Aspects for Improvement of Performance in Fault-Tolerant Software

Diana Szentiványi
Department of Computer and Information Science, Linköping University, Sweden
diasz@ida.liu.se

Simin Nadjm-Tehrani
Department of Computer and Information Science, Linköping University, Sweden
simin@ida.liu.se

Abstract

This paper[*] describes the use of aspect-oriented programming to improve performance of fault-tolerant (FT) servers built with middleware support. Its contribution is to shift method call logging from middleware to application level in primary-backup replication. The novelty consists in no burden being placed on application writers, except for a simple component description aiding automatic generation of aspect code. The approach is illustrated by describing how synchronization aspects are weaved in an application, and modifications of an FT-CORBA platform to avoid middleware level logging. Evaluation is performed using a telecom application enriched with aspects, running on top of the aspect-supporting platform. We compare overheads with earlier results from runs on the base-line platform. Experiments show a drop of around 40% of original overheads. This is due to methods starting execution before previous ones end, in contrast to ordering enforced at middleware level where methods are executed sequentially, not adapting to application knowledge.

1. Introduction

The price of making a system fault-tolerant (FT) in terms of performance, flexibility, and other attributes is seldom quantified in a research environment with non-trivial industrial applications. To build up such analyses one needs a systematic study of different ways for improving fault tolerance in the same application. A basic premise is that providing support in middleware makes the task of building fault-tolerant services simpler, as application writers then mainly concentrate on the functional aspects of their code. At the other extreme is the opposite premise that each application writer should make his/her application as fault-tolerant as needed, having full control on performance penalties. Thus, a central issue in making any quantitative comparison is the performance/availability trade-off. For an application that is not time-critical, choosing the middleware support may be quite appropriate, providing flexibility, low cost of maintenance, and a high code quality with full transparency. However, making this very decision has been hampered by lack of systematic studies of the above-mentioned trade-offs. This paper studies a variation that is in between the above two extremes and compares it with an instance of the middleware-supported fault tolerance.

In earlier work we have quantitatively studied the price for obtaining high availability through building fault tolerance capabilities as part of a middleware[8]. Our experiments were performed with the code of a service extracted from the Operation and Management (O&M) layer of a mobile radio network, provided by Ericsson Radio Systems AB. These studies showed comparative measures of the timeliness penalty (roundtrip time overheads) for a range of FT mechanisms built within our FT-CORBA infrastructure. Other authors have also studied CORBA-based FT infrastructures and evaluated them (typically on non-industrial applications), e.g. Felber et al.[2],Narasimhan et al.[6]. Based on these studies, as well as the O&M based study, one can pinpoint what caused the major performance penalty: the call ordering needed in connection with logging method call information ([7],[8]).

In the context of *primary-backup* replication, to be able to restore the state of the backup to the state of the primary replica upon failure, periodic object state reading and storing is combined with update method[1] call information logging. A call record consists of method name and parameter list, thus enough information for replay.

Considering a worst case scenario where update operations performed on the server object are not commutative, the order of execution of those operations on the primary replica has to be the same as that of the logged call information. The reason is that when a failure occurs and a backup becomes primary, restoration of state

[1] Update methods (operations) are those which change the state of the server object, when executed

involves transfer of the latest recorded state as well as replaying methods from the call log arrived since the state recording took place. Otherwise, if the order of method call records in the log is different from the order in which those methods were actually executed on the primary, the state of the new primary will be inconsistent with that of the old one at the moment of its failure.

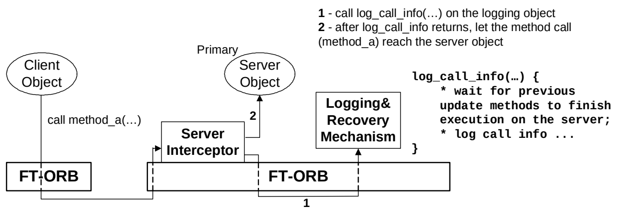
Although the FT-CORBA standard does not mandate a serialization of update operations on the primary, an ordered execution of the calls and preservation of this order in the log is needed. This is due to the fact that logging is performed at the middleware level and execution of the operations is performed in an independent manner at application level.

