Componentization of Fault Tolerance Software for Fine-Grain Adaptation

Thomas PAREAUD[1], Jean-Charles FABRE[1,2], Marc-Olivier KILLIJIAN[1]
[1] LAAS-CNRS ; Université de Toulouse ; 7, avenue du Colonel Roche, F-31077 Toulouse, France
[2] Université de Toulouse ; INP
{tpareaud, fabre, marco.killijian}@laas.fr

Abstract

The evolution of systems during operational lifetime is becoming a core assumption of the design. This is the case for resource constrained embedded systems. Such an evolution may be driven by environment or the execution context. The adequacy of the service delivery with respect to the current operational conditions depends on the ability to tune the software configuration accordingly. This is true for application services, but also for dependability services, in particular the fault tolerance software. This paper presents a design of fault tolerance software for its runtime adaptation. This design relies on a reflective framework and open component based software engineering (CBSE) techniques. We demonstrate in this paper the feasibility of adapting componentized fault tolerance at a meta-level of the application.

1. Introduction

On-line evolution of software is becoming a requirement for many types of systems in different application domains, those being critical or not. This problem is clearly of interest for large-scale systems because evolution is a core assumption of their design, e.g. service-oriented architectures. It is worth noting however that it is becoming the case also for systems in the small, e.g. embedded system, in the context of ubiquitous computing, mobile systems, long living space systems, for instance. Considering evolution, the application software is often the target for adaptation. However, changes may also regard threats, fault assumptions, or resources. Then, the fault tolerance software can be subject to adaptation with respect to fault models (*Context-aware Fault Tolerant Computing*) or resources (*Resource-aware Fault Tolerant Computing*).

The adaptation of non-functional mechanisms raises two major issues. The first one refers to the assessment of a dynamic system, i.e. monitoring operational conditions, to take the decision of adapting the system. The second issue refers to the on-line modification of the software devoted to fault tolerance.

In this paper, we focus on the design of the system enabling the modification of its fault tolerance software at runtime. This is a fundamental issue to perform fine grain adaptation and to minimize space and CPU resources. It is important to mention that coarse reconfiguration is always possible at the expense of global freezing of the system, resources (memory, time) consumption overhead, and often the loss of past execution history. An appropriate design for on-line modification is mandatory to tackle fine-grain adaptation. The proposed design relies on both an architectural design and technologies providing both separation of concerns and concepts for software modification.

The overall approach relies on the recursive use of the reflection paradigm to build the architecture. This first implies the definition of an on-line model of the functional software to separate fault tolerance and functionalities in two different abstraction levels. Then, we propose to decompose fault tolerance into small components in order to modify the fault tolerance software at runtime. This component technology relies on a middleware implementing a reflective component model. A case study will illustrate this design and demonstrate the feasibility of the adaptation.

2. Problem statement, proposed approach

The problem is the following. Let S_1 and S_2 be two fault tolerance configurations, how should we modify S_1 using a set of algorithms switches and parameters updates, in order to install S_2 maintaining system correctness and availability.

From a design viewpoint, the fault tolerance software has to be as independent from the application as possible. In general, fault tolerance software is very

difficult to modify, because it is often merged with functional algorithms. Substitution of fault tolerance algorithms implies that they are separated from functional ones. As far as on-line fault tolerance adaptation is concerned, separation of concerns must be effective at runtime, and not only at compile time. Reflection [1] and more recently Aspects [2], have shown their interest regarding separation of concerns, i.e. mapping different software concerns to different levels of a system. For example, in [3], reflection has been used to separate fault tolerance and application in a CORBA system.

The reflection principles rely on an ordered set of *meta-levels*. The level associated to the functionalities of the system is called the *base level*. A meta-level has a representation of the lower levels called a meta-model. This representation is synchronized, i.e. causally connected, with the levels it represents. When a modification is made to the representation, the corresponding level is impacted. Conversely, the evolution of the levels will be visible through their representation. *Reflective mechanisms* are needed to synchronize the representation and the implementation. In practice, each level is implemented in a *separate software layer*.

