Fault-Tolerant Replication Based on Fragmented Objects

Hans P. Reiser[1], Rüdiger Kapitza[2], Jörg Domaschka[1], and Franz J. Hauck[1]

[1] Distributed Systems Lab, University of Ulm, Germany
{hans.reiser, joerg.domaschka, franz.hauck}@uni-ulm.de
[2] Department of Distributed Systems and Operating Systems,
University of Erlangen-Nürnberg, Germany
kapitza@informatik.uni-erlangen.de

Abstract. This paper describes a novel approach to fault-tolerance in distributed object-based systems. It uses the fragmented-object model to integrate replication mechanisms into distributed applications. This approach enables the use of customised code on a per-object basis to access replica groups and to manage consistency. The addition of fault tolerance to the infrastructure has only little overhead, is fully transparent for clients, and does not require internal modifications to the existing middleware. Semantic annotations at the interface level allow the developer to customise the provision of fault tolerance. Operations can be marked as read-only to allow an execution with weaker ordering semantics or as parallelisable to allow true multithreaded execution. A code-generation tool is provided to automatically produce object-specific fragment code for client access and for replica consistency management, taking into account the annotations, the interface specification, and the non-replicated implementation. A further contribution of our code-generation approach is the support of deterministic multithreading in replicated objects.

1 Introduction

The development of fault-tolerant applications in distributed systems is a complex task. It can be simplified for the developer by providing support for fault-tolerance at the middleware level. Replication support has previously been added to middleware systems like CORBA in various ways, for example using the interception approach [13], the integration approach [3], or the service approach [4]. The approaches differ in their properties regarding transparency, efficiency, and portability; each approach has its specific advantages, but also disadvantages.

Recent publications (e.g., [2, 5]) indicate that the current support for replication in general-purpose distributed object middleware is not yet sufficient in several regards. One of the limitations is the lack of interoperability between multiple middleware infrastructures. For example, typical fault-tolerant CORBA systems require all replicas to run on the same ORB implementation. Often, clients must use the same manufacturer's ORB to benefit from the fault-tolerance mechanisms. Consequently, heterogeneity in terms of middleware platform or programming language—an important feature and one of the main objectives of CORBA—lacks support.

F. Eliassen and A. Montresor (Eds.): DAIS 2006, LNCS 4025, pp. 256–271, 2006.

Replica nondeterminism is a source of problems, both in passive replication in case of replica faults and in active replication [23]. If potentially nondeterministic actions are not simply prohibited in replica implementations (a popular approach), they have to be intercepted and coordinated by the infrastructure. One problematic source of nondeterminism is multithreaded execution of object operations. Many fault-tolerant infrastructures solve this problem by forcing a strictly sequential execution of all client requests in total order. This solution has serious limitations, as it not only lacks performance, but is also inherently deadlock-prone. Only few systems (e.g, [15, 8, 23]) offer a multithreaded solution with an adequate deterministic thread scheduling strategy. Such approaches require that the service uses specific locking methods that can be intercepted by the fault-tolerance infrastructure.

Furthermore, most existing fault-tolerant object middleware systems provide no mechanism to use semantic knowledge about the replicated object. Often, this is not the most efficient solution: If, for example, it is known that some methods are read-only or parallelisable, weaker ordering semantics than total order can be employed to improve efficiency. A replication infrastructure can provide such optimisations automatically only if it has access to semantic information explicitly expressed by the object developer.

The contribution of this paper are approaches to handle several of these problematic issues in fault-tolerant middleware systems. Our implementation provides an infrastructure for fault-tolerant distributed applications in the AspectIX middleware based on fragmented objects. The fragmented-object model [12, 7, 21] is a versatile approach to design complex distributed services that do not strictly adhere to a simple client-server structure. It supports dynamic loading of object-specific fragment code at the client side and at replica locations. This flexibility of a fragmented-object middleware enables the integration of fault-tolerance support without requiring internal middleware modifications. The access to replica groups remains fully transparent for clients. At the same time, the directly loaded object-specific fragment code avoids the overhead of interception strategies or other delegation approaches.

At the core of our architecture, we provide a code-generation tool that automatically creates client-side access fragments and server-side replica fragments based on the non-replicated object implementation, the interface definition, and semantic annotations. This way, the transition from an existing implementation to a replicated one is automated as much as possible, with only minimal developer intervention required. Annotations can be provided to specify if an object operation interacts with the replica and modifies its state, if it is a read-only operation, if it is parallelisable with other methods, or if it is a method that can be computed locally at the client side without interacting with the replica group.