In this paper we provide a solution that is in between the FT-CORBA based support by middleware and the support for FT at individual application level. More specifically, we show how recent techniques for building adaptable components, specifically aspect orientation [5], can be used to shift some of the code generation to the application level (thus gaining performance by utilising application-specific knowledge). The contributions of the paper are twofold. First, the paper shows a description of how to use aspect-oriented programming for building FT applications on top of an FT supporting middleware. Second, it presents a quantitative evaluation of the performance gains by studying the decrease in overheads, when running the same O&M application, in comparison with earlier baselines. The experimental platform is still FT-CORBA, but the application code modifications are made so that the granularity of unit on which thread synchronisation is performed is shifted from method level (the earlier FT-CORBA solution) to statement level. We believe that the methodology is general enough for application on any FT middleware, other than CORBA, since the basic primitives of logging are inherent in all FT solutions. Our base FT infrastructure is a collection of service object on top of our extension of an existing Java implemented ORB [1]. Similarly, when adding aspects to support the application writer we chose the Java-based AspectJ[2]. The choice of AspectJ was motivated by its richness of pointcut designators and its generality. Again, we believe that the same principles can be used when using a different programming language, e.g. C++.

The paper is organised as follows: section 2 presents the motivation for the work, section 3 describes aspect-oriented programming concepts, section 4 describes our approach, and section 5 describes the implementation of the approach. Section 6 presents evaluation results, while section 7 presents our conclusions and discussions.

2. Motivation

Figure 1 presents a scenario in which a client communicates with a replicated server in a primary-backup setting, in a typical FT-CORBA implementation. Consider the serialization of update method call dispatches needed in the server side interceptor. Of course, if the underlying platform does not support multithreaded dispatch and execution of operations on the server object, the requirement of order preservation is satisfied with no additional intervention. On the other hand, for a multithreaded dispatch policy, method calls on the server object have to be “manually” stopped from performing changes on the object state in a different order than that reflected in the log. If stopping is done at the middleware level (as illustrated in Figure 1), then the granularity of thread synchronization is the whole method. In particular, in typical FT-CORBA implementations (see [7]), the logging of method call information is done at an interceptor level where an update method call is stopped until all preceding ones sent their reply to the client. This can lead to large average queuing (and thus roundtrip) times that may be unacceptable in some applications - especially if method calls come with a high arrival rate.


1 - call log_call_info(…) on the logging object
2 - after log_call_info returns, let the method call (method_a) reach the server object
Primary
Client Object
Server Object
call method_a(…)
Server Interceptor
Logging& Recovery Mechanism
FT-ORB
FT-ORB
1
2

```
log_call_info(…) {
    * wait for previous
    update methods to finish
    execution on the server;
    * log call info ...
}
```

Figure 1 Client-server communication in a FT-CORBA primary-backup setting

This paper presents the following alternative approach. We perform the logging of method call information in the method code itself (i.e. at application level). Method calls still must inflict their changes on the object state in the same order that they were logged. The difference now is that, synchronization is not done at whole method level, but at state variable access level.

Let us take an example Java class `ExampleClass` containing methods `method_a`, and `method_b` as in Figure 2. Consider the scenario where `method_a` is logged before `method_b`. When logging is performed at a platform (interceptor) level, the call to `method_b` is delayed until execution of `method_a` is ended. On the other hand, with application level logging and field level synchronization, the call to `method_b` is delayed only by the execution time of the first line in `method_a` (the assignment `v1=v2/3`).

[2] AspectJ is a trademark of Xerox Corporation. AspectJ web page: http://www.eclipse.org/aspectj

```
void method_a(Str[] val){                    void method_b(){
 StructureC loc;                              v1=v7-v1
 v1=v2/3;                                    }
 v2=17;
 v3=v4 || v3;
 loc=v5.field2;
 loc.set_a(3);
 v5.meth_(v6,4);
 for(int i=0;i<val.length;i++)
   System.out.println(val[i].the_v);
}
```

Figure 2 Two methods that have to be synchronized

3. Background

Aspect-oriented programming [5] is a very elegant way to cope with the introduction of non-functional concerns in a program, separate from functional ones. Besides, this is possible, at least theoretically, on top of any programming language. Non-functional properties can be fault tolerance ([3],[4]), synchronization, security, etc. An overall picture of a component modified by aspects, by so-called aspect weaving, is shown in Figure 3.