We introduce a reflective architecture composed of three layers (cf. figure 1). Each layer focuses on a separated crosscutting concern of the system. The base level implements the functionalities of the system, the first meta-level deals with fault tolerance of the system, the second meta-level controls the adaptation of the fault tolerance software.

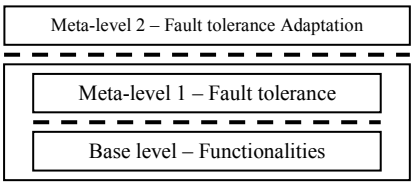

Figure 1. System reflective architecture

The manipulation of the fault tolerance software at runtime requires the software to be designed using abstractions that enable the modification of the software at runtime, both in terms of parameters and algorithms. We propose to use a reflective component model. This technology enables to represent the software as an architecture of inter-connected boxes. Then, it is possible to create new boxes, to remove some existing boxes, to connect or disconnect boxes. Such a model is provided by some component models, which provide ways to manipulate software at runtime. These boxes are *software components*. A component has several interfaces (represented by circles) and receptacles (represented by half-circle). A receptacle is connected to one or several interfaces. Components interact through connexions. Component architecture refers to a set of inter-connected components. The componentization process consists in defining the services provided by the components, i.e. their interfaces.

The fault tolerance software can take advantage of the representation of the functional layer provided by the reflective component model. The switch between two fault tolerance algorithms becomes possible for the adaptation layer, and consists in the substitution of some inter-connected boxes.

In the remainder of this paper, we show how the link between the functional layer and the fault tolerance software can be realized using the component technology. Then, we propose a decomposition of fault tolerance software into components, in order to enable its modification at runtime.

3. Modelling application for fault tolerance

The meta-level for fault tolerance involves the introduction of a structural and behavioural model of the functional part of the system. A model has been proposed in [3] for implementing replication protocols on CORBA. We decided to reuse this model. It is composed of four reflective mechanisms:

- Reification of an incoming interaction
- Reification of an outgoing interaction
- Intercession of method calls
- State handling of the component (save/restore)

The adaptation assumes the use of component technology for the two lower layers of the system. Thus, in order to be consistent with this assumption, we use components to provide these reflective mechanisms. The next two sub-sections detail the implementation of these reflective mechanisms using component technology.

3.1. State handling

The management of state information is mandatory for most of the fault tolerance strategies. Saving the correct state of the component before a failure occurs, is essential for later recovery, and can be done using checkpointing techniques. The recovery operation in this case is also called *rollback*. The problem of the checkpointing is a really difficult problem in complex software architectures. When checkpointing is not well designed, it may lead to serious problems and prevent successful recovery. Moreover, the state is a very complex notion. The state of a component is composed of internal data structures, executions stacks but also of the state of the opened channels such as sockets or files. A part of this state is thus dependent of the lower layers of the system (middleware layers, the kernel).

Such a complex state can be captured using multi-layer reflective technology [4].

State handling can be realised in several ways. First, the state save and the restore operations can be delegated to a third party [5], provided by the execution platform. Second, inheritance mechanisms [6] can be use to delegate the implementation of the operations to the application developer. Third, reflective language (OpenC++ [7], AspectJ [8] or the java.lang.reflect library of Java) can be used. We propose to equip every component with the *IState* interface, which provides save and restore methods that can be implemented with any of the above mentioned solutions.

3.2. Behaviour control

The capture of the behaviour of the functional components synchronise fault tolerance mechanisms to the base level of the architecture. This behaviour is observable through interactions between components by means of their connected interfaces. We consider two kinds of interactions. The first one corresponds to a call to the target component. They are called *incoming interactions*. The second one corresponds to interactions issued by the target component itself. They are called *outgoing interactions*. In order to capture the behaviour and apply fault tolerance processing, these interactions have to be synchronously reified to the fault tolerance layer.

