Continuous Migration Container System for Upgrading Object

N. Yoosanthiah and A. Khunkitti

Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang
Ladkrabang , Bangkok 10520 , Thailland
Email: netsuda@yahoo.com and akharin@it.kmitl.ac.th

Abstract: During system resource improvement process that based on Object-Oriented technology could be affect to the continuous system performance if lack appropriate management and control objects mechanism. This paper proposes a methodology to support continuous system performance and its stability. The adoption is based on Java Container Framework and Collections Framework for object collection. Also includes Software Engineering, Object Migration and Multiple Class Loaders mechanism accommodate to construct Continuous Migration Container (CMC). CMC is a runtime environment provides interfaces for management and control to support upgrading object process. Upgrade object methodology of CMC can be divided into two phase are object equivalence checking and object migration process. Object equivalence checking include object behavior verification and functional conformance verification before object migration process. In addition, CMC use Multiple Class Loaders mechanism to support reload effected classes instead of state transfer in migration process while upgrading object. These operations are crucial for system stability and enhancement efficiency.

Keywords: Java Container Framework, Multiple Class Loaders, Continuous Migration Container and Object Equivalence Checking

1. INTRODUCTION

Nowadays the system applicability based on Object-Oriented has been used widespread. In composite with new technology to improve the capability of device or environment to support variety works developed. There are essential needs to enable or enhance the existing resources represent objects to updating system performance. Hence systems must serve management and control the transactions in continuous pattern while the changing occur.

Several researches system stability maintenance in upgrading transition in the following approaches:

Dynamic Load Distribution [1] approach supporting Parallel Objects (PO) programming environment has provide a transparent and dynamic distribution of system load via remote creation and migration of objects. At least one execution thread is associated with each object. This approach have perspective that allocation policies cannot be completely automated, but new created object allocation need also to be directed by user.

The principle of Dynamic Class Loading in JVM [2] provides mechanism for dynamically loading software components on the Java platform. It is often desirable upgrade software component in a long-running application. By organizing software components in separate class loaders and avoid dealing with schema evolution with new classes that loaded by a separate loader. This approach depend on loaded class caches because JVM cannot trust any user-defined load class method to maintain them for mapping class names and initiating loaders to class type. These can affect downtimes that occur from the system that loaded class cache.

Transparent Dynamic Reconfiguration for CORBA [3] approach support distributed system provides the ability to maintain or upgrade without being taken off-line. The mechanisms drive the system under reconfiguration to a safe state, which avoids abortion interactions. This approach focuses on reconfiguration of non-redundant objects and has impact on execution during reconfiguration. There are effects of reconfiguration on the performance of the new object right after this process.

Location-Aware Lifecycle Environment framework (Loc-ALE) [4] provides a simple management interface that control the lifecycle (LCManager) of CORBA distributed objects. Every object creation, movements or deletions requests are routed through LCManager. The advantages are that the LCManager presents object fault-tolerance and mobility support facilities without breaking client held references. But LCManager as a single point of the system, so there are problem of bottleneck within a location domain.

From above, this paper proposes a mechanism of *Continuous Migration Container (CMC)* for maintain stability while upgrading object by use Java Container Framework [6] which provide objects environment based on Collections Framework [7]. Container infrastructure composes management and control interfaces that can be applied to object migration and multiple class loaders that support new object creation and existing object equivalence checking before initiate upgrading process.

This paper is further structured as follows. Section 2 represents continuous upgrading object approach. That is a support guide for management and control objects under *Continuous Migration Container (CMC)* environment implementation, Section 3 describe about infrastructure and operation approach. Finally, the conclusion and future work in Section 4.

2. CONTINUOUS UPGRADING OBJECT APPROACH

This section is a object migration concept which process based on software engineering [5] that provide upgrading object operation within the system compose as Figure 1.

1. Object Creation: newly object (Onew) being created to the existing object (Oold) within the system.

2. Object Validation: functional and operation compatibility validation check point between Onew and Oold that can divided into 2 phase.

2.1 Check Authentication: verify agent username and password permitted to operate.

2.2 Check Equivalence: verify functional equivalence between Onew and Oold consist of:

Behavior verification is descriptive check both objects regarding the field, parameter, method, modifier and return type to compare similarity.

Execution verification is executed objects operations check within their method and compare results are conformable execution.

