Towards Adaptive Fault Tolerance in Embedded Systems

Thomas PAREAUD[1], Jean-Charles FABRE[1], Marc-Olivier KILLIJIAN[1]

1: LAAS-CNRS, 7 av. du Colonel Roche, 31077 Toulouse Cedex 4, France

Abstract: Dynamic systems become more and more widespread in many application fields. This observation highlights the trend that systems are no more considered as running in a static, predefined environment. Evolution has to be taken into account and fault tolerance does not make an exception. When the executive or environmental context has changed, hypothesis or fault model may become outdated or invalid. On-line adaptation of fault tolerance has to be tackled.
This paper deals with the design of fault tolerance for its on-line adaptation. It describes a reflective architecture, suitable for modifying fault tolerance at runtime. Then it shows how fault tolerance may be componentized into small components to enable its runtime modification.

Keywords: Fault tolerance adaptation, Reflective architecture, Component Models

1. Introduction

Many systems, critical or not, may see their environment changed during service delivery. On-line fault tolerance adaptation becomes necessary for many systems. Thus, it may concern either embedded critical systems, ubiquitous or mobile systems. On-line adaptation of fault tolerance mechanisms depends on two different factors. The first one is the evolution of fault hypothesis: several fault models have to be taken into account according to current context of execution (context-aware fault tolerant computing). The second one is the variation of available resources (resource-aware fault tolerance computing).

From the architectural point of view, a classic approach to implement fault tolerance consists in designing a dedicated middleware which provides the fault tolerance mechanisms, such as the Delta-4 system [1]. The goal of our work is to provide a fault tolerance middleware with adaptive capabilities that enables to modify fault tolerance at runtime to fit executive constrains (resources, context). To this aim, our approach relies on reflective technologies and open components models.

In order to adapt fault tolerance at runtime, we have to deal with several issues. The first one concerns the runtime supervision of operational context of the system which triggers the adaptation. The second one concerns the design of fault tolerance in order to be able to modify it at runtime. The last one deals with the executive constraints when modifications are applied to the system.

In this paper, we focus on designing fault tolerance for its runtime adaptation. We tackle this problem in two ways. First, we propose a reflective multi-layered architecture based on component technology that separates the different issues of adaptive fault tolerant system (separation of concerns). Secondly, we provide guidelines for the decomposition of fault tolerance strategies into components to facilitate their modification at runtime. These guidelines are illustrated by two well-known strategies that have been implemented.

This paper is organized as follow. Section 2 describes the problem statement and introduces our approach. Section 3 shows how to use components technology to provide reflective mechanisms for the reflective layer in charge of fault tolerance. The guidelines for fault tolerance decomposition are proposed in section 4, and examples of fault tolerance components in section 5. Then, a case of study is described section 6. Our conclusions and perspective of this work are given in section 7.

2. Problem statement and approaches

2.1. Problem statement and motivations

The adaptation of a fault tolerance strategy attached to an application means that, under some new operational conditions, the current fault tolerant strategy is to be modified. We assume that some on-line assessment and/or runtime monitoring are able to state when operational conditions change and trigger the adaptation. On-line assessment is out of the scope of this paper.

We consider that adaptation leads to two kinds of modifications. The first one is the modification of fault tolerance algorithms. The second one is the modification of parameters of the fault tolerance algorithms.

Let S1 and S2 be two fault tolerance strategies. Our goal is to modify algorithms and parameters to turn current strategy S1 into strategy S2, taking into account the functional specifications of the system and fault tolerance properties during adaptation. Whatever the solution is, clearly the application software must be separated from the fault tolerance software. This is a first objective of the proposed approach (*objective 1*).

Now, to perform this change the first option is to stop S1 as a whole and restart S2 from some past history of S1. This implies mastering in deep the state and

the behaviour of the system. This is not easy if the strategy is seen as a single software component.

Instead, we propose a solution that takes advantage of Component Based Software Engineering techniques. The idea is to design the strategies as a set of cooperating fine-grain software components. A set of small software components can be dynamically attached together to form a strategy. The adaptation can thus rely on the ability to attach and detach components (*objective 2*).

