Design Patterns for Releasing Applications in C++ implementations of JTRS Software Communications Architecture*

Michael Barth, Jonghun Yoo, Saehwa Kim, and Seongsoo Hong

Real-Time Operating Systems Laboratory, School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea
{barth, jhyoo, ksaehwa, sshong}@redwood.snu.ac.kr

Abstract

The Software Communications Architecture (SCA), which has been adopted as an SDR (Software Defined Radio) Forum standard, provides a framework that successfully exploits common design patterns of distributed, real-time, and object-oriented embedded systems software. We have fully implemented the SCA v2.2 in C++. In the process, we have encountered the lack of a suitable design pattern for releasing the SCA applications. Unfortunately, design patterns for releasing objects have been neither extensively addressed nor well investigated as opposed to creational design patterns. This is largely due to the fact that such releasing design patterns are highly dependent on the programming languages. In this paper, we investigate three viable design patterns for releasing the SCA applications in C++ SCA implementations and discuss their pros and cons. In addition, we select the most portable and thus most reusable pattern, which we named Vulture design pattern, among those alternatives and detail our specific implementation.

1. Introduction

As the complexity of embedded software is constantly increasing and software development time is dramatically declining, the need for software reuse has intensified in recent years. To foster software reuse, software component technology has become a major trend of the embedded software market. Software component technology takes aim at creating new software systems that unify disparate technologies through the combination of deployable software, as opposed to ground-up development. The representative examples of such trend are military radio systems, automobile systems, and robotics technologies.

The Software Communication Architecture (SCA) [1] [2] [3] is a key enabling component technology for the systems, which are real-time, embedded, distributed and object-oriented. The SCA specification is published by the Joint Tactical Radio System Joint Program Office (JTRS JPO). In addition, it is adopted by the Software Defined Radio (SDR) forum as the standard software structure of SDR systems. The goals set for the SCA are

* The work reported in this paper was supported in part by the Ministry of Information and Communication under a project entitled “Embedded Component Technology and Standardization for URC.”

increasing flexibility, upgradeability and interoperability of globally deployed systems, as well as reducing the costs for supportability, system acquisition and operation.

Figure 1 depicts the overall structure of the SCA. The SCA is composed of applications and an operation environment (OE) layer. The SCA applications layer includes diverse types of applications depending on the SCA implementation appliance. The OE is comprised of the RTOS (real-time operating system) and the RT-CORBA (Common Object Request Broker Architecture [4]) implementations as the distributed middleware, and the Core Framework (CF) as the deployment middleware. The RTOS and the RT-CORBA middleware control CPU scheduling, resource allocation, and the inter software components communication. The basic function of the Core Framework (CF) is to manage the SCA application life cycle following the Factory design pattern [5].

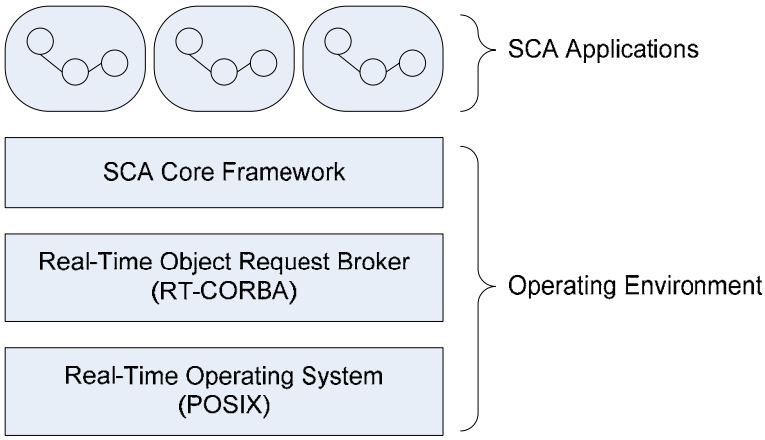

Figure 1. Structure of SCA software

We have fully developed an SCA v2.2 compliant C++ [6] implementation called The Robot Software Communication Architecture [7] [8] [9] [10] (RSCA), which is specialized for the robot applications. In the process of implementing RSCA, we have encountered the lack of a suitable design pattern for releasing the SCA applications. Unfortunately, design patterns for releasing objects have been neither extensively addressed nor well investigated as opposed to creational design patterns. This is largely due to the fact that the releasing patterns are highly dependent on programming languages. We have found that it is hard to overcome the memory leakage without applying a suitable design pattern for releasing the SCA applications. Specifically for releasing the SCA applications, the SCA specifies a dedicated CORBA operation called *releaseObject()*. All resources allocated for the SCA application must be released as a result of *releaseObject()*. The releasing process must include two operations: (1) the SCA application CORBA object deactivation and (2) the SCA application servant deletion. However, the two operations cannot be processed in one CORBA operation due to the nature of CORBA. The application servant deleting operation must be performed outside the *releaseObject()*.

