JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 7, July-August 2004

Online Upgrade of Object-Oriented Middleware

Apostolos Zarras, Computer Science Department, University of Ioannina, Greece.

Abstract

A system that relies on object-oriented middleware comprises computational objects that are specific to the system's purpose and middleware objects used for the transparent integration of the former. The efficient maintenance of such a system involves the dynamic upgrade of the aforementioned entities. So far, there have been various approaches dealing with the online upgrade of computational objects. This paper examines the second part of the problem: the online upgrade of middleware. Starting from the identification of requirements for the consistent upgrade of middleware, we result in the design and experimental evaluation of an architectural style for CORBA-based upgradeable middleware.

1 INTRODUCTION

Object-oriented middleware is the current practice in the development of open distributed processing systems (ODPSs). An ODPS consists of basic architectural elements that we call *objects*. An object provides one or more *interfaces* that can be used by others to access, or modify its internal state. We can characterize objects as either *computational*, providing functionality that is specific to the system's purpose, or *middleware*, providing reusable solutions to problems encountered in many different kinds of ODPSs [Bernstein96]. Middleware objects are parts of an overall infrastructure that consists of a base communication mechanism, often-called *broker*, and a set of *complementary services*. The broker masks differences in data representation and communication mechanisms to enable interoperation between computational and middleware objects. The set of complementary services may include repository services (e.g., naming, trading, etc.), security services and coordination services (e.g., transactions, fault tolerance, persistence, etc.).

The evolution of the middleware concept led in the development of various standards and infrastructures like CORBA, J2EE and COM+, whose use reduces the developer's effort when building an ODPS [Zarras04]. However, the ODPS development process is iterative and the maintenance of the system involves the upgrade of the computational, or the middleware objects that constitute it. The need to upgrade may arise from rapidly

Cite this article as follows: Apostolos Zarras: "Online Upgrade of Object-Oriented Middleware", in *Journal of Object Technology*, vol. 3, no. 7, July-August 2004, pp. 121-140.
http://www.jot.fm/issues/issue_2004_07/article3

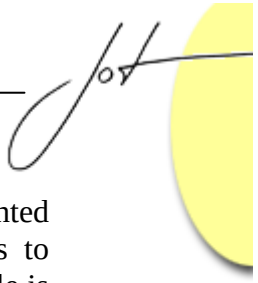
fluctuating environmental conditions found, for instance, in the area of mobile computing. The availability of more efficient and reliable infrastructures further triggers the need to upgrade the ODPS to meet the often-conflicting dependability and performance requirements found in areas such as telecommunications, real-time and embedded systems.

In the late 90's, there has been a substantial research effort towards facilitating the upgrade of ODPSs. Those efforts gave birth to *reflective middleware infrastructures*, which expose functionality that allows monitoring and adapting, at runtime, computational and middleware objects [Blair *et. al.*98, Kon *et. al.*02]. An issue, however, that raised in the case of reflective middleware infrastructures is that they offer too much flexibility [Blair *et. al.*00]. More specifically, they allow building upgradeable ODPSs, which can be changed in ways that may jeopardize their overall integrity. The previous led us in the development of a middleware service for the consistent upgrade of *computational objects* in CORBA-based ODPSs [Bidan *et. al.*98]. Other similar approaches dealing with the upgrade of computational objects have been proposed since then [Rodr. *et. al.*99, Alm. *et. al.*01]. Moreover, the OMG recently adopted a standard for the online upgrade of CORBA objects [CORBAUv1.0].

This paper examines in depth the second part of the upgrade problem: *the online consistent upgrade of middleware objects*. The approach we present here relies on our previous efforts presented in [Blair *et. al.*00]. In particular, in Section 2, we briefly describe the architectural style of conventional CORBA-based ODPSs (we use the term *conventional* to refer to ODPSs whose middleware objects cannot be upgraded). In Section 3, we detail the problem of upgrading middleware in conventional CORBA-based ODPSs. In Section 4, we identify requirements for the consistent upgrade of middleware objects. Moreover, we examine traditional dynamic reconfiguration approaches and we argue about their suitability for the middleware upgrade problem. Based on the requirements analysis, in Section 5, we propose a CORBA-based architectural style for ODPSs, whose middleware objects can be upgraded without compromising the ODPSs' integrity. In Section 6, we evaluate the proposed architectural style. Finally, Section 7 summarizes this paper with our contribution and the future perspectives of this work.

2 CONVENTIONAL CORBA-BASED SYSTEMS

A conventional CORBA-based ODPS is organized in a set of capsules for the purpose of encapsulation of processing and storage. A capsule comprises a number of basic units of computation and data-store, which are called *servants* [CORBAv3.0.2]. A servant is an implementation-language specific element (e.g., a C++ object, a Java object, etc.). The capsule is responsible for creating references to *provided CORBA objects* that conceptually represent the encapsulated servants. A CORBA object offers an *interface* that defines the object's type. Interfaces are specified using CORBA IDL, a purely declarative language that supports interface inheritance. In particular, an interface specification includes a number of *operations* and *attributes* that can be used towards accessing and changing the object's internal state. An attribute actually corresponds to operations that can be used to



access or modify the attribute's value. Interface operations and attributes are implemented by servant-specific functionality (e.g., C++, or Java class methods). All references to conceptual CORBA objects inherit from a standard interface, named Object. The capsule is further in charge of obtaining references to required CORBA objects, i.e., CORBA objects provided by other capsules and used in the context of this one. Holding a reference to a CORBA object allows invoking its operations. An invocation results in a CORBA *request* delivered to the servant that is represented by the referenced CORBA object. Technically, request delivery relies on the proxy-pattern [Shapiro86].

To improve the understanding of the middleware upgrade problem, we distinguish between references to computational and references to middleware objects. Each capsule holds a reference to a middleware object that realizes the standard ORB interface, which defines operations for the initialization and management of the CORBA base communication mechanism, *i.e.*, the CORBA broker. In addition, the ORB object offers operations for obtaining references to other middleware objects that are parts of the implementation of complementary CORBA services (e.g., naming, trading, transactions, etc. [CORBASrvs]). Each capsule further holds references to a number of objects that realize the standard POA interface. Each POA object manages the reference creation, activation, deactivation and request flow for a subset of CORBA objects that are included in the capsule. The previous is achieved according to a number of POA policies, which are configured properly by the capsule. The POA references held by the capsule are organized in a tree structure, which determines the order of the POA objects' activation/deactivation. Moreover, it determines the order of activating/deactivating the objects managed by the POA objects.

3 THE PROBLEM OF ONLINE MIDDLEWARE UPGRADES

Regarding conventional CORBA-based ODPSs, we can distinguish two cases of online middleware upgrade situations that may arise in practice; each one of them has a different impact on the overall ODPS architecture.

The first case amounts to dynamically upgrading the implementation of a standard middleware service used in the ODPS for another one that still complies with the same CORBA standard. The new service implementation may be either a later version of the old one (both implementations come from the same vendor) or, completely different from the implementation that is currently in use (the implementations come from different vendors). Performing the upgrade involves identifying all capsules that hold references to middleware objects that are part of the old service implementation (*i.e., the affected capsules*), initializing the new service implementation within those capsules, and changing the old references into references to middleware objects that are part of the new service implementation.