The on-line adaptation itself can be seen as a software runtime entity responsible for changing the configuration of the fault tolerance software, taking care of synchronisation issues (w.r.t to the behavioural aspect) and of state issues (history of the previous strategy). The solution must exhibit the adaptation as independent from the fault-tolerance software, so that it makes easier the definition and the implementation of an adaptation strategy (*objective 3*).

We expect the proposed approach to have the following benefits:

- it enables mastering behavioural and state issues more easily since the connection between fine-grain software components can be controlled at runtime;
- it saves memory space since many of these fine-grain components can be shared between several strategies ;
- the system performance can be increased as only few light-weight components need to be changed to switch to a new strategy
- for a given set of fine-grain components, several strategies can be defined and modified at runtime by the adaptation software.

In order to fulfil these objectives, the approach is based on separation of concerns architectural frameworks and open component models.

2.2. Reflective architecture

Reflection and more recently aspect oriented programming, have shown their efficiency to separate different crosscutting concerns, such as fault tolerance for instance. A software system can thus be conceived as multiple layers. These layers are called meta-level in contrast with the base layer that they control. A meta-model of the base layer is provided to the meta-level. Control is realized through the use of reflective mechanisms (reification, introspection and intercession).

We propose to use a reflective multi-layered architecture (cf. Figure 1) in order to separate, on the one hand, the processing of the fault tolerance algorithms from functional ones, and on the other hand, the adaptation mechanisms from the rest of the system.

The functional part of the system corresponds to the base layer. Then, the fault tolerance software is located in the first meta-level. Finally, fault tolerance adaptation is handled by the second meta-level.

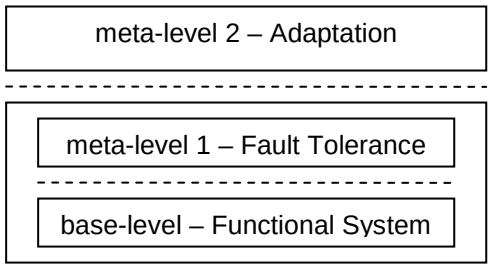


Figure 1 : Reflective architecture

Although adaptation focuses on fault tolerance, it also needs to control the functional layer of the system. For example, adding a replica in a replication-based strategy involves, among other things, base level state manipulation. Thus, the second meta-level is a reflective layer of the whole fault tolerant system composed of the two lower layers.

2.3. Componentized layers

Separation of fault tolerance and functional algorithms is a first step to tackle fault tolerance adaptation, but it is not enough. We need mechanisms to manipulate algorithms. Component models seem to be a good way to deal with algorithms manipulation. We propose to use component models to implement both the base level and the fault tolerance layer.

Indeed, a component is an abstraction of algorithms and its interface describes the service provided by the component. The content is the set of algorithms that implement the service specified through the interface. The notion of receptacle represents the dependencies with other algorithms, in fact the explicit link with other components needed to realize the service. By definition, a component model provides reflective mechanisms to load, unload, connect and disconnect components. These mechanisms are a foundation for the building of the meta-model to tackle adaptation issues.

Because we need to decompose fault tolerance software into several small algorithms, the selected component model must provide means to define and manage small components. In our opinion, component models that best fit these constraints are Fractal [2] and OpenCOM [3]. In this work, we use OpenCOM essentially because it is more flexible from an implementation viewpoint. The same work may be done using Fractal as well.

The corner stone of any reflective architecture is the notion of meta-model. The meta-model captures the essential behavioural and state elements of the base level to perform some actions at the meta-level. In

other words, the meta-model is an input of the meta-level software to enable control over the base-level software.

In order to be able to use such reflective mechanisms for adaptation, the meta-model of base level for fault tolerance has to be implemented using components technology and fault tolerance has to be componentized to enable adaptation layer to manipulate fault tolerance algorithms at runtime.

The next section shows how the meta-model for fault tolerance can be realized with components. Section 4 provides guidelines for the fault tolerance componentization.