Table 1. Reflective mechanisms

Interfaces	Details
IReifyIncomingInteractions	Reification of a incoming interactions
IReifyOutgoingInteractions	Reification of a outgoing interactions
IIntercessionIncomingInteractions	Enable to call a method of the functional component
IIntercessionOutgoingInteractions	Enable to call a method of a component connected to the wrapper

We introduced the reflective model for fault tolerance using a composite component called the *ApplicationController*, which wraps each functional component. The *ApplicationController* plays the role of a meta-object protocol since it provides the reflective mechanisms in order to link fault tolerance level with functional level. It intercepts interactions with the wrapped component, and reifies them at the fault tolerance level. A component in charge of the fault tolerance of C_2 can be connected to reflective interfaces and receptacles of the *ApplicationController*, which are detailed in the table 1.

The figure 2 illustrates these reflective mechanisms. Three functional components are connected together. We wrapped C_2, to add fault tolerance service on it. The wrapper reifies incoming interaction (1) to the fault tolerance component (2). Then, the fault tolerance processes verification for instance (3) and calls the original method (4). The functional component processes the call (5). Then the component starts a call from its receptacle toward a connected component. It is reified to the fault tolerance level (6). The fault tolerance component processes the reification (7) and returns to the base level for the outgoing interaction (8). The outgoing interaction is then done (9). This picture does not illustrate the return path of the execution.

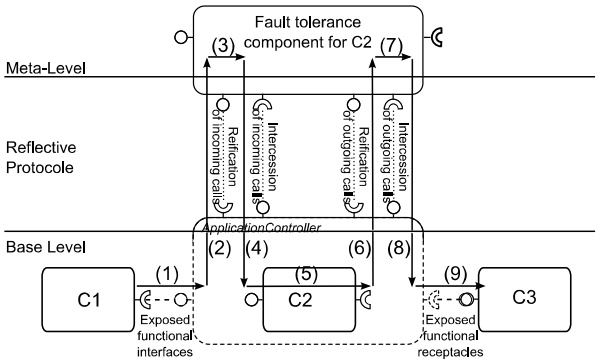

Figure 2. Capture of the control flow

An *ApplicationController* was implemented using the OpenCOM component model [9] in Java.

4. Component design for adaptation

In order to adapt the state component architecture, it is necessary to deal with two fundamental issues. The first one is to put the execution of the system in a state where modifications of the component architecture are possible without compromising the consistency of the whole execution. This problem involves being able to monitor the execution within the component architecture. In a component architecture, the execution is observable at the connection level. The second one is to transfer the state of the component architecture before adaptation to the component architecture after adaptation. This problem involves being able to introspect the state of component and update it. This state transfer necessarily introduces mapping functions, to convert the state before adaptation, in a state compatible with components after adaptation. There is a trade-off between the granularity of the componentization and the complexity of execution and state management.

The decomposition determines the elementary blocks that can be modified. Now, the real question is what has to be modified to go from one configuration to another. The fault tolerance software is built from

generic services (algorithms and protocols) as well as from specific fault tolerance mechanisms and protocols. During adaptation, the generic services are likely to remain unchanged, whereas, fault tolerance mechanisms should be modified. These fault tolerance mechanisms can be, for example, replication protocols which depend on generic services. The fault tolerance mechanisms are likely to contain a very simple state, or even to be stateless.

Our decomposition process is two-fold: on the one hand, we consider the generic services on top of which fault tolerance mechanisms are developed, and on the other hand, we analyse the differences and common points between various fault tolerance protocols in order to find the good elementary blocks for adaptation.

4.1. Generic services

In our opinion, generic services are linked with fundamental resources management. These services can be classified into three main categories: time, communication and storage services.

Time service provides both a local or global dating service (*IClock* interface) and an alarm service (*IAlarmMng* and *IAlarmListener* interfaces).

We limit our decomposition to high level communication abstractions providing services such as: the message send service (*ISend* interface), the regular reception service (*IRegular-Receive* and *IInformationReceive* interfaces), a listening message reception service (*IRegularReceive-Listener* and *IInformationReceiveListener* interfaces), a peer connection service (*IConnect* interface), a group connection and management service (*IGroup* interface). The adaptation of communication protocols is out of the scope of this paper (see [10, 11])

We introduce the storage service (*IStorage*) that provides means to save and restore persistent data. The implementation can range from a local disk to a safe distributed storage.