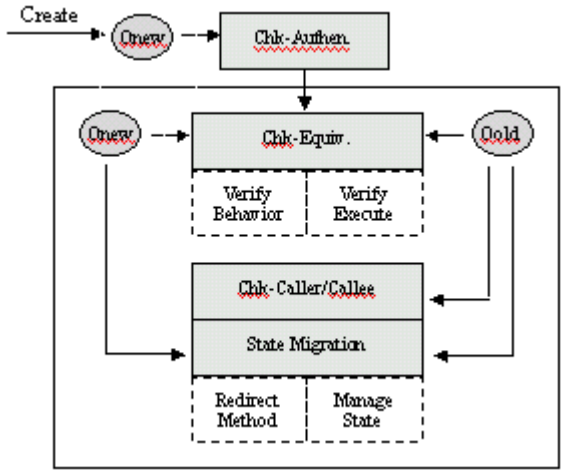

Fig. 1 Object Migration Approach

3. Object Transfer: check effected objects for support state transfer process.

3.1 Check Call Method: check effected objects, which are called or call to Oold's method. Then the system will obtain information about caller and callee objects and their status for support state migration process.

3.2 State Migration: redirect messages and manage call status between effected objects to migrate.

Case of Oold's call methods are waiting for execution, terminated and recall to Onew for proceeds. In the case of call methods are executing, will be checked:

If these executing have long-time operation, pack and migrate method's state for continue executing to Onew.

If their have short-time operation, freeze method execution and return value to related object and sends these status for along continue into Onew.

4. Destroy Old Object: termination of Oold when there are non remaining transactions or reference to Oold.

3. CONTINUOUS MIGRATION CONTAINER (CMC) IMPLEMENTATION

From section 2, the environment to manage and control objects should be provided. This paper proposes *Continuous Migration Container (CMC)* is a runtime environment for manage component execution and maintain continuous performance while upgrading object.

3.1 CMC System

System of CMC is shown in Figure 2 consists of the following elements:

- *Component Manager:* is the central component of CMC that interacts with other component as a Runtime [6] provide interface to the container with infrastructure service like transaction and security. This component is user interface that accessible everywhere in the container and usually works as a single point of access to obtain infrastructure services and other elements of CMC.
- *Component Handler:* usually take a component request as input and invokes it on the target component instance. In addition, provide hook at various well-defined points (e.g. before and after method invocation) and adopted from Reactor pattern [9] for support handler dispatching object's state. This component can be divided into *Reactor* provides interface to call methods and *Event Handler* provides a standard interface for dispatching event object's state, will describe in 3.2.2.
- *Component Meta-Data*: contains information about necessary interface and implementation classes of the components within container. This information include object's signature which useful for object behavior verification in equivalence checking process.
- *Component Registry*: is a place where all the components are registered in the container. Responsible maps a unique key with the component to provide a registry for the location of object.
- *Component Factory*: is an interface for the components. These factories are used to create and find object on behalf of Component Manager. CMC uses factory to manage lifecycle of components.

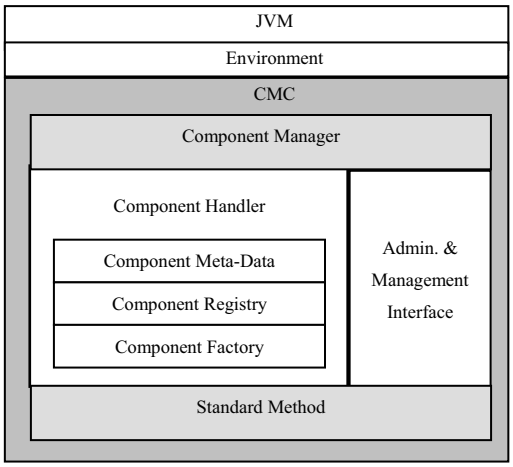

Fig. 2 CMC System

- *Administration & management Interface:* include method to provide equivalence checking to support object migration process.
- *CMC Standard Method:* provide method implementation for support CMC operation together with all of above component.

3.2 Upgrading Object Methodology of CMC

Since newly object creation is completely deployed into CMC and duplicate object discovery checking, the migrating operation for upgrading object perform include the following steps as show in Figure 3.