3. Meta-model of base level for fault tolerance

Several previous studies [4, 5] have shown the benefits of reflective mechanisms for implementing fault tolerance. We propose to use the same mechanisms to perform:

- State handling through intercession and introspection mechanisms;
- Behavioural control through:
  - Reification of incoming and outgoing calls;
  - Intercession of incoming and outgoing calls.

Introspection is an observation of the base level on-demand by the meta-level. Reification is an action carried out at the base level to spontaneously inform the meta-level of some event. Intercession consists in modifying the base level software either by invoking a method (behavioural intercession), or by modifying a part of the internal state (structural intercession).

3.1. State handling

State acquisition and restoration are necessary for checkpointing-based mechanisms used in many fault tolerance strategies like Primary Backup Replication or Recovery Blocks [6] for instance. Saving and restoring process' state have been addressed in many works [7-9]. It is worth noting that such mechanisms can lead to major troubles when they are not designed and implemented correctly.

Indeed, the state of a component can be a very complex notion. It is composed of local process data, process execution information, and state of operating system resources like opened files or sockets in use, etc. The component state is thus partially dependent on lower layers of the system such as the kernel and the middleware on top of which the application components are running.

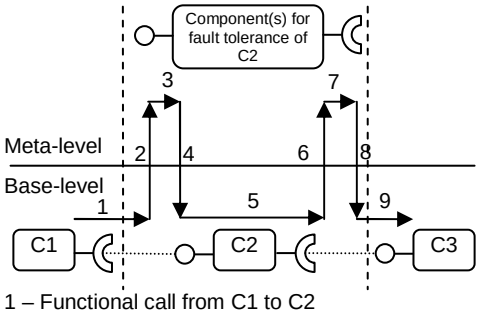
The acquisition and the restoration of the state can be realized by a third party [10] or implemented using traditional object-oriented facilities, like inheritance of abstract classes, overloaded by a programmer [11]. In our implementation we have used the second solution which suits component technology better. Every functional component has an interface named *IState* which defines the save and restoration mechanisms. The developer of the functional components has to implement this interface. By the way the management of state issues is delegated to the component developer. To do this job, the reflective capabilities of a language (AspectJ [12] or the java.lang.reflect library in Java) can be of great help, other mechanisms provided by the underlying platform as well.

3.2. Behaviour control

As stated earlier, we need the component behaviour to be observable and controllable through incoming and outgoing calls. Incoming calls are the ones that the component user makes by calling the interface of the component. Outgoing calls are the ones that the component makes through one of its receptacles to a service on which it relies.

Control mechanisms like reification of calls enable the insertion of fault-tolerance software both before and after functional processing. The reification mechanism routes the initial functional call to meta-level as shown in Figure 2. Then some fault tolerance related actions can be performed before delegating the real call to the base level. The functional component is invoked and returns the result to the meta-level.

Figure 2 does not illustrate the return execution path. Nevertheless, fault tolerance related actions should be realized before returning the result to the base level functional call.



1 – Functional call from C1 to C2
2 – Interface call reification to meta-level
3 – Fault tolerance treatments before calling
4 – Intercession of C2 by doing incoming call
5 – Functional treatment of C2, then call to C3
6 – Receptacle call reification to meta-level
7 – Fault tolerance treatments before calling
8 – Intercession of C2 by doing outgoing call
9 – C3 execution relative to outgoing call from C2

Figure 2 : Behaviour capture

3.3. Implementation of meta-model

In practice, the meta-model defined in the previous section must correspond to a real component able to provide control facilities for a given functional

component. We propose to implement this meta-model as a wrapper of the functional component (a composite component in OpenCOM). This wrapper (cf. Figure 3) owns two interfaces that provide intercession mechanisms and two receptacles for reification mechanisms. It is named *ApplicationController* because it provides control facilities over the application composed of functional components.

The *ApplicationController* exposes both functional interfaces and receptacles. This wrapper can been implemented with the OpenCOM component model using proxy components able to intercept incoming and outgoing calls and route them through reification receptacles to the connected component at the meta-level.