The solution would be simple if the SCA was implemented in a language with the automated memory management capabilities. For instance, Java™'s [11] sophisticated garbage collector would simply delete the

SCA application servant after the deactivation of the SCA application CORBA object, since the deactivation operation removes the Java™ object reference to the SCA application servant from the Portable Object Adapter (POA). However, the solution might not be easy if the SCA was implemented in a language without such capabilities. The SCA does not specify the method to delete the SCA application servants, as the deleting mechanism is highly implementation-dependent.

In this paper, we propose three alternative solutions of design patterns for releasing the SCA applications, which can be applied in a language-independent manner. The first solution deletes the SCA application servants using advanced CORBA features. In the second solution, a clean-up thread is launched by the SCA application servant that safely deletes the servant after the deactivation of the application CORBA object completes. The third solution improves the previous one by relocating the clean-up thread launch to the SCA domain manager. In our RSCA implementation, we have adopted the third solution since it is the most portable and thus most reusable design pattern among the three alternative solutions. We named the design pattern we adopted, *Vulture* design pattern, which comes from the scavenging habits of the vultures, feeding from carcasses of dead animals. We explain the detailed RSCA implementation regarding this.

The remainder of the paper is as follows. In Section 2, we describe the problem in the process of releasing the SCA applications in detail. Then, we introduce three alternative design patterns to solve the problem and present their pros and cons in section 3. In section 4, we detail our specific implementation of the Vulture design pattern and give our rationale. Finally, we conclude this paper in Section 5.

2. Problem Description

The SCA specification applies the Factory design pattern to create the SCA applications of a certain type. For simplicity, we abstract from the SCA only a subset of the SCA interfaces relevant to our considerations. Figure 2 depicts the creational pattern used by the SCA for the application life cycle. This pattern consists of three types of CORBA objects: (1) *Application* objects, (2) *ApplicationFactory* objects and (3) a *DomainManager* object.

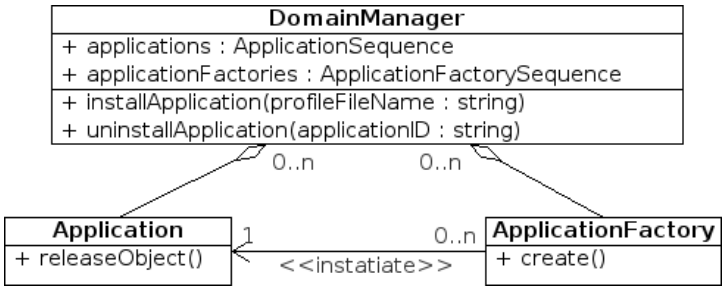

Figure 2. *Application* Factory design pattern in SCA.

Figure 3 shows how those CORBA objects interact. At the beginning of the SCA application life cycle, the SCA client installs the SCA application package via *DomainManager installApplication()*. Then, the

DomainManager creates an *ApplicationFactory* instance for a particular *Application* object (1). In order to instantiate the *Application*, the SCA client gets all *ApplicationFactory* references from the *DomainManager*, using the *applicationFactories()* operation[†], (2). After that, the SCA client selects among those references an accordant *ApplicationFactory* reference and invokes the *ApplicationFactory create()* operation (3). Then, the *ApplicationFactory* creates a new *Application* instance.

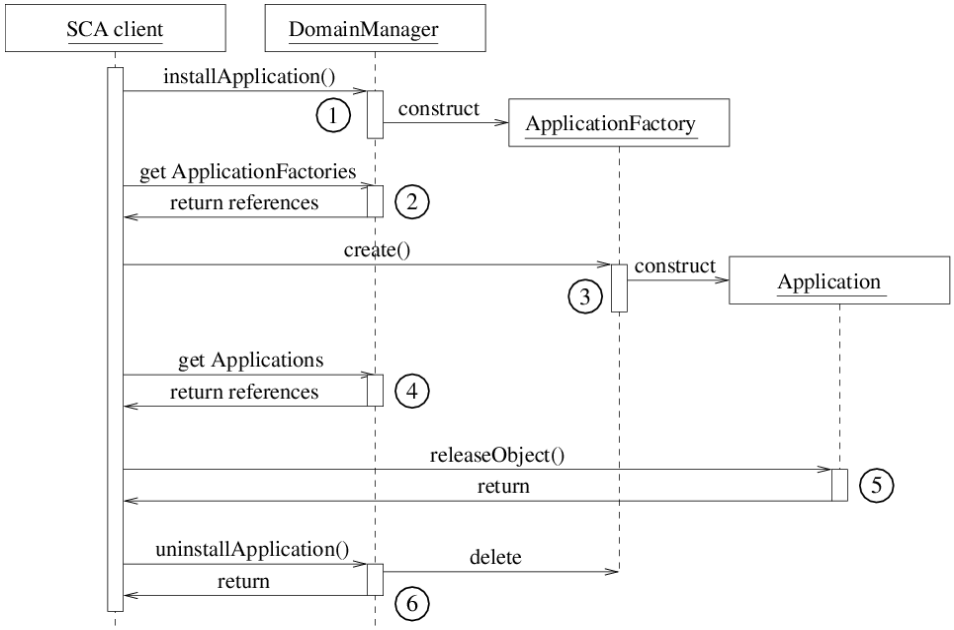

Figure 3. Interaction between the *DomainManager*, *ApplicationFactory* and *Application* objects.