4.2. Taxonomy-based componentization

We have used the fault tolerance's taxonomy proposed in [12] to guide the componentization of fault tolerance. This taxonomy provides a semantic decomposition of fault tolerance into several standard mechanisms.

The *error detection* is in charge of detecting the presence of an error in the system. The *recovery* is in charge of transforming the erroneous state of the system into a fault-free state. Recovery is decomposed into *error handling* and *fault* handling. Error handling aims at eliminating error from the system's state using rollback, rollforward or compensation techniques. The *fault handling* prevents faults to be reactivated. It contains *diagnostic* that identifies the causes of the errors and *isolation* which excludes faulty components in the future service delivery. It also includes *reconfiguration* that reassigns tasks among non-faulty components and *reinitialization* that updates the system tables.

Reconfiguration in fault tolerant computing differs from adaptation in two ways. Firstly, reconfiguration refers to fault handling mechanisms, whereas adaptation focuses on the modification of the fault tolerance software due to operational conditions. Secondly, the overall component architecture of the system is not modified by the reconfiguration; that is not the case for fault tolerance adaptation.

Table 2. Fault tolerance services

Interface Name	Provided Service
IErrorDetectionNotification	Notification of error detection
IErrorHandling	Error handling (rollback, rollforward, compensation)
IErrorDiagnostic	Diagnostic of error
IComponentIsolation	Isolation of functional components
IReinitialization	Application of records updates
IFunctionalReconfiguration	Reconfiguration of functional components
ILog	Log of data
ICheckpointing	Take a Checkpoint of the state
IInterReplicasProtocol IInterReplicasProtocolListener	Protocol between replicas
IElection	Election service

This decomposition can be improved for adaptation. Some other abstractions of fault tolerance can be added. In order to perform the rollback on a component, it is necessary to previously save its state. More mechanisms are needed: the *log* and the *checkpointing* services. Moreover, fault tolerance often relies on the use of replication. In fault tolerance strategies based on replication, replicas are synchronized using a macro-protocol (notion of *inter-replicas protocol*). In primary-backup or leader-follower replication for instance, when an error is detected, an *election* has to be launched in order to determine which replicas will become the new leader one. This election can rely on a predetermined order or a distributed consensus algorithm. Election is an additional possible service. The list of fault tolerance services is given in the table 2.

This decomposition is very interesting from the adaptation viewpoint. It offers the opportunity to observe the key steps of the execution of the fault tolerance software at the interface of the component. For instance, an error has been detected if the *IErrorDetectionNotification* service is called, or an error handling is on-going if a call on the

IErrorHandling interface is not terminated. Thus, at runtime, some information about the fault tolerance execution can be captured.

5. Case study

To illustrate this decomposition, we propose to consider a conventional problem of automatic control. The controlled system is an inverse pendulum located on a cart. The pendulum can rotate around a shaft, which connects it with the cart. The cart can move on rails using an electrical engine. A sensor measures the angle of the pendulum and the position of the cart. An actuator controls the acceleration of the engine. The goal is to make the cart moving to a wanted position, provided by a console, by keeping the pendulum in equilibrium in the inversed position.

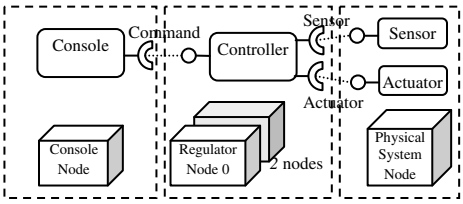

Figure 3. Case study

The system is implemented as depicted in the figure 3. The *Controller* contains a periodic task. This task reads the sensor values, calculates the acceleration to reach the position specified using the console, and applies it using the actuator. Its period is 50ms. The controller is naturally robust and can exceptionally miss three consecutive deadlines before loosing the system control.