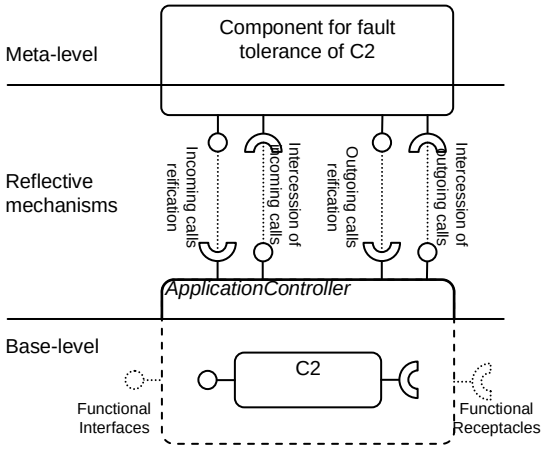


Figure 3 Application Controller

3.4. Framework for distribution

Most of fault tolerance strategies which aim at tolerating hardware faults rely on replication: a functional component is replicated on several nodes of a distributed system.

Clearly, distribution is part of the fault tolerance meta-model since it is a way to control interaction. We thus introduce distribution in the component model to manage the replication of components. To this aim, specific composite components have been introduced. These composite components provide an abstraction of distribution for implementing replication. Two types of composite components are required:

- *SingleReplica* grouping functional and fault tolerance components but corresponding to a unique replica.
- *ReplicasGroup* grouping all replicas of a replication strategy that can be called altogether.

The distribution framework is illustrated in Figure 4. It enables considering the group of replicas as a unique component that behaves as a reliable functional component. Stub and skeletons in the picture act as in any distributed model, but hide remote interaction among replica groups.

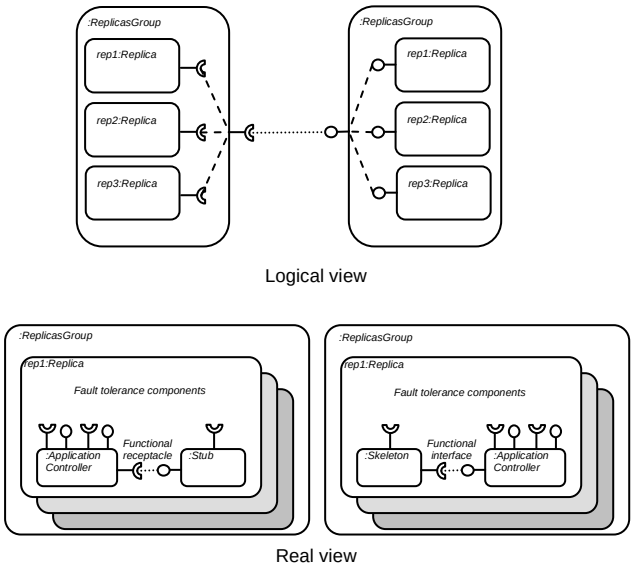


Figure 4 : Distribution framework

This design is quite interesting for adaptation because it provides opportunities to address the functional synchronisation of replicas for fault tolerance adaptation at the *ReplicasGroup* level. It can be used to synchronise the state of all replicas. In addition, it can be more efficient to adapt the fault tolerance mechanisms attached to a functional component when the later is in a particular state, e.g. no request in progress.

4. Fault tolerance componentization

4.1. Problem analysis

Our goal is to guide the componentization process of fault tolerance software in order to facilitate runtime adaptation.

During runtime adaptation, two kinds of changes are applied to the system. The first one focuses on parameters of the system, and the second one focuses on algorithms.

When component models are used to develop algorithms, the smaller piece of algorithm that can be changed is a component. This component's boundaries are the interfaces it implements and the other interfaces it depends on.

Thus, interface definition is a very important part of the componentization process of fault tolerance for its online adaptation. These interfaces reflect on the one hand, algorithms that may be dynamically changed, and on the other hand, algorithms that should never be modified because they provide generic services.

Moreover, adaptation implies being able to:

- control execution of components in order to realize system adaptation when they are in a suitable state;
- transfer data from previous existing components to newly inserted components. This transfer needs the introduction of translation function which converts the state from previous fault tolerance components to new ones.

The same service can be provided by a set of fine grain cooperative components or by a single coarse one, depending on the choices of decomposition.