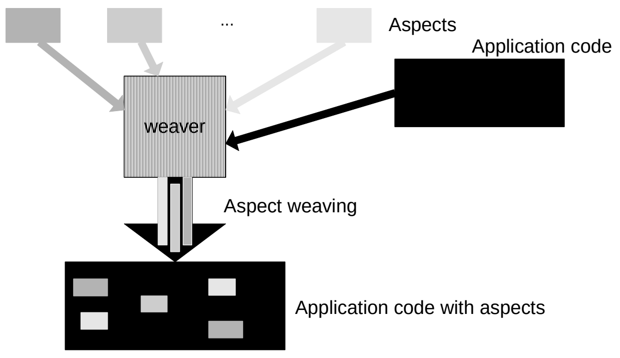


Figure 3 Aspect weaving

3.1. Aspects, join points, pointcuts and advices

Throughout the rest of the paper we will refer to *aspect* as a piece of code that is designed and implemented separate from the application code. The aspect code is meant to change the application code with regard to some non-functional property. It is "weaved" in the application code and thus parts of it (the *advices*) are executed at the specified *join points*.

A *join point* is a well-defined point in the execution of a program. For example, join points defined by AspectJ are, among others, start of execution of a certain method, call to a certain method, read/write (get/set) access to a field of a class. The code of an aspect consists of *pointcut* and *advice* definitions.

A *pointcut* is a program element that picks out join points, as well as data from the execution context of the join points. Pointcuts are used primarily by advices.

An *advice* contains the code that will be executed when the application program execution reaches a join point present in the set defined by the pointcut. The advice code can be executed *before*, *after* or *around* (instead) of the application code at the join point. These three keywords are used when defining the respective advices inside the aspect. The example in Figure 4 illustrates a *before* advice. Thus, if the aspect code is weaved in the application code, the new application object will print out the given text whenever it starts executing `method_a`. The italic words are keywords in AspectJ. The code is written using the AspectJ syntax for defining aspects, pointcuts and advices.

Aspect definition

Pointcut definition

Advice definition (to be executed before the join point, i.e. *before* method_a starts executing)

```
public aspect ExampleAspect {

  pointcut execmethod_aPointcut(Str[] val):
     execution(method_a(Str[]));

  before():execmethod_aPointcut(Str[]){
     System.out.println("before execution of method_a");
  }
}
```

Figure 4 A "`before`" advice

3.2. Enriching by "Introduction"

Besides enriching existing methods of a class (component) by adding pieces of code that execute at join points, it is possible to enrich the component itself by adding new methods and fields. This way of enriching is called *introduction* in AspectJ.

For example, our aspect (`ExampleAspect`) can be extended to introduce a new method `method_c` in the class `ExampleClass` (see Figure 5). Of course, it is possible to define pointcuts (and thus, advices) related to the new method, as well.

Introduction (of a new method) in class ExampleClass

Body of the new method

```
public aspect ExampleAspect {

  ExampleClass.method_c(int i, boolean b){
    v1=i/7+20;
    v3=b or v4;
    v7=v1-v2;
  }
  pointcut execmethod_aPointcut(Str[] val):
     execution(method_a(Str[]));

  before():execmethod_aPointcut(Str[]){
     System.out.println("before execution of method_a");
  }
}
```

Figure 5 Enriching by "introduction"

3.3. Aspects defined per control flow

In section 5.1, the notion of aspect defined percflow will be used. Therefore, an explanation of this AspectJ defined notion is needed here.

If aspect AspectForMethod_A is defined with the attribute percflow(Pointcut_p), where Pointcut_p is defined inside the body of the aspect (exec_method_a_p in Figure 6) then one object of type AspectForMethod_A is created each time the flow of control reaches a join point from the set defined by exec_method_a_p. The difference as compared with a non-percflow aspect AspectForMethod_A is as follows: the variables defined inside the body of the aspect (is_first and counter in Figure 6), do not become instance variables of the aspect-woven class (here, ExampleClass), initialized at the aspect instance creation, but they are defined as local variables within the scope of exec_method_a_p, each time an instance of AspectForMethod_A is created, i.e. each time method_a executes; also, the code of advices (before and after in Figure 6) described in the body of the aspect, is executed at join points selected by pointcuts defined inside AspectForMethod_A (get_field in Figure 6), but *only* those reached while in the scope of exec_method_a_p.

```
public aspect AspectForMethod_A percflow(exec_method_a_p(Str[])){

    int counter;
    boolean is_first;

    public AspectForMethod_A(){
        counter=0;
        is_first=true;
    }

   pointcut exec_method_a_p(Str[] val):execution(method_a(Str[]))
&& args(val);

   pointcut  get_field():get(int v1);

   before(): get_field(){
     if(is_first){
        ...
     }
   }
   after(): get_field(){
     counter++;
  }
}
```

Figure 6 Percflow aspect example

4. Aspects for fault-tolerant applications

In our approach, the application writer will be provided with aspects incorporating method logging clauses, as well as variable level synchronization clauses. The aspect code will be automatically generated from a simple component description provided by the application writer. Aspects can easily be weaved in the application code, to provide better performance of the application on top of an FT-CORBA middleware that uses a separate method information logging mechanism, and a different synchronization mechanism.