Our objective is to tolerate the crash of the controller using two fault tolerance strategies based on distributed replication, namely the leader-follower replication (LFR) and the primary backup replication (PBR). Regarding resource usage, LFR uses more memory and CPU since two replicas are running in parallel, but less network bandwidth since no checkpoints are required.

These strategies are detailed in the next two sections. The section 6.3 provides the cost of these strategies.

5.1. Leader Follower Replication

The Leader Follower Replication (LFR) is a semi-active replication strategy: a replica has two modes of operation, the leader mode in which the replica is active, performing the normal service, and the follower mode in which the replica processes input requests but does not interact directly with other functional components. One replica is leader, others are followers.

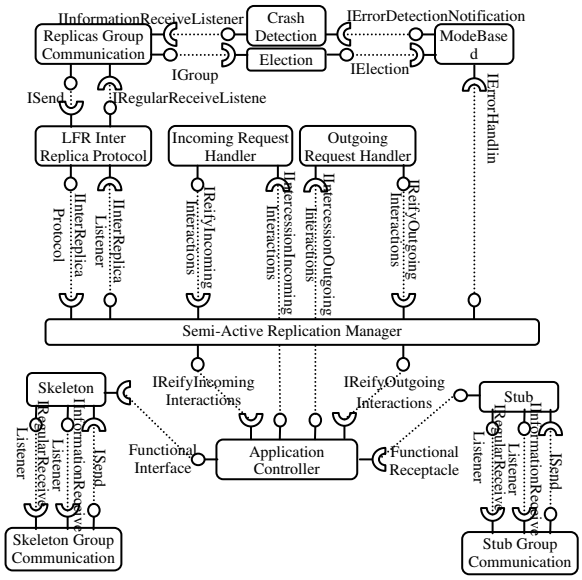

Figure 4. LFR replica component architecture

When the leader crash is detected, a follower is elected as the new leader. The recovery consists in sending all unacknowledged replies to clients and sending requests for which replies has not been forwarded by the leader. Then, the former follower replica follows its execution in leader mode.

In our case, the sensor and the actuator can receive the same request twice. This is not a problem since the fresher is the data; the more relevant data is used for controlling the process. In the same manner, the client ignores replies that have been received several times.

In our implementation (cf. figure 4), we introduce stubs and skeletons, to interact with the group communication implementation providing reliable atomic broadcast and group membership. The crash detection mechanism leads the member to leave the group of replicas.

5.2. Primary Backup Replication

The Primary Backup Replication (PBR) is a passive replication strategy. The primary is the only active replica. It processes requests, and periodically checkpoints its state into a fault tolerant disk, shared among the replicas. The backups receive the requests and save it until the primary signals to the backups that the result has been sent to the client.

When the primary replica crashes, a backup is elected to be the new primary replica. The most recent checkpoint is rolled back. Then, the functional component is started. The replica behaves as the new primary.

The figure 5 shows the corresponding configuration.

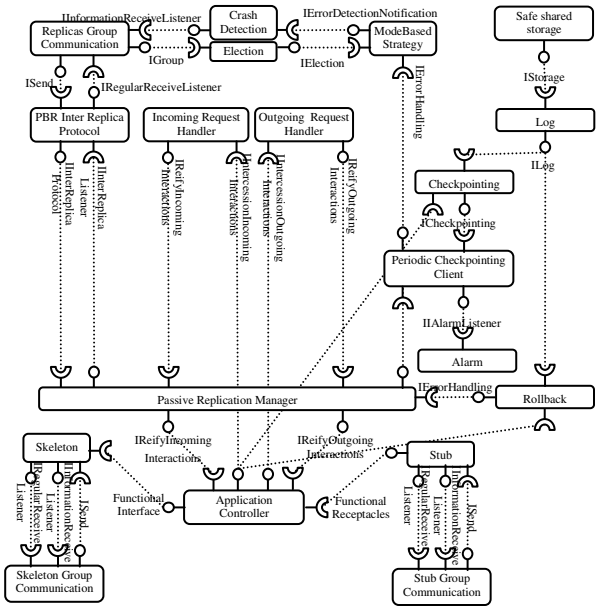

Figure 5. PBR replica component architecture

6. Implementation of adaptation

6.1. Modification of fault tolerance software

The comparison of the two component-based designs, leader follower replication and primary backup replication, shows that several components belong to both software configurations. However, some components differ between the leader follower replication and the primary backup replication, such as the inter-replica protocol or the semi-active replication manager which has to be replaced by the passive replication manager. In addition, new components in primary backup replication are needed to perform the checkpointing.