De facto, dealing with several small components may increase the complexity of execution control whereas using a unique big component will make the state transfer functions be more complex. A trade-off has to be done during componentization process.

4.2. Fault tolerance persistent state

In a component based fault tolerance implementation, the state of the fault tolerance mechanisms is spread over every components involved in the fault tolerance strategy. This concerns both fault tolerance components and the wrapped functional components.

When an adaptation occurs, the state of the current strategy has to be transferred to the new one. A part of this state is persistent during adaptation, that is to say, it remains identical before and after adaptation. To optimize the state transfer, persistence of this state has to be analyzed.

This state is composed of values of internal variables, stored data, communication channels with transient messages, or timers. For instance, in primary-backup and leader-follower replication, group communication is generally used. Changing system configuration from the primary-backup strategy to the second one implies to transfer possible transient messages. This process may be quite complex. By using generic services like group communication services, it is thus possible to simplify the state transfer function.

In our componentization process, we identify, in a first step, the generic services on which fault tolerance relies. Thereby, when possible, these services will be persistent, as well as their own state.

4.3. Identified services and components

In this section, we first propose several services that we identified as generic services. These services are persistent and thus, store state information that may remain valid for several fault tolerance strategies. Their interfaces are not detailed in this paper, because it focuses on the method of decomposition of fault tolerance software.

Generic services

We identified five groups of generic services which can be useful in a fault tolerant system: communication service (messages sending/receiving and group management), election services, storage services (distributed, local, stable or unstable), clock services and timer services.

These services may correspond to several components providing different properties. For example, a clock service may be implemented by either a local clock, a global clock or a logical clock. Communication services can be implemented by a peer-to-peer communication component, or a reliable group multicast component.

Moreover, depending on the underlying hardware, a reliable distributed storage can be implemented using a reliable data bus to a shared disk or using the reliable group multicast component and a set of local non-reliable disks.

Fault tolerance services

Our componentization process of fault tolerance relies on the taxonomy of dependability [13].

Fault tolerance consists in using core mechanisms to build a fault tolerance strategy. This strategy is based on a fault model for the considered system.

Fault tolerance mechanisms fall into three categories:

- Error detection mechanisms, which detect when the service provided by a component in the system deviates from the specified correct service. Each component of this category provides the same service which is an error notification service. A detection component has a receptacle connected to a component which provides the notification of some detection;
- Error recovery mechanisms, which eliminate errors from the system. Each component of this category provides the same service, which is an error recovery service;
- Fault recovery mechanisms, which prevents fault from being reactivated again. They can be divided into four services which are:
  - Diagnosis, which identifies the possible origins of the detected errors in terms of both location and type;
  - Isolation, which performs physical and logical exclusion of the faulty components of the system from further participation in service delivery;
  - Reconfiguration, which either switches to spare components or reassigns tasks among non-failed components;
  - Reinitialization, which checks, updates and records the new configuration and updates systems tables and records.

The recovery mechanisms consist in reconfiguring functional components as a response to error detection. However, in this work, reconfiguration refers to modification of the fault tolerance software (algorithms and parameters) induced by contextual adaptation reasons.

A strategy consists in using these different services to react to error detection. Thus, a strategy has a core component, to which errors are notified. Then this core component recovers the detected errors and handles the faults of the system by calling the services provided by recovery mechanisms. The core component necessarily implements the error notification interface to be notified of error detection. Moreover, it has receptacles connected to recovery mechanisms on which it depends.

Some more generic fault tolerance services have been defined. They are:

- Checkpointing that consists in saving functional state into a storage;
- Logging that keeps a trace of execution into a log;
- Inter Replica Protocols, which is a synchronisation protocol between the replicas.

5. Components examples

5.1. Generic services

Group communication

We have implemented a multi-threaded communication group component based on Spread [14]. This component provides asynchronous message transfer services based on a publish/subscribe approach. This component has two receptacles which are connected to *Info-MessagesListener* and *RegularMessagesListener* interfaces. They realize spontaneous message delivery to connected components. These messages are of two kinds: regular messages are messages sent by third party and information messages are group information messages which signal join and leave operations.