To illustrate this approach, we have used an existing FT-CORBA platform [7], and performed some modifications to support aspect-oriented application extensions. The modifications to the middleware were mainly within the interceptors that now forward the method logging operation execution to the application level. The application code is automatically modified using corresponding aspects.

The exact changes in the FT-CORBA infrastructure that are concerned with the logging mechanism (in order to perform method logging at the application level) are as follows:

(a) Interceptors used in primary-backup replication, were adapted to throw an exception containing the call information that has to be logged from the application level. Throwing an exception is needed, because the method call (in the form it comes from the client) has to be stopped from reaching the application object. After the exception is thrown from the interceptor, other middleware levels are informed to take the necessary steps so that the application receives the call to the right (pseudo) method (the one that incorporates logging clauses and whose signature has an extended set of parameters - see section 5). These changes solely affect the server side interceptor in the FT-CORBA infrastructure. Thus, an application may choose to use the standard (FT-CORBA) mechanisms (i.e. those that do not throw the mentioned exception), or the version supporting aspects. This is simply done by using different interceptors when deploying the enhanced infrastructure. Applications with different performance requirements can thus exist side-by-side using different interceptors.

(b) The original logging mechanism was extended with an operation for storing call information, without waiting for previous calls to finish execution. Of course, if the application writer wants to use the regular FT-CORBA platform, he/she can choose to use the new logging object, and calling the (old) logging operation that includes waiting, or the old implementation of the logging object.

In Figure 7 we illustrate the extension of our FT-CORBA infrastructure with the addition of support for aspects, and logging performed from application level.

An application writer that will use the support from a regular FT-CORBA infrastructure has to add a couple of small extensions to the application: methods for getting and setting the object state. Logging of information about method calls in the right order is entirely taken care of by the infrastructure. Similar additions to the application code are also necessary when the application writer chooses to use our FT-CORBA infrastructure with support for aspect orientation.

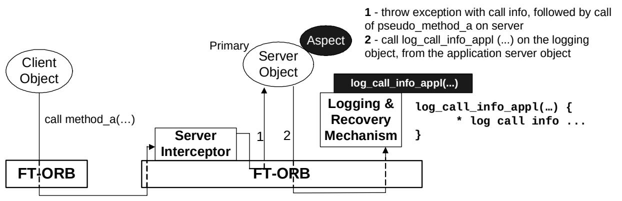

1 - throw exception with call info, followed by call of pseudo_method_a on server
2 - call log_call_info_appl (...) on the logging object, from the application server object
Aspect
Primary
Server Object
Client Object
log_call_info_appl(...)
Logging & Recovery Mechanism
log_call_info_appl(…) {
* log call info ...
}
call method_a(…)
Server Interceptor
1
2
FT-ORB
FT-ORB

Figure 7 Client-server communication using the aspect and FT supporting middleware

To aid the automatic generation of the aspect code, the application writer has to provide a component description that we proceed to explain below.

4.1. Context

Synchronization and method logging aspects will be weaved in the code of update methods. Synchronization at variable level involves the fields of the object/component that are accessed inside an update method. With respect to taking and releasing the lock corresponding to a variable, the first and last access to that variable inside the method are important.

To simplify the implementation, we make no distinction between read/get and write/set accesses to a variable. Also, the accesses are counted in terms of number of appearances of that variable in an instruction (for example: in the assignment `a=a/2+a`, the number of accesses to `a` is three, although to compute the right hand side expression `a` has to be read only once).

4.2. Component description

The component description provided by the application writer has to contain the following:

- A table (`TypeConv`) of Java type to IDL type mappings; the table is used when method call information is logged; thus, the mappings have to be given only for types of parameters or results of the update methods;
- A list (`Interface`) with update methods, in terms of method signatures, with parameter types and names;
- A list (`State`) with the types and names of object fields, that are accessed in update methods, as well as the types for state elements that are accessed via local variables;
- A table (`FieldAccess`) of method names and corresponding accessed fields (with names), or types of local variables, followed by number of accesses in that method.

Note that all of this information can, in principle, be automatically deduced by static analysis of the program. However, for the purpose of this study, we assume these as given.