The representation of the component architectures depicted in the figure 4 and 5 is available on-line thanks to the component model, i.e. OpenCOM in our case. This representation is the architectural meta-model of the component based software. The component model enables to manipulate at runtime this model by creating, removing connecting and disconnecting components. The script shown in 6 modifies the leader-follower component architecture, to obtain the primary-backup component architecture.
In brief, components to be replaced are stopped; new components are created, inserted into the software configuration and finally started. The later implies taking care of the state of the computation in the small, i.e. the components itself, and in the large, i.e. the whole software configuration (see section 6.2).

```
IOpenCOM runtime = null;
// deletion of the useless components
// this action stops, disconnects and destroys the components
IUnknown comp_to_delete =
runtime.getComponentPIUnknown("Semi-Active Replication
Manager");
runtime.deleteInstance(comp_to_delete);
comp_to_delete = runtime.getComponentPIUnknown("LFR Inter
Replica Protocol");
runtime.deleteInstance(comp_to_delete);

// creation of the new components
IUnknown iUnkStorage =
runtime.createInstance("services.storage.SafeSharedStorage",
"Storage");
[…]
IUnknown iUnkAlarm =
runtime.createInstance("services.time.PeriodicAlarm",
"Alarm");

// we get some identification of existing components
// in order to connect them with the new inserted ones
IUnknown iUnkController =
runtime.getComponentPIUnknown("Application Controller");
[…]
IUnknown iUnkCommunication =
runtime.getComponentPIUnknown("Group Communication");

// connection of the newly inserted components
runtime.connect(iUnkAlarm, iUnkCkptClt,
"services.time.IAlarm");
[…]
runtime.connect(iUnkRepMng, iUnkStrategy,
"ft.recovery.IErrorHandling");

// start the new components
((ILifeCycle)iUnkRepMng.QueryInterface("opencom.ILifeCycle"))
.startup();
[…]
((ILifeCycle)iUnkAlarm.QueryInterface("opencom.ILifeCycle")).
startup();

// start the controller if primary
if (isPrimary)
  ((ILifeCycle)
iUnkController.QueryInterface("opencom.ILifeCycle")).startup(
);
```

Figure 6. Reconfiguration script

6.2. Synchronization of adaptation

In order to use this script for adaptation, it is necessary to deal with two problems: synchronisation of the replicas execution with respect to adaptation (i.e. the script execution) and state transfer.

The first issue implies putting the execution of the system in a state where modifications can be done consistently. Indeed, the fault tolerance software is composed of parallel executions of the replicas. These executions are synchronized to keep the consistency of the replicas. For being able to adapt the system, we have to tackle two major problems. On the one hand, the components to be changed are concurrently accessed by system tasks and the tasks devoted to adaptation (namely the execution of the script). When replacing a component, all receptacles connected to services are disconnected, leading to potential errors to current execution. It is thus important to prevent such situation. A sort of critical section must be set around the modified sub-architecture to ensure that there is no execution in progress within it when modifications are applied. On the other hand, replicas are maintained synchronized using the inter-replicas protocol. Indeed, the replicas state might be inconsistent and then lead to

errors in the fault tolerance software, i.e. bad state of the replicas after changing their mode of operation.

The second issue is to transfer the state of the system to the new configuration. In fact, the system state is composed of the functional state of the application and the state of the fault tolerance software. Depending of the replication strategy, the replicas can be or not in the same state regarding the processing of requests. For instance, with the LFR strategy, the replicas are supposed to be in the same state, assuming their determinism and the fact that they have processed the same ordered set of input messages. In a PBR replication strategy, the primary is the only replica that has an up to date state; the backup is waiting for a recovery signal to upload the state from stable storage. Regarding the fault tolerance software, the state of inserted components has to be initialised with a value that takes into account the past execution of the system. This value may depend on the state of the set of removed components. The mapping of the state of removed components to newly inserted ones is a complex issue we do not tackle in this paper. Solving the problem in general is the subject of our current research.