The *Application* releasing process is similar to the *Application* instantiation process. First, the SCA client retrieves all *Application* references from the *DomainManager* (4). Second, the SCA client selects an accordant *Application* reference among those references and calls the *releaseObject()* operation (5).

In order to uninstall the *Application*, the *DomainManager* provides the *uninstallApplication()* operation. The *DomainManager uninstallApplication()* operation deletes a particular *ApplicationFactory* and all *Applications* that the *ApplicationFactory* has created before (6).

The *Application* releasing problem originates from the *Application releaseObject()* operation. An invocation by the SCA client of the *Application releaseObject()* operation must firstly deactivate the *Application* CORBA object and secondly delete the *Application* servant from memory. For the former functionality, the *Application*

[†] According to the CORBA C++ mapping, the *DomainManager* attributes "*applications*" and "*applicationFactories*" are mapped to the *applications()* and *applicationFactory()* operations.

servant can issue the deactivation of the *Application* CORBA object, which is incarnated by the servant, using the POA *deactivate_object(in ObjectId oid)* operation inside the *releaseObject() operation*. However, adhering to the OMG CORBA specification [4], the *deactivate_object(in ObjectId oid)* operation, which is in charge of breaking the bond between a CORBA object and its servant, is non-blocking and does return immediately. After the call of *deactivate_object()*, an ORB keeps the CORBA object activated until all active requests to the CORBA object are processed. Active requests are those requests that have arrived before *deactivate_object()* was called. Since the *releaseObject()* request arrives before *deactivate_object()* is called within *releaseObject()*, there is at least one active request to the *Application* servant. If *deactivate_object()* called inside *releaseObject()* would block, i.e. not return immediately, the *releaseObject()* operation would never return, because the *deactivate_object()* operation would be deadlocked waiting on *releaseObject()* that invoked it. Consequently, the ORB will only deactivate the *Application* CORBA object after the SCA client call of the *releaseObject()* returns to the SCA client. Therefore, it is impossible to simply delete an *Application* servant within *releaseObject()* because the *Application* servant incarnates the active *Application* CORBA object during the runtime of *releaseObject()*. Deleting a servant, which incarnates an active CORBA object, would cause an undefined ORB behavior.

The *Application* releasing problem is the latter functionality of the *Application releaseObject()* operation, i.e. the question how to delete the *Application* servant from the memory after *releaseObject()* returns to the SCA client. This problem only occurs when C++ is used as the implementation language for SCA. In contrast to C++, a programming environment with an automated memory management system would automatically delete the *Application* servant from the memory. For instance, Java™'s garbage collector would delete the *Application* servant after the *Application releaseObject()* operation returns to the SCA client.

C++ does not have any automated memory management capability. Therefore, the *Application releaseObject()* operation causes a memory leakage when the *Application* servant gets deactivated, but not deleted from memory. Since it is expected that many SCA applications will be created and uninstalled frequently by the nature of the application domains of SCA, systems that use a C++ implementation of the SCA may suffer from a serious memory leakage problem.

3. Alternative Solutions

In this section, we propose three alternative design patterns to solve the *Application* releasing problem, and we will discuss our specific implementation in the next section. The first solution (S1) is an approach that relies on the CORBA's servant manager interface. The second solution (S2), which is similarly implemented for the MS Windows platform by OSSIE [12], introduces a clean-up thread inside the *Application* servant. Finally, the third solution (S3) challenges the disadvantages of S2 by moving the clean-up thread to the *DomainManager* servant. We will explain each of these solutions in detail in the following subsections.

3.1 Servant Manager Approach

A servant manager [4] is a CORBA object that the *DomainManager* registers with its POA to assist or even to replace the function of the POA's own Active Object Map. When dispatching a request, the POA extracts the object id, which is normally embedded in the object reference for the target object, and uses this reference to look up the servant that corresponds to the target object in its Active Object Map.

There are two types of servant managers: (1) servant activators and (2) servant locators. For the POA with the *RETAIN* value for the servant retention policy, the servant manager must implement the *ServantActivator* interface. Also, in case of the *NON_RETAIN* value, the servant manager must implement the *ServantLocator* interface respectively. This subsection gives a solution that makes use of the *ServantActivator* interface.