For our class `ExampleClass` the description is presented in Figure 8; the methods in which aspects have to be weaved are `method_a` and `method_b`. Method `method_a` accesses the fields `v1`, `v2`, `v3`, `v4`, `v5`, `v6`, `v7`, and a local variable of type `StructureC`, which when being modified changes the state of the object (via field `v5`). The description is written according to syntax that we impose: the italic words in the figure are keywords; also the use of ":" in the `FieldAccess` section is our choice.

```
#BTypeConv
Str[]  StrList
#ETypeConv

#Binterface
void method_a(Str[] val);
void method_b();
#EInterface

#BState
int v1,int v2, boolean v3, boolean v4, StructureA v5,
StructureB v6, int v7, StructureC
#EState

#BFieldAccess
method_a v1:1, v2:2, v3:2, v4:1, v5:1, v6:1, StructureC:1
method_b v1:2, v7:1
#EFieldAccess
```

Figure 8 Component description

5. Implementation issues

The FT-CORBA infrastructure requires the application writer to indicate which replication style they wish to use in their application (primary backup with cold or warm passive replication, or active replication). In this section we explain how the support for the two primary-backup mechanisms is implemented within the aspect-orientation framework. Support for active replication can also be provided with little effort. However, queuing time not being the largest part of the overhead in that case, no big performance improvement is expected.

For warm/cold passive replication, method call information that is logged consists of the name of the method (e.g. `method_a`), the list of call parameters encapsulated in a list of middleware specific types[3], and the unique request identification information (retention identifier, client identifier). When logging is done at the infrastructure level, the latter piece of information is easy

[3] In our example the argument `val` of type `Str[]` is encapsulated in the single element of an array of elements of type `Parameter`.

to obtain. On the other hand, at the application level, this information is not available unless the method is called with these extra parameters as well.

Our approach is to generate one separate aspect code file for each method in the list `Interface`. Each aspect so generated, combines the introduction of a new *pseudo* method in the application code, with the definition of advices that contain code for method logging, and the definition of advices for variable level synchronization. The advices will be executed as the specified join points are reached. The new (pseudo) method is introduced in the class, in order to be able to extend the parameter list of the original method, without the application writer being obliged to write a new method with an extended signature. The body of the pseudo method is simply a call to the original method with the original set of parameters.

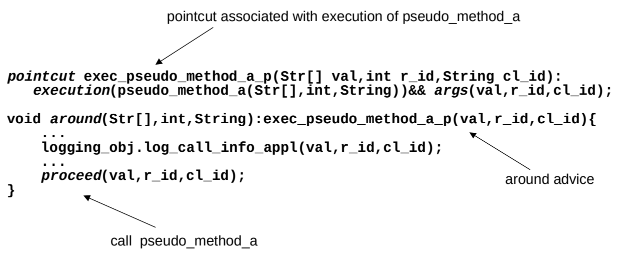
5.1. Method execution related advices

To illustrate how an aspect file looks like, let us take the example of `method_a` from our `ExampleClass`. There will be one pseudo method introduced in the class, that is `pseudo_method_a`. The parameters of `pseudo_method_a` are the parameters of `method_a`, plus the extra parameters needed, as mentioned before in case of primary-backup replication. The join point for which the method related advice is defined is the execution of the pseudo method. The aspect code is defined to activate its advices at the given join points, whenever the method `pseudo_method_a` is executed, i.e. the control flow reaches any join point defined by pointcut `exec_pseudo_method_a`. To achieve this we need an aspect defined with the attribute **`percflow,`** which is written as follows:

```
public aspect Asp_METHOD_A
   percflow(exec_pseudo_method_a_p(Str[],int,String)){
   …
}
```

The reason for defining all aspects to be of type **`percflow`** is the need to introduce some extra local counter variables in the methods, used as explained below.

The advice code that will be executed at the method execution join point is of type **`around`**. This means that the code written inside the advice is run instead of the code of the method. The new execution starts with logging the method call information obtained from the parameters of the call. Further, the numbers of threads currently accessing fields that will be read/written by this method (thread) are assigned to the extra local counter variables. The global variables containing the numbers are incremented. These global variables are static fields in an aspect class (not detailed here) specially defined to contain them. Finally, the (pseudo) method itself is called (by using the AspectJ keyword `proceed`). A fragment of the aspect code used for the extension of `method_a` is shown in Figure 9.



```
pointcut exec_pseudo_method_a_p(Str[] val,int r_id,String cl_id):
   execution(pseudo_method_a(Str[],int,String))&& args(val,r_id,cl_id);

void around(Str[],int,String):exec_pseudo_method_a_p(val,r_id,cl_id){
    ...
    logging_obj.log_call_info_appl(val,r_id,cl_id);
    ...
    proceed(val,r_id,cl_id);
}
```

Figure 9 An "`around`" advice

The reason for the somehow strange combination of introducing a new method in the application class and then replacing it with an around advice is that the local variables defined inside the percflow aspect cannot be accessed in the pseudo method itself as method (only introduced) in the application class.