Our replication system allows multithreading inside actively replicated objects. A deterministic thread scheduler supports an arbitrary number of reentrant mutex locks, condition variables that allow threads to block and be woken up by other threads, and timeouts on blocking synchronisation operations. Language-specific synchronisation statements need to be mapped to the synchronisation interface of the scheduler. Our Java-based prototype provides a code-generation tool that automatically transforms native Java synchronisation statements. This allows application developers to use

Java-specific constructs (e.g., `synchronized` statements) to express the required coordination, as they would do in non-replicated code. The multithreading issues of replication remain fully transparent for the application developer. Semantic annotations can be used to further improve the thread-scheduling mechanism. For example, multiple methods that are marked as parallelisable can all be executed in parallel without coordinating their lock acquisitions.

This paper is structured as follows: Section 2 discusses established approaches to fault tolerance in traditional middleware and surveys in more detail the fragmented-object model. Section 3 presents the realisation of fault-tolerant replication in the AspectIX middleware based on fragmented objects. It describes our code-generation process, which considers semantic annotations, and discusses the advantages of our approach regarding multithreading. Section 4 evaluates our system. Finally, Section 5 concludes.

2 Background

2.1 Approaches to Replication Support in Distributed Object Middleware

Replication adds redundancy to a system, which makes it possible to tolerate the failure of some of the nodes on which an object is located. Some strategy, like active or passive replication, is required to keep the replicas in a consistent state. In passive replication, only a single designated primary replica executes all operations; secondary replicas are able to take over the primary's functionality if it fails. In active replication, all replicas execute all operations. This causes more overhead than passive replication in failure-free executions, but allows faster reaction to failures (ideally, the failure of a single node remains fully unnoticed by clients). In addition, keeping all replicas constantly up-to-date allows using load-balancing for read-only requests, which are handled by only one replica.

Several research projects have investigated ways for adding fault-tolerance mechanisms to distributed object middleware. For fault-tolerance in CORBA systems, the OMG provides the FT-CORBA specifications [16, Chap. 23] as a general standard, without specifying exact implementation details. The implementation of this standard in existing CORBA middleware is usually based on the interception approach, the service approach, or the integration approach.

The interception approach initially was propagated by the Eternal system [13]. Eternal intercepts IIOP messages between the ORB and the operating system. This way, any off-the-shelf ORB can be used, and replication becomes fully transparent for clients and servers. However, such interception requires adequate support at the operating-system level. Fault-tolerance mechanisms are fully separated from the ORB core; information about remote invocations can only be obtained by parsing the marshalled invocation data.

A prominent example for the service approach is OpenDREAMS [4]. This system encapsulates fault-tolerance mechanisms inside a CORBA service. The only direct interaction of application objects is with a local Object Group Service (OGS), which in turn coordinates with other OGS instances and executes the requested operations at the replicas. This approach does not offer replication transparency, as clients are aware of the OGS, and adds an additional step of indirection, which increases latency. Its outstanding benefit is that no proprietary extensions to the ORB or the operating system are needed.

In the integration approach, the ORB is directly modified to provide the desired support for fault tolerance. In general, this provides the most efficient solution. However, this approach usually inhibits any interoperability with clients running on standard off-the-shelf CORBA platforms. Orbix+Isis [3] and Electra [11] are examples where the integration approach has successfully been used.

Replication may also be added to other non-CORBA middleware infrastructures. For example, the AROMA system [14] transparently enhances the Java RMI system with mechanisms for consistent object replication. It modifies the Java RMI infrastructure to intercept remote invocations and maps them to a reliable, totally ordered group communication protocol. As another example, the .NET remoting infrastructure provides the possibility to load custom stub code ("real proxy") instead of the default stub. This makes it possible to transparently add custom support for replication [19].

Our fault-tolerance infrastructure is based on the AspectIX middleware, which provides support for *fragmented objects*. This support is implemented as an extension to a standard CORBA ORB. With fragmented objects, custom fragment code can be loaded transparently at client side and replica side on a per-object basis. Our fault-tolerance architecture provides code-generation tools to create fragment code for fragmented objects. This code can be used on any middleware that supports our fragmented-object model; to this extent our approach is portable and not restricted to a specific vendor's ORB. Furthermore, it provides client-side transparency and has optimal efficiency, as the client directly invokes fragment methods without unnecessary indirection steps.

2.2 The Fragmented-Object Model

In a traditional client/server system based on remote method invocations, the functionality of an object completely resides on a single node. For transparently accessing the object, the client-side middleware instantiates a stub that handles remote invocations (Fig. 1a). Usually, the stub code is automatically generated from an interface specification. All objects with the same interface share the same stub code. The middleware runtime systems instantiates the stub as soon as the client binds to an object reference. The bind operation is either requested explicitly by the client, or it is performed implicitly when an object reference is passed to the client through the marshalling mechanisms of the ORB.