In our case study, the replication manager component is the only one that contains a state that depends of the past of the execution. This state is composed of saved requests, requests received but not yet processed, and replies for requests which have been processed but whose replies have not been acknowledged by the client. During adaptation, the new replication manager has to be initialised with the messages stored by the old replication manager.

As a proof of concepts of our proposed architecture and approach, we propose to use an ad-hoc solution to these two problems to perform the adaptation from LFR to PBR in our case study. We assume that no crash occurs during adaptation.

Ideally, the system must be in some particular execution state where modifications can be applied to the system without endangering consistency. Such a state is called a *Suitable Adaptation State* (SAS). In our case study, one SAS example is when the state of fault tolerance software is empty: there is no waiting message, and the replicas are synchronized. Such a state for the LFR strategy is the following:

- Each replica has processed the same number of cycles of the periodic task of the controller
- Each replica has processed the same number of requests from the *Console*
- There is no execution on the each replica

In this SAS, no state transfer is necessary to adapt from LFR to PBR. The adaptation process falls into four steps:

1. Trigger the adaptation, providing the signal to switch the configuration,
2. Put the system in the SAS, and maintain the system in it using locks
3. Modify of the system using the script,
4. Let the system leave the SAS by releasing the locks.

To reach the expected SAS, we publish the state of the replica (number of cycles and messages) and control its execution until all replicas have performed the same number of cycles, and processed the same number of input messages. This approach is sufficient to reach this non-optimal SAS.

6.3. Early experimental measurements

The system owns five characteristic time values (cf. figure 7): the task period (T_{period}), the fault tolerant task execution time (T_{task}) which includes functional execution time and fault tolerance execution time, the time taken to synchronize the execution before locking the system (T_{synch}), the adaptation script execution time (T_{script}), and finally the total time taken to react to the trigger (T_{adapt}). T_{adapt} is important for the decision part of the adaptation. It is the time needed for the new configuration to be operational.

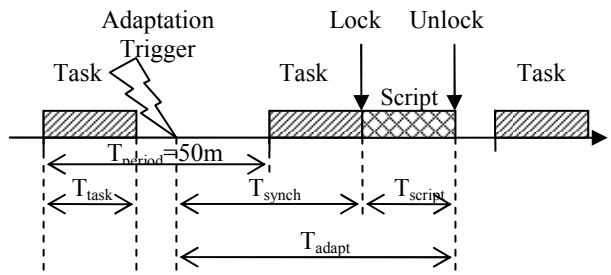

Figure 7. Time slices of execution

A correct adaptation involves being able to freeze the system without the fall of the pendulum. Because the system can naturally miss three deadlines, this constrains the script execution time to:

$$T_{script} < 4 \times T_{period} - T_{task}$$

The two configurations LFR and PBR with two replicas have been tested on four distributed Pentium 1,5Ghz under Linux kernel 2.6.18 connected by an Ethernet network. We have used the OpenCOM component model. We measure the round trip time of a request-reply seen by a client (the console) and by the wrapped functional component (the controller).

The obtained measures are compared with two other configurations, on providing locally connected functional components (Default) and the other providing distributed functional component. The results are provided in table 3.

Table 3. Timing measurements

	Default	Distributed	PBR	LFR
Target position request processing time average	0,046	1,875	2,102	1,988
Controller-system round trip time average	0,051	2,040	2,312	2,513

We observe that distribution increases the processing time (nearly 2ms in this example). On the contrary, fault tolerance strategies make this value increasing of less than 0.5ms. As far as the functional component is concerned, the periodic task does not take more than T_{task}=6ms to be achieved with the fault tolerance strategies attached.

