The Research of Component-based Dependable Encapsulation

SHUANGHUI YI 1, DONGHONG LI 1
1 Beijing Institute of System and Engineering
P. O. Box 9702-19, Beijing 100101
CHINA
1 yishuanghui@gmail.com

Abstract: According to the component-based design and developing pattern, we present a plug-in architecture for the dependable component-based software. After that, the dependable encapsulation is put forwards, which mainly focuses on the component security and availability attributes. Finally, the developing and implementing method of security interceptor, high available load balancing and fault tolerance services have been put emphasis on in this paper.

Key-Words: Component, Dependable encapsulation, Security, Load balancing

1 Introduction
Component is the basic element of the current software system, so its reliability, security, availability are the important factors to affect the correct running of current system. Component-based software development approach is based on the idea to develop software systems by selecting appropriate off-the-shelf components and then to assemble them with a well-defined software architecture. Component-based dependable software development becomes the hot topic in software development.

Dependability is considered to cure the security threat, abnormal behaviour and untrustworthy crisis of the information system. Dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable to the user(s) [1]. As developed over the past three decades, dependability is an integrating concept that encompasses many attributes, such as reliability, availability, security, safety, and maintainability, etc. The dependability specification of a system must include the requirements for the dependability attributes in terms of the acceptable frequency and severity of failures for the specified classes of faults and a given use environment. One or more attributes may not be required at all for given system.

In this paper, we focus the dependability attributes on security, reliability and availability [1, 2]. We introduce the component-based dependable software design and development, supporting software system to meet requirement of the dependability.

2 The Plug-in Architecture for Dependable Component-Based Software
Component-based design and developing pattern has become the mainstream model for software development, such as J2EE, dot NET and so on, which were well-known for us. Compared to traditional software development pattern, the Component-Based Software Development (CBSD) has some obvious advantages. Firstly, Component-Based software is made up of a series of components, and software functions and non-functional properties are implemented separately, which makes the software structure more clear; Secondly, some components can be developed independent of specific applications, which are called reusable components, and application developers can take advantage of existing structures, improve application development efficiency, reduce software development time; Thirdly, component containers usually support the Inversion of Control(IoC), Dependency Injection(DI), call-back, configurable components, interceptors, single-case and other design patterns[2], which provide a richer support for component-based software development, deployment and running.

Compared with traditional software, component-based software dependability has its distinctive features. Firstly, since the component-based software is the combination of some ready-made components [3](COTS, MOTS, GOTS, etc.) or new developed component with a certain way, the dependability of component-based software focuses on how to ensure the dependability of components used and new combined components; Secondly, components usually run in a certain container.
environment, the dependability property can be injected or combined by the mechanism of container, which is significantly different from the traditional hard-wired programming implementation.

As Fig.1, the container handles two kinds of requests, one from the client, and the other from the dependable modules which in fact intercept the client’s request. Once the component requires the dependability assurance which is described in the component deployment script, container could wake up the dependable modules and invoke it to transparently intercept the client’s request.

As we know, the life of components is maintained by the container, so the dependability of components could not be separated from the container support. Some dependability, such as fault tolerance, failover and load balancing, could not be implemented with single-node container. Meanwhile, in order to meet the needs of the user's specific dependability, the dependable component-based software system should have an open, scalable and transparent architecture.

Therefore, based on the container cluster technology, we brought forward distributed container architecture, which provide component configuration and AOP [4] interceptor patterns to support component fault tolerance, load balancing and state recovery to increase the security, reliability or availability.

As Fig.2, according to the different requirements of application components, container can dynamically load dependable service plugins for each component or service module, which is transparent to the client calling the components. At the same time, containers are interconnected with each other to achieve the component state synchronization by the listening mechanism.

The dependable components calling process is as the following:

1) The client calls the component method (function), the container intercepts the method call and assigns to the component instance;
2) Before delegating method calls to the component instance, the corresponding interceptor could be activated (such as security interceptor to authenticate the user's call);
3) At the end of the interceptor chain, the method call is delegated to the real component instance;
4) The running results will be cached and returned to the client by the activation mechanism of the container.