The Spread communication library provides a multicast protocol which insures messages delivery and order. This component implements a *Group* interface which enables to know the current user identifier, the groups it belongs to and members of each of these groups. The members are totally ordered and consistent into a group.

Another non-threaded communication group component has been realized. It implements blocking *RegularMessagesReceive* and *Info-MessagesReceive* interfaces instead of listener receptacles.

Election

The election component has been realized using the *Group* interface of the communication component. It implements the *Election* interface. Then, the replica is elected if it is the first in the totally ordered list of the replicas group.

Storage

Several storage components have been realized. All of these components implement the *Storage* interface. The first one is a local unsafe storage, which does not insure write atomicity. The second one is a local atomic storage, which insures atomicity on write. The third one is a distributed safe storage. This storage service relies on a group communication service and on a local atomic storage. It insures that data stored on each node of the system is the same and that it was written atomically.

5.2. Fault tolerance generic services

Log

The log component implements the *Log* interface. It uses a storage service to create log pages that can be accessed through its interface. The reliability of the log service depends on the type of storage used.

Some client components can be designed. For instance, there could be client components which log incoming or outgoing requests and returned responses.

Checkpointing

The checkpointing component implements *Checkpointing* interface. It gets the state of functional component and logs it using the log service.

We have considered two client components for the checkpointing service. The first one requests checkpointing for each incoming call. The second requests periodic checkpointing (using a timer).

Inter replicas protocol

The inter replica protocols use the group communication to provide synchronization among replicas. This component implements the *InterReplicaProtocol* interface and a receptacle where *InterReplicaProcolEventListener* interfaces can be connected. This component synchronises a given event on replicas. Components to be synchronised implements the *InterReplicaProtocol-EventListener* interface and are connected to the receptacle. A synchronised event is then signalled to the listening components by calling the method corresponding to the event.

5.3. Error detection

Crash detection

We implement a node crash detection component using the *InfoMessagesListener* interface of communication component, assuming that a crash always makes the node to leave the replicas group. This component has a receptacle which is connected to an error notification service. When a crash of a node is detected, the error is notified by this component to the one connected to this receptacle.

5.4. Error handling

Rollback

Rollback consists in putting the functional component in an error-free state obtained before failure. Our rollback component reads previously logged requests, responses and checkpoints. Then, it restores the state of functional component from the checkpoint. At last, it replays incoming requests, and replaces responses of outgoing request by the logged ones.

This component is an error handler. So it implements the *ErrorRecovery* interface which is called to recover when an error has been notified and has to be recovered.

Compensation

Compensation is an error handling mechanism. It consists in using execution redundancy to mask errors. It may be used in several strategies, like leader-follower replication or triple modular replication for instance. Moreover, this compensation may be realized at the replica side or at the client side.

In the case of leader-follower compensation mechanism at a replica side, the results of incoming requests are returned only by the leader replica, the outgoing requests are only sent by leader replica, and the result is then provided to all replicas.

6. Case study

6.1. Functional level description

We now propose a case study (Figure 5) that is composed of four functional components which are sensor, actuator, automatic controller and the command console. The command console captures a reference (for instance from a keyboard) and applies it to the controller. The controller tries to make the system to behave as the reference specifies it. It periodically reads measures from the physical system using the sensor and calculates control outputs to be applied to the system through the actuator.

We aim at making the controller reliable by inserting fault tolerance strategies at its meta-level. The fault model considered here is the crash fault model of nodes. We want controller to tolerate '*n*' crashes. Then, functional software is replicated among '*n+1*' different nodes. We propose to study two classic strategies which are leader-follower replication strategy and primary backup replication strategy.