In the fragmented-object model, the distinction between client stubs and the server object is no longer present. From an abstract point of view, a fragmented object is a unit with unique identity, interface, behaviour, and state, as it is in classic object-oriented design. The implementation of these properties, however, is not bound to a certain location, but may be distributed arbitrarily on various fragments (Fig. 1b). Any client that wants to access the fragmented object needs a local fragment. In addition, there can be fragments that are deployed on nodes without a client. The client interface of a local fragment is identical to that of a traditional stub. However, the local fragment can be specific for exactly that object. Two objects with the same interface can lead

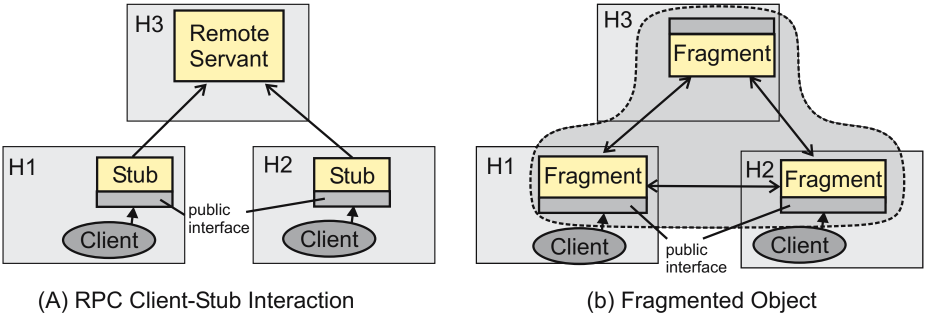


Fig. 1. Traditional Client/Stub Structure vs. Fragmented Object

to completely different local fragments. This internal structure allows a high degree of freedom on where the state and functionality of an object is provided, and how the interaction between fragments is done. The internal distribution and interaction is hidden from the outer interface. In addition, the distribution of functionality to fragments can even be changed dynamically at runtime.

The AspectIX middleware provides support for fragmented objects. Unlike other fragmented-object middleware infrastructures such as FOG [12] and Globe [7], it even supports implicit binding of fragmented objects upon the receipt of a marshalled object reference. All reference-related operations are handled by a generic reference manager and pluggable profile managers [6]. AspectIX uses CORBA IORs as references and also provides interoperability with standard CORBA applications. In addition to CORBA IIOP profiles, a custom IOR profile (called APX) for fragmented objects is supported. When binding to such a reference, fragment-specific code is transparently loaded; for this purpose, AspectIX provides a Dynamic Loading Service (DLS [9]) that enables the lookup, selection, and loading of platform-dependent code at run-time. From a client point of view, the interface of a local fragment is identical to that of a standard CORBA object.

3 The AspectIX Replication Architecture

3.1 Overview

On top of the basic infrastructure for fragmented objects, we provide support for fault-tolerant active replication of distributed objects. This chapter first outlines our architecture, which encapsulates replication inside a fragmented object. We discuss the internal implementation structure of the fragments, illustrate the use of the AspectIX profile in the IOR to reference replica groups, and describe the run-time infrastructure, which is used to create and administrate replica groups. Subsequently, we explain how semantic properties can be defined by developer annotations. Finally, we focus on the code-generation process that automatically produces fragment code from the interface definition, the semantical annotations, and the non-replicated implementation.

3.2 Fault-Tolerant Replication with Fragmented Objects

A fault-tolerant service in the fragmented-object model is represented by a single distributed object which is composed of replica fragments and access fragments (Fig. 2). The development process consists of defining the global object interface in CORBA IDL, implementing the functional parts of the service, and creating the fragment code. The creation of fragment code is done automatically by tools; these tools can make use of additional semantic annotations provided by the developer, as we will describe in Section 3.3. This enables the generation of a customised layer between the client and the core framework and also between the framework and the replica implementation.

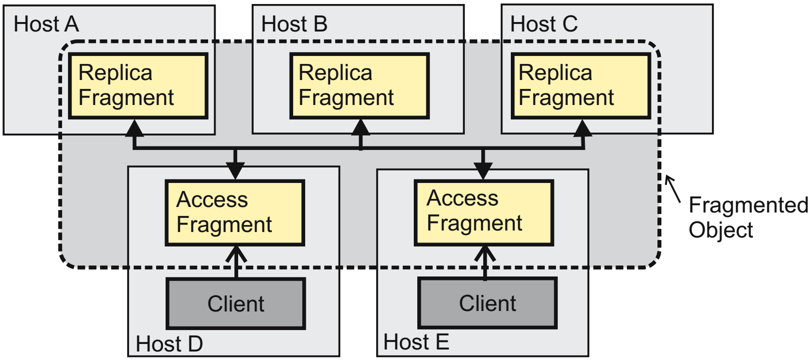


Fig. 2. Replication with Fragmented Objects

Details of the Fragment Architecture. The layered design of access and replica fragments in our architecture is shown in Fig. 3. Access fragments are used by client applications; the replica fragments contain the object state. Replica fragments do not support direct client access. Instead, an access fragment is instantiated at the same location as the replica fragment. For simplicity, this detail is not shown in Fig. 3.