The second and more complicated case amounts to dynamically upgrading the implementation of the CORBA broker for a new one that still complies with the CORBA standard. As with middleware service upgrades, the new broker implementation may be

either a revised version of the old one or, it may be completely different from it. The middleware upgrade in this case affects all of the ODPS capsules. More specifically, it amounts to initializing the new broker implementation in each capsule, obtaining references to new ORB and POA objects, creating new references to provided computational CORBA objects that represent the encapsulated servants and obtaining new references to the required computational and middleware objects.

4 BACKGROUND AND REQUIREMENTS ANALYSIS

As we discussed in the previous Section, the main goal of the online middleware upgrade is to dynamically change the implementation of *some* or, in the worst-case, *all* middleware objects used in an ODPS architecture. Moreover, we have to change references to old middleware objects into references to new middleware objects. Hence, we can reduce the problem of middleware upgrade into a problem of *dynamic reconfiguration*. Pioneer work in the field of dynamic reconfiguration includes [Kramer *et. al.*90]. In this paper, the authors provide a foundation of properties that should characterize a process, which soundly manages changes in the configuration of a running system. The first essential property discussed in the paper is the one of *preserving the correct execution of the system*. As said in the paper, *"changes should leave the system in a consistent state"*. A consistent state is one from which the ODPS can continue providing correct service rather than progressing towards an error state. The definition of consistent states for a particular ODPS involves specifying certain *invariants* (e.g., safety and liveness properties) that must hold during the ODPS execution. Moreover, defining consistent states relates to the specification of a *fault model* regarding the middleware and the computational objects that constitute the ODPS. The second essential property discussed in [Kramer *et. al.*90] is the one of *introducing minimal disruption*, while changing the configuration of the system. Upgrading some ODPS objects should not impose suspending the execution of the whole system. At this point, it is worth noticing that in real-time, critical systems, disrupting the execution of the system also implies that correct execution is not preserved. Hence, for the case of middleware upgrades we can combine the two properties discussed in [Kramer *et. al.*90] into the following:

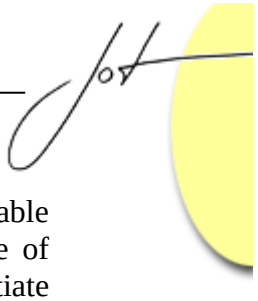
- *The middleware upgrade process should leave the ODPS in a correct state, while minimizing disruption.*

A third property identified in [Kramer *et. al.*90] is *reachability of changes*. In analogy to the previous, the second requirement for consistent middleware upgrade is that:

- *The middleware upgrade process should take place within finite time.*

Leaving the System in a Correct State

Based on the properties identified in [Kramer *et. al.*90], Kramer and Magee proposed a corresponding reconfiguration strategy, which is further enhanced in [Goud. *et. al.*96]. According to this strategy, the process that manages changes first identifies a minimal set



of architectural elements *affected* by those changes. Non-affected elements should be able to operate normally, as if there was no reconfiguration in progress. Taking the case of substituting an old element with a new one, the affected elements are those that may initiate requests to the element. Moreover, to substitute the old element the following steps should be taken:

1. Block all the affected elements.
2. Wait until the old element is not engaged in serving any pending requests, or in any request that it initiated.
3. Remove all links to, and from the old element and then remove the element itself.
4. Create the new element and setup the communication links as prescribed by the architecture of the ODPS.
5. Unblock all the elements that were blocked during the first step.

By taking a closer look at the previous strategy, we can conclude that it is not directly applicable in middleware upgrades. In particular, the strategy would leave the ODPS in a correct state after the upgrade, but it would not minimize disruption. In the case of a broker upgrade, for instance, all of the ODPS capsules are affected. Similarly, if we aim at upgrading a frequently used middleware service like the ones for naming, trading, security, transactions, etc. most of the ODPS capsules are affected by the change. In general, it is a common case that *most of the ODPS capsules are affected by middleware upgrades*. However, if we cannot minimize the number of ODPS capsules that are disrupted by the upgrade process, we can still minimize the duration of this disruption. In other words, our requirement for minimal disruption is refined as follows:

- *Whatever is the disruption introduced by the upgrade process, it does not last for long.*

The duration of a service upgrade equals to the time it takes the middleware objects that are parts of the service to reach a state where they are not engaged in serving any pending requests, or in any request that they initiated. From now on, we call such states of middleware objects, *idle states*. The duration of a broker upgrade equals to the time it takes the middleware objects of the broker to reach an idle state. Recall that a broker upgrade also involves upgrading the references to the middleware objects that are part of the services used in the ODPS. In order to do the previous we have to re-initialize the services; the re-initialization must take place at the time when the middleware objects that are part of the services are in an idle state. Hence, the duration of the broker upgrade also includes the time it takes the services to reach such a state.

In a broker upgrade, *pending requests* are those created when a required computational object is invoked because their delivery involves the implicit use of middleware objects that are part of the broker. Moreover, pending requests are those created when a middleware object is explicitly invoked. From the point of view of the broker, a *request stops being pending* at the time when the invoked object finishes serving it. In the case of a service upgrade, *pending requests* are those created when a middleware object that is part of the service is explicitly invoked. It is worth noticing that from the point of view of some services, *a request stops being pending with the completion of another request*. The previous refers to services that enable the execution of *sessions i.e.*, sets of requests that

share common properties. A typical example is the CORBA Transaction Service (TS) [CORBASrvs]. A CORBA transaction is a set of CORBA requests that executes atomically. From the point of view of TS, a request that creates a new transaction remains pending until the completion of a corresponding request that commits, or aborts the transaction. Another example is the CORBA Security Service (SS) [CORBASrvs] that enables creating security sessions consisting of requests, which share common security properties. Finally, consider the CORBA Concurrency Service (CS) [CORBASrvs] that provides a simple locking mechanism, which allows executing a set of requests on an object, in isolation from other requests. In this case, a request that acquires a lock remains pending until the completion of a corresponding request that releases the lock.

To reduce the duration of waiting, and consequently the duration of the disruption introduced by the upgrade process we have to change the middleware objects while there *exist pending requests*. The previous implies that the state of the new middleware objects must contain the necessary information for completing requests that were pending at the beginning of the upgrade. Hence, the upgrade process involves transferring state information from the old middleware objects to the new ones. Technically, transferring state information depends on the state transfer facilities that are provided by both the old and the new middleware objects and is feasible only for some states of the old middleware objects, which we call *safe states*.

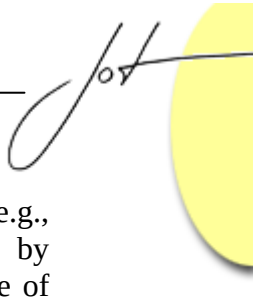
An important remark resulting from the above discussion is that safe state detection mechanisms *depend on a particular upgrade situation*. Hence, building upgradeable middleware requires knowing, in advance, the exact upgrade situations that may occur. The previous issue is a usual limitation of typical reconfiguration approaches that are based on state transfer [Warren *et. al.*95, Hauptm. *et. al.*96]. This limitation becomes a major drawback in the case of middleware upgrades because the requirements of the middleware-based ODPSs and the availability of services provided by existing middleware infrastructures may change in any arbitrary way. Thus, there is no way to know, or even predict, in advance, the upgrade situations that may arise during the lifetime of the ODPS. To overcome this problem we rely on the following idea:

- *The middleware upgrade process relies on state detection mechanisms that adapt to a particular upgrade situation.*

Reachability of Changes

Regarding reachability of changes, Kramer and Magee propose constraining the behavior of the ODPS. More specifically, they assume that the interactions between the ODPS objects complete within finite time. Following their approach in the case of middleware upgrades means that the responsibility of performing the upgrades within finite time is mainly assigned to the ODPS developer, while the upgrade process passively waits for the middleware objects to reach an idle/safe state.