A *ServantActivator* object must support the *incarnate()* and the *etherealize()* operations. When the POA with the *RETAIN* policy value receives a request for a target object, it searches for the servant for this object in its Active Object Map. If there is no entry for the target object and the *ServantActivator* object is registered with the POA, then the POA invokes the *incarnate()* operation on the *ServantActivator* object, passing the object id of the target object. The implementation of the *incarnate()* operation returns a servant instance according to the object id or raises an exception. The *etherealize()* operation is the opposite of the *incarnate()* operation. The POA is supposed to invoke *etherealize()* in response of the explicit object deactivation via *deactivate_object()*. The implementation of the *etherealize()* operation then deletes all resources associated with the servant.

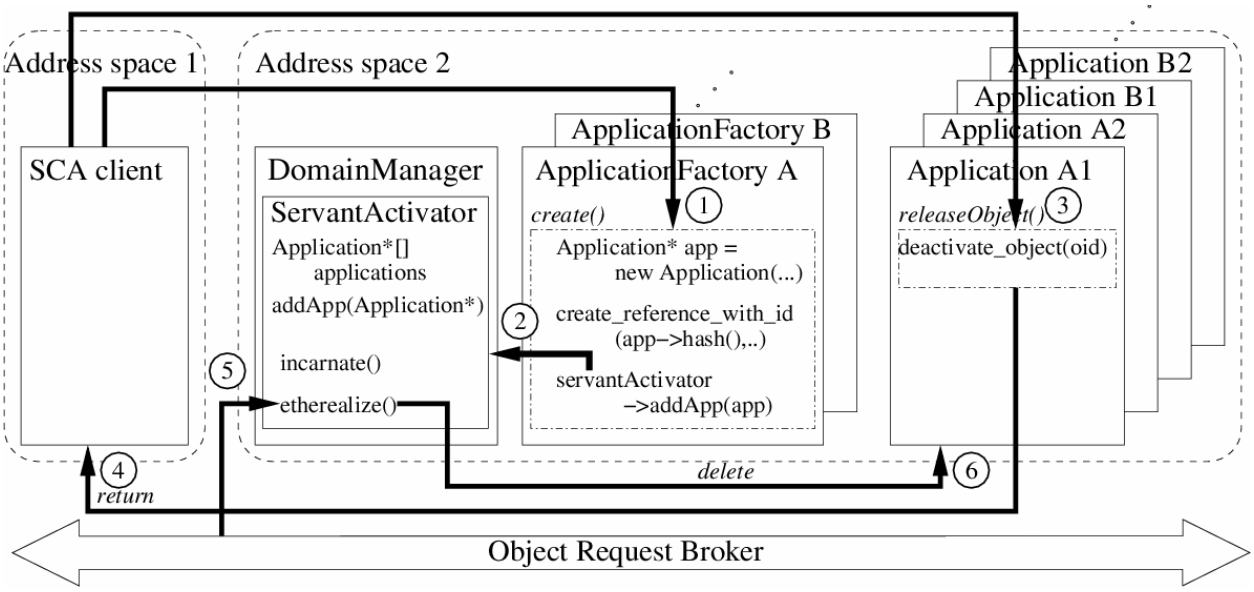

Figure 4. *Application* servant deletion via a *ServantActivator* object.

Figure 4 depicts our servant activator solution for solving the *Application* releasing problem. An SCA client runs in the address space 1. The *DomainManager* servant, the *ApplicationFactory* servants and the *Application* servants run in the address space 2. The *DomainManager*, the *ApplicationFactory*, and the *Application* CORBA objects share the same POA. The *ServantActivator* servant is created inside the *DomainManager's* constructor. Our *ServantActivator* maintains a list of *Application* servant pointers. To add an *Application* servant pointer of a newly created *Application* to this list, we have implemented the *addApp(Application*)* function.

In order to create a new *Application* instance the SCA client invokes the *create()* operation on an *ApplicationFactory* (1) after getting the *ApplicationFactory* references from the *DomainManger*. Afterwards, the *ApplicationFactory* creates an *Application* servant and a CORBA *Application* object with a unique hash that serves as CORBA object id. Then the *ApplicationFactory* adds via the *addApp()* function the *Application* servant pointer to the *ServantActivator*'s list of the *Application* servant pointers (2). When an SCA client attempts to invoke an operation on a specific *Application* CORBA object, the POA will call the *ServantActivator incarnate()* operation, passing the *Application* hash embedded embedded in the *Application* object reference. The implementation of *incarnate()* will return an *Application* servant, which will serve the SCA client request, with an accordant hash,

When the SCA client releases the *Application* (3), the *Application* servant deactivates its corresponding *Application* CORBA object and returns to the SCA client (4). Finally, after all active requests to the *Application* CORBA object are processed, the POA calls the *ServantActivator etherealize()* operation, passing the *Application* hash, and the implementation of *etherealize()* deletes the *Application* servant, respectively (5).