Of course, the server side ORB has to know to call the pseudo method instead of the original one, since the client will transparently call the latter. The code of the skeleton generated at the server side is augmented with calls to pseudo methods, according to the component description provided by the application writer. At runtime, the skeleton object is directly provided with the name of the pseudo method to be called: in the point immediately over the server interceptor, the method name is simply changed from the original (as called by the client) to the pseudo method's name. The server interceptor is responsible for throwing the exception that will inform other ORB levels about the name change, and of course about the extra parameters[4].

Note that e.g. `method_a` still exists in the class `ExampleClass`. However, it will not be called when the aspect-supporting infrastructure is used. Instead, `pseudo_method_a` is called with the functionality of `method_a` preserved inside it.

5.2. Field level synchronization advices

Other join points used to define advices are the get/set accesses to object fields and local variables that can modify the state. To select get/set accesses to e.g. a field called `v1` of type `int`, the following pointcut is written in AspectJ:

```
pointcut get_set_v1_p():get(int v1)||set(int v1);
```

[4] The IDL compiler was changed to cope with the addition of pseudo method calls to the skeleton's code.

To select get/set accesses to e.g. the type `StructureC` the pointcut looks as follows:

```
pointcut access_StructureC_p():target(StrcutureC);
```

The variable level synchronization takes place as follows. Before the first access to a field in a method, the current thread gains the semaphore corresponding to that variable, after all previous accesses by other methods (threads) are finished. It is known when this happens, because the local counter variables are decreased whenever the semaphore corresponding to that field is signalled. The semaphore is signalled in a method, after the last access to the field in that method. The first access to a variable in a method is detected in the advice code that is always executed before the variable access. To detect the final access, counting of all get/sets is done and the number compared with the value specified in the component description file. The semaphore variables are also static fields in the same aspect class that contains the global counter variables.

6. Evaluation of the approach

We performed experiments in a replicated setting by using the aspect-supporting FT-CORBA platform built as described above. The O&M service is a so-called “Activity Manager” whose role is to create activities and jobs to be scheduled at later times. The update operations performed on the activity manager server object by client requests modify its state by mainly creating jobs and activities, starting and terminating those, as well as reporting status changes in activity executions. The application has defined fifteen update operations, out of which we experimented with six.

The O&M service was augmented by the aspects generated based on the component description. Our goal was to compare average roundtrip time overheads obtained in this setting with the average overheads measured using the baseline FT-CORBA platform. Overhead values were computed by subtracting the roundtrip time obtained in a non-replicated scenario from the time taken in the replicated one.

Six update methods (called `m_1`, `m_2`, etc. in our result tables, and referred also as method 1, 2, etc.) were used in the tests and average overhead percentages were computed for each.

We performed our experiments on a set of SUN Ultra SPARC workstations running SunOS 5.8. The machines were connected in a LAN. We did not have control over the link traffic or the load on the machines. The reason for not performing measurements in a controlled host environment was to mimic the realistic setting in which the service will eventually run.

The parameters we varied during the experiments were: the replication style (warm or cold), the number of replicas, the checkpointing interval, and the number of times the client called the methods on the server, i.e. the number of iterations of the loop which the client used. The measurements obtained after each varied parameter will be referred to as one experiment. Thus, in case of cold passive replication, for each server method called by the client, and for each infrastructure, we obtained 12 numbers[5] designating average roundtrip time overheads, corresponding to 12 combinations of parameters:

– number of replicas: 2 (1 value)
– checkpointing intervals: 1, 5, and 10 seconds respectively (3 values)
– number of iterations: 100, 200, 400 and 800 respectively (4 values)

In case of warm passive replication, for each method called by the client, and for each infrastructure we obtained 36 numbers[6] designating average roundtrip time overheads, corresponding to 36 combinations of parameters:

– number of replicas: 2, 3 , and 8 respectively (3 values)
– checkpointing intervals: 1, 5, and 10 seconds respectively (3 values)
– number of iterations: 100, 200, 400 and 800 respectively (4 values).