An alternative is to assign the responsibility of upgrading middleware within finite time entirely to the upgrade process, by putting it in charge of enforcing an interaction that drives the middleware objects into an idle/safe state. The state of the middleware objects



changes as a result of the interaction between them and the computational objects (e.g., computational objects may initiate and terminate transactions, or security sessions by invoking specific operations provided by the middleware objects). Moreover, the state of the middleware objects changes because of events that do not relate with the computational objects that use them (e.g., network timeouts may cause the middleware to re-transmit requests issued by ODPS objects). Finally, the state of the middleware objects may change due to interaction between them (e.g., a transaction manager using a resource manager and a locking mechanism). Based on the above, the upgrade process must behave as follows:

- Play the role of the computational objects, by issuing requests that cause the middleware objects to reach an idle/safe state (e.g., issue requests for aborting all pending transactions).
- Play the role of the software, or hardware that the middleware relies on, by generating independent events, which cause the middleware objects to reach an idle/safe state (e.g., introduce network and hardware failures, making the middleware rollback all pending transactions).
- Play the role of the middleware objects, by generating events that make the middleware objects look as if they are in an idle/safe state (e.g., raise exceptions, which notify the computational objects that all pending transactions are rolled-back).

Assuming that it is always possible to enforce an interaction that drives the middleware objects to an idle/safe state, we can apply this idea to upgrade them within a finite time. However, the upgrade process must disturb the execution of the ODPS as less as possible. Consider, then, the case of aborting some transactions that were just about to complete before the upgrade. Typically, the ODPS would have to retry them after the upgrade. Hence, the requirement for minimal disruption is not preserved. Furthermore, enforcing an idle/safe state is misleading when we assess the ODPS quality (e.g., if well-formed transactions are aborted due to an enforced interaction, the ODPS user gets the impression that the overall system is not reliable).

Hence, actively driving middleware objects to an idle/safe state is a rather dangerous approach, while passively waiting for them to reach such a state establishes a strong dependency between the upgrade process and the behavior of the ODPS. Based on the previous remark, we follow an approach that stands between the two and relies on the idea below:

- *Assign to the upgrade process part of the responsibility for upgrading middleware within finite time, by putting it in charge of making long waits less likely.*

To achieve the above we use mechanisms that *selectively block invocations issued to computational* and *middleware objects.* The purpose of blocking is to prevent the creation of requests that keep the middleware objects away from an idle/safe state. The requests that need to be blocked to facilitate reaching an idle state do not depend on a particular upgrade situation. For example, preventing the creation of requests that initiate new transactions is a way to help a transaction manager reach an idle state. On the other hand, the requests that need to be blocked to reach a safe state depend on a particular upgrade situation. Suppose,

for instance, that we can change the transaction manager even if there exist pending transactions, as long as the manager is not engaged in performing a two-phase-commit protocol between the participants of a pending transaction. In this case, the blocking mechanisms should prevent requests for committing pending transactions. Hence, the blocking mechanisms that contribute in reaching a safe state are specific to the upgrade situation that takes place. In order to deal with the lack of *a-priory* knowledge regarding this situation, we rely on the idea that we already proposed for the case of state detectors. In particular:

- *The middleware upgrade process is based on blocking mechanisms that adapt to a particular upgrade situation.*

5 CORBA-BASED ARCHITECTURAL STYLE FOR UPGRADEABLE MIDDLEWARE

Based on the requirements analysis detailed in the previous Section, we extend conventional CORBA-based ODPS architectures with additional elements that detect and assist in reaching an idle/safe state and elements used for the coordination of the overall upgrade process. In particular, Figure 1 gives the architectural style of CORBA-based ODPS architectures, whose middleware can be upgraded.

Basic Architectural Elements

Detecting an idle/safe state involves monitoring the interaction between computational and middleware objects and inspecting the state of the latter. Blocking requests that drive middleware objects away from an idle/safe state also requires monitoring the interaction between computational and middleware objects. Technically, we achieve the previous using a set of wrapper objects in every ODPS capsule. Each wrapper corresponds to a required computational or middleware object and embeds a reference to that object. The wrapper's interface is identical with the interface of the embedded object and it is further derived from the UpgradeableObject interface defined in Figure 2(a). At this point, we distinguish between wrappers that monitor pending requests from the broker's point of view and wrappers that monitor pending requests from the point of view of services that allow the creation and execution of sessions (see Section 4). Hereafter, we call the former kind of wrappers *broker-specific* and the latter *session-specific*.

The wrappers are actually pseudo CORBA objects (*i.e.*, they are used only within the capsule that contains them) and hence, there is no need to employ the proxy pattern in order to communicate with them. The reason why we use wrappers instead of CORBA interceptors (*i.e.*, user-specific objects, which we can register to the CORBA broker; thereafter the broker calls operations on those objects before issuing a request and after delivering one) is that interceptors are parts of the broker implementation, which eventually becomes the subject of an upgrade. This fact unnecessarily complicates the middleware upgrade and strengthens the dependency between the duration of the upgrade process and the implementation of CORBA broker in use.

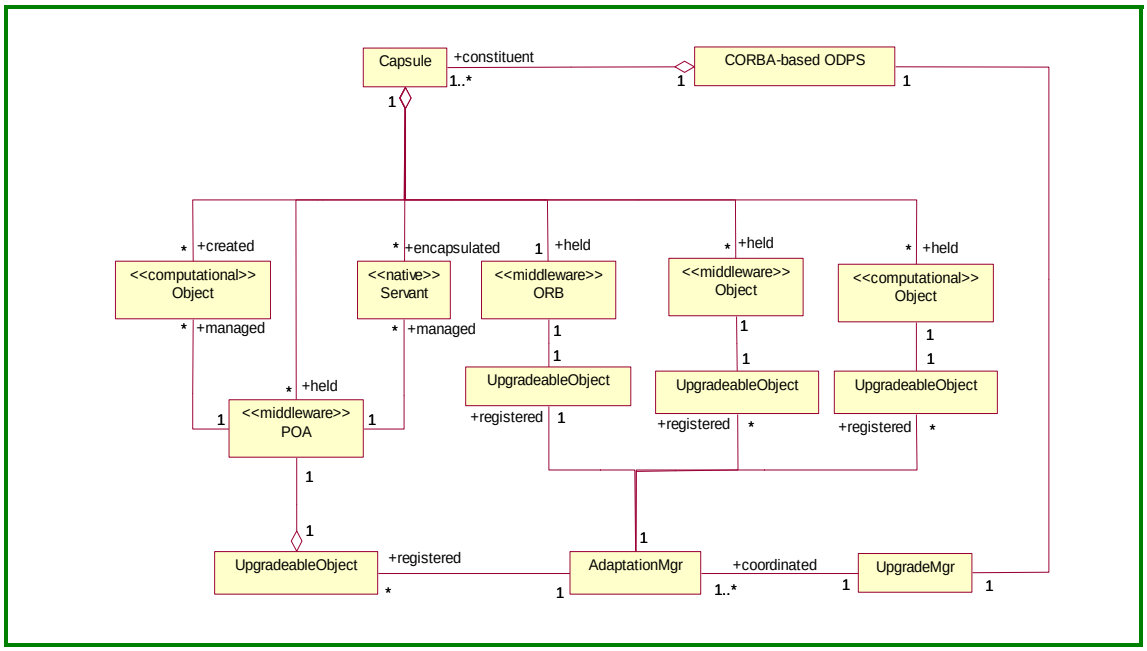


Figure 1: Extending the conventional CORBA architectural style to support middleware upgrades.