In this solution, the *Application* servants are deleted safely and immediately. Besides, this solution requires only one callback CORBA object, that is, the *ServantActivator*. However, this solution heavily uses advanced CORBA features. For instance, the old BOA does not provide servant managers. Furthermore, some lightweight ORBs like ORBitcpp [13] do not support the servant manager interfaces. Since we are considering alternative ORBs with better performance [14], we further investigate other solutions to ensure portability and thus reusability.

3.2 Thread per Application Approach

This solution relies on a clean-up thread instead of a call back object of S1. Figure 5 shows this approach. For simplicity, we assume that the *Application* has been already created. At first the SCA client invokes the *releaseObject()* operation on this *Application* reference (1). The *releaseObject()* operation un-registers the *Application* from the *DomainManager* and launches a clean-up thread. After the *releaseObject()* operation has returned to the SCA client (2), the clean-up thread deletes the *Application* servant (3).

Basically, the clean-up thread maintains the C++ *Application* servant pointer and the POA reference, wherein the *Application* CORBA object is registered. The clean-up thread deactivates the *Application* CORBA object and then periodically checks whether the ORB has actually succeeded the deactivation. If the *Application*

CORBA object is not active, the clean-up thread deletes the *Application* servant from the memory.

The clean-up thread deletes the *Application* servant only if the corresponding *Application* CORBA object is not active. Therefore, this approach does not make assumptions about the time it takes for *releaseObject()* to return and thus can safely release the *Application* solving our problem.

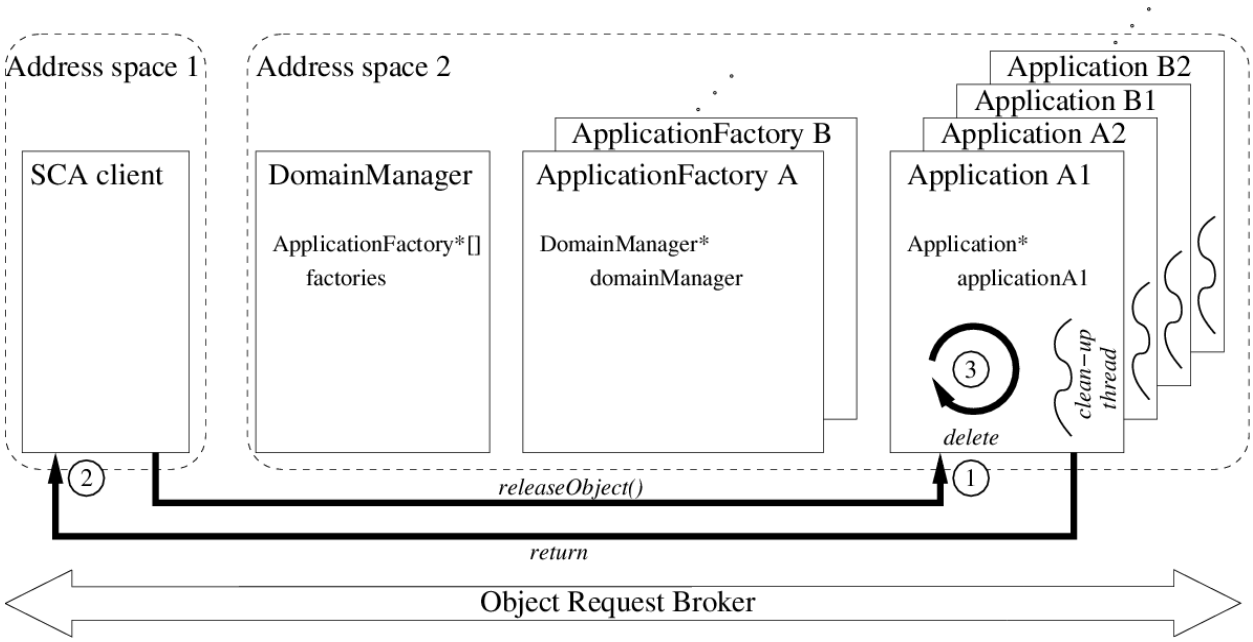

Figure 5. Clean-up thread deletes *Application* servants.

However, the time when the ORB deactivates the CORBA object is unpredictable. The exact point of the time, when the ORB turns the state of the *Application* CORBA object from active to deactivated, depends upon the requests arrived before *deactivate_object()* was called. As a consequence, the *Application* runs after the call of *deactivate_object()* as a sort of zombie process and occupies memory without knowledge of the SCA framework. Since there can be many *Applications*, there might be a lot of such zombie *Applications*. Because each *Application* in zombie state occupies a considerable amount of memory and maintains its own clean-up thread, which needs additional resources, the system might run out of memory. Furthermore, the start of a clean-up thread might fail, for example, due to a lack of memory. Therefore, the *Application* servant might not be deleted. Hence, it is better that the clean-up thread for a particular *Application* servant is already running before the *Application* gets created. Finally, the SCA framework is not aware of the exit status of the clean-up thread.