Then, the free time for executing the script is 194ms. In practice, the script execution time is less than 100ms. Because the time constraints were fulfilled during the experiments, the pendulum did not fall.

7. Conclusion

Fault tolerance adaptation is a two side problem. First, on-line assessment has to be performed in order to decide of adaptation. Then, the fault tolerance software has to be modified in order to correspond to the wanted configuration. This paper focuses on the second aspect. It shows how fault tolerant systems can be designed to enable the modification of fault tolerance software at-runtime.

We proposed a reflective architecture with three levels, which separate the functionalities, the fault tolerance and the adaptation issues that enable fault tolerance and adaptation to be addressed separately. The component based design of fault tolerance software offers opportunities to be adapted at runtime, using the architectural representation provided by the component model.

However component models are not sufficient to solve the adaptation of the fault tolerance software. It is necessary to deal with two other issues that are the management of execution during adaptation and the transfer of the state from old to new configurations in order to maintain the replication consistency. We demonstrate that these two problems can be solved using ad-hoc solutions. The proposed solution consists in freezing each synchronized replica in a predefined Suitable Adaptation State (SAS). This type of ad-hoc solution works fine on a case-by-case basis.

A significant improvement of fine grain adaptation would be to identify possible SAS automatically. The aim is to minimize the part of the execution that has to be frozen during adaptation. This is possible using a generic model of the behaviour of the software at runtime, which is the focus of our current work.

8. References

[1] P. Maes, "Concepts and experiments in computational reflection", Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, Orlando, Florida, October 4-8, 1987.

[2] G. Kiczales, J. Lamping, et al., "Aspect-Oriented Programming", European Conference on Object-Oriented Programming, Springer-Verlag, Jyväskylä, Finland, June 9 - 13, 1997.

[3] M.-O. Killijian, J.-C. Fabre, et al., "A Metaobject Protocol For Fault-Tolerant CORBA Applications", 17th IEEE Symposium on Reliable Distributed Systems, IEEE Computer Society, West Lafayette, Indiana, USA, October 20-23, 1998.

[4] F. Taïani and J.-C. Fabre, "A multi-level meta-object protocol for fault-tolerance in complex architectures", International Conference on Dependable Systems and Networks, IEEE Computer Society, Yokohama, Japan, June 28 - July 1, 2005.

[5] J. S. Plank, M. Beck, et al., "Libckpt: Transparent Checkpointing under UNIX", USENIX Technical Conference, USENIX, New Orleans, Louisiana, USA, January 16-20, 1995.

[6] S. K. Shrivastava, G. N. Dixon, et al., "An Overview of the Arjuna Distributed Programming System", IEEE Software, vol. 8, pp. 66-73, 1991.

[7] M.-O. Killijian, J. C. Ruiz-Garcia, et al., "Portable Serialization of CORBA Objects: a Reflective Approach", Conference on Object-Oriented Programming Systems, Languages and Applications, ACM Press, Seattle, Washington, USA. SIGPLAN Notices 37(11), November 2002, 2002.

[8] G. Kiczales, E. Hilsdale, et al., "An Overview of AspectJ", European Conference on ObjectOriented Programming, Springer-Verlag, London, UK, 2001.

[9] G. Coulson, P. Grace, et al., "Towards a Component-based Middleware Architecture for Flexible and Reconfigurable Grid Computing", International Workshops on Enabling Technologies, Infrastructure for Collaborative Enterprises, IEEE Computer Society, Modena, Italy, Junes 14-16, 2004.

[10] N. T. Bhatti, M. A. Hiltunen, et al., "Coyote: A System for Constructing Fine-Grain Configurable Communication Services", ACM Transactions on Computer Systems, vol. 16, pp. 321-366, 1998.

[11] S. Lin, F. Taïani, et al., "Facilitating Gossip Programming with the GossipKit Framework", Distributed Applications and Interoperable Systems, Springer Berlin / Heidelberg, 2008, vol. 5053, pp. 238-252.

[12] A. Avizienis, J.-C. Laprie, et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Trans. Dependable Secure Computing, vol. 1, pp. 11--33, 2004.