Creating the wrapper objects is a responsibility of the ODPS capsules. More specifically, when a capsule obtains a reference to a required CORBA object it creates a corresponding wrapper and associates it with the reference by setting the theObjectRef attribute. The capsule should always use the wrapper to invoke operations on required CORBA objects. In other words, to invoke an operation on a required CORBA object, the capsule uses the corresponding operation of the wrapper that encapsulates a reference to it.

Each capsule further creates an object that implements the AdaptationMgr interface defined in Figure 2(c). The AdaptationMgr object coordinates the upgrade within the capsule. Moreover, the capsule registers *structural information* to the AdaptationMgr object, using the registration operations showed in Figure 2(c). The structural information serves for creating new references to computational and middleware objects (e.g. ORB, POA objects) and for reconstructing the associations between them in the case of a service or a broker upgrade, (e.g., reconstructing the associations between POA objects, servants and provided CORBA objects). More specifically, the wrappers that embed references to POA objects are registered using the registerPOA() operation, which accepts as arguments both the wrapper that embeds the POA object and the wrapper that encapsulates its parent POA object. Similarly, the wrappers that contain references to provided computational objects are registered, using the registerProvidedObject() operation, which takes as arguments the wrapper of the computational object, the wrapper of the POA object that manages the computational object's lifecycle and the object's name. Finally, information regarding required references to computational objects and middleware objects that are part of complementary CORBA services is registered using the registerRequiredObject() operation. This operation takes as arguments the name of the object or the service, a wrapper to a computational or middleware object, and a Boolean parameter whose value determines if the wrapper is broker-specific or session-specific.

(a) UpgradeableObject interface.

(b) UpgradeMgr interface.

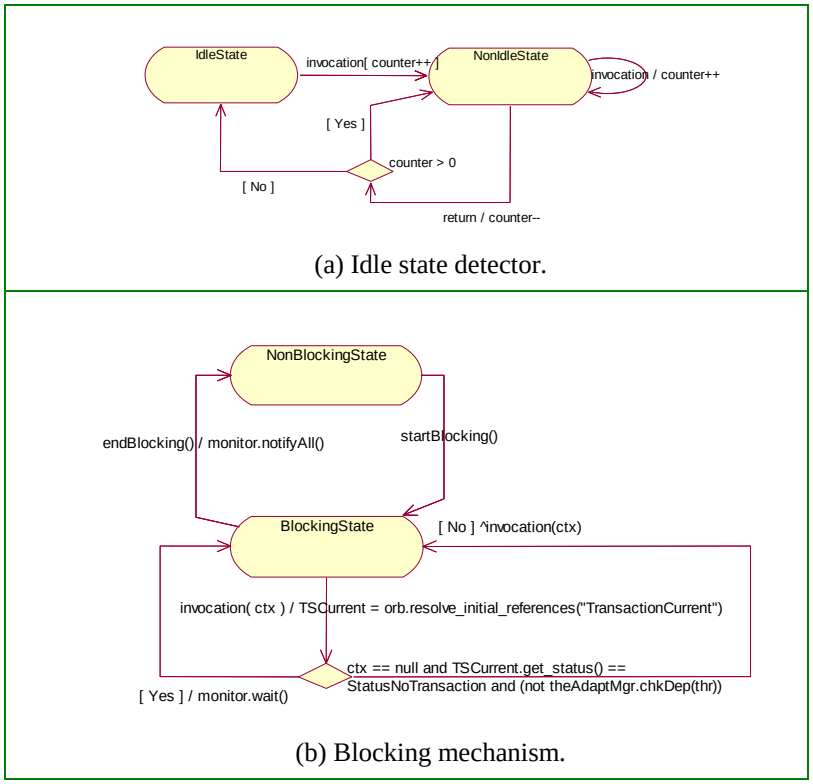
(c) AdaptationMgr interface.

Figure 2: Interfaces of the basic entities that facilitate the middleware upgrade process.

The creation of an AdaptationMgr object is followed by the object's registration to a central control unit that implements the UpgradeMgr interface (defined in Figure 2(b)) and coordinates the overall upgrade process. The UpgradeMgr object further serves as a lightweight naming facility; it maintains a registry of name contexts that contain associations between names and CORBA object references created by capsules. The use of the registry is crucial for upgrading references to required computational objects. As with the case of interceptors, we avoid using an implementation of the standard CORBA naming service because the former may eventually become the subject of an upgrade. Capsules that create references to provided computational CORBA objects register them to the UpgradeMgr object using the createdObjectRef() operation. On the other hand, capsules may obtain references to required computational objects using either the lookupObjectRef() operation of the UpgradeMgr object, or any other standard CORBA repository service (e.g., naming, trading).

The Behavior of the Wrapper Elements

In general, we decompose a wrapper implementation into the state detection part and the blocking mechanism part. Initially, each ODPS capsule contains wrappers whose implementation detects and facilitates reaching an idle state since this kind of support does not depend on a particular upgrade situation.



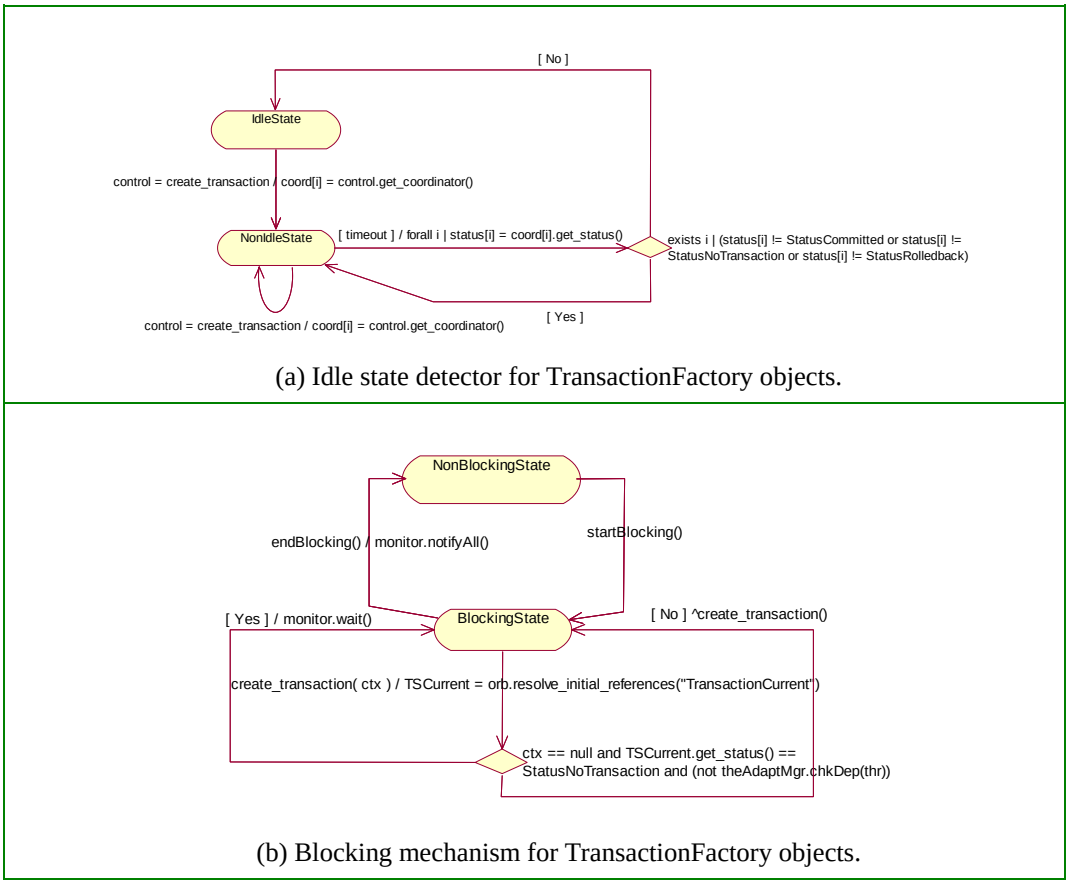
(a) Idle state detector.