Choosing only one value for the number of replicas parameter, in cold passive replication, was due to the fact that the parameter is expected not to influence at all the overheads. In case of warm passive replication, the overhead does not change much either with the number of replicas, but more than in case of cold passive.

The two charts (Figure 10 and Figure 11) present the average roundtrip overhead computed over the above averages obtained in each experiment. That is, for each of methods 1 to 6, and each platform, a comparison is presented between:

– cold passive: the average of the 12 numbers obtained, and
– warm passive: the average of the 36 numbers obtained.

The charts show a general drop in the presented average values. This is especially true for cold passive replication

[5] This means a total of 12×6=72 numbers for the original FT-CORBA and 72 numbers for the aspect supporting FT-CORBA

[6] This means a total of 36×6=216 numbers for the original FT-CORBA and 216 numbers for the aspect supporting FT-CORBA

where the drop is sometimes over 50%. Due to the fact that the call to `m_1` does not interfere with so many other method calls, it shouldn't be a surprise that in case of method 1 the results do not show an improvement. In fact, the synchronization clauses in the application code can themselves be causing an overhead. In addition, there is a further explanation for the phenomenon: when a method call arrives in the server's synchronization-induced wait queue, it is possible that a call to `get_state (`method called from time to time to save the state of the object during the checkpoint operation) is there somewhere in the front. Now, `get_state` cannot start executing until all previous methods in the wait queue finish their execution; this is valid in both platforms (the FT-CORBA and the aspect-supporting FT-CORBA). Therefore, the method call that arrives after the call to `get_state` (e.g. `m_1`), but before `get_state` finished execution, has to wait until *all* methods arrived before `get_state` (those for which `get_state` is itself waiting) finish execution. In such a situation our variable level synchronization does not improve performance.

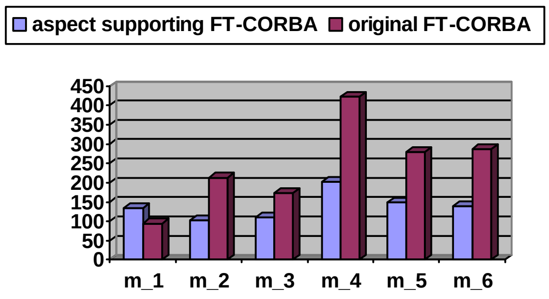


Figure 10 Average over average overhead percentages for cold passive replication

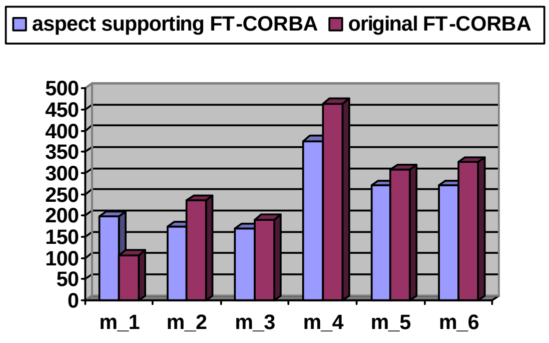


Figure 11 Average over average overhead percentages for warm passive replication

The figures confirm our hypothesis, indicating: in a setting with multithreaded server call dispatching, with a minimal specification of application level knowledge, the average overhead percentages in primary-backup replication is generally improved – with around 40%.

7. Conclusion and discussion

In this paper we presented the results of the successful merging of two areas: middleware supporting fault tolerance with aspect oriented programming for weaving of fault tolerance aspects in the application. Aspects are shown to be a powerful tool for adaptive development of non-functional requirements (e.g. real-time properties[9]). To our knowledge this is the first work that uses aspects for improving performance of a fault-tolerant application on top of a standard middleware. Of course, the improvement is most significant when considering the worst-case scenario, i.e. high degree of importance of update method execution order. However, when the middleware builder and the application writer do not have the chance to communicate, and the middleware has to be built general enough to be able to accommodate many different applications, the worst case scenario assumption is not unreasonable. What we proposed in this work does not remove the worst-case scenario assumption, but aids the application writer to adjust his/her application to deliver better performance and still keep ordering (serialization) requirements. The success of using aspects should be seen in the modification of the application code with no programmer intervention leading to a gain in performance of server request processing. In some cases, the overhead percentage dropped to half of the overhead rate in a regular FT-CORBA implementation.

As mentioned before, there is a potential for reducing the application writer input to only writing application code (i.e. no added effort due to FT and aspect-orientation support requirements). This could be implemented if standard techniques for static analysis are used to deduce the variable usage in methods instead of the programmer writing the component description.