Starting at the client side, the client application accesses the fault-tolerant fragmented object via its interface like any other CORBA object. The generated access fragment may contain optional developer code that is directly embedded (see Section 3.3). Furthermore, it contains a *Context Handler*, code for marshalling and unmarshalling of requests (equal to a standard client-side stub), and code for remote communication.

If client *A* invokes a method at a replicated object *B*, a node failure during the invocation can make it necessary to repeat the invocation. This happens if the client *A* communicates with a replica fragment that fails. The re-invocation contacts a different replica of *B*. Alternatively, *A* can be replicated itself. In this case, the access fragment provides client-side duplication suppression by selecting one replica of *A* to actually make the remote invocation. If this selected replica fails, another replica of *A* repeats the invocation. To preserve the at-most-once invocation semantics of CORBA, in both cases the repetition of the invocation needs to be detected and filtered out at the replica

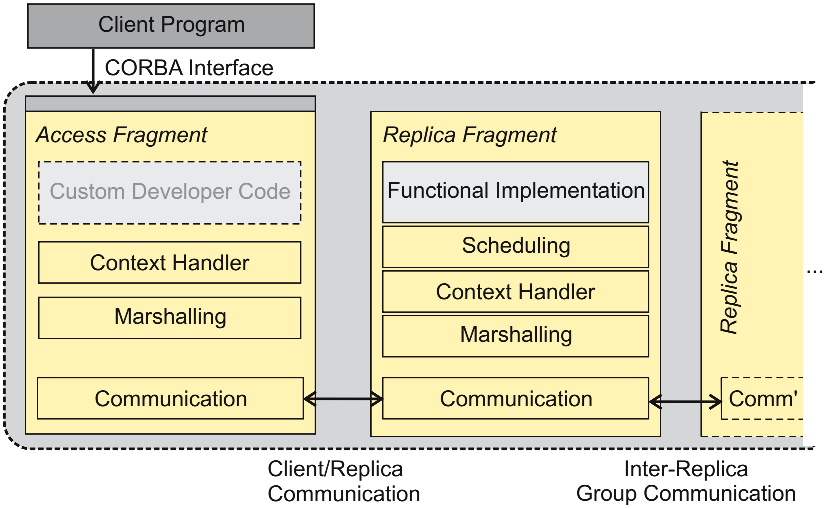


Fig. 3. Architectural Overview

fragments of *B*. For this purpose, the *Context Handler* adds context information with a consistent unique ID that identifies the request.

At the network level, a totally ordered group communication system provides the basis for consistent active replication. Currently, our prototype uses the JGroups system [1], which is based on the closed-group approach. This implies that group communication is only used between replicas; any replica can act as a gateway to communicate between an access fragment and the replica group. However, our implementation easily supports the exchange of the network layer, such that group communication systems with an open-group model—like our own group communication framework [18]—can be used. This potentially improves performance (e.g., the access fragment can directly multicast its request to all replicas). For tolerating Byzantine failures, an appropriate variant

With the gateway approach, the *Communication* element in the access replica is responsible for transmitting calls to one available replica. If this replica fails, the *Communication* transparently reconnects to another replica fragment and reissues the call. If the call has already been processed by the replicas, this is detected and the invocation result is returned from a cache. The *Communication* component in replica fragments is responsible for passing requests to group communication. All requests that are received from group communication are placed into a totally ordered queue for subsequent processing in the upper layer.

The upper layer consists of three different components: First, the *Marshalling* component deserialises requests and serialises replies. The replica side of the *Context Handler* component is used for the suppression of duplicated requests. The *Scheduling* component is responsible for the internal message management and for the deterministic multithreading support, which is explained in detail in Section 3.5. On the top end, requests are passed to the functional implementation that was provided by the developer; this implementation may partially get modified by the code generation process (see Section 3.4)

IOR References to Replica Groups. As described in Section 2.2, the AspectIX middleware uses CORBA IORs to reference fragmented objects via an APX profile. This profile contains a unique object ID, a specification of the initial fragment type to load, and contact information of other fragments. The initial fragment type can be specified in a language-independent way; equivalent fragment implementations may exist for various programming languages or execution platforms. The Dynamic Loading Service (DLS) of AspectIX loads the appropriate code based on the specification from the IOR profile [9].

The initially loaded fragment evaluates the contact information from the IOR profile. In the gateway approach, this information consists of a list of all replica gateway addresses of the group. In the open-group approach, address information for the group communication system (like a multicast address for group discovery or the addresses of gossip servers) can be stored in the contact information. The removal or addition of replicas triggers the creation of a new version of the replica group's IOR. The standard approach for updating the client's IORs, which is also used in FT-CORBA, is to include the client's current IOR version in each invocation request to the replica group. If this version is out-of-date, the replica will send the current version in the reply to the client.

