuse of existing technologies.

The Darwin architectural description language (ADL) [9] is also related to our work. Darwin is a declarative ADL designed to provide a general-purpose notation for specifying both static and dynamic structures of systems composed from components.

Dynamic service composition is closely related to service discovery. An overview of some service discovery technologies was given in [4]. This reference also advocates usage of artificial intelligence for service discovery, as well as comprehensive service description accompanied with mechanisms for advanced "fuzzy" matching. Their solution for comprehensive service description and advanced "fuzzy" matching is called XReggie. XReggie has similar goals to our XML specification of service components and advanced "fuzzy" matching used in ICARIS. Another similarity is

that both approaches are based on Jini and XML. However, their specification does not contain all the information necessary for dynamic service composition.

7. Conclusions

Dynamic service composition supports business agility, flexibility, and availability. Consequently, it is useful for engineering e- and m-commerce software systems where these features are of critical importance. However, dynamic service composition cannot be used in all circumstances requiring dynamic system evolution. In many circumstances—like composition of security, finance, or telecommunication services—dynamic service composition adequately addresses the need for dynamic system evolution. In other cases, it can be a valuable element of a wider solution for dynamic system evolution. Another problem is that dynamic service composition is a challenging, inherently complex process that requires addressing a number of issues.

During the development and experimental usage of the ICARIS architecture we have obtained valuable experience and a number of insights related to dynamic service composition. ICARIS addresses an important problem with dynamic service composition – the lack of appropriate infrastructure support that is based on existing, proven, and widely used technologies. The main conclusion of our project is that it is possible to design and implement such an infrastructure using Java, Jini, JavaBeans, and XML. The experiences with ICARIS could be very useful for the ongoing and future research on dynamic service composition, like the emerging solutions for dynamic composition of e-business Web services.

Every step was taken in the design of ICARIS to avoid composing services together that could not be composed and to minimize unexpected runtime feature interactions. One of the measures taken to avoid complications is to bundle an XML service specification with each service component that describes the dependencies, constraints, or potential incompatibilities for the component. The XML service specification of each service component was parsed before attempting the composition to ensure that the composition was possible and to minimize unexpected feature interactions. We have not encountered cases where our approach was not able to provide service composition, although there were some cases when the composed services did not function as expected. In such rare cases, ICARIS requires human input to decide whether the side effects are neutral or service affecting and what do with the service composition. Additionally, to prevent unexpected runtime feature interactions, we have used a pipe-and-filter architecture for creating stand-alone composite services.

We believe that the main value of ICARIS is for composing moderately complex service components, using a pipe-and-filter architecture where the interaction between components is limited. Our application of ICARIS to dynamic composition of security associations showed the viability and feasibility of our concepts and demonstrated applicability of dynamic service composition for addressing the issues of security and building trust in on-line business systems, especially e- and m-commerce systems. While we have successfully completed the ICARIS project, we continue working on dynamic software upgrading with minimal disruption to consumers and on dynamic adaptation of service components with multiple classes of service.

Acknowledgements

The authors would like to thank Dr. Mark Vigder from National Research Council Canada and reviewers and participants of WOOBS'01 (Workshop on Object-Oriented Business Solutions 2001) for very useful comments on earlier versions of this paper.

References

1. Apache Software Foundation: Xerces Java Parser Readme. WWW page (2001). On-line at: http://xml.apache.org/xerces-j/index.html
2. Bosch, J.: Superimposition: A Component Adaptation Technique. Information and Software Technology, Vol. 41, Issue 5 (1999) 257-273
3. Casati, F., Ilnicki, S., Jin, L.-J., Krishnamoorthy, V., Shan, M.-C.: Adaptive and Dynamic Service Composition in *eFlow*. Tech. Rep. HPL-2000-39. Hewlett-Packard Company (March 2000). On-line at: http://www.hpl.hp.com/techreports/2000/HPL-2000-39.pdf
4. Chakraborty, D., Chen, H.: Service Discovery in the Future for Mobile Commerce. ACM Crossroads, Vol. 7, Issue 2 (Winter 2000) 18-24. On-line at: http://www.acm.org/crossroads/xrds7-2/service.html
5. Feng, N., Ao, G., White, T., Pagurek, B.: Dynamic Evolution of Network Management Software by Software Hot-Swapping. In Proc. of IM 2001, IEEE Publications (Seattle, USA, May 2001) 63-76
6. IAIK-Java Group: IAIK-JCE Toolkit. Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Graz, Austria (2000). On-line at: http://jcewww.iaik.tu-graz.ac.at/
7. Kniesel, G.: Type-Safe Delegation for Run-Time Component Adaptation. In Proc. of ECOOP '99 (LNCS 1628), Springer-Verlag (Lisbon, Portugal, June 1999) 351-366
8. Leymann, F.: Web Services Flow Language (WDFL 1.0). White paper. International Business Machines Corporation (May 2001). On-line at: http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
9. Magee, J., Tseng, A., Kramer, J.: Composing Distributed Objects in CORBA. In Proc. of ISADS'97, IEEE Computer Society Press (Berlin, Germany, April 1997) 257-263
10. Mennie, D., Pagurek, B.: An Architecture to Support Dynamic Composition of Service Components. Presented at WCOP 2000 (Sophia Antipolis, France, June 2000). On-line at: http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/mennie.pdf
11. 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 (2000). On-line at: http://www.sce.carleton.ca/netmanage/papers/MennieThesis.pdf
12. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1998)
13. Tosic, V., Mennie, D., Pagurek, B.: Software Configuration Management Related to Management of Distributed Systems and Services and Advanced Service Creation. In Proc. of the SCM-10 workshop at ICSE 2001 (Toronto, Canada, May 2001). On-line at: http://www.ics.uci.edu/~andre/scm10/papers/tosic.pdf
14. Truyen, E., Jorgensen, B. N., Joosen, W., Verbaeten, P.: On Interaction Refinement in Middleware. Presented at WCOP 2000 (Sophia Antipolis, France, June 2000). On-line at: http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/truyen.ps