From Design for Adaptation to Component-Based Resilient Computing

Miruna Stoicescu, Jean-Charles Fabre, Matthieu Roy

From Design for Adaptation to Component-Based Resilient Computing†

Miruna Stoicescu[1,2], Jean-Charles Fabre[1,3], Matthieu Roy[1,2]
[1] *CNRS, LAAS, 7 Avenue du colonel Roche, F-31400 Toulouse, France*
[2] *Univ de Toulouse, LAAS, F-31400 Toulouse, France*
[3] *Univ de Toulouse, INP, LAAS, F-31400, Toulouse, France*

***Abstract*—The evolution of systems during their operational lifetime is becoming ineluctable. Dependable systems, which continuously deliver trustworthy services, must evolve in order to comply with changes having different origins, e.g. new fault tolerance requirements, or changes in available resources. These evolutions must not violate their dependability properties, which leads to the notion of resilient computing. This paper presents a methodology for developing adaptive fault tolerance mechanisms, from the design to the actual runtime reconfiguration, leveraging component-based middleware which enable fine-grained manipulation of software architectures.**

***Keywords*-resilience, adaptation, software architecture**

I. Introduction

Nowadays, systems are becoming increasingly complex, their development most often being an iterative process. The capacity to easily evolve in order to efficiently cope with change is a requirement of utmost importance. This issue is even more complex in the case of critical systems, which cannot be stopped for a long period of time and which must continuously provide trustworthy services. Their properties must not be violated by changes occurring in the environment or in the system itself and evolution must be performed seamlessly with service delivery. This leads to the notion of resilient computing [1].

Our focus is on fault-tolerant systems, designed according to the principle of *separation of concerns*, which facilitates the introduction of fault tolerance mechanisms in an application in a flexible way for subsequent modification and reuse. According to this principle, software architectures consist of two abstraction levels where the base level provides the required functionalities and the top level contains the fault tolerance mechanism(s) [2]. This model is based on the idea of behavioural reflection [3].

The choice of an appropriate Fault-Tolerance Mechanism (FTM) is based on several criteria: fault model, application assumptions such as determinism and state accessibility, available resources. Evolution in terms of these dimensions can require changing the FTM in operation to maintain dependability properties. In this paper, we first discuss the problem statement in more detail in Section II. Section III presents our "design for adaptation" approach for developing a toolbox of FTMs. Section IV describes the mapping of FTMs to components, Section V explains how runtime adaptation of FTMs can be implemented and Section VI presents further benefits of component-based architectures. In Section VII we present related work before concluding in Section VIII.

†This work has been supported by the French National Research Agency ANR, contract nr. ANR-BLAN-SIMI10-LS-100618-6-01, and by the INRIA collaboration program Serus

II. Problem statement and overall approach

A. Problem Statement

The evolution of systems concerns several aspects, covering system configuration, environmental changes, application changes due to bug correction or evolution in the functional specifications. This is particularly the case for autonomic computing systems, mobile systems, or sensor network systems. As far as dependability is concerned, the evolution of non-functional mechanisms is also an issue. Their evolution may be due to environmental changes, runtime support changes, non-functional specification changes in the lifetime of the system. This is the point we address in this work.

The evolution of a dependable system relates first to the required dependability properties and, in particular, its fault tolerance capabilities (*FT*), the available resources (*R*) and the application assumptions at a given point in time (*A*). *FT* encompasses one or several FTMs that are obviously strongly related to the fault model and the required dependability properties (reliability, availability, integrity, confidentiality). To run the selected mechanisms, a set of resources *R* is required and should be available. Last but not least, the characteristics of the application *A* have an impact on the validity of the selected FTMs. An application can be zero-default from a software development viewpoint, have an accessible state or not, be deterministic or not, etc. A lack of resources (e.g., network bandwidth), a change in the application assumption due to versioning (e.g., non deterministic version), an evolution of the non-functional requirements (e.g., transient faults to be tolerated due to hardware aging or environmental modifications) implies a change of the running FTM. At any point in time, (*A*,*R*,*FT*) must be valid in order to fulfill dependability properties.

To this aim, we investigate in this work the use of CBSE technologies and companion middleware support for

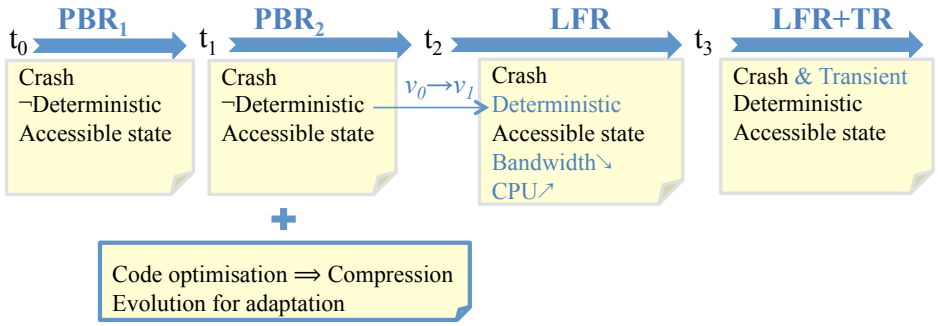

Figure 1. Transitions between FTMs - a scenario

two reasons: 1) the fine-grained development of software packages using autonomous components following the SCA paradigm, and 2) the capabilities to perform modifications of the software architecture at runtime. This technology is used to develop FTMs that can be easily updated.

B. Overall Approach

Our overall approach is composed of several steps, starting from the design of FTMs to simplify the implementation of changes at runtime.

- The very first step is of utmost importance: the design of FTMs must be done with adaptation in mind, i.e. they must be finely-grained decomposed, easily specialisable and also composable with each other (e.g. it should be easy to compose a duplex strategy tolerating crash faults with a mechanism tolerating transient faults).
- The detailed design of the mechanisms must then be mapped to a component-based middleware, in such a way that resulting components are small enough to hold a simple state that can be easily transferred to a new component.
- Development of transition algorithms to transform a running component-based implementation of an FTM into a variant, this implying removal of components, insertion of components, mastering component life-cycle at runtime and bindings.
- Monitoring both system state and distributed properties are important issues that are not tackled in this paper. A conventional monitoring of resources and related dependability measures are required to provide information to the system manager. He is responsible in a first step for developing a valid FTM and the corresponding transition algorithm. He is also responsible for triggering the change at runtime.

We have developed a number of Fault Tolerance Design Patterns considering various fault models and application assumptions in UML. The main effort was to consider adaptation when designing the mechanisms as a first objective. We validated this approach by implementing and testing these FT Design Patterns in C++. A full account of this work can be found in [4]. The basic philosophy, an example of such a design and lessons learnt are presented in Section III.

From this design, the next step was to map it to components and to develop transition scripts. The current implementation is done on the OW2 FraSCAti middleware as a proof of concept. In summary, the various FTMs are viewed as a graph of components at runtime. Several of them share some root components, so the evolution of a given mechanism leads to modifying the graph by removing, replacing, adding components to the mechanism in operation. A so-called transition algorithm performs the updates.

Based on CBSE and related middleware facilities, this approach provides an interesting perspective to master evolution of dependable systems, making them resilient. The considered mechanisms do not need to be all available at a given point in time. Monitoring the system in operation gives insights to the system designer to design a new appropriate mechanism and develop the appropriate transition algorithm. The benefit here is that only new components and the transition algorithm need to be uploaded at runtime. The system manager can then trigger the execution of the transition algorithm. This approach has also the benefit of reducing the resource overhead and the interruption time for resource constrained systems.

C. An Illustrative Example

We consider in the practical section of this paper two well-known variants of a duplex protocol tolerating crash faults: PBR and LFR strategies. PBR stands for Primary Backup Replication: a first active replica (the master) executes service requests and propagates checkpoints to a so-called passive replica (the slave). LFR stands for Leader Follower Replication: Both replicas process input requests, and only the master delivers response messages to the client. In both cases, a crash detection of the master triggers a recovery action by which the slave becomes the master. In terms of application assumptions the two strategies differ: LFR requires determinism of the application behaviour but does not require state capture, the slave executing the same input

requests. Conversely, PBR does not require determinism of the application behaviour but imposes state capture that can be a very tricky action. In terms of resource cost, PBR requires more network bandwidth in normal operation.

Figure 1 gives an example of a possible sequence of updates at runtime. At t_0, an application is loaded with a PBR_1 strategy. At some time later t_1, some update is performed leading to a PBR_2 strategy. A new evolution now at t_2 is done to reduce the bandwidth used in normal operation. An *LFR* strategy is selected. At t_3, the system manager observes a high number of transient faults and decides to complement the duplex strategy with a Time Redundancy (*TR*) strategy.

III. Design for adaptation

In this section, we describe the process of developing our toolbox of reusable and customizable FT Design Patterns following a "design for adaptation" approach. The basic idea of this approach is that we must analyze the various FT protocols we target and to identify and isolate the different variation points between them. By identifying and isolating these differences (in this object-oriented design, we isolate the variation points in specific object methods), moving from one protocol to another becomes extremely intuitive and can be done without unnecessarily duplicating elements which are common to the initial and the final mechanism from our transition.

We consider this "design for adaptation" approach as a means of achieving what we call "cold resilient computing" because for moving from one mechanism to a new one, we must stop the application and restart it. By mapping this design on platforms supporting runtime reconfiguration, we reach what we call "hot resilient computing", the topic of Section V. This toolbox is the result of several design loops, each loop corresponding to a refinement step towards achieving the optimal representation of the various concerns and protocols and, in particular, towards a generic protocol execution scheme.

A. Initial design

The requirements we set for building our toolbox were to develop FT mechanisms targeting the crash fault model, in a first step, which can be attached to query-response systems (processing incoming requests) and which preserve the "only-once" semantics. We used IBM Rational Software Architect v8.0 for UML design and attached the FT protocols to a very simple application for validation, a basic calculator service.

To really perceive in deep the way to adapt FTMs, we start with a design of PBR (and its corresponding implementation) whose core is a class encompassing general fault tolerance concerns, duplex concerns and the PBR protocol itself. The aim is to reach a clean separation between these concerns and to be able to easily develop duplex variants and non-duplex protocols.

FTM	Before	Proceed	After
TR	Capture state	Compute	Restore state
Assertion	Assert-input	Compute	Assert-output
PBR (P)	Nothing	Compute	Checkpoint to B
PBR (B)	Nothing	Nothing	Process checkpoint
LFR (L)	Forward request	Compute	Notify F
LFR (F)	Receive request	Compute	Process notification

Figure 2. Generic execution scheme of FT strategies

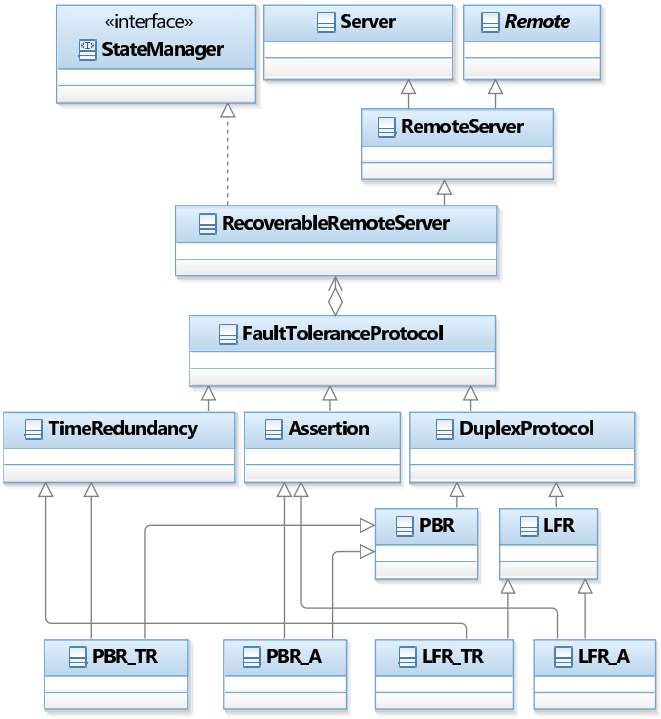

Figure 3. Excerpt from FT Design Patterns Toolbox (UML class diagram)

B. Design for adaptation steps

By analyzing the different inter-replica protocols, we identified the variation points between them and reached a generic execution scheme, inspired from aspect-oriented programming. The main idea is that for adding FT to ordinary request processing, some actions must be performed *before*, e.g. apply assertions on the input parameters. If these actions are successful, processing *proceed*; then some actions are performed *after*, e.g. apply assertions on the result and, if these actions are successful as well, the result is sent to the client. This is, in short, the *Before-Proceed-After* execution scheme, which is similar to aspect-oriented programming, in which function calls can be intercepted and different actions can be performed before and after calling the actual function. Figure 2 summarizes the description of the FT strategies we

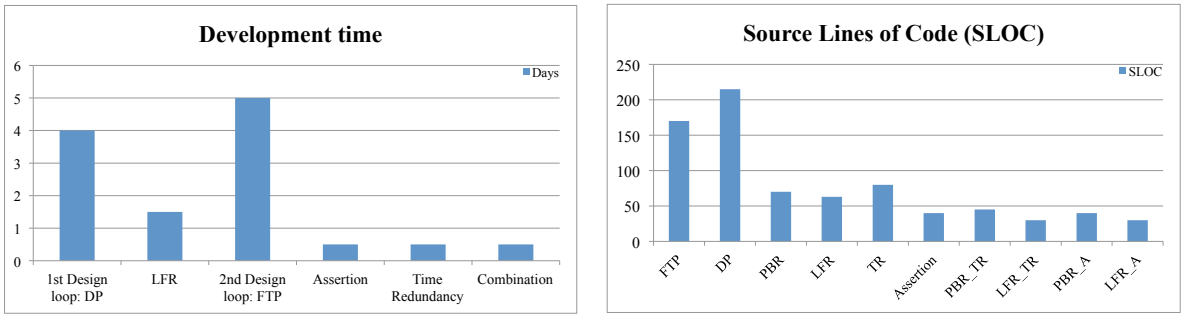

Figure 4. FT toolbox design and implementation: development time (left) and source lines of code (right)

are currently considering, according to our scheme.

When designing a duplex mechanism targeting the crash fault model, this execution scheme can be translated into *Sync_before-Proceed-Sync_after*, which is a generic representation of an inter-replica protocol. Basically, there is a form of inter-replica synchronization taking place before request processing and another synchronization after the processing. The three steps in this execution scheme represent the variation points between different protocols and combinations of protocols.

This execution scheme enables us to factorize what is common to all duplex strategies in a class, DuplexProtocol in Figure 3 and then specialize, through inheritance, the various protocols, PBR and LFR. Other duplex variants can be easily added to our toolbox by following this design approach, i.e. describing the protocol according to our generic scheme and inheriting, either from DuplexProtocol, or from PBR or LFR, if we want to develop a variant of one of these. Decoupling the duplex base-class from specific duplex protocols represents the first design loop for adaptation.

Going further in a new design loop for adaptation, another separation/factorization can be done between what is common to all FT strategies, duplex or not, and what is specific to duplex ones. Communication with the client, preservation of "only-once" semantics and request forwarding to the actual functional service in the processing step are therefore encapsulated in a class, FaultToleranceProtocol in Figure 3. This enables us to place non-duplex protocols in our framework, i.e. protocols targeting other fault models, namely value faults (transient and permanent): TimeRedundancy and Assertion, which follow the same generic execution scheme (see Figure 2). Other non-duplex protocols can be easily added to our framework by inheriting from FaultToleranceProtocol and the two subclasses (TimeRedundancy and Assertion) can be tailored to the user's needs, through inheritance as well, especially in the case of Assertion, which usually demands the definition of an application-dependent assertions. Decoupling general fault tolerance concerns from duplex ones and developing non-duplex protocols represent our second design loop.

C. Combining FT strategies

Having developed the protocols resulting from the two design loops for adaptation, our next aim was to combine mechanisms targeting different fault models, i.e. a duplex protocol (PBR, LFR) with a mechanism tolerating value faults (Assertion, Time Redundancy). Mechanism combination is extremely intuitive and results from inheriting the corresponding duplex protocol and the value fault tolerance mechanism, as shown in Figure 3. We obtain four composed mechanisms: PBR_TR, LFR_TR, PBR_A, LFR_A. Our *Before-Proceed-After* scheme enables nesting the execution steps of the base classes. For instance, PBR_TR inherits from PBR and TimeRedundancy and has the following steps: sync_before() of PBR, before() of TimeRedundancy, processing(), after() of TimeRedundancy, sync_after() of PBR.

D. Development efficiency

The implementation of our first experimental validation of FTMs and composed mechanisms in C++ is almost immediate thanks to our two design loops, but their significance is nonetheless powerful: they extend the fault model initially considered (the crash fault model) to crash faults and value faults. During the development of this FT toolbox, we observed that our design approach was very efficient, in terms of development time of new mechanisms and in terms of code reuse and lines of code to be produced (see Figure 4). Our statistics, which are based on the work done by a pair of junior developers, clearly show that the development of a concrete protocol takes less than a quarter of the time spent on a design loop for adaptation.

The toolbox we presented in this section validates our "design for adaptation" approach, showing that careful design, evolution prediction, modularity and clean separation of concerns, in the large and in the small, enable us to reach a generic protocol execution scheme which is the cornerstone of our framework.

In the next sections we show the process of moving from this toolbox to a component-based implementation in which transitions between FT mechanisms are executed at runtime.

IV. From objects to components

The design we obtained is essential for understanding and achieving a detailed implementation of FTMs, validated on a simple service. Now, this design must be mapped to components for adaptation, in particular at runtime.

A. Service Component Architecture

Service Component Architecture (SCA) [5], [6] provides a set of specifications for building and composing loosely-coupled, tailorable applications encompassing a wide range of technologies. The main idea of this paradigm is that applications are built from *bricks* (i.e., components) exposing their functionalities in the form of services and consuming services provided by other components through references. By separating the interfaces (services and references) from the actual implementation, this approach facilitates reuse, evolution and technology-agnosticism as components consume services provided by other components without being aware of how they are implemented and whether the implementation changes over time.

B. OW2 FraSCAti

The component-based middleware on which we develop our toolbox of adaptive FTMs is OW2 FraSCAti v1.4 [7], a platform providing runtime support for SCA and developed according to SCA principles. The OW2 FraSCAti runtime provides support for SCA composite definitions following the SCA Assembly Model V1.0 specification, Java component implementation (SCA Java Component Implementation V1.0 and SCA Java Common Annotations and APIs V1.0), remote component bindings using Web Services (Soap or RESTful) and Java RMI protocols.

C. Runtime reconfiguration support

FraSCAti goes beyond SCA specifications by also providing support for runtime reconfigurations of component-based architectures and reflective capabilities. Runtime architecture exploration and reconfiguration can be performed in several ways, and one of the most convenient ones is by using FScript. Dedicated to querying and modifying component-based applications, this scripting language provides support for atomic changes [8], thus guaranteeing that a reconfiguration either takes a component-based architecture to a consistent state or leaves it unmodified in its initial state, should there be a bug in the script. This is an essential property for our transitions as the addition of dynamics to FTMs should not impair the reliability of the mechanisms.

The runtime reconfiguration requirements we have identified for building a resilient computing framework are the following:

- *access* to components' state and properties;
- *control* over components' *lifecycle* (start, stop);
- *control over interactions* between components, for creating or removing bindings.

Furthermore, to maintain consistency, several issues must be carefully considered:

- components must be stopped in a *quiescent* state, i.e. when all internal processing has finished;
- incoming requests on stopped components must be buffered;
- the state of old components must be mapped to the ones replacing them.

FraSCAti and FScript fulfill the first two requirements, leaving component state management to the designer. Several options can be envisaged:

- stateful components can explicitly provide a state management service (in the form of a Java interface with `getState()` and `setState()` methods) which is called when needed;
- the designer can create a `@State` annotation and place it on the attributes considered part of the component state and develop the corresponding component controller which interprets this annotation;
- we can design the architecture in such a way that components which are subject to runtime reconfigurations have no state at all.

In our current implementation, the application server implements a state management interface which enables us to capture and restore state for protocols demanding it, e.g. PBR, TR. Components which are part of the actual FTM are stateless, as a result of careful design for reaching the desired granularity, which is further detailed. If stateful components are manipulated at runtime, we must define a method (e.g. a specific action in FSript) for capturing the state of old components and restoring it on the newly instantiated ones.

D. Towards component-based design

Building on the lessons learnt during the previous step of our development process, namely design for adaptation, we design the FTMs targeting crash faults, i.e. Primary-Backup Replication and Leader-Follower Replication, according to the *Before-Proceed-After* general protocol execution scheme. The transition from an object-oriented design to a component-based architecture is not an automatic process, neither in terms of entities nor in terms of interactions between them. A component-based architecture could be designed using either a top-down or a bottom-up approach.

Given the freedom to design components as fine-grained as we want, the obvious question which arises is: what is the *right* granularity? First, we need to define what *right* means, in the context of our work. We aim at facilitating the transitions between mechanisms and as the unit of runtime manipulation is the component, it means that for performing fine-grained reconfigurations it is necessary and sufficient to encapsulate the differences between closely-related FTMs in individual components .

As described in Section III, for performing a transition between PBR and LFR, we would need to change

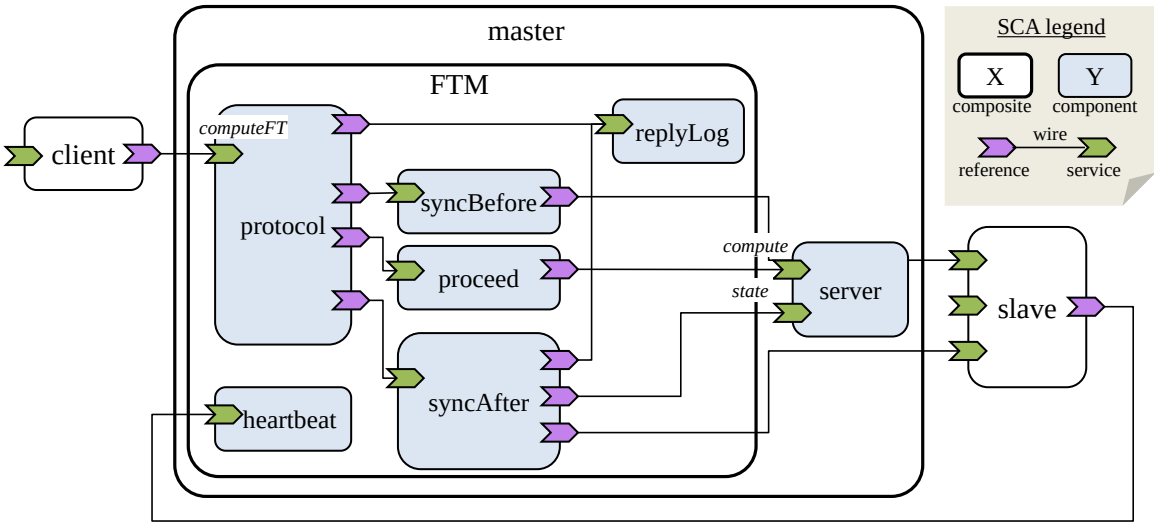

Figure 5. PBR (Primary-Backup Replication) component-based design for adaptation

the methods sync_before() and sync_after(). Therefore, when moving towards a component-based implementation, it seams ideal to encapsulate each step of the *Before-Proceed-After* execution scheme in an individual component.

This analysis gives us the *right* component granularity: we have reached the suitable level of decomposition when we are able to isolate in separate components the differences between closely-related FTMs in what could be regarded as a *separation of concerns* in the small. In practice, for decomposing the PBR entity (and its base classes) from the class diagram in Figure 3, we use 4 components, represented in Figure 5: a stateful one, `protocol`, encapsulating the functionalities of the classes PBR inherits from (FTProtocol, DuplexProtocol) and 3 stateless ones, `syncBefore`, `proceed`, `syncAfter`, corresponding to the methods sync_before(), processing(), sync_after().

During the process of mapping our object-oriented design to components, we observed the following:

- Relationships such as composition, association and aggregation are usually translated into a reference-service interaction;
- Translating inheritance usually needs to be tackled on a case-by-case basis;
- In some cases, 1 object maps to 1 component, e.g. FTProtocol to `protocol`;
- In some cases, 1 method maps to 1 component, e.g. sync_before() to `syncBefore`.

Apart from granularity, another problem to tackle is the generality of interfaces representing services provided by components. Components provide and consume services, therefore they have interface-typed members instead of classic references to objects. In order to replace a component A with a component A', A and A' must provide the same services to the consumers which are not modified by the reconfiguration. Therefore, from the very beginning, components which may be changed in order to execute a transition from one mechanism to other(s) must implement generic interfaces. This can be obtained by using polymorphism, for instance, on the types of parameters taken as input by methods in the given interfaces. Replacing a component does not necessarily imply a 1-to-1 correspondence between old and new components.

In the next section, we describe in details the architecture of our PBR component-based implementation and the transition towards LFR through reconfiguration scripts executed at runtime. Other possible transition scenarios are also discussed.

V. Component-based implementation and runtime reconfiguration

A. PBR component-based architecture for adaptation

Figure 5 shows the component-based design of Primary-Backup Replication, with a detailed view of the primary entity. In the current implementation, we name the server entities `master` and `slave` rather than primary and backup in order to have a uniform naming convention when passing to another duplex protocol, i.e. Leader-Follower Replication. We can see that the *Before-Proceed-After* execution scheme is mapped onto individual components, as explained in Section IV. All the components represented in Figure 5 are implemented in Java. Our application consists of three composite components and their interactions: the `client`, the `master` processing the requests and the `slave` processing the checkpoints sent by the master. In parallel, a failure detector periodically checks the liveness of the master using a heartbeat mechanism. Should the master crash, the slave substitutes for it.

The component-based design emphasizes the separation of concerns between the functional layer of the application

(the actual server) and the non-functional one (the FTM protocol, encapsulated in a composite component, master-side and slave-side). The application itself (i.e. the functional layer) is a basic calculator service, the aim being to trigger the execution of the FTM protocol.

B. From PBR to LFR at runtime

Now, we focus on the actual transition between two FTMs targeting the crash fault model, i.e. PBR and LFR. This is achieved through a reconfiguration process, involving the replacement of old components with new ones and dealing with distributed computing aspects.

1) Replacing components: As previously discussed in Section IV-C, when replacing components, we must deal with state management, more precisely capturing the state of the old component, disconnecting it, mapping the state on the new component and connecting it to the overall architecture. In our architecture, when moving from PBR to LFR, we must replace `syncBefore` and `syncAfter` (see Figure 2).

These components, identified as possible points of variation, together with `proceed`, are rendered stateless in our design by having them pull information from stateful, stable components (not subject to change when moving from one FTM to another) such as `protocol`, in the form of parameters in service calls, and `replyLog`. Having dealt with the state management issue, two ways of performing the "mechanics" of component replacement are possible.

- The first one consists in deploying, from the beginning, a component-based architecture containing all the foreseeable `syncBefore`, `proceed`, `syncAfter` components and only wiring the ones which belong to the initial FTM, i.e. PBR, while the others are left unwired. A script would only need to unwire the old components and wire the new ones. With this solution, all possible reconfigurations and transitions must be foreseen from the beginning, in the initial architecture.
- The second solution, which is the one we implemented, consists in deploying an architecture corresponding to the first mechanism, i.e. PBR, and, in the runtime reconfiguration script, loading the new components from separate description files. This enables us to have a separation between the fault-tolerance concerns and the adaptation ones and to enrich our framework with new reconfiguration scenarios. There is a trade-off between flexibility and reconfiguration speed: given the fact that the new components are not instantiated a priori, they must be loaded on-the-fly from a separate file, therefore reconfiguration is slower than in the first case. Our implementation is at an early stage, therefore we do not have precise measures for each step of the reconfiguration process, this being part of our future plans.

Figure 6 contains the script executing the transition between PBR and LFR, master-side, i.e. the transition between Primary and Leader. In short, this script does the following:

- load `syncBeforeLFR` and `syncAfterLFR` from a separate description file (lines $3-11$);
- disconnect the old `syncBefore` and `syncAfter` from all their services and references (lines $16-22$ and $31-36$);
- connect `syncBeforeLFR` and `syncAfterLFR` to all the necessary services and references (lines $23-25$ and $37-39$);
- delete old components (lines $26-28$ and $40-42$).

Concerning the components that must be stopped, FraSCAti requires us to stop a component only if it has a reference wired (i.e. connected through a local binding, inside the same process) to a component which is going to be removed. Buffering incoming requests, putting components in a quiescent state when executing various actions such as wiring references, binding and unbinding services and references are platform-specific and transparent to the user.

2) Distributed computing issues: Reconfiguration mechanics is executed by local scripts, meaning that there is a component implemented in FScript on the master and another one on the slave, in order to have access to the component graph corresponding to each machine. For the time being, reconfiguration is triggered by the system manager through a service provided by the master, which runs the master script and then, if no problem was encountered, the slave one. Script termination is a property guaranteed by the framework, as previously mentioned.

The inverse transition (LFR to PBR) is also possible, through a similar procedure (having a separate description of the corresponding components, their Java implementation and the associated scripts), provided that our `server` component implements a state management interface enabling us to capture state and to map it.

C. Composing FTMs

Fault tolerance requirements represent one of the three axes in our change model frame of reference described in Section II. The fault model can change during the system's lifetime by being extended. Composing FTMs tolerating different fault models can sometimes be necessary, especially if fault tolerance requirements were not completely defined during the specification phase of the system's development cycle. A typical scenario, as shown in Figure 1, requiring the composition of FTMs can be the following: at initial time, a duplex strategy was selected for a given function to comply with the crash fault model, as requested in the specification; at a later time, the monitoring of the system reveals a high number of transient physical faults impacting the results of the function. It is decided to combine a time redundancy (TR) strategy with the duplex strategy to comply with this new situation.

```
action changeStrategyPbr2Lfr() {

sca-new("pbr2lfr.composite");
master=$domain/scachild::master;
ftm=$master/scachild::ftm;
protocol=$ftm/scachild::protocol;
add-scachild($ftm,$domain/scachild::pbr2lfr/scachild::syncBeforeLFR);
add-scachild($ftm,$domain/scachild::pbr2lfr/scachild::syncAfterLFR);
sca-remove("pbr2lfr");
new_sync_before=$ftm/scachild::syncBeforeLFR;
new_sync_after=$ftm/scachild::syncAfterLFR;
old_sync_before=$ftm/scachild::syncBefore;
old_sync_after=$ftm/scachild::syncAfter;

-- Replacing SyncBefore
set-state($old_sync_before,'STOPPED');
set-state($protocol,'STOPPED');
remove-scawire($protocol/scareference::syncBeforeService,$old_sync_before/scaservice::execute);
remove-scawire($protocol/scareference::syncAfterService,$old_sync_after/scaservice::execute);
set-state($protocol,'STARTED');
old_binding=$old_sync_before/scareference::synchronizeService/scabinding::syncBefore-RESTful-stub;
remove-scabinding($old_sync_before/scareference::synchronizeService,$old_binding);
add-scawire($protocol/scareference::syncBeforeService,$new_sync_before/scaservice::execute);
add-rest-binding($new_sync_before/scareference::synchronizeService,"http://localhost:8082/SyncBeforeService");
set-state($new_sync_before,'STARTED');
set-state($ftm,'STOPPED');
remove-scachild($ftm,$old_sync_before);
set-state($ftm,'STARTED');

-- Replacing SyncAfter
set-state($old_sync_after,'STOPPED');
remove-scawire($old_sync_after/scareference::replyLogService,$ftm/scachild::replyLog/scaservice::replyLogService);
old_binding=$old_sync_after/scareference::stateAccessService/scabinding::syncAfter-RESTful-stub;
remove-scabinding($old_sync_after/scareference::stateAccessService,$old_binding);
old_binding=$old_sync_after/scareference::synchronizeService/scabinding::syncAfter-RESTful-stub;
remove-scabinding($old_sync_after/scareference::synchronizeService,$old_binding);
add-scawire($protocol/scareference::syncAfterService,$new_sync_after/scaservice::execute);
add-rest-binding($new_sync_after/scareference::synchronizeService,"http://localhost:8083/SyncAfterService");
set-state($new_sync_after,'STARTED');
set-state($ftm,'STOPPED');
remove-scachild($ftm,$old_sync_after);
set-state($ftm,'STARTED');
}
```

Figure 6. The script implementing the transition between PBR and LFR, master-side (Primary to Leader)

For facilitating mapping to components, we can combine the execution steps of TR (see Figure 2) in one single component `proceedTR`. In practice, to add TR, given our PBR component-based architecture or the LFR mechanism obtained after executing the transition described in the previous section, we would only need to change the `proceed` component which, in its current configuration, only calls the actual service and sends back the result. A `proceedTR` implementation would need to do the following:

- capture the state of the server prior to computation;
- call the service and store the result;
- restore the state of the server to the initial one;
- call the service a second time and store the result;
- compare the results: if they are identical, send the result, if not send an error code.

For the runtime reconfiguration we need the implementation of the new `proceedTR` component and a script performing the following actions:

- load the description of the new component;
- stop the `protocol` component;
- unwire `protocol` from `proceed`;
- wire `protocol` to `proceedTR`;
- start the `protocol` component;
- wire `proceedTR` to the calculator service provided by the server;
- wire `proceedTR` to the state management service provided by the server because, unlike the old `proceed`, the new one also needs to capture and restore state;
- delete `proceed`.

Our current work focuses on completing the implementation of this scenario.

VI. Further benefits of a component-based implementation

In this section, we describe two ways of exploiting the support and capabilities of a component-based architecture: first, we show how to exploit the separation of concerns provided by our component-based architecture for replacing the functional layer (the application) with another one and reusing the FTM; next, we describe the recovery procedure of PBR, which is implemented in a reconfiguration script.

The component-based design emphasizes the separation of concerns between the functional layer of the application (the actual server) and the non-functional one (the FTM protocol, encapsulated in a composite component, master-side and slave-side). Notice that it is not difficult to develop a more sophisticated application, given the fact that SCA is particularly useful for precisely identifying where we need to modify the implementation. To be more precise, we need to change:

- the `server` component both in terms of functional service/application implementation and in terms of state management service, as state information is application-dependent;
- the `client` component, to stay consistent with the application;
- the `syncAfter` components, master-side and slave-side, which respectively pack and unpack checkpoints, and therefore need to know contained inside checkpoints;
- the `replyLog` component, which needs to store enriched results;
- the data structures, e.g. ClientRequest, CheckpointMessage, ServerState, which are application-dependent.

However, some components do not need specific modifications. For instance, components which only delegate service calls (`protocol`), components that forward incoming information (`syncBefore` and `proceed`), or components that only slightly modify service calls (`proxy` only adds an identifier to the client request) are left unmodified.

We have also experimented implementing the recovery procedure in PBR in a reconfiguration script, which gave us the possibility to experiment with dynamics at runtime in the small, i.e. inside one particular mechanism. We target the PBR component-based implementation described before and develop the recovery procedure in a script which manipulates components on-the-fly. When the master fails, the client needs to be connected to the former slave, which will become the new master. To that aim, Figure 7 presents the associated script that implements the following actions:

- a reference to the requested operation is obtained by introspecting the client component (`compute`, lines $2-5$);
- the binding between the client and the former service provider is disconnected (line 6);
- a new connection between the client and the service provided by the new master is established (line 7).

Reconfigurations scripts can be executed step-by-step using the the OW2 FraSCAti interactive Explorer (a sort of testing phase), but can also be directly executed. In practice, such reconfiguration scripts (e.g. the recovery one and the transition between FTMs) are actually components implemented in FScript and publishing services in the form of Java interfaces. Reconfiguration is triggered in special conditions (in this case because of the master failing) by components which have a reference on the Java interface/service provided by the component implemented in FScript.

```
action switchServer(){                                                  1
client_machine = $domain/scachild::client_machine;                      2
proxy = $client_machine/scachild::proxy;                                3
refcompute = $proxy/scareference::compute;                              4
bind_old = $refcompute/scabinding::proxy-RESTful-stub;                  5
remove-scabinding($refcompute,$bind_old);                               6
add-rest-binding($refcompute,                                           7
      "http://localhost:8081/ComputeService");                          8
}                                                                       9
```

Figure 7. Script-based runtime recovery procedure inside PBR

VII. Related Work

One of our first areas of interest was Autonomic Computing (AC) [9], more precisely self-healing systems, i.e., systems which are able to repair themselves. We consider that runtime adaptation of FTMs and, more generally, resilient computing are part of this trend. Similarly, in [10] , the author states that dependability and fault tolerance are not only "specifically aligned to the self-healing facet" of AC but, on a closer view, "all facets of Autonomic Computing are concerned with dependability" (i.e., self-configuration, self-optimization and self-protection as well).

In the field of adaptive fault tolerance (AFT) [11], several projects exist, especially targeted at CORBA-based applications [12], but evolution is tackled differently: adaptation has a parametric form (e.g. number of replicas) or it is performed at compile time or, if done at runtime, has a coarse-grained nature. [13] describes a component model based on the CORBA Component Model for building distributed applications with fault-tolerance requirements which are able to change their FT strategy at runtime. Leveraging the flexibility provided by a components, it is the closest work we have found but reconfiguration is more coarse-grained than in our approach and only the crash fault model is taken into account.

In the area of reconfigurable software architectures, RAINBOW [14] builds on the use of architectural models for problem diagnosis and repair. The proposed framework includes a monitoring layer composed of two types of entities, namely probes which gather basic data from the running system and gauges which perform computations on the data in order to obtain measures of the system properties. An architecture manager is in charge of maintaing the architectural model at runtime and of verifying that the constraints on the system elements are maintained. The project is very complex as it includes a very expressive ADL called ACME [15], a system in charge of verifying constraints, called ARMANI, a library of gauges, etc. In [16], the authors describe their experience in associating an enriched version of ACME with the OpenCOM middleware (a component-based middleware) for providing programmed and ad-hoc

changes at runtime while maintaining certain constraints. Although these projects do not tackle adaptation in a fine-grained way as we do and they do not target fault-tolerance mechanisms, they are interesting from a methodological point of view.

Concerning design patterns for fault tolerance, [17] describes a generalized FT design pattern for strategies targeting software faults. This could be a source of inspiration for enriching our current toolbox of fault tolerance strategies.

VIII. Conclusion

Our approach to resilient computing relies on several steps, from a design for adaptation down to the verification of the consistency of distributed updates of a real system. A cornerstone in this process is the manipulation of software architecture at runtime. A component-based middleware is an essential piece of this process. The experiments described in this paper show that combining careful design with the exploration and control capabilities provided by such a platform enable resilient computing to be addressed.

In our future work, we aim at exploring several directions: enriching the existing component-based FT toolbox with different transition scenarii; integrating our mechanisms in a framework which takes into account fault tolerance requirements, generates the most appropriate FTM to connect to an application and triggers adaptation according to changes in the initial requirements; use the lessons learnt from this first implementation for building our adaptive FTMs on a platform for networks of environmental wireless sensors.

References

[1] J. Laprie, "From dependability to resilience," in *International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, AK, USA, pp. G*, vol. 8, 2008.

[2] J.-C. Fabre, "Architecting dependable systems using reflective computing: Lessons learnt and some challenges." in *WADS'09*, 2009, pp. 273–296.

[3] P. Maes, "Concepts and experiments in computational reflection," *ACM Sigplan Notices*, vol. 22, no. 12, pp. 147–155, 1987.

[4] V. Gibert, M. Machin, J.-C. Fabre, and M. Stoicescu, "Design for adaptation of fault tolerance strategies," LAAS-CNRS, Tech. Rep. 12198, April 2012.

[5] D. Chappell, "Introducing sca," *Avalaible at http://www. davidchappell. com/articles/Introducing_SCA. pdf*, 2007.

[6] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 2009.

[7] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures," *Software: Practice and Experience*, 2011.

[8] P. David, T. Ledoux, M. Léger, and T. Coupaye, "Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures," *Annals of Telecommunications*, vol. 64, no. 1, pp. 45–63, 2009.

[9] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[10] R. Sterritt and D. Bustard, "Autonomic Computing-a means of achieving dependability?" in *Proc. IEEE Engineering of Computer-Based Systems*, 2003.

[11] K. Kim and T. Lawrence, "Adaptive fault tolerance: Issues and approaches," in *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*. IEEE, 1990, pp. 38–46.

[12] E. Shokri, H. Hecht, P. Crane, J. Dussdault, and K. Kim, "An approach for adaptive fault-tolerance in object-oriented open distributed systems," in *Object-Oriented Real-Time Dependable Systems, 1997. Proceedings., Third International Workshop on*. IEEE, 1997, pp. 298–305.

[13] J. Fraga, F. Siqueira, and F. Favarim, "An adaptive fault-tolerant component model," in *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. Proceedings. Ninth IEEE International Workshop on*. IEEE, 2003, pp. 179–186.

[14] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the first workshop on Self-healing systems*, ser. WOSS '02. New York, NY, USA: ACM, 2002, pp. 27–32. [Online]. Available: http://doi.acm.org/10.1145/582128.582134

[15] D. Garlan, R. Monroe, and D. Wile, *Acme: Architectural description of component-based systems*. Cambridge University Press, 2000.

[16] T. Batista, A. Joolia, and G. Coulson, "Managing dynamic reconfiguration in component-based systems," *Software Architecture*, pp. 1–17, 2005.

[17] F. Daniels, K. Kim, and M. Vouk, "The reliable hybrid pattern: a generalized software fault tolerant design pattern," in *Int. Conf. PloP97*, 1997, pp. 1–9.