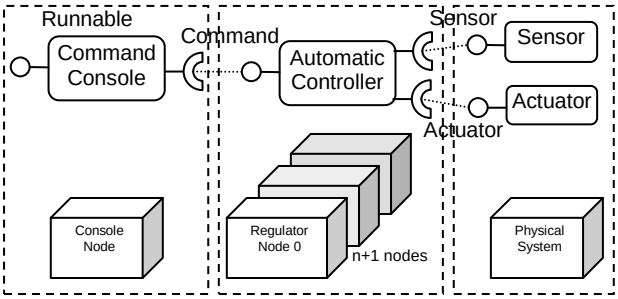


Figure 5 : Case study

6.2. Leader Follower Replication strategy

The leader follower replication strategy (LFR) consists in replicating execution on more than two replicas. One of these replicas is the leader. The others are the followers. All of them are active in the sense that they execute requests.

The leader provides the answers for incoming requests, and is the only one sending outgoing requests to other components. The responses of outgoing requests are then provided to the followers.

When the leader fails, one of the followers is elected as the new leader replica.

Thus, this strategy is based on crash detection, election and compensation. Compensation uses the inter replica protocol service, which depends on the group communication service. We focus on error recovery and do not consider fault handling. The componentized strategy is presented Figure 6.

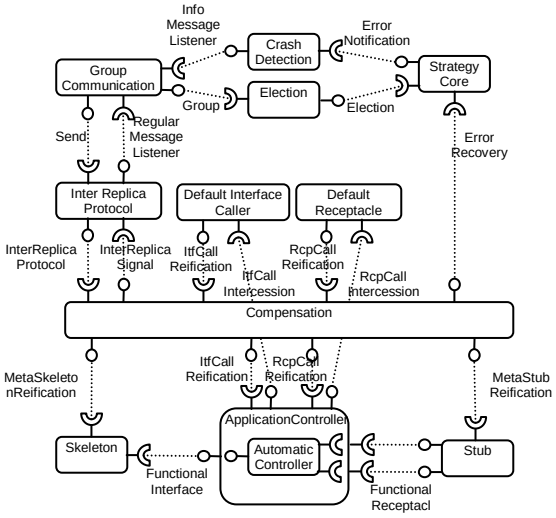


Figure 6 : A replica in LFR strategy

6.3. Primary Backup Replication strategy

In the primary backup replication strategy (PBR), a replica is the primary and the others are the backups. The primary is active and processes incoming requests. Others are passive. The state of the primary replica is saved into a stable shared storage. On primary failure, a backup is elected as new primary. Its state is then restored from checkpoints stored before the primary failed.

There are many ways to implement this strategy. In the work reported in this paper, we choose to save the state when an incoming call occurs on a stable storage, shared among the alive replicas. Then, the incoming call is logged. Outgoing calls do not have to be logged in this example because the controller needs the most recent measures of sensors and applies directly computed outputs to the system. The components architecture of a replica is depicted in Figure 7.

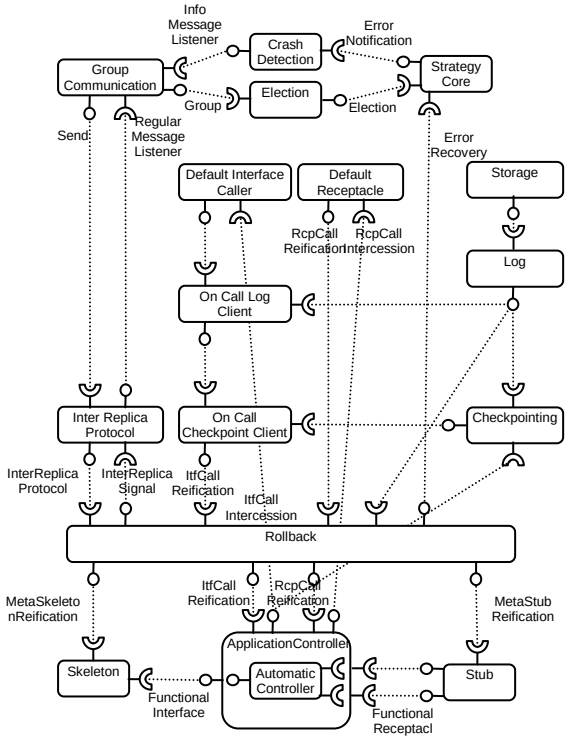


Figure 7 : A replica in PBR strategy

7. Conclusion

Several operational constraints involve mastering dependable system on-line evolutions. These evolutions, induced by resource variations, changes of model or evidences of non-respect of hypothesis, are currently discussed in the ResIST[1] network of excellence. Fault tolerance mechanisms are naturally spotlighted. This evolution is of interest in embedded systems with constrained resources but also in systems with a natural environment variability.