(b) Blocking mechanism.

Figure 3: The behavior of broker-specific wrappers that detect idle state.

As already mentioned the broker-specific wrappers provide different interfaces depending on the type of the objects, whose references they encapsulate. However, the way they monitor requests is the same for all of them and, hence, we can generate their implementation based on the IDL specifications of their interfaces. In particular, the behavior of the state detection part of broker-specific wrappers conforms to the pattern described in the state-chart diagram of Figure 3(a). Initially, each wrapper is in the IdleState. Upon the invocation of an operation, the wrapper gets into the NonIdleState and a counter is increased. Subsequent invocations further increase the counter, while the wrapper remains in the NonIdleState. When an invocation is done, the counter is decreased; if the counter remains greater than zero the wrapper stays in the NonIdleState, else the wrapper gets back in the IdleState. The behavior of the blocking part conforms to the pattern described in Figure 3(b). In particular, when the startBlocking() operation is called, the wrappers get into the BlockingState. For every subsequent invocation of a wrapper operation x(), the wrapper checks whether any of the following conditions hold:

- x() is nested within another invocation y() that started before the beginning of the upgrade. In this case, performing x() is necessary for the completion of y().
- x() is performed within a thread that is involved in serving or issuing requests that are necessary for driving a session into completion.



(a) Idle state detector for TransactionFactory objects.

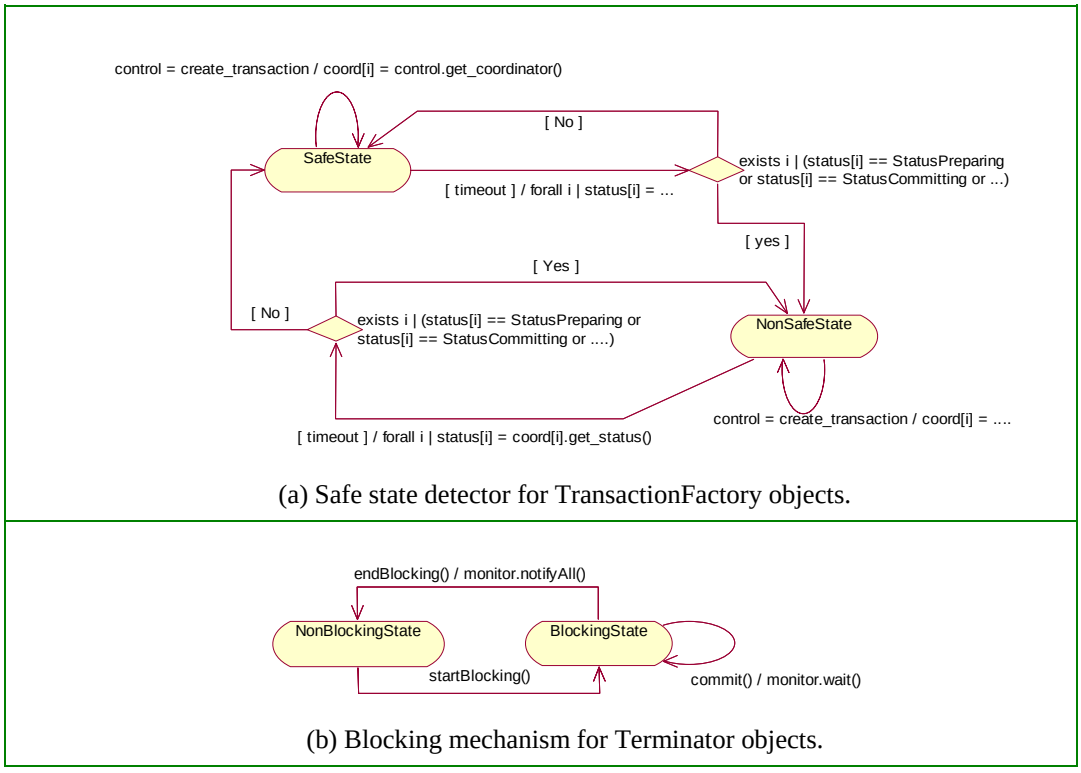
(b) Blocking mechanism for TransactionFactory objects.

Figure 4: The behavior of session-specific wrappers that detect idle state for CORBA TS.

Checking the first condition is rather simple in CORBA; every operation invocation on a target object can be associated with a Context object, which contains named values and is transferred to the target object, along with the request that is generated from the invocation. In our case, if y() is a non-nested invocation on a wrapper, it is associated with an empty context ctx. The wrapper implementation of y() appends to ctx the string-ified reference of the target object that is included in the wrapper and calls y() on the object. If the target object implementation of y() involves invoking x() on another wrapper, it associates ctx with this invocation. Consequently, the wrapper implementation of x() shall receive a non-empty context, which signifies that the invocation is nested.

Checking whether the second condition holds is slightly more complicated. Most of the CORBA services that enable creating sessions (e.g., CORBA TS, CORBA SS) also provide means for obtaining references to Current objects that offer operations, which allow discovering whether a thread that holds a reference to a Current object is involved in the execution of a session. Upon an invocation, the wrappers use references to service-

specific Current objects to check for the existence of such kind of associations between the invoking thread and pending sessions. For middleware services that do provide implementations of the Current interface (e.g., the CORBA CS), special care must be taken. More specifically, the AdaptationMgr object in each capsule provides operations for creating, removing and checking for dependencies between threads and pending sessions. Creating those dependencies is a responsibility of the session-specific wrappers.



(a) Safe state detector for TransactionFactory objects.

(b) Blocking mechanism for Terminator objects.

Figure 5: The behavior of session-specific wrappers that detect safe state for CORBA TS.

At this point, it is worth noticing that the decision of blocking an invocation may also involve checking the existence of synchronization dependencies between the invoking thread and threads that are involved in the completion of other pending requests, or sessions. However, CORBA does not provide standard specifications of thread synchronization services, or standard specifications of thread models. Hence, examining the previous issue is out of the context of this paper.

Session-specific wrappers that detect idle state depend on the particular service that enables the creation and execution of the corresponding sessions. Here we take CORBA TS as a representative example. Among the basic TS interfaces, we have the TransactionFactory, the Terminator and the Coordinator interfaces with operations for creating new transactions, committing/aborting transactions and checking the status of transactions, respectively. For TransactionFactory objects, we use session-specific wrappers whose state detection and blocking parts behave as described in the state-charts of Figure 4. According to Figure 4(a), a TransactionFactory wrapper is initially in IdleState. Upon the invocation of the create_transaction() operation, the wrapper gets into

the NonIdleState and obtains a reference to a Coordinator object that is responsible for the newly created transaction; the reference is added in a sequence of references maintained by the wrapper. The wrapper uses this sequence to periodically check the status of the all pending transactions. More specifically, the wrapper calls the get_status() operation on each one of the referenced Coordinator objects. If the values returned by all calls are StatusCommitted, StatusNoTransaction or StatusRolledBack, the wrapper returns to IdleState; otherwise, it remains in NonIdleState. According to Figure 4(b), the blocking part of the wrapper, blocks invocations on the create_transaction() operation, except if they are nested in invocations that are needed for completing other pending requests or sessions.

As discussed in Section 2, the ability to upgrade middleware, while there exist pending requests reduces the disruption introduced by the overall upgrade process. The behavior of broker-specific and session-specific wrappers that detect safe state depends on the particular upgrade situation. Again, we take a representative example of wrappers that detect safe state for CORBA TS. In particular, suppose that we can upgrade the TS implementation for a new one if there exist pending transactions, as long as the middleware objects that are part of TS are not engaged in performing a two-phase commit protocol. To detect this kind of safe states we have to use wrappers to TransactionFactory objects that behave as given in the state-chart diagram of Figure 5(a). Each of those wrappers holds a sequence of references to Coordinator objects that are responsible for pending transactions. While being in the NonSafeState, the wrapper checks the status of pending transactions using the aforementioned references. If the status reported by any of the Coordinator objects is StatusCommitting, StatusPreparing, or StatusRollingBack, the wrapper remains in NonSafeState. Otherwise, it gets into SafeState. The blocking part of the wrappers to the TransactionFactory objects is not interesting, since it is not necessary to block the creation of new transactions to drive TS objects in a safe state. On the other hand, reaching a safe state requires blocking invocations of the commit() operation, issued to Terminator objects; these are the invocations that actually trigger the execution of the two-phase-commit protocol. To achieve the previous, the wrappers to Terminator objects behave as in Figure 5(b). Technically, if a set of broker-specific or session-specific wrappers that detect safe state is available at the time when a particular upgrade situation occurs, it substitutes the corresponding set of wrappers that detect idle state. The UpgradeMgr object invokes either the upgradeBrokerWrappers() or the upgradeSessionWrappers() operation on every AdaptationMgr object. The aforementioned operations take as input parameter the name of a helper library that contains functionality, which is dynamically loaded by each AdaptationMgr object. The AdaptationMgr object uses this functionality to create new wrappers in place of the old ones. Moreover, it uses the state of the old wrappers for the initialization of the new ones.

The Behavior of the Coordination Elements

The sequence diagram in Figure 6 describes the overall upgrade of a broker, which is triggered by calling the upgradeBroker() operation on the UpgradeMgr object. The upgrade of a service is performed similarly by calling the upgradeService(). The

upgradeBroker() operation takes as input parameter the name of a helper library that contains functionality for the initialization of the new broker implementation.

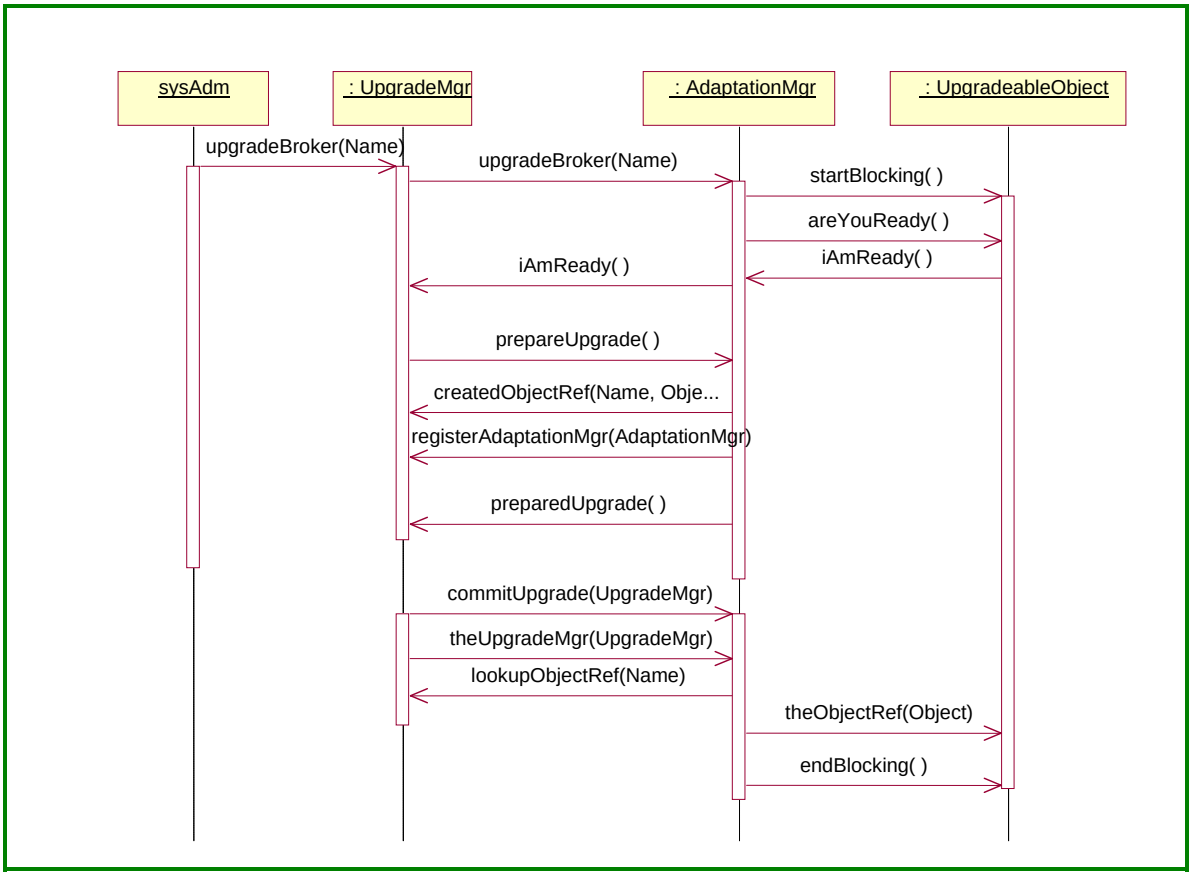


Figure 6: Coordinating the middleware upgrade.

In response to the upgradeBroker() invocation, the UpgradeMgr object notifies the AdaptationMgr objects that coordinate the middleware upgrade in the individual ODPS capsules about the beginning of the upgrade and about the helper library that is going to be used during the upgrade. Each AdaptationMgr object dynamically loads the functionality contained in the helper library and initializes the blocking and the state detection mechanisms. Then, it waits until all of the encapsulated wrappers report that they reached an idle/safe state. At this point, the AdaptationMgr object calls the iAmReady() operation on the UpgradeMgr object to notify the latter that it is safe to upgrade the broker from the point of view of the capsule for which the AdaptationMgr object is responsible. The UpgradeMgr object waits until the reception of notifications by all of the AdaptationMgr objects and then, invokes the prepareUpgrade() operation on these objects. In response to this call, each AdaptationMgr object creates a reference to an ORB object that relies on the new broker implementation. Moreover, it uses the structural information that was registered to it, to reconstruct the POA tree (recall that this information was provided by the capsule with the use of the registerPOA() operation). The new tree shall comprise references to POA objects that rely on the new broker implementation. If the ODPS architecture contains safe state detection mechanisms instead of idle state ones, the state of the old broker objects

must be transferred into the new broker objects. Then, the AdaptationMgr object creates a new computational object for every old one and activates it using the appropriate POA object. The AdaptationMgr object creates references to the new computational objects and registers them to the UpgradeMgr object using the createdObjectRef() operation. Then, the AdaptationMgr object creates a new reference to itself, registers it to the UpgradeMgr object and calls the preparedUpgrade() operation. The accomplishment of all tasks mentioned up to this point, relies on the use of the functionality that is provided by the dynamically loaded library. When all of the AdaptationMgr objects are done, the UpgradeMgr object creates a new reference to it and makes it public. Then, it calls commitUpgrade() on every AdaptationMgr object. In respond to this call, each AdaptationMgr object looks in the UpgradeMgr object registry for new references to required computational objects. The new references are set in place of the old ones in every wrapper used in the capsule for which the AdaptationMgr object is responsible.

6 EXPERIMENTAL RESULTS

To evaluate the proposed architectural style we used a typical client-server benchmark that consists of a server capsule and one or more client capsules. Each client capsule holds a reference to a CORBA object that represents a servant provided by the server capsule. The client repeatedly calls an operation on the CORBA object. Each invocation passes a string (328 bytes), which is echoed by the server object. We deployed the client and the server capsules on different Sun Blade100 workstations, running Solaris v5.8. The workstations are connected with an idle 100Mbps Ethernet. The experiments we performed were aiming at measuring: (1) the performance overhead introduced by the additional elements we add in conventional CORBA-based ODPS architectures, to be able to upgrade their middleware parts; (2) the disruption introduced by the upgrade process itself.

Performance Overhead Introduced in Conventional CORBA Architectures

To measure the overhead introduced in conventional CORBA-based ODPS to support the middleware upgrade process we compared the time for a client invocation in a conventional ODPS architecture that comprises a client and a server capsule, with the time for a client invocation in the corresponding ODPS architecture that supports the middleware upgrade process. For each one of the aforementioned architectures we produced two different implementations based, respectively, on the OMNI v2.7.0 and the MICO v3.2.7 implementations of the CORBA standard.

Figure 7 gives the results obtained from this experiment. The overhead introduced is quite small (5% for OMNI v2.7.0 and 4% for MICO v3.2.7). One would expect it to be larger, since the requests, issued by the client, go through an additional layer of invocation. However, this is not the case since the wrapper objects we use in the architecture are pseudo CORBA objects. Consequently, invocations on the wrapper objects are cheap, as they do not rely on the proxy pattern. Considering that OMNI and MICO are among the most efficient implementations of the CORBA standard, providing

rather fast base communication channels [DSCG], we expect that the overhead in cases of other CORBA implementations is going to be negligible.

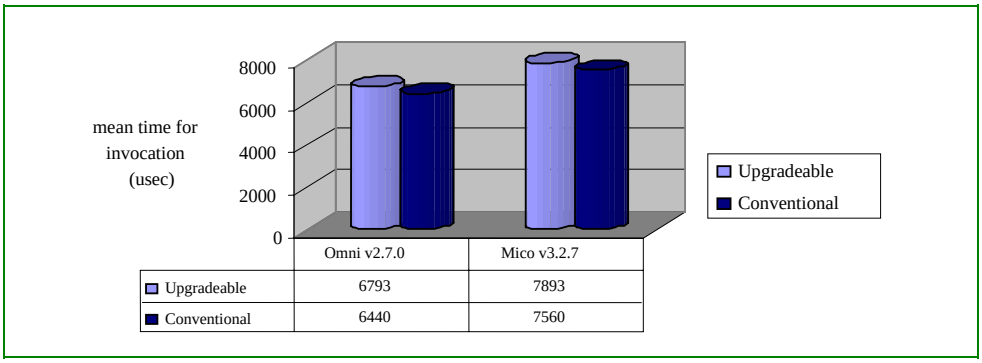


Figure 7: Performance overhead introduced in conventional CORBA ODPS architectures.

Disruption Introduced by the Upgrade Process

In this experiment, we measured the time for changing the OMNI v2.7.0 broker for an OMNI v3.0.4 broker in 4 different ODPS architectures that comprise 1 server capsule and 1 up to 4 client capsules. We performed the upgrades at a time when the middleware was in an idle state.

Figure 8, shows the results obtained. More specifically, we can remark that the fluctuation of the time it takes to upgrade the broker is impressive, especially as we increase the number of the client capsules that continuously access the server. However, the proportion of this time spent for the coordination of the overall upgrade process is quite stable and predictable; the rest of it is spent while waiting for the broker to reach an idle state. The previous remarks give us a strong hint about the benefits we have if we can change the broker without having to wait for it to complete all pending requests. More specifically, if it was possible to transfer any state of the old broker into the new broker, the disruption introduced by the upgrade would be minimal and equal to the communication overhead introduced for coordinating the upgrade.

The architectural style we proposed in this paper specifically deals with the technical aspects that concern the aforementioned kinds of upgrades. Most importantly, it deals with the uncertainty introduced at the level of the state detection and the blocking mechanisms, when facing the fact that we cannot a-priori know the upgrade situations that may arise during the lifetime of the system. However, our solution strongly relies on the assumption that the standard middleware infrastructures provide functionality that enables accessing and modifying the state of middleware objects. The previous is currently not the case with CORBA, where the provision of standard introspection and state transfer mechanisms is limited compared to the facilities provided by other non-standard reflective middleware infrastructures [Blair *et. al.*98]. Based on the previous, we can derive the conclusion that it is time to develop infrastructures that bridge the gap between CORBA that imposes too many constraints on developers and, completely open reflective infrastructures that leave them without any protection against the flexibility they provide. The architectural style proposed here may constitute a first step towards this direction.

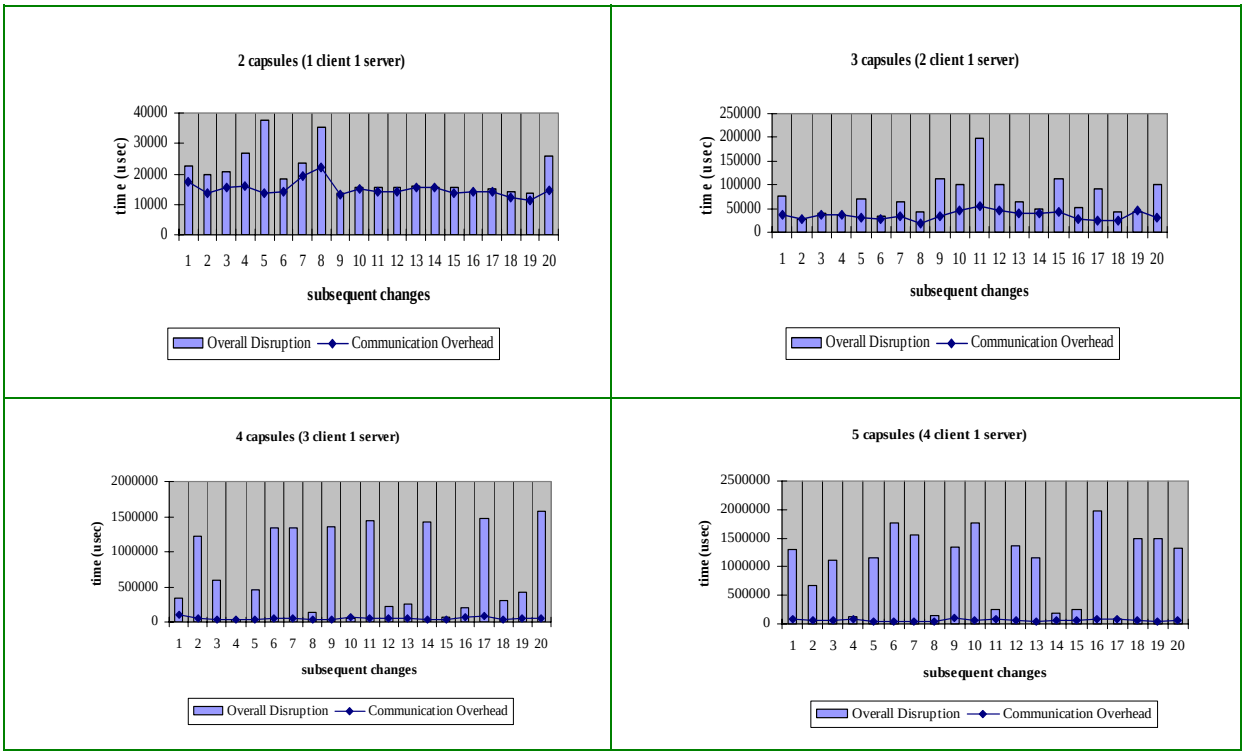


Figure 8: Overall disruption vs. communication overhead.

7 CONCLUSION

Dynamic middleware upgrade is an issue in the development of tomorrows open distributed processing systems. In this paper, we examined the different dimensions of this problem. Starting from the basic requirements for consistent middleware upgrade, we went through the examination of different techniques used in the past in the field of dynamic reconfiguration of software architectures, and reached our main contribution: *An architectural style for open distributed processing architectures whose middleware can be upgraded.*

The approach for consistent middleware upgrade proposed in this paper can be combined with offline analysis of changes in the middleware parts of ODPS architectures. Into this context, in [Issarny et. al.02] we proposed an approach for the performance and reliability analysis of middleware architectures. Currently, we work towards extending this approach for analyzing the impact of middleware changes in ODPSs with real-time requirements. Moreover, we concentrate on refining the proposed architectural style with respect to middleware infrastructures used in wireless, resource-constrained execution environments.

REFERENCES

[Bernstein96] P. A. Bernstein. Middleware: "A Model for Distributed System Services". In *Communications of the ACM*, vol. 39 no. 2, pp. 86-98. 1996.

[Zarras04] A. Zarras. "A Comparison Framework for Middleware Infrastructures". In *Journal of Object Technology*, vol. 3 no. 5, May-June 2004, pp. 103-123. http://ww.jot.fm/issues/issue_2004_05/article2

[Blair *et. al*.98] G. Blair and G. Goulson and M. Papathomas. "An Architecture for Next Generation Middleware". In *Proceedings of the 1st IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pp. 191-203, 1998.

[Kon *et. al*.02] F. Kon, F. Costa, G. Blair, and R. H. Campbell. "The Case of Reflective Middleware". In *Communications of the ACM*, vol. 45 no 6, pp.33-38, 2002.

[Blair *et. al*.00] G. Blair, L. Blair and V. Issarny, P. Tuma and A. Zarras. "The Role of Software Architecture in Constraining Middleware Adaptation in Component-Based Middleware Platforms". In *Proceedings of the 2nd IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, 2000.

[Bidan *et. al*.98] C. Bidan, V. Issarny, T. Saridakis and A. Zarras. "A Dynamic Reconfiguration Service for CORBA". In *Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems*, pp. 35-42,1998.

[Rodr. *et. al*.99] N. L. R. Rodríguez and R Ierusamlimschy. "Dynamic Reconfiguration of CORBA-Based Applications". In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics*, LNCS 1725, pp. 95-111, 1999.

[Alm. *et. al*.01] J-P. A. Almeida, M. Wegdam, M. van Sinceren and L. Nieuwenhuis. "Transparent Dynamic Reconfiguration for CORBA". In *Proceedings of the 3rd IEEE International Symposium on Distributed Objects and Applications*, pp. 197-207, 2001.

[CORBAUv1.0] CORBA Online Upgrades v.1.0. OMG Document, ptc/2002-07-01. http://ww w.omg.org/technology/documents/specialized_corba.htm.

[CORBAv3.0.2] Common Object Request Broker Architecture (CORBA/IIOP)

v.3.0.2. OMG Document, formal/2002-12-06. http://www.omg.org/technology/documents/for mal/corba_iiop.htm.

[Shapiro86] M. Shapiro. "Structure and Encapsulation in Distributed Systems: The Proxy Principle". In *Proceedings of 6th Interantional Conference on Distributed Computer Systems*, pp.198-204, 1986.

[CORBASrvs] CORBAServices Specification. OMG Document. http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm.

[Kramer *et. al*.90] J. Kramer and J. Magee. "The Evolving Philosophers Problem". In *IEEE Transactions on Software Engineering*, vol. 15 no. 1, pp. 1293-1306, 1990.

[Goud. *et. al*.96] K.M. Goudarzi and J. Kramer. "Maintaining Node Consistency in the Face of Dynamic Change". In *Proceedings of the 3rd IEEE International Conference on Configurable Distributed Systems*, pp. 62-69, 1996.

[Warren *et. al*.95] J. Warren and I. Sommerville. "Dynamic Configuration Abstraction". In *Proceedings of the 2nd Joint ACM SIGSOFT Symposium on the Foundations of Software Engineering and European Software Engineering Conference (FSE-ESEC'95)*, pp. 173-190. 1995.

[Haupt *et. al*.96] S. Hauptmann and J. Wasel, "On-line Maintenance with On-the-Fly Software Replacement". In *Proceedings of the 3rd IEEE International Conference on Configurable Distributed Systems*, pp. 70-80. 1996.

[GSCG] CORBA Comparison Project Web site. Distributed Systems Group, Charles University. http://nenya.ms.mff.cuni.cz/projects.phtml?p=cbench&q=3.

[Issarny *et. al*.02] V. Issarny, C. Kloukinas and A. Zarras. "Systematic Aid for Developing Middleware Architectures". In *Communications of the ACM*, vol. 45, no. 6, pp. 53-58. 2002.

About the author

Apostolos Zarras got his Ph.D. in the year 2000 from the University of Rennes I, France. From 2000 to 2002, he worked as a research engineer at INRIA-Rocquencourt, France. Currently he is a visiting assistant professor at the University of Ioannina, Greece. His current research interests include model driven architecture development, adaptive middleware, and quality analysis of software systems. He can be reached at zarras@cs.uoi.gr.