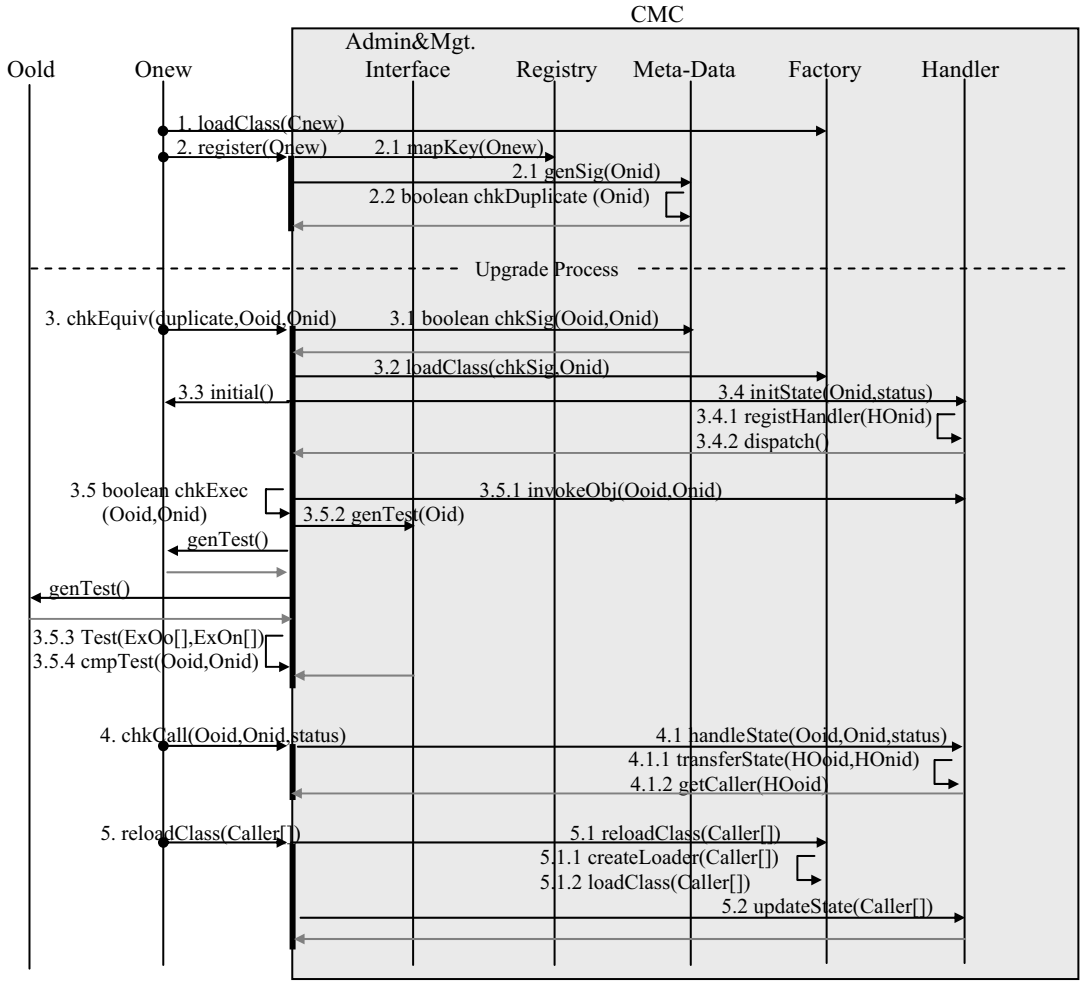

Fig 3. Object Upgrading Process

3.2.1 Check Equivalence

This process provides objects equivalence checking for verify compatibility between Onew and Oold within CMC operation before migration process. In addition, these operations are implemented within objects as interfaces are associated with *Admin. & Management Interface* as Figure 4.

This process can be divided into two partition-checks:

- *Object Behavior Verification*

The first partition-check to determine the compatibility between Onew and Oold directly to get the information about object signature and descriptor e.g. field name, field type, method's name and return type, method's parameter type, constructor's name and parameter type from *Component Meta-Data* to compare. Usually Onew may have any additional signature which may effect to the system but not.

- *Object Functional Conformance Verification*

Since the object behavior process is completed, the next step is to verify functional conformance, which controlled *by CMC Standard Method.* By comparing methods if they give the same direction of result or not. The operation will conduct the test of all methods generation of Oold and Onew through *Administration & Management Interface*. Thus all of objects within CMC, interface for functional conformance checking must be implemented.