In this paper we have dealt with a key issue of dynamic fault tolerance adaptation: design for adaptation. Firstly, we have proposed a reflective architecture which separates functionalities of the system, fault tolerance mechanisms and algorithms responsible for on-line adaptation. Secondly, we have proposed a componentization method of fault tolerance to enable its modification at runtime.

We have built some foundations of the adaptive fault tolerance middleware.

Our current work deals with the two other issues of adaptation: the first one is the assessment leading to the modification of fault tolerance mechanisms, and the second one is the runtime meta-model of the system to apply modification during its service delivery. In other words, these two issues are the content of the meta-layer dedicated to the adaptation processing at runtime.

Based on the componentization of fault tolerance, our approach to dynamically adapt fault tolerance addresses two questions: “when” and “how”.

The “when” refers to the execution state of the fault tolerance software where the modification on the components is valid. This means that the modification does not make inconsistent neither the fault tolerance software processing nor the functional one.

The “how” refers to the modifications to perform. It firstly concerns architectural changes of the system which has to insure architectural dependencies. Secondly, the issue of knowledge (past activity) transfer from the old to the new fault tolerance software version has to be taken into account. This state transfer must insure the data value consistency among components after the system adaptation.

[1] Resilience for Survivability in IST

8. References

[1] D. Powell: "*Delta-4: A Generic Architecture for Dependable Distributed Computing*", vol. 1: SpringerVerlag, 1991.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani: "*The Fractal Component Model and Its Support in Java*", Software Practice and Experience, vol. 36 (11-12)(Experiences with Auto-adaptive and Reconfigurable Systems), 2006.

[3] G. Coulson, P. Grace, G. S. Blair, L. Mathy, D. Duce, C. Cooper, W. K. Yeung, and W. Cai: "*Towards a Component-based Middleware Architecture for Flexible and Reconfigurable Grid Computing*", Workshop on Emerging Technologies for Next generation Grid (ETNGRID-2004), 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, Italy, 2004.

[4] M.-O. Killijian, J.-C. Fabre, J. C. Ruiz-Garcia, and S. Chiba: "*A Metaobject Protocol For Fault-Tolerant CORBA Applications*", 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98), West Lafayette, Indiana, USA, 1998.

[5] F. Taïani and J.-C. Fabre: "*A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures*," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 270-279.

[6] B. Randell: "*System structure for software fault tolerance*", IEEE Transactions on Software Engineering, vol. 1(2), 1975.

[7] R. Koo and S. Toueg: "*Checkpointing and rollback-recovery for distributed systems*", IEEE Transaction on Software Engineering, vol. 13, 1987.

[8] K. M. Chandy and L. Lamport: "*Distributed snapshots: determining global states of distributed systems*", ACM Transaction on Computer Systems, vol. 3, 1985.

[9] D. B. Johnson and W. Zwaenepoel: "*Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing*", Journal of Algorithms, vol. 11(3), 1990.

[10] J. S. Plank, M. Beck, G. Kingsley, and K. Li: "*Libckpt: Transparent Checkpointing under UNIX*", USENIX Winter, 1995.

[11] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington: "*An Overview of the Arjuna Distributed Programming System*", IEEE Software, vol. 8(1), 1991.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold: "*An Overview of AspectJ*", Proceedings of the 15th European Conference on ObjectOriented Programming (Ecoop'01), London, UK, 2001.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr: "*Basic Concepts and Taxonomy of Dependable and Secure Computing*", IEEE Transactions on Dependable and Secure Computing, vol. 1(1), 2004.

[14] Y. Amir, C. Danilov, and J. R. Stanton: "*A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication*", 2000 International Conference on Dependable Systems and Networks, New York, NY, USA, 2000.

9. Glossary

LFR: Leader Follower Replication

PBR: Primary Backup Replication