Although the same-order requirement described before is imposed in order to avoid state inconsistencies when a backup is instated as new primary, there is no change in failover time as compared with the original FT-CORBA setting. The failover time mainly consists of the time taken to install the latest state on the new primary, plus the time for replaying the methods from the call log. Both of these times are independent of the application being weaved with aspects or not.

As expected, there are also some drawbacks to the approach. Of course, the gain in performance applies when the field accesses inside methods are organized such that the extra synchronization code execution time can be neglected. In particular, there are cases where even by synchronizing at variable level a method cannot start executing before a previous one has completely finished.

Imagine the case where `method_a` in our example is changed so that the line `v1=v2/3` is moved from the beginning to the end of the method body (Figure 12).

```
void method_a(Str[] val){                        void method_b(){
 StructureC loc;                                  v1=v7-v1;
 ~~v1=v2/3;~~                                        }
 v2=17;
 v3=v4 or v3;
 loc=v5.field2;
 loc.set_a(3);
 v5.meth_(v6,4);
 for(int i=0;i<val.length;i++)
   System.out.println(val[i].the_v);
 v1=v2/3;
}
```

Figure 12 Changed `method_a`

In a scenario in which the call to `method_a` is logged first, `method_b` cannot start execution, until `method_a` releases the lock on `v1`, i.e. finishes its execution.

If we think about code maintenance, as soon as the component description changes, new aspects have to be generated and weaved in the code. Of course, it is possible that the server class was only extended with new methods, and perhaps new state variables. In this case, the new aspects can be weaved directly in the already "aspected" code. The worst-case happens when the functional code has to be modified by weaving in totally new aspects. This can be solved by "putting together" the non aspect weaved code and the new aspect files.

In the context of modifying the application code by weaving aspect code, security problems can be considered. However, in the present setting, we assume that the middleware together with the aspect generating software is provided by a trusted party. Also, the application writer who augments his/her code with the aspects generated based on the component description does not impair security more than he/she would if no external code would be weaved in the application. Thus, the new usage of the (aspect-supporting) FT-CORBA middleware will presumably not impair the security of the application more than the original FT-CORBA infrastructure.

Future works include the study of other method structures (creating different synchronization scenarios) and extensions of the technique to other non-functional properties.

8. Acknowledgements

This work was supported by the Project TRANSORG, part of the European Commission IST initiative and is included in the cluster of projects EUTIST-AMI on Agents and Middleware Technologies applied in real industrial environments (www.eutist-ami.org). It was also supported by the European project Safeguard (IST-2001-32685).

The authors also wish to thank Călin Curescu and the anonymous reviewers for their valuable comments.

References

[1] J. Daniel, M. Daniel, O. Modica, and C. Wood. Exolab OpenORB. Webpage http://www.openorb.com/.

[2] P. Felber, R. Guerraoui, and A. Schiper: Replication of CORBA Objects. *Volume 1752 of Lecture Notes in Computer Science,* Springer Verlag, Berlin, 2000, pp. 254-276.

[3] A. Gal, O. Spinczyk, and W. Schröder Preikschat: On Aspect-Orientation in Distributed Real-Time Dependable Systems. *In Proceedings of the Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 7-9, 2002, pp. 261-270.

[4] J. Herrero, F. Sanchez, and M. Toro: Fault Tolerance AOP Approach. *In Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns,* Lancaster University, UK, 24 August, 2001, pp.44-52

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin: Aspect-Oriented Programming. *Invited talk* in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* Finland, Springer-Verlag LNCS 1241, June 1997, pp.220-242 .

[6] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith Using Interceptors to Enhance CORBA. *IEEE Computer,* 32(7), 1999, pp. 62-68.

[7] D. Szentiványi and S. Nadjm-Tehrani: Building and Evaluating a Fault-Tolerant CORBA Infrastructure. *In Proceedings of the DSN Workshop on Dependable Middleware-Based Systems,* June 23-26, 2002,Washington, DC, pp. G-31—G-38.

[8] D. Szentiványi and S. Nadjm-Tehrani: Middleware Support for Fault Tolerance. To appear as a chapter in the book *"Middleware for Communications"* edited by Qusay H. Mahmoud, published by Wiley and sons.

[9] A. Tesanovic, D. Nyström, J. Hansson, and C. Norström: Towards Aspectual Component-Based Development of Real-Time Systems. *In Proceeding of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003),* Springer-Verlag, Feb. 2003.