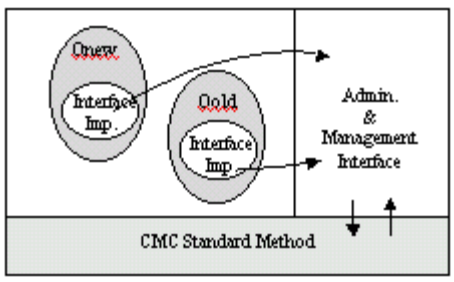

Fig. 4 Interface implementation

After test case generation of Oold and Onew complete, CMC will compare result of testing all methods by test generation for check functional conformance.

When the equivalence checking process has been done, it will initiate *Object Migration Process*. This phase perform check effected objects call or called to Oold's method for support state transfer process. Object migration phase compose two steps: check call method and state transfer or reload effected classes.

3.2.2 Check Call Method

After the equivalence checking process has been done, *CMC Standard Method* invoke chkCall() on *Handler* to handle event by using Object Behavioral Pattern [8] and capture state of Oold by Object Serialization.

Since *Component Manager* receive chkCall(), delegate freeze() for enforce to stop execution of CMC and handleState() is invoked for handle state of Oold to check call method as Figure 5.

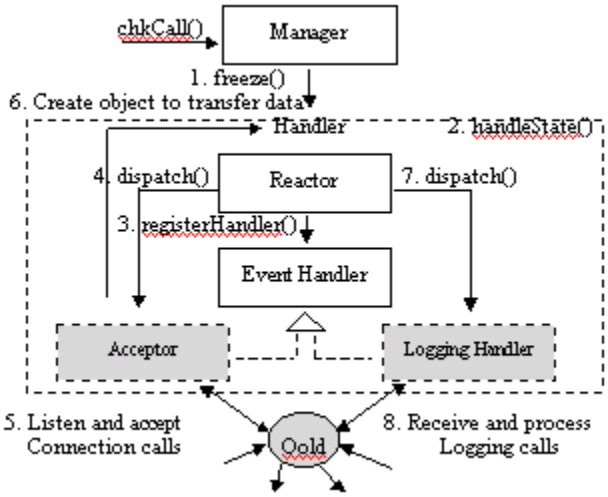

Fig. 5 State Handler for Check Call Method of Oold

Component Handler provide handle Oold's states and event as following:

- *Reactor:* Define an interface for registering, removing and dispatching Event Handler object. An implementation of this interface provides mechanisms perform event demultiplexing and dispatching of Oold's event handlers in response to events.
- *Event Handler:* Define base class specifies an interface use by Reactor to dispatch call methods of Oold defined by objects that are pre-registered to handle event. There are two subclasses: Acceptor and Logging Handler are implements the callback method that process events in object.
- *Acceptor:* provide a factory for creating Logging Handler object to establish connection to Oold for listen and accept connection request of Oold.
- *Logging Handler:* responsible for receiving and processing logging records sent from Oold.

When a logging record arrives, the *Reactor* automatically dispatches the handle event method of the associated Logging Handler object. This object then receives and processes the logging record.

3.2.3 Reload Effected Class

Because of *State Transfer* operation similar redirect messages from effected objects that coming to Oold outgoing to Onew, which can use class loader mechanism. JVM uses class loaders to load class files and create class objects that is a powerful mechanism for dynamically loading software components on the Java platform [9] and responsible to determine when and how classes can be added into a running Java environment [10]. *CMC Standard Method* has method for load class as following algorithm:

```
MyClassloader(String nameClass)
        // check nameClass has been loaded
        if (nameClass =  loaded class)
         return previous loaded class
        // check nameClass is system class
       else if (nameClass = system class)
         // primordial class loader perform load class
              Load class from class path
              else
                    read class file into array of byte
                    construct class object and its method
                    specify class loader of nameClass
                    load class from class loader
              end if
       end if
end
```

Furthermore CMC's operation use separate class loaders called *Multiple class loaders* for support the key technique is to load the caller class of Oold, class of Oold and class of Onew into separate class loaders. This mechanism provides reload effected classes instead of state transfer in migration process. Thus, this step is called alternative name as *Reload Effected Classes*. CMC perform reload classes operation as Figure 6.