OSSIE's SCA implementation makes use of a similar, but weaker approach for the MS Windows platform. Inside OSSIE's SCA *Application* servant, *releaseObject()* starts a clean-up thread. The OSSIE clean-up thread sleeps for one second before it calls *deactivate_object()* on the corresponding *Application* CORBA object. After one second, the OSSIE clean-up thread deletes the *Application* servant without checking whether the *Application* CORBA object is active or not. Because there is no guarantee that *releaseObject()* will be finished

within one second, the OSSIE clean-up thread might delete an SCA *Application* servant that incarnates an active SCA *Application* CORBA object.

3.3 Singleton Domain Manager Thread Approach

All disadvantages of S2 can be eliminated if the clean-up thread is launched by the *DomainManager* servant. This solution is shown in Figure 6. For simplicity, we assume that the SCA client already holds the *Application* reference. At first, the SCA client invokes the *releaseObject()* operation on this *Application* reference (1). The *releaseObject()* operation un-registers the *Application* from the *DomainManager*. Then the *Application* servant delivers a C++ pointer of itself using the non CORBA operation *addDefunctApp()* to the *DomainManger* servant (2). Then, the *DomainManager* servant passes this C++ *Application* servant pointer to the clean-up thread. After the *releaseObject()* operation has returned to the SCA client (3), the clean-up thread deactivates and deletes the *Application* servant like in S2 (4). Note that this solution requires the *Application* servant to hold a C++ pointer to the *DomainManager*. The *Application* servant gets this pointer via its constructor.

The *DomainManager* keeps track of whether the number of *Applications* in the zombie state reaches a critical point to jeopardize a proper RSCA behavior.

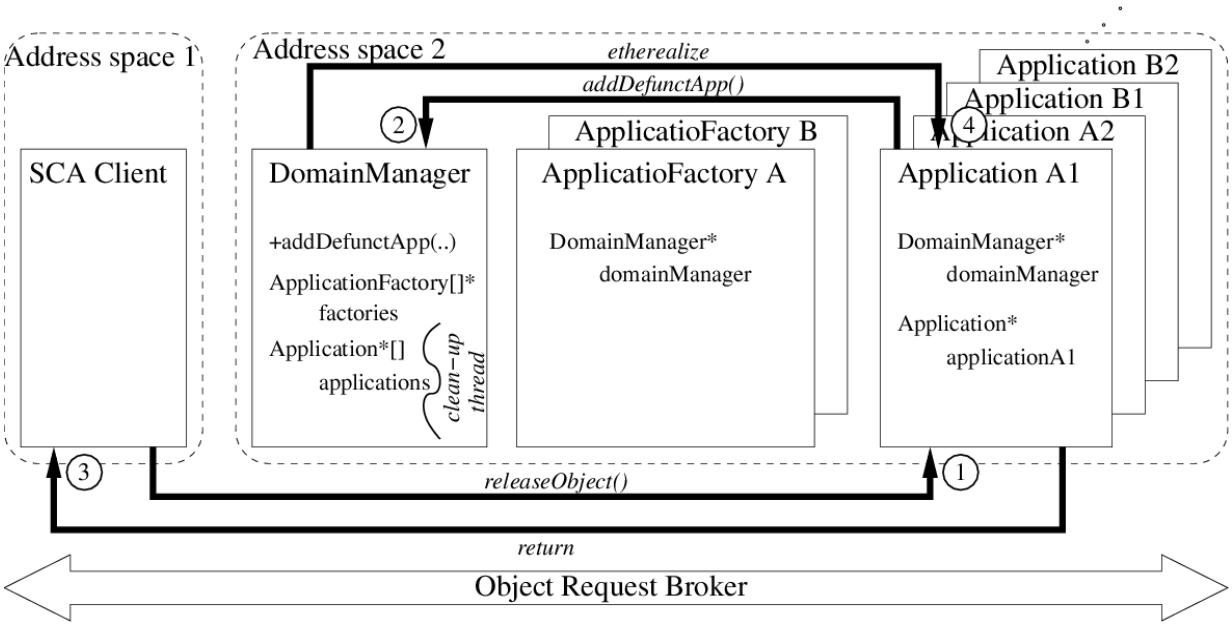

Figure 6. Clean-up thread is launched by the *DomainManager* servant.

This approach also offers a convenient way to delete *ApplicationFactories*. Since an *ApplicationFactory* object does not have an operation to release itself, the *DomainManager*'s *uninstallApplication()* operation is responsible to delete *ApplicationFactories*. However, like an *Application* CORBA object, the *ApplicationFactory* object must be deactivated before the *ApplicationFactory* servant gets physically removed from the memory. The same clean-up thread, which is in charge of deleting *Application* servants, can safely

deactivate and delete the *ApplicationFactory* servants.