This approach is practical in many situations; however, it does not provide any guarantees that all client-side IORs will get updated in time before they no longer provide valid contact information. To improve this situation, we additionally support the concept of a lifetime specified within the IOR references [10]. During the specified lifetime, the replica group guarantees that the IOR information can be used to contact the group; in the gateway approach, this means that at least one of the replica gateway addresses included in the IOR remains accessible. Optionally, the address of a location service can be specified in the IOR, which manages an up-to-date contact information for the replica group. This is useful if the distribution of a fragmented object changes frequently and the risk of stale references is high.

Run-Time Infrastructure. Similar to other fault-tolerant middleware infrastructures, AspectIX uses the factory pattern to create and set up replicas. Replication groups are implemented as self-managing entities; this reduces the complexity of the necessary infrastructure compared to other systems that require a dedicated replication manager. In addition, the management automatically benefits from the same fault-tolerance mechanisms as the replicated object itself.

Starting a new replicated service involves several steps. First, a factory must be acquired via a factory finder. A factory finder represents a search scope for possible places of execution, as defined by the CORBA Life Cycle Specification [17]. Currently, our factory finder is implemented straightforwardly in plain CORBA and well-known on every node within a domain. This way, a node can register its local factories and it can lookup factories from all other nodes. Multiple factory finders can be provided for fault tolerance.

Our generic factory for object creation offers two methods: one for setting up an initial replica of a replicated fragmented object and another one for setting up additional

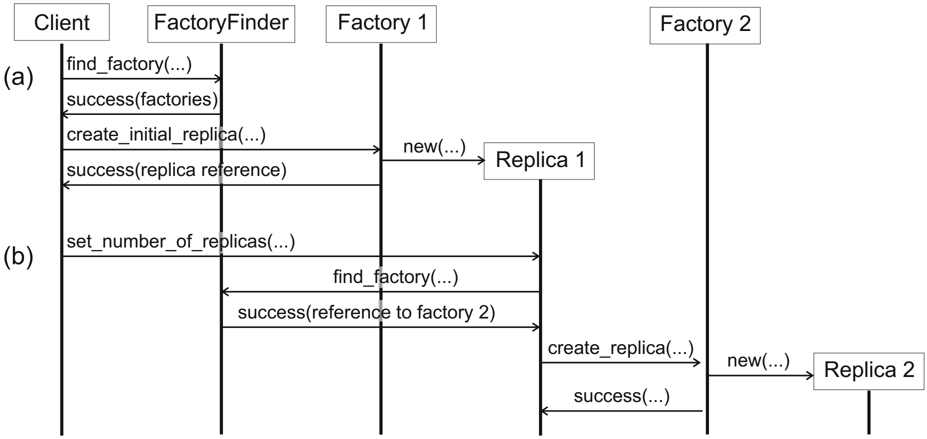


Fig. 4. Creation of first and additional replicas

replicas. After lookup of one or more factories via the factory finder, one of them is requested to instantiate the first replica via `create_initial_replica()` (see Fig. 4 (a)). The factory creates the initial replica and activates the fragment object. Afterwards, the object is returned to the calling client and, as it is a fragmented object, a local access fragment is dynamically instantiated. This results in a simple client/server structure with only one replica. The management code within this replica is able to control the creation of additional replicas.

A management interface of the fragmented object is used to adjust the desired number of replicas (see Fig. 4 (b)). If the client increases the number of replicas, the existing replica group is triggered to add the necessary number of additional replicas. The replica-side fragment contacts the factory finder to request additional factories. In the next step, a reference to the fragmented object is passed to a factory. At the factory side, the fragmented object is transparently bound by the middleware, which loads the initial fragment. Under control of the factory, the local fragment is reconfigured to be a replica fragment. The state of the existing replica group is transferred to the new replica similar to the CORBA Life Cycle Service. The addition of replicas is repeated until the desired replication level is reached or until no additional factories are found. The failure of a replica in the group is detected by a failure-detection mechanism at the group-communication level. After detecting a failure, the replica group automatically sets up a new replica in the same way, as long as another factory is available.

3.3 Semantic Information from Object Developers

A simple scheme for generating client-side and replica-side fragment code only uses IDL interface information, like a traditional CORBA IDL compiler does for stubs and skeletons. Additionally, our architecture allows the developer to express semantic knowledge in order to improve and customise the replication mechanisms. Currently, we support several annotations on a per-method basis:

```
interface CC_processor {
  transaction_id charge(in card_data card, in float amount)
      raises(CardNotValid, TransactionFailed);

#pragma annotate(readonly)
  boolean validate_card(in card_data card) raises(CardNotValid);

#pragma annotate(local)
  boolean validate_card_checksum(in card_data card)
      raises(CardNotValid);
};
```

Fig. 5. IDL with semantic annotations

- `readonly`: A method marked as read-only does not modify the relevant replica state. Instead of executing this method in total-order at all replicas, it is sufficient to invoke it on one available replica.
- `parallelizable(methodlist)`: A method marked as parallelisable with respect to a set of other methods can be executed in parallel with the specified list of other methods. This allows true multithreading.
- `local`: The implementation of a method marked as local will be placed in the client-side fragment. This way, methods that need no access to the replica state can be executed locally at the client, while still being conceptionally part of the distributed object.
- `intercepted`: A method marked as intercepted will execute custom code at the client-side before and after invoking the remote method at the replica group. This mechanism can be used for local preprocessing, for caching, or for the accumulation of multiple client invocations into one remote invocation to the replica group.