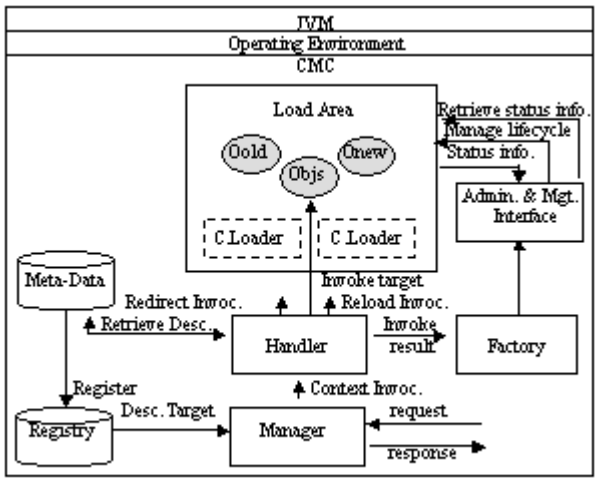

Fig. 6 Reload effected classes of CMC

Referring to the reload method, the processRequest() redirects all incoming requests to a service object stored in a private field to invoke the "run" method on the service object. In addition, the updateService() allows a new version of Oold (Onew) to dynamically loaded, replacing the existing service object. updateService() callers could supply the location of new class files. All of calls will be redirected to the new object referenced to by services. These methods are the following algorithm:

```
reloadClass()
        private Object service;
        updateService(String location)
        processRequest()
end
updateService(String location)
         create MyClassLoader (location)
         // load class which want to redirect
         Class c = loadClass(nameClass)
         service = create new instance of c
end
processRequest()
         Class c = get class of service
         Method m = get method (run) of c
         invoke m
end
```

After call updateService(), Onew will process all of calls in the future. However the Oold may not have finished processing some of the earlier calls. Thus two classes may coexist for a short period of time. Until all uses of the Oold are completed, all references to Oold are dropped and unloaded by *Admin. & Management Interface* through Garbage Collection.

4. CONCLUSION AND FUTURE WORK

The modification of the existing resource to update and performance enhancement is essential for the systems that based on Object-Oriented technology. Typically, the process may have the impact on continuous of system operation.

This paper proposes mechanisms that provide system stability and continuous performance by CMC construction. This is a runtime environment that equipped interface for continuous upgrading object and support object equivalence checking by objects behavior and functional conformance verification before migration process. In addition, object behavioral pattern and object serialization mechanisms are effectively handle event for capture and restore object's state. Finally, CMC use Multiple Class Loaders mechanism to support reload effected classes instead of state transfer in migration process. These operations can manage and control objects still perform together with newly object and existing object by continuous system performance.

Currently, we work according to interface implementation proposed that can check equivalence object for upgrade process preparation. Simultaneously get up object serialization and multiple class loaders in depth to improve algorithm for captures object states and reload effected classes in accurately the future.

REFERENCES

[1] Antonio Corradi, Letizia Leonardi and Franco Zambonelli, "Dynamic Load Distribution in Massively Parallel Architectures: the Parallel Objects Example", IEEE MPCS'94, May 1994.

[2] Sheng Liang and Gilad Bracha., "Dynamic Class Loading in the Java™ Virtual Machine", Annual ACM SIGPLAN Conference (OOPSLA'98), Canada, October 1998.

[3] J.P.A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In Proceedings of the 3rd International Symposium on Distributed Objects and Applications, IEEE Computer Society, September 2001, pp. 197--207.

[4] Lpez de Ipia D. and Lo S. "LocALE: a Location-Aware Lifecycle Environment for Ubiquitous Computing", Proceedings of the 15th International Conference on Information Networking (ICOIN-15), February 2001.

[5] Ian Sommerville, Software Engineering, Third Edition, Addison-Wesley, 1998, pp. 379-398.

[6] Sanjay Dalal, "A Java Container Framework for Server Component Models", eCommerce Server Division, BEA Systems, Inc., pp. 10-16.

[7] http://www.java.sun.com/java_tutorial_3nd/collections

[8] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", in Pattern Languages of Program Design, AddisonWesley, 1995, pp. 529-545.

[9] Li Gong, "Secure Java Class Loading", 1069- 780/98 IEEE, November-December 1998, pp. 58-61.

[10] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio, "A Formal Specification of JavaTM Class Loading", Kestrel Institute July 21, 2000.