4. Our Implementation

We have applied S3 to our RSCA implementation as the solution to the *Application* release problem. We named this releasing pattern *Vulture* design pattern, which comes from the scavenging habits of the vultures, feeding from carcasses of dead animals, looking down from the sky. Their habits are analogous to the aspect of the clean-up thread in the design pattern that monitors the deactivated objects and sweeps them away.

Our RSCA was implemented using the gcc compiler and Linux v. 2.4.20. The RSCA is highly portable and currently the TAO real-time ORB 1.3.1 [15], omniORB 4.0.5 [16], and ORBExpress [17]. We have implemented the RSCA's clean-up thread as a Pthread [18]. In contrast to RSCA, the OSSIE project uses ACE [19] tasks.

In order to apply the Vulture design pattern, the *DomainManager* servant maintains two lists of object descriptors: (1) a list of the *Application* descriptors and (2) a list of the *ApplicationFactory* descriptors. An object descriptor encapsulates an *Application* servant pointer *application_ptr* or an *ApplicationFactory* servant pointer *applicationFactory_ptr* as well as a POA reference *poa* and the CORBA object id *oid*. To notify the *DomainManager* that the *Application releaseObject()* operation was called, we have implemented the *DomainManager addDefunctApp(applicationDescriptor*)* function. The *Application* must pass, using this function, the object descriptor of itself to the *DomainManager* when *releaseObject()* is invoked by an SCA client. Since an *ApplicationFactory* is created by the *DomainManager*, the *DomainManager* adds an accordant entry to the *ApplicationFactory* descriptors list when an SCA client uninstalls an *Application*.

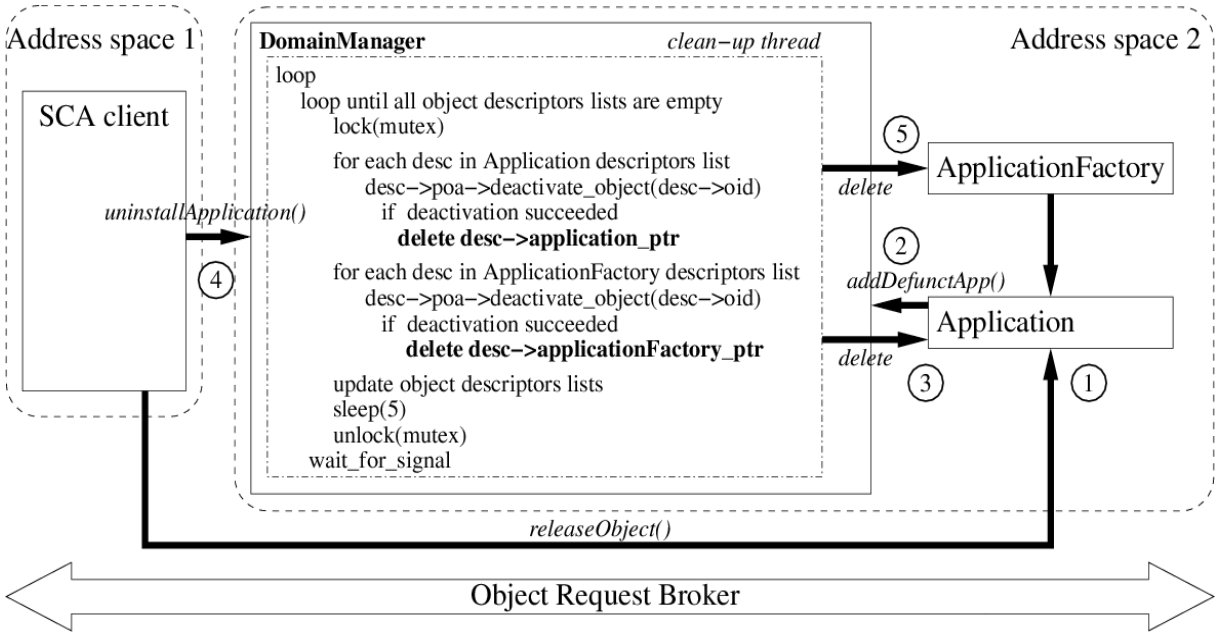

Figure 7. Our clean-up thread implementation.

In Figure 7, the *DomainManager* launches the clean-up thread inside its constructor with an argument comprised of a pointer to the list of the *Application* descriptors, a pointer to the list of the *ApplicationFactory* descriptors, and a mutex to protect the object descriptor lists.