Our current implementation uses annotations embedded as `#pragma` instructions within the IDL file, as the example in Fig. 5 illustrates. Our flexible IDL compiler IDLflex [22] allowed us to implement this solution easily.

3.4 Code Generation for Fragments

In our replication infrastructure, the creation of fragment implementations is automated by a code-generation tool. Two basic fragment types are required (see Section 3): A replica fragment for consistency management and a client-side access fragment. The current prototype of the code-generation tool is based on IDLflex [22], an IDL-compiler that generates customisable code. IDLflex parses CORBA IDL, evaluates an XML-based mapping specification, and uses this specification to create arbitrary output code. It includes two standard mapping specifications for the Java programming language, one for standard CORBA and one for AspectIX fragmented objects.

For replication support, the IDL-compiler was extended to support semantic annotations in IDL files, expressed as `#pragma annotate` statements. Within a custom

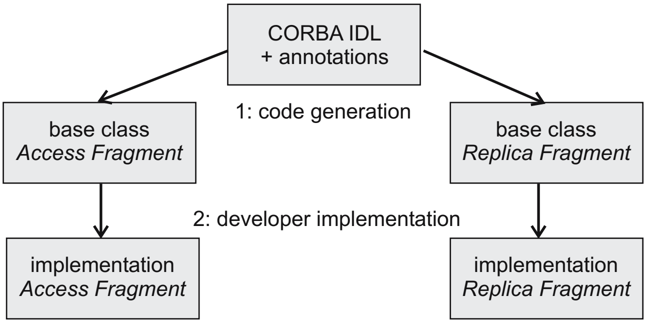


Fig. 6. Development Process of Fault-Tolerant Fragmented Object

mapping specification, these annotations are evaluated and used to control the code-generation process.

The development process of a fault-tolerant fragmented object is illustrated by Fig. 6. The annotated IDL is used to create a base class for the access fragment and the replica fragment. Additional developer code can be added to the access fragment if required by *local* or *intercepted* operations, as we describe below. The implementation of the replica fragment is similar to that of a non-replicated CORBA servant. The main differences are that (1) it has to inherit from the generated replica base class and (2) it has to implement methods for state transfer.

The generated code depends on the code annotations. If at least one *read-only* method is present, the generated code for the *Communication* component will examine all invocation requests. If the requested method is marked as read-only, it will be passed directly to the implementation of one replica, bypassing the totally ordered group communication.

The *Scheduling* component interacts with the deterministic thread-scheduling support of our replication infrastructure (see Section 3.5). The generated code of the component knows which methods are marked *read-only* and *parallelisable*. This information is made available to the thread scheduler in order to maximise the concurrency of request execution.

Specifying a method as *local* causes the method's implementation to be placed in the access fragment instead of the replica fragment. Such method implementation must not access replica state. This approach is useful for methods that, for example, validate client data in a state-independent way or that provide static information to the client.

For each method annotated as *intercepted*, an abstract method is created in the access fragment; the actual method implementation must be provided by the developer. A protected method is provided in the access fragment for accessing the real implementation at the remote replica group; this method can be used within the developer code in the access fragment. Applications for such interceptions are client-side caching strategies or the accumulation of multiple client method invocations with subsequent manipulation of the replicated object's state with only one invocation to the replica group. Our current prototype requires that the developer manually implements the additional client-side code.

Another aspect of the creation of replica fragment code is the ability to modify the functional object implementation. This approach is used to intercept native Java synchronisation code. Synchronisation operations need to be intercepted by our deterministic thread scheduler (see Sect 3.5). By replacing all relevant statements with custom code, such interceptions is possible without internal modification to JVM or operating system. The same approach could be used to intercept Java API calls to nondeterministic methods like the generation of time-stamps or random numbers.

3.5 Multithreading in Actively Replicated Services

Active replication requires a deterministic execution of all state-modifying actions. If multiple threads are allowed to access the replicated object in parallel, the order in which threads access shared data may vary between replicas; this can lead to an inconsistent object state.

The popular solution of using a single-threaded execution has several drawbacks. If a replica group issues a nested invocation, it has to idle until this invocation returns; if a second thread used this waiting time for computations, it would result in improved performance. The single-threaded approach is also deadlock prone: If such a nested invocation calls back a method on the first replica group, this call is blocked by the waiting thread, resulting in a deadlock. Similarly, with a single-threaded model, condition variables that suspend the current thread until woken up by another thread cannot be used.