This implementation structure has mainly the following advantages:

- Interceptor mechanism reflects the good idea "separation of concerns [5]," which can separate functions from non-function services, not interfere with each other, and reduce the coupling degree of component systems;
- To promote the transparency of dependable service to the user, and reduce the complexity of client applications;
- Interceptor is implemented as hot-plugins [6], and ensures scalability of system applications, so new dependable requirements could be implemented by adding new interceptors into container.

3 The Dependable Encapsulation Based On The Container Service

As described in the last chapter, the architecture provides the opportunity to implement the dependability attributes as the container service.
Based on this idea, we have implemented a dependable interceptor to achieve a dependable component by injecting security interceptor into components. With the container load-balancing cluster technology and the distributed architecture, it is easy for components to get the ability of load balancing and fault tolerance, then achieve correct and efficient, high throughput and scalability purposes.

3.1 Security for Components
Security services are implemented by interceptors, which is responsible for monitoring messages between components and clients, moreover, security manager and domain mapping manager are designed under the unified management of distributed container environment, which have their unique ID. But only the use of security interceptor cannot be intercepted by a distributed component of the container under the unified management of the environment, so they designed a security manager and domain mapping manager. Security manager and domain mapping manager have their unique ID in the distributed container system-wide, while security interceptor for each container.

Fig. 3 The Security Interceptor for Dependable Components

As Fig.3, Security interceptor gets the security context from the component requests, that are the user identity and the token, and then takes access control checks. Finally, security manager requests the authentication server to authenticate it according to the security context. The access control checks is based on the each component’s privilege information stored in the container, so it should not be done by the security manager, which is responsible for the application server and could not get the meta-information about components. About access control, we have to mention the domain mapping manager. The user identity of security context is the physical status, which should be mapped to logic role with access rights to make access control. The domain mapping manager provides the interface between physical users and their roles.

3.2 High Availability for Components
The load balancing service could promote the high availability of components, which can be implemented in different levels, such as the application server (JNDI [7] naming server, containers, etc.), the load balancer or the client proxy. For different environments, different implementations have their advantages and disadvantages. This paper mainly focuses on the J2EE applications based on transactions, so takes the solution by high available naming service (HANS), which maintains the mapping relationship between JNDI and a set of object reference or client proxy. When the client requests the component located on the distributed container nodes, the HANS could select a node according to some load balancing algorithm and return the specific object reference back to the client. As Fig.2, the so-called HANS is made up of distributed naming service located on each container node. In addition, it is easier to get the high-availability global naming service information (such as load status of each node), and thus easier to implement dynamic load balancing algorithm.

Fig. 4 The Architecture of High Availability for Components

For the interactive transaction environment with a large number of customer applications, in order to reduce the cost of the server, this paper designed the client load-balancing interceptors. The client interceptors maintain a set of objects reference information which encapsulates the deployment information of the component, such as the address of each node or service. When the client requests,
the client interceptor could transfer the requests to the selected object reference according to the load balancing algorithms and other information (such as the load of each node).

After that, the load balancing service should have a certain degree of fault tolerance to avoid the requests failure in case the back container node fails. Especially for the stateful components, we have to consider the state synchronization and consistency [8] between components and their redundant copies located on the other container nodes. Therefore, we divided the distributed container system into primary container and backup containers. When the application server starts, load balancing service could select a container as the backup container for restoring the state of the component instance when the client invokes the component or the state of the invoked component instance changes.

4 Conclusion

In this paper, the dependability development architecture is introduced. And the related technologies are discussed based on the dependable software architecture. Dependable encapsulation can change the existed dependable component to new dependable component in specific environment. On the other hand, Dependable encapsulation can change the undependable component to dependable component also. Currently, we designed and implemented an Eclipse plug-in tool to develop dependable component-based software based on JBoss application server environment. The tool environment includes a distributed container environment, dependable service configuration tool, and dependable encapsulation tool. With this tool, developer could inject dependability attributes into components when creating new components and loading dependability attributes for the existing components, etc.

References:


