Architecting Resilient Computing Systems: Overall Approach and Open Issues

Miruna Stoicescu, Jean-Charles Fabre, Matthieu Roy

Architecting Resilient Computing Systems: Overall Approach and Open Issues*

Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy

CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse , France
Université de Toulouse ; UPS, INSA, INP, ISAE ; UT1, UTM, LAAS

Abstract. Resilient systems are expected to continuously provide trustworthy services despite changes in the environment or in the requirements they must comply with. In this paper, we focus on a methodology to provide adaptation mechanisms meant to ensure dependability while coping with various modifications of applications and system context. To this aim, we propose a representation of dependability-related attributes that may evolve during the system's lifecycle, and show why this representation is useful to provide adaptation of dependability mechanisms at runtime.

1 Introduction

One of the main challenges nowadays, as stated by IBM in The Vision of Autonomic Computing [12], is managing systems for which the total cost of ownership is ever-increasing as they continuously evolve. The solution to this problem would be for such systems to become autonomous, to a certain extent, and no longer depend on humans for performing basic management tasks.

Autonomic Computing is also enticing for ubiquitous systems based on technologies such as Wireless Sensor Networks. The aim of Autonomic Computing is described in [22] as addressing "today's concerns of complexity and total cost of ownership while meeting tomorrow's needs for pervasive and ubiquitous computation and communication".

Our current work shares this vision while adding a fault tolerance axis. A self-healing system is able to identify when its behaviour deviates from the expected one and to reconfigure in order to correct the deviation. We understand a self-healing system as a context-aware fault tolerant system. To ensure a safe adaptation, a validation step has to be added to this scheme, guaranteeing safety during any reconfiguration. More precisely, we define the dynamic adaptation, or self-healing, process as a *two-step permanent loop* consisting of a monitoring service and an adaptation engine. The monitoring service is in charge of observing the system, measuring certain parameters and resource properties and informing the adaptation engine. The latter must analyze the values, compare the observed behaviour to the expected one, decide if an adaptation is needed, choose a reconfiguration strategy and apply it.

* This work is supported by ANR, contract ANR-BLAN-SIMI10-LS-100618-6-01.

Our aim is to develop a method to define fault tolerant applications and to adapt them during the system's lifetime. The basic idea is to break them down into fine-grained components and to minimize the modification to be performed to update the fault tolerance mechanisms with respect to operational conditions.

This paper presents *on-going work* on a methodology for developing resilient applications for systems ranging from workstations to smart sensors. These applications are context-aware and must be able to adapt their fault tolerance mechanisms as a consequence of changes in their requirements and/or the environment. The rest of this paper is organized as follows: Section 2 gives our view on this matter and a representation of the problem, Section 3 thoroughly describes our approach, Section 4 presents our work plan, Section 5 presents a case study, Section 6 presents related works and finally, in Section 7 we give the concluding remarks.

2 Problem Statement

The evolution of systems may have an impact on dependability, i.e., various changes may invalidate some dependability properties and thus call for new solutions. In our work, we consider that any application is attached to one or several fault tolerance mechanisms (FTMs). Among the evolution scenarios that can occur during the operational life of a fault-tolerant application, we focus on those that may have an impact on the FTMs.

Beforehand, we define a frame of reference as a 3-axis space, each axis representing multiple variables, namely application assumptions, fault tolerance requirements, and system resources. A fault tolerant application may evolve in this space during the lifetime of a system due to:

- changes in the application assumptions which influence the choice of the fault tolerance mechanism,
- changes in the fault tolerance requirements,
- changes in the system resources (e.g., number of processors, memory resources, network bandwidth).

The essence of a *resilient application* [14] lies in the fact that when faced with any of these changes or with a combination of them, it must adapt itself while ensuring the observance of the dependability criteria. The 3-axis space gives an idea of the change model that we consider.

In Fig. 1, a cloud represents an application attached to a FTM. For a given application, each cloud represents a region in this 3-axis space. We view the evolution of a system during its operational life as a trajectory in a space characterized by the parameters which can cause the aforementioned evolution. The three evolution scenarios can be aggregated into this frame of reference for this space, the three axes being: the application assumptions (i.e., whether it is deterministic or not), the fault tolerance requirements (i.e., the fault model) and the current system resources.

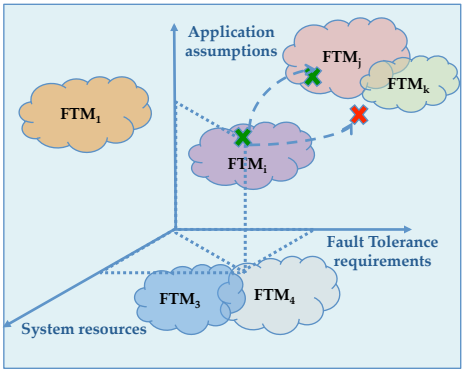

Fig. 1. Context-aware fault tolerant computing

It is the responsibility of the designer of the system to design the set of FTMs and foresee all possible evolutions and changes of the system, so that the trajectory it follows during its operational life is entirely covered by FTMs. We claim that a system's evolution can be represented in this vector space as shown in Fig. 1. The combination of these three types of information indicates the most convenient FTM to use. A solution in this space addresses a specific fault model, relies on some specific application assumptions and has a cost determined by a function with multiple resource parameters.

Given this frame of reference, we consider that each FTM covers a certain region of space as shown in Fig. 1. Obviously, the regions associated to different FTMs may partially overlap. To link the system's current state with this representation, a monitor keeps track of changes in terms of the three axes and allows placing the system status in a certain region of this space, which, by hypothesis, is covered by at least one FTM.

Once our view of the problem stated, a certain number of issues arise, such as: how do we associate FTMs with the combination of the three measures? how do we describe these FTMs in a way which allows us to change them dynamically? how do we actually perform the transition from one FTM to another? The following sections present our view on these matters.

3 Our Approach

The work reported in this paper is based on former work carried out on resilient computing at LAAS. We investigated the on-line adaptation of component-based FTMs (primary-bakup and semi-active replication) and applied it to an automatic control application (an inverse pendulum control system located on a cart). The modeling of FTMs using Petri nets was used to control their execution and to synchronize the adaptation, i.e. the change of mechanisms. An interesting point was the analysis and the definition of suitable adaptation states for performing the adaptation in a consistent manner. A full account of this work

can be found in [8]. Another aspect of the work was the monitoring of timed properties of real-time applications, using Timed Automata. A monitor checking behavioural and timing properties was developed and implemented in Xenomai. Such a monitoring system was used to provide early error detection for fault tolerance strategies (see [9]). The approach proposed in this paper is based on the lessons learnt from these works and tries to define a whole development and runtime process to address the problem of dependable systems evolution.

Separation of concerns is a generally accepted idea for introducing FTMs in an application in a flexible way which allows subsequent modification and reuse. Based on the principle of behavioural reflection [16], we describe our approach using reflective component models (see Section 4.3). According to these principles, software architectures consist of two abstraction levels where the base level provides the required functionalities and the top level contains the FTM(s) [7]. As we target the adaptation of FTMs, we must manage the dynamics of the top level, which can have two causes:

- The application level remains unchanged but the FTM must be modified either because the fault model changes or because an event in the environment, such as resource availability, makes it unsuitable or, at least, non-optimal from a performance viewpoint.
- Changes in the top level are indirectly triggered by modifications in the application level which make the FTM unsuitable. In this case both levels are modified.

In order to achieve the adaptation of the FTMs in both scenarios, we build on the representation from the previous section and refine it in three steps.

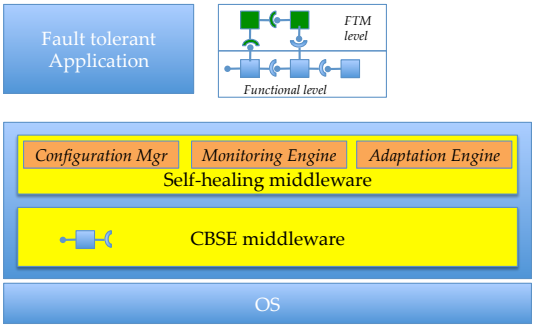

Fig. 2. The architecture of a reconfigurable fault tolerant application

Fig. 2 shows the "big picture" of the system architecture that we target. Based on a CBSE middleware, we develop a self-healing middleware populated with design patterns for fault tolerance and key services:

- a FTM repository in charge of storing the descriptions of FTMs in the form of component-based software architectures (see Section 4.2),

– a monitoring engine in charge of observing changes in the available system resources detailed in the next section,
– an adaptation engine in charge of performing the actual modifications on the FTMs for executing the necessary transitions.

3.1 Description of the frame of reference

The three parameters labeling our axes are actually multidimensional vectors but for the sake of visibility we have chosen an elementary representation.

– The fault tolerance requirements are part of the non-functional specifications of an application. Our main focus is on the fault model, i.e., the types of faults which must be tolerated by the application. We base our fault model classification on known types [1], e.g., physical faults, design faults, leading to value faults or crash faults only.
– The application assumptions group the characteristics of an application which have an impact on the choice of the FTM but are not the same as the functional specifications of the application. These characteristics include: whether the application is stateless or stateful, whether its state is accessible or not and whether the application is deterministic or not.
– The system resources axis groups information such as the number of available processors, the available bandwidth, the memory resources, ... Some of these variables can be precisely determined a priori (e.g., the number of processors) for a given FTM, while others are application-dependent (e.g., network bandwidth for replica synchronization). All these parameters must be continuously monitored.

3.2 Fault Tolerance Mechanisms

In order to place the FTMs in the previously described frame of reference, we must first be able to classify them using the given criteria. In our approach, there are three steps in the selection process for finding the most adequate FTM for an application: first, the fault model is identified, then the application assumptions and finally the currently available system resources. The same process will be applied for classifying them, resulting in a tree detailed in Section 4, Fig. 3.

3.3 System evolution

The status of a fault-tolerant application represents a point in the space given by our frame of reference. The application being fault-tolerant, this point must be in a region covered by a certain mechanism. A change (in terms of one of the three axes) is equivalent to a new status of the application, therefore a new point. As our purpose is to guarantee dependability when facing changes, the application must always be "accompanied" by an adequate FTM on its evolutionary trajectory. Therefore, we must provide a FTM encompassing a region

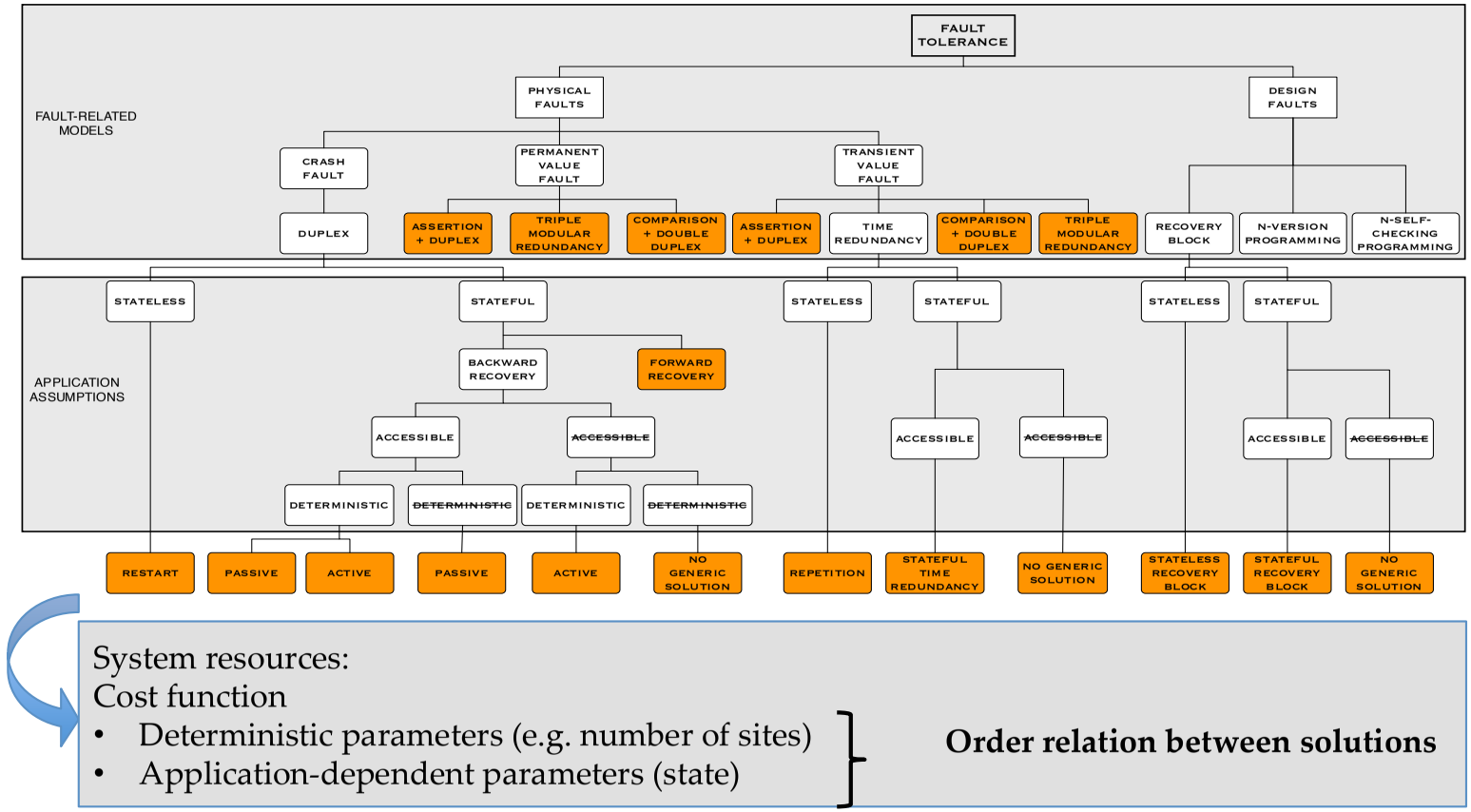

Fig. 3. Classification of fault tolerance mechanisms

which contains the new point. We must design and build our FTMs in view of such transitions and adaptations. The "distance" (as will be defined later on in Section 4.4) between two mechanisms as represented in our frame of reference should be equivalent to the difference between their implementations. If the new mechanism is close to the old one, we should be able to generate it through minor adjustments. The transition between distant mechanisms will most likely demand more complex modifications or even complete replacement.

The planned development process for achieving the smooth and safe transition between FTMs consists of the following elements:

- a complete classification of the FTMs we intend to use;
- a method for describing and storing the architecture of FTMs;
- a set of tools allowing us to develop applications which are easily reconfigurable at runtime;
- one or several algorithms for performing the transition between mechanisms.

The next section describes each stage of this process.

4 Development Process

4.1 Classification of fault tolerance mechanisms

We classified the FTMs according to the three groups of criteria, which were already described. This led to the tree-graph representation from Fig. 3, where the nodes represent our criteria and the leaves represent the mechanisms. It only gives a partial classification because the layer corresponding to the system resources criteria is not included. For the time being, we consider that this axis gives an order relation between mechanisms as certain costs in terms of resources can be associated with each one. We can say, for instance, that active replication is more demanding in terms of CPU operations than passive replication, as all requests are processed at all replicas. More precisely, some parameters depend on the used strategy (e.g., number of sites) while others are application-dependent (state size, synchronization frequency, checkpoint size), the latter being used to determine the cost function (for bandwidth usage, for instance).

The FTMs we are considering are well-established solutions for certain use-cases/scenarios. This has led us to the concept of design patterns for fault tolerance (*Fault Tolerance Design Pattern* — FTDP), by drawing a parallel with the design patterns from software engineering. Therefore, each leaf of the tree in Fig. 3 represents a FTDP and also a more accurate view than the one in Fig. 1 as here we refine the axes variables and labels.

4.2 Description of design patterns for fault tolerance

Each FTDP[1] has an associated software architecture. In order to operate a transition between them, we must be able to manipulate the architecture through

[1] Recall that FTDP stands for Fault Tolerance Design Pattern

its description file. This is a crucial step because from the descriptions of two design patterns A and B, we must be able to easily "compute" $A \cap B$, A/B and $A \backslash B$, if we are to perform mechanism replacement at a finer grain, as further discussed.

A decision must be taken regarding what needs to be stored off-line, before the application executes. One idea would be to describe the architecture of all the FTDPs we intend to use and store these files. Another idea would be to group closely related design patterns, choose the most "representative" one for each class and store its description as well as describe the strategy for passing to another one.

An ADL, such as ACME [11], can be used for describing these architectures. ADLs facilitate the construction of models in which the architecture of a system is described as a composition of components and connectors. Some of them also include properties and constraints which can be placed on those entities. The use of an ADL enables us to reason about a software architecture during the specification stage. In order to maintain this reasoning consistency at runtime, we will use a component-based middleware for implementing the system. This stage is detailed in the next subsection.

4.3 Runtime support and reconfiguration

A design principle that is commonly accepted in the area of reconfigurable systems is the use of component-based technologies (CBSE) [6] for developing the management framework. CBSE middleware provide means to develop and run software as a graph of interconnected components. Applications built using a component-based middleware consist of components and connectors and can be modified at runtime by using the methods provided by the middleware for stopping, unbinding, binding and starting components. In this paper, this technology will be used to implement FTDPs whereas the application will be seen as a black box.

As there are many available platforms, during this project stage we will focus on finding a component-based middleware tailored to our needs. It should provide runtime reconfiguration facilities, have a small memory footprint, suiting target systems such as smart sensors with limited resources and provide the right level of design complexity for our problem. A first step in this direction will be to define a minimal set of instructions we need for our reconfigurations, detailed in the next section.

4.4 Transitions between mechanisms

A crucial point concerns the granularity of the reconfiguration. There are two obvious approaches for replacing a FTM with another one:

- an atomic approach: the old mechanism is replaced by the new one,
- a differential approach: transition operates at a finer level by adding/removing components corresponding to the difference between the two mechanisms.

The *distance* between the two mechanisms plays an essential part in the choice of the approach. The function for computing this measure is, intuitively, the sum of three basic distances, namely application assumptions, fault model complexity and number of components to be changed. Regarding application assumptions, a stateless deterministic application has an associated weight lower than a stateful non-deterministic application. Regarding fault model complexity, design patterns that tolerate crash faults have a lower complexity than design pattern that tolerate design faults. However, the most influencing parameter of this distance is the number of components to be changed between two component-based implementations of fault tolerance mechanisms.

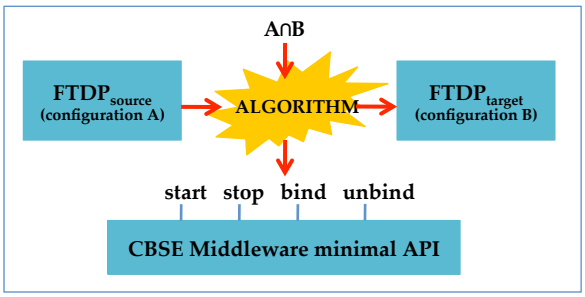

Fig. 4. Transition between FTMs using a minimal CBSE support

The transitions between mechanisms will most likely be described through state-machines. The algorithms behind the transitions could lead to design patterns for system evolution.

The transition algorithm is executed by the adaptation engine represented in Fig. 2 and based on a minimal interface enabling components to be manipulated at runtime (see Fig. 4). Our final aim is to define the minimal interface required to install a component at runtime within a software configuration. Such an interface consists of at least four operations to manipulate components and bindings/connections: stop, start, bind, unbind. It also needs customized, user-defined methods like setState and getState operations for stateful objects. The stop operation is certainly more subtle than the others: its semantics is "stop after termination of all operations in progress". This means that all input requests are queued while requests already in progress terminate.

Moreover, any algorithm that performs an online adaptation of a mechanism based on a CBSE middleware must rely on the notion of *transaction*: when a transition from a design pattern to another one is triggered, the different components that have to be replaced must be changed on an *all-or-nothing* approach, i.e., the set of basic replacements must be encapsulated in a global transaction — when all components are modified, then the modification can be committed; in case of a single failure, the whole transaction is rolled back.

5 Case Study

The detailed classification of our FTMs given in Fig. 3 is the basis for their adaptation[2]. Any final FTDP is associated to a software architecture, i.e. the FTM, ideally a component graph. Our final aim is to examine various transitions of the change model. This will consist in choosing a starting point, a destination point and several intermediate points, which all represent system state regions, associating the most adequate FTM to each one, and develop algorithms to perform safe transitions between mechanisms.

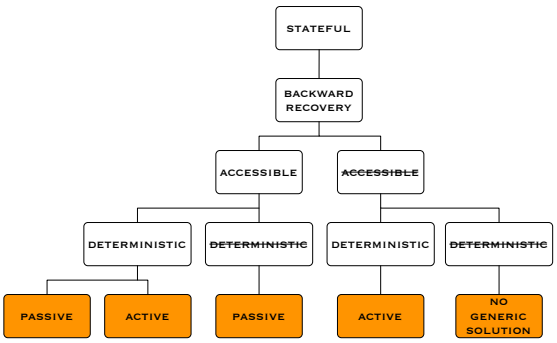

Fig. 5. DUPLEX/STATEFUL subtree

The FTDPs, more precisely their architectural description (component view) is the basis for a first analysis of the adaptation. Starting from one mechanism, called FTDP_{source}, the work consists in analyzing the changes to be performed to reach another mechanism, called FTDP_{target}. For each transition between FTDP_{source} and FTDP_{target}, an algorithm has to be defined off-line to perform the change. This algorithm thus takes two configurations derived from the design descriptions given as inputs and performs the operation using the API of a CBSE middleware providing control over the components' lifecycle and bindings.

As an illustration, let us consider two variants of duplex strategy, namely a passive replication design pattern as a FTDP_{source}, and an active replication design pattern as a FTDP_{target} (see Fig. 5). Passive replication is a checkpointing-based fault tolerance strategy for crash faults, with only one replica processing requests (PBR, *Primary Backup Replication* model). Active replication, in this case, is a semi-active fault tolerance strategy for crash faults, with both replicas processing requests and only one delivering output messages (LFR, *Leader-Follower Replication* model) (see [17]). Here, we consider simplified implementations of both strategies we developed in UML using RSA (*IBM Rational Software*

[2] Notice that the rightmost leaf indicates that an ad-hoc solution must be defined, relying on a roll-forward recovery approach, i.e., defining intermediate valid states in the computation from which the application can restart.

Architect) tool. The complete design has been created and the passive replication strategy is described by the class diagram shown in Fig. 6. We assume that objects can be mapped to components to enable easy manipulation at runtime. In the design provided here, we consider the following objects mapped to components: ProxyServer, Communication, Server (ServerPBR), DuplexProtocol, RemoteCall and the Client.

The proposed simplified design perfectly illustrates our method, i.e., it entails minimal modifications to execute a transition to an active replication strategy. The distance being small in this case, the transition corresponds to replacing the DuplexProtocol component implementing the PBR model with a DuplexProtocol component variant implementing the LFR model instead. The DuplexProtocol component performs the main operation of the passive replication strategy (run method), by getting the state of the server using the CaptureState method inherited from ServerPBR. The LFR variant of DuplexProtocol activates the requested server method and synchronizes with the follower.

With respect to the state of the protocol, i.e., either the DuplexProtocol is stateless or stateful, a transfer function must be activated to initialize the new DuplexProtocol component. In our case, to simplify implementation, we considered it stateless. Notice that this assumption greatly simplifies switching, as explained in [8].

6 Related works

As already mentioned in Section 1, one of our first areas of interest was Autonomic Computing (AC), more precisely self-healing systems, i.e., systems which are able to repair themselves. However, as [22] also points out, many of the "hot" issues within AC have been at the core of other disciplines, e.g., fault tolerant computing and artificial intelligence for a long time. The novelty lies in the "holistic aim" of regrouping all relevant research areas in this common project. Focusing on the intersection between AC and fault tolerance computing, which is our main research axis, the same author, in [23], reaches the conclusion 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).

A complex example of (re)configurable framework which also touches the areas of ubiquitous systems, AC and fault tolerant computing is Gaia [19], which builds on the core concept of *active spaces*. Heterogeneous devices, such as PCs, sensors, smart-phones, blend in the environment and interact with the user through a uniform interface. A method of adding an autonomic dimension to this framework is the use of a planner from the field of artificial intelligence, as presented in [18], which has the role of creating a path between the current state and a goal state given by a user or by a developer. This planner can be considered analogous to our transition algorithm. In [3], the authors add a fault tolerance dimension to Gaia by emphasizing the importance of dependability

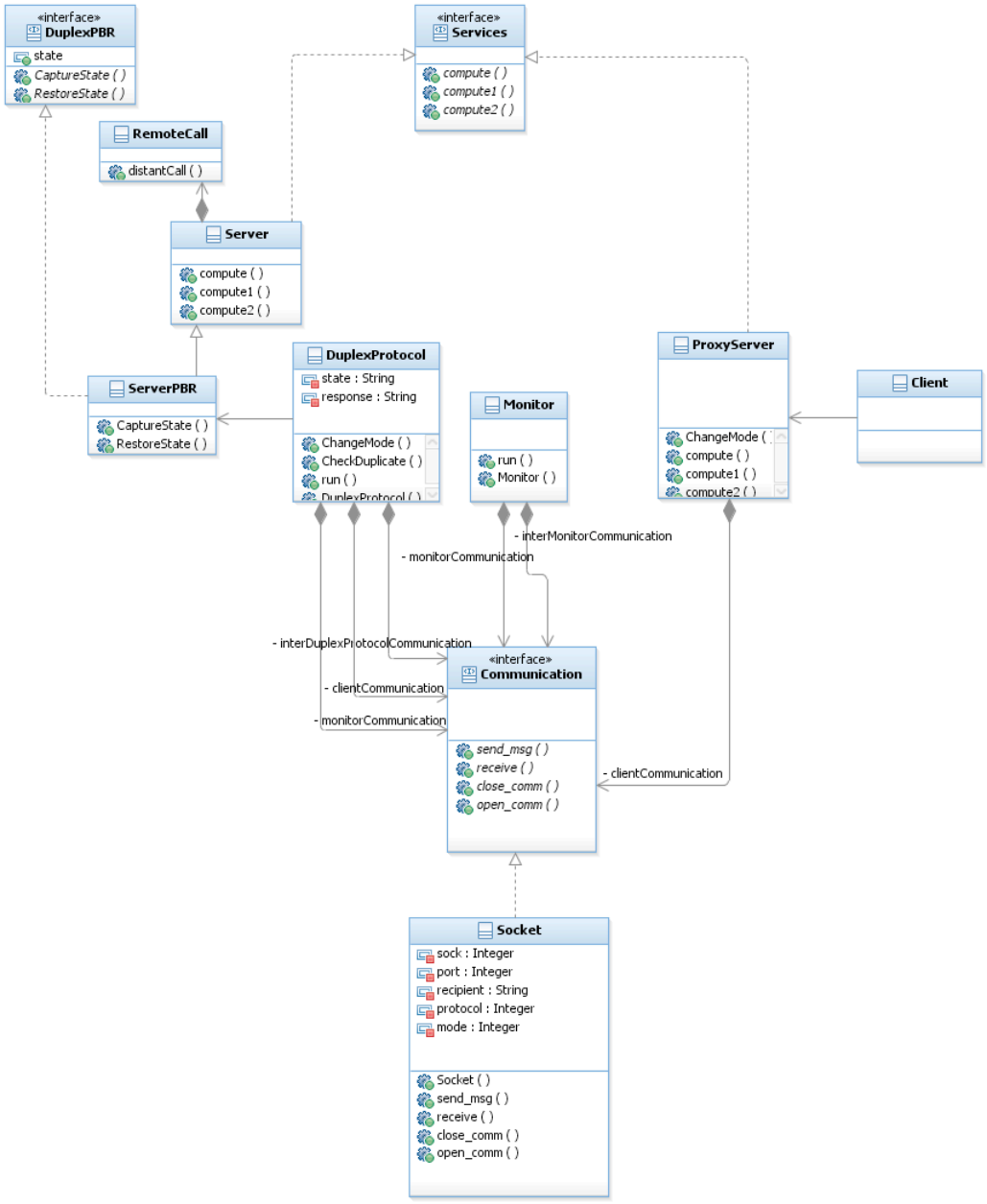

Fig. 6. UML class diagram for passive replication

in pervasive systems i.e. systems which interact closely with the users, possibly even in a healthcare context. They only endow Gaia with the property of tolerating fail-stop faults. Although this framework tries to cover issues from many areas including dependable computing, the proposed solution appears to be closely linked to an underlying operating system called 2K, the case studies focus only on managing media presentations with a basic support for fault tolerance. RUNES (Reconfigurable Ubiquitous Networked Embedded Systems) [5] is also a project addressing several of the areas mentioned above: it is an easily tailorable middleware component framework for resource-constrained systems (e.g., sensor networks) which aims at the reconfiguration of heterogeneous systems. The scenario they address in [4] involves fire management in a road tunnel that is instrumented with sensors and actuators communicating with devices such as PDAs belonging to firemen.

In the area of reconfigurable software architectures, RAINBOW [11] 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 englobes a very expressive ADL called ACME [10], already cited, a system in charge of verifying constraints, called ARMANI, a library of gauges, etc. The idea of placing an ADL on top of a CBSE middleware, as we intend to do, is the topic of [2], in which the authors describe their experience in associating an enriched version of ACME with the OpenCOM middleware for providing programmed and ad-hoc changes at runtime while maintaining certain constraints. Drawing the parallel between the entities described by the ADL and those provided and manipulated by the CBSE will also be an important step in our research.

Although there are a lot of available component-based platforms, not all of them are equally suitable for runtime reconfiguration and for dealing with self-healing. In [21], the authors identify the necessary runtime abstractions that a component model must include in order to efficiently support an autonomic adaptation service. What we need is not a component model which merely allows runtime reconfiguration but one which is designed for it. An example is [20] which builds, as RUNES, on the idea of a basic runtime kernel and a variety of additional services in the form of modules which can be plugged in. A fertile source of inspiration are the projects aimed at providing reconfiguration/adaptation properties to wireless sensor networks such as the works described in [24], [25] and [26] which present a middleware for wireless sensor networks and a component model enabling fine-grained adaptation.

The idea of creating design patterns for fault tolerance based on meta-level architectures and fault tolerance mechanisms is also described in [15]. [13] presents reflective design patterns implementing the separation of concerns as well as a

hierarchy of design patterns in the form of a tree-graph where each level acts as a refinement step.

7 Conclusion and perspectives

In this paper, we presented our approach to design and implement *resilient systems*, i.e., systems meant to cope with continuous evolution while guaranteeing dependability. Hence, our work lies at the intersection of three research domains, namely Fault Tolerance, Autonomic Computing and Ubiquitous Systems.

To that aim, we propose to represent various dependability-related attributes of the system in a three-axes space. Interestingly, this space allows us to link application and dependability-related context with fault tolerance strategies. More precisely, we classified classical fault tolerance patterns with regards to this space, and showed how every fault tolerance pattern defines a region in this space.

In this formalism, the evolution of the system can be represented as a path in this space, and evolution induces modifications of fault tolerance strategies, as derived from our classification of fault tolerance patterns.

To illustrate our method, we show, on a simple active to passive replication evolution, how we can provide an algorithmic solution of this evolution, based on a simplified UML description of such replication strategies.

To go further this simple example, we are currently investigating the possibility to automatically link fault tolerance design patterns (such as the one provided in UML) to component-based technologies (CBSE) for developing the management framework. Using such approach will allow us to define precise algorithms for adaptation, and will ease the development of a generic framework for adaptation of dependable systems.

References

1. Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1:11–33, January 2004.
2. T. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. *Software Architecture*, pages 1–17, 2005.
3. S. Chetan, A. Ranganathan, and R. Campbell. Towards fault tolerance pervasive computing. *Technology and Society Magazine, IEEE*, 24(1):38–44, 2005.
4. P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G.P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. The RUNES middleware for networked embedded systems and its application in a disaster management scenario. 2007.
5. P. Costa, G. Coulson, C. Mascolo, L. Mottola, G.P. Picco, and S. Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 14(2):149–162, 2007.
6. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems (TOCS)*, 26(1):1–42, 2008.

7. J-C. Fabre. Architecting dependable systems using reflective computing: Lessons learnt and some challenges. In *WADS'09*, pages 273–296, 2009.
8. J.C. Fabre, M.O. Killijian, and T. Pareaud. Towards On-line Adaptation of Fault Tolerance Mechanisms. In *EDCC*, pages 45–54. IEEE, 2010.
9. J.C. Fabre, T. Robert, and M. Roy. Early Error Detection for Fault Tolerance Strategies. In *RTNS*. IEEE, 2010.
10. D. Garlan, R.T. Monroe, and D. Wile. *Acme: Architectural description of component-based systems*. Cambridge University Press, 2000.
11. David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM.
12. J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
13. L. Lamour, F. Cecília, and M.F. Rubira. Reflective Design Patterns to Implement Fault Tolerance.
14. J-C. Laprie. From dependability to resilience. In *DSN, Anchorage, AK, USA, pp. G8-G9*, volume 8, 2008.
15. M.L.B. Lisbôa. A new trend on the development of fault-tolerant applications: software meta-level architectures. *J. of the Brazilian Computer Society*, 4(2), 1997.
16. P. Maes. Concepts and experiments in computational reflection. *ACM Sigplan Notices*, 22(12):147–155, 1987.
17. D. Powell. Distributed fault tolerance—Lessons learnt from delta-4. *Hardware and Software Architectures for Fault Tolerance*, pages 199–217, 1994.
18. A. Ranganathan and R.H. Campbell. Autonomic pervasive computing based on planning. 2004.
19. M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, pages 74–83, 2002.
20. D. Romero, G. Hermosillo, A. Taherkordi, R. Nzekwa, R. Rouvoy, and F. Eliassen. The DigiHome Service-Oriented Platform. *Software: Practice and Experience*, 2011.
21. S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In *Proceedings of the 30th international conference on Software engineering*, pages 101–110. ACM, 2008.
22. R. Sterritt. Autonomic computing. *Innovations in systems and software engineering*, 1(1):79–88, 2005.
23. R. Sterritt and D. Bustard. Autonomic Computing-a means of achieving dependability? In *Proc. IEEE Engineering of Computer-Based Systems*, 2003.
24. A. Taherkordi, F. Eliassen, R. Rouvoy, and Q. Le-Trung. ReWiSe: A New Component Model for Lightweight Software Reconfiguration in Wireless Sensor Networks. In *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*, pages 415–425. Springer, 2010.
25. A. Taherkordi, Q. Le-Trung, R. Rouvoy, and F. Eliassen. WiSeKit: A Distributed Middleware to Support Application-Level Adaptation in Sensor Networks. In *Distributed Applications and Interoperable Systems*, pages 44–58. Springer, 2009.
26. A. Taherkordi, R. Rouvoy, Q. Le-Trung, and F. Eliassen. Supporting lightweight adaptations in context-aware wireless sensor networks. In *Workshop on Context-Aware Middleware and Services/COMSWARE*, pages 43–48. ACM, 2009.