To provide support for multithreading, we integrated our deterministic thread scheduler for active replication [20] into our AspectIX replication architecture. Our scheduler uses an algorithm similar to that by Zhao et al. [23], improved by support for condition variables and for native Java synchronisation mechanism. It provides a non-preemptive, deterministic mechanism for thread scheduling. A new thread is only created or an existing thread is only resumed, if all other existing threads have reached a safe state, i.e., have terminated or have blocked waiting for a mutex, for a condition variable notification, for a timeout, or for a nested invocation reply. All decisions are fully deterministic, and consequently remain consistent among all replicas. Lock requests, lock releases, and condition variable access need to be passed to our thread scheduling algorithm.

We do not want to modify the execution environment (i.e., the JVM), but still want to allow implementing synchronisation in the replicated object with native Java mechanisms (`synchronized` statements, etc). To intercept these statements, our code generation tool automatically transforms these statements into appropriate synchronisation calls to the deterministic scheduler in the replica fragment, as described above. This approach requires that the synchronisation of a replicated object is fully encapsulated within the replica fragment. That is, we assume that lock object instances used by the replica implementation are not used for synchronisation in code outside the replica fragment, but within the same JVM (Java virtual machine). Developer code within the access fragment has no direct access to the object state and thus does not require synchronisation.

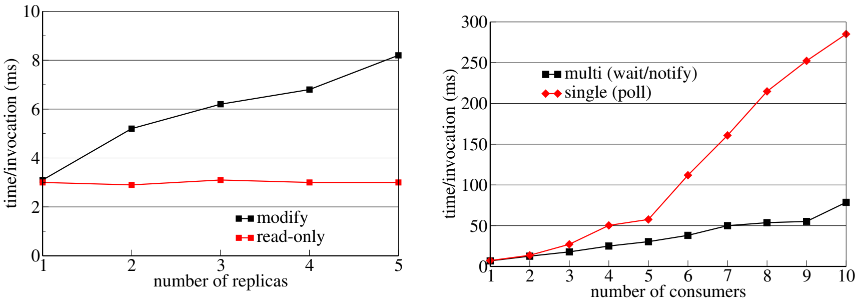
Based on the semantic annotations, the scheduling algorithm can be further improved. A thread may be created or resumed not only if all other threads have reached a safe state, but also if it is marked as parallelisable with all other threads that have not terminated. Special care needs to be taken for read-only methods, which are only executed in one replica. These methods are not allowed to use wait/notify operations on condition variables, as this could lead to an inconsistent scheduling of other modifying methods.

4 Evaluation

Our semantic annotations at the interface level offer ample opportunities to tune and optimise the implementation of replicated objects. In many cases these are very application specific, e.g., if a resource intensive subtask is moved from server to client side. In this section, we present two general examples, which show the possible speed-up of our approach. All presented measurements were made on AMD Opteron 2.2 GHz Linux server machines connected via a switched 100Mbit/s ethernet, using our AspectIX ORB with JGroups 2.2.9.1 for group communication and Java SDK-1.5.0. The JGroups stack was configured to use TCP connections and TOTAL ordering.

In the first example, we measured the difference in invocation time between a read-only method and a modifying method. The read-only method invocation is not distributed via the group-communication framework but instead sent directly to one of the replicas. Fig. 7(a) shows the average time per invocation, obtained from at least ten runs with 5000 client invocations. A single client accesses a replica group with the number of replicas increasing from one to five nodes. The invocation cost for modifying methods is dominated by the cost of the totally ordered group communication. This underlines the benefit from using semantic knowledge about object methods for building an efficient fault-tolerant replication system.

The second example analyses our support for condition variables in the deterministic scheduler. We implemented a simple replicated counter that is increased by a producer and decreased by a variable number of consumers. Without condition variables, the consumers must use polling (for our measurements, a one ms delay between retries was used); with condition variables, a consumer call can block within the counter object at a condition variable until it is woken up by a producer call. We again show the



(a) Read-only versus modify calls

(b) Consumer/producer szenario

Fig. 7. Efficiency gains by read-only annotations and by multithreading

average time per consumer invocation, averaged over repeated experiments with 5000 calls per client. The time is expected to increase linearly with the number of consumers, as multiple consumers compete for values produced by a single producer. As shown in Fig. 7(b), the multithreaded approach with condition variables outperforms a single-threaded implementation; the benefit increases with a rising number of clients.

5 Conclusions

We have presented an architecture for fault-tolerant replication of objects in distributed systems based on the fragmented-object model. A fragmented-object middleware loads custom code at client side and server side on a per-object basis. This enables the implementation of generic fault-tolerance support fully transparent for clients, without internal modifications to the middleware, and without the overhead of indirections that take place in the interception or service approach to fault-tolerance support.

The core of our architecture is a code-generation process that automatically produces client-side access fragments and replica-side consistency management fragments, based on the interface specification, the functional implementation provided by the object developer, and semantic annotations. Our current prototype uses semantic annotations inside the IDL interface definitions, and supports the Java programming language.