When the SCA client calls *releaseObject()* on an *Application* (1), the *Application* creates an object descriptor of itself and passes the descriptor to the *DomainManager* via *addDefunctApp(applicationDesc*)* (2). Afterwards, the *DomainManager* adds the *Application* descriptor to the *Application* descriptors list and signals the clean-up thread. The clean-up thread now starts to iterate through the *Application* descriptors list and to deactivate the *Application* CORBA objects for each entry. The clean-up thread checks whether the deactivation succeeded for every entry (Figure 7). If the *Application* CORBA object is deactivated the clean-up thread deletes the *Application* servant and removes the *Application* descriptor from the *Application* descriptors list (3). In case of a still ongoing deactivation, the clean-up thread will reiterate through the *Application* descriptors list after sleeping 5 seconds. If both object descriptors lists are empty, the clean-up thread will start to wait for a wake up signal.

When the SCA client calls the *uninstallApplication()* on the *DomainManager* (4), the *DomainManager* creates an object descriptor for the *ApplicationFactory* that the SCA client intends to uninstall, adds this descriptor to the *ApplicationFactory* descriptors list, and wakes up the clean-up thread. The clean-up thread will iterate through the *ApplicationFactory* descriptors list deactivating and deleting the *ApplicationFactory* objects, respectively (5).

5. Conclusion

In this paper, we have investigated design patterns for releasing the SCA applications in a C++ SCA implementation. The goal is to release the SCA applications safely and cleanly without any memory leakage. Since it is expected that many SCA applications will be instantiated and released during the SCA run time, the SCA platform will suffer from out of memory or unexpected ORB behavior if SCA applications are released improperly.

We have given three alternative design pattern solutions to complete the SCA application release process. The first solution deletes SCA application servants using advanced CORBA features. This solves the problem simply and cleanly, but is not portable on old or on some lightweight CORBA implementations. In the second solution, a clean-up thread is launched by an SCA application servant that safely deletes this application servant after the deactivation of the application CORBA object completes. This solution has a better portability characteristic than the first one since it does not depend on the advanced CORBA features. However, the exact behavior of the clean-up thread is unknown by the SCA domain manager; therefore, the *Application* objects might run in some kind of zombie state. The third solution improves the previous one by relocating the clean-up thread launch to the SCA domain manager. Among those three solutions, we have selected the third one and applied it to our RSCA implementation. We named the design pattern we adopted, *Vulture* design pattern, which comes from the

scavenging habits of the vultures. The clean-up thread in the Vulture design pattern of RSCA deletes the *Application* servants without corrupting with the POA's Active Object Map, guaranteeing no memory leakage caused by *Application* servants.

References

[1] Joint Tactical Radio Systems. "Software Communications Architecture Specification V2.2." November 2002.

[2] Joint Tactical Radio Systems. "Support and Rationale Document for the Software Communications Architecture Specification (v2.2)." December 2002

[3] Joint Tactical Radio Systems. "Software Communications Architecture Specification V2.2. API Supplements." November 2002.

[4] Object Management Group. "The Common Object Request Broker Architecture: Core Specification Revision 3.0." December 2002.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides "Design Patterns" July 1997.

[6] American International Standards Institute. INCITS/ISO/IEC 14882-2003 "Programming languages – C++" 2003.

[7] S. Hong, J. Lee, H. Eom, and G. Jeon. "The Robot Software Communications Architecture (RSCA): Embedded Middleware for Networked Service Robots" Accepted to the 14th IEEE International Workshop on Parallel and Distributed Real-Time Systems, Island of Rhodes, Greece, April 25-26, 2006

[8] J. Lee, S. Kim, J. Park, and S. Hong. "Q-SCA: Incorporating QoS Support into Software Communications Architecture for SDR Waveform Processing" Accepted for the Journal of Real-Time Systems

[9] S. Kim, J. Masse, and S. Hong. "Dynamic Deployment of Software Defined Radio Components for Mobile Wireless Internet Applications." In Proceedings of International Human.Society@Internet Conference (HSI), June 2003.

[10] RSCA project home page, http://rsca.snu.ac.kr

[11] James Gosling, Bill Joy, Guy L. Steele Jr, and Gilad Bracha "The Java Language Specification, Third Edition", June 2005.

[12] OSSIE project home page, http://ossie.mprg.org

[13] ORBitcpp project home page, http://orbitcpp.sourceforge.net/

[14] Tuma P, and Buble, A. "Overview of the CORBA Performance", In Proceedings of the 2002 EurOpen.CZ Conference, Znojmo, Czech Republic, September 2002.

[15] F. Kuhns, D. D. Schmidt, et al. "The Design and Performance of a Real-Time Object Request Broker." In Proceedings of IEEE Real-Time/Embedded Technology and Applications Symposium, May 2000.

[16] omniORB project home page, http://omniorb.sourceforge.net

[17] ORBExpress home page http://www.ois.com

[18] IEEE and The Open Group. IEEE Standard 1003.1-2001. IEEE, 2001.

[19] Douglas C. Schmidt “An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse” In USENIX login magazine, Tools special issue, November 1998.