We use the semantic annotations for selecting consistency mechanisms for method invocation, strategies for thread scheduling, and for supporting client-side computations. Methods can be marked as read-only, which allows their execution without strict total-order requirements. They can be marked as parallelisable, which enables parallel execution by our multithreading support for actively replicated objects. Furthermore, parts of the functional object implementation can be directly placed at the client side.

Code transformation is also used for removing nondeterministic behaviour in object implementations. We specifically use this to allow multithreading in actively replicated objects; native Java synchronisation mechanisms are replaced with code that allows interception by our deterministic thread scheduler.

Our prototype assumes that a fragmented object middleware is used, but our concept is not strictly limited to such a platform. Other platforms that provide means to load custom object-specific code at the client side, for example based on the smart-proxy principle, can similarly use our code-generation concept for supporting fault tolerance. One advantage of the fragmented-object approach is its flexibility that even supports dynamic reconfiguration at run-time.

References

1. Bela Ban. Design and implementation of a reliable group communication toolkit for Java. Technical report, Dept. of Computer Science, Cornell University, 1998.
2. Ken Birman. Can web services scale up? *Computer*, 38(10):107–110, 2005.
3. Kenneth P. Birman and Robbert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993.
4. Pascal Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

5. Pascal Felber and Priya Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Trans. Comput.*, 53(5):497–511, 2004.
6. Franz J. Hauck, Rüdiger Kapitza, Hans P. Reiser, and Andreas I. Schmied. A flexible and extensible object middleware: CORBA and beyond. In *Proc. of the Fifth Int. Workshop on Software Engineering and Middleware*. ACM Digital Library, 2005.
7. Philip Homburg, Leendert van Doorn, Maarten van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge. An object model for flexible distributed systems. In *Proceedings of the 1st Annual ASCI Conference*, pages 69–78, 1995.
8. Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Sergio Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 164, Washington, DC, USA, 2000. IEEE Computer Society.
9. Rüdiger Kapitza and Franz J. Hauck. DLS: a CORBA service for dynamic loading of code. In *Proc. of the OTM'03 Conferences (DOA, Sicily, Italy, Nov. 3-7, 2003)*, number 2888 in LNCS. Springer, 2003.
10. Rüdiger Kapitza, Hans P. Reiser, and Franz J. Hauck. Stable, time-bound references in context of dynamically changing environments. In *MDC'05: Proc. of the 25th IEEE Int. Conf. on Distributed Computing Systems - Workshops (ICDCS 2005 Workshops)*, 2005.
11. Silvano Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the Conference on Object-Oriented Technologies, (Monterey, CA), USENIX*, pages 135–146, 1995.
12. Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in distributed computing systems*, pages 170–186. IEEE Computer Society Press, 1994.
13. Luise E. Moser, P. M. Melliar-Smith, and Priya Narasimhan. Consistent object replication in the eternal system. *Theor. Pract. Object Syst.*, 4(2):81–92, 1998.
14. Nitya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Transparent consistent replication of Java RMI objects. In *DOA*, pages 17–26, 2000.
15. Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 263, Washington, DC, USA, 1999. IEEE Computer Society.
16. Object Management Group (OMG). Common object request broker architecture: Core specification, version 3.0.2. OMG document formal/02-12-02, 2002.
17. Object Management Group (OMG). Life cycle service specification, version 1.2. OMG document formal/02-09-01, 2002.
18. Hans P. Reiser, Udo Bartlang, and Franz J. Hauck. A reconfigurable system architecture for consensus-based group communication. In *Proc. of the 17th IASTED Int. Conf on Parallel and Distributed Systems (Phoenix, AZ, USA, Nov 14-16, 2005)*, 2005.
19. Hans P. Reiser, Michael J. Danel, and Franz J. Hauck. A flexible replication framework for scalable and reliable .NET services. In *Proc. of the IADIS Int. Conf. Applied Comuting 2005, Vol I, Algarve, P*, pages 161–169, 2005.
20. Hans P. Reiser, Franz J. Hauck, and Rüdiger Kapitza. Deterministic multithreading for replicated CORBA applications, 2006. submitted for publication.
21. Hans P. Reiser, Franz J. Hauck, Rüdiger Kapitza, and Andreas I. Schmied. Integrating fragmented objects into a CORBA environment. In *Proc. of the Net.ObjectDays (Erfurt, Germany)*, 2003.

22. Hans P. Reiser, Martin Steckermeier, and Franz J. Hauck. IDLflex: a flexible and generic compiler for CORBA IDL. In *Proc. of the Net.ObjectDays (Erfurt, Germany, Sep. 10-13, 2001)*, 2001.
23. Wenbing Zhao, Louise E. Moser, and P. M. Melliar-Smith. Deterministic scheduling for multithreaded replicas. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 74–81, Washington, DC, USA, 2005. IEEE Computer Society.