HAL
open science

Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS

Michaël Lauer, Matthieu Amy, Jean-Charles Fabre, Matthieu Roy, William Excoffon, Miruna Stoicescu

To cite this version:

Michaël Lauer, Matthieu Amy, Jean-Charles Fabre, Matthieu Roy, William Excoffon, et al.. Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS. HASE 2016 - IEEE 17th International Symposium on High Assurance Systems Engineering Symposium, Jan 2016, Orlando, FL, United States. pp.94-101, 10.1109/HASE.2016.30 . hal-01288098

HAL Id: hal-01288098

https://hal.science/hal-01288098v1

Submitted on 14 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS

Michael Lauer, Matthieu Amy, Jean-Charles Fabre, Matthieu Roy, William Excoffon, and Miruna Stoicescu[1]

LAAS-CNRS, Université de Toulouse,
CNRS, INPT, UPS, Toulouse, France

[1]Presently with ESOC/ESA, Darmstadt, Germany,
on behalf of GMV

***Abstract*—Systems are expected to evolve during their service life in order to cope with changes of various natures, ranging from fluctuations in available resources to additional features requested by users. For dependable embedded systems, the challenge is even greater, as evolution must not impair dependability attributes. Resilient computing implies maintaining dependability properties when facing changes. Resilience encompasses several aspects, among which evolvability, i.e., the capacity of a system to evolve during its service life. In this paper, we discuss the evolution of systems with respect to their dependability mechanisms, and show how such mechanisms can evolve accordingly. From a component-based approach that enables to clarify the concepts, the process and the techniques to be used to address resilient computing, in particular regarding the adaptation of fault tolerance (or safety) mechanisms, we show how Adaptive Fault Tolerance (AFT) can be implemented with ROS. Beyond implementation, we draw the lessons learned from this work and discuss the limits of this runtime support to implement such resilient computing features in embedded systems.**

I. Introduction

Evolution during service life is inevitable in many systems today. A system that remains dependable when facing changes (new threats, change in failures modes, updates of applications) is called resilient. The persistence of dependability when facing changes is called resilience [1]. *Resilient computing* encompasses several aspects, among which evolvability, i.e., the capacity of a system to evolve during its service life. On the other hand, dependability relies on fault-tolerant computing at runtime, enabled by fault tolerance mechanisms (FTMs) attached to the application. As such, one of the key challenges of resilient computing is the capacity to adapt the FTMs attached to an application during its operational life.

One important aspect of a dependable system design is the definition of the fault model. This fault model considers both hardware and software faults may lead to failure modes that impair the correct behavior of the system. In critical systems, such failure modes may violate safety properties. The role of the safety analysis (e.g. using the FMECA method) is to identify the failure mode and then define the safety mechanisms to prevent the violation of safety properties. Such safety mechanisms rely on basic error detection and recovery mechanisms, namely fault tolerance techniques following Laprie's terminology. Such safety mechanisms are based on Fault Tolerance Design Patterns that can be combined together. The safety analysis is often done a priori according to the fault model that had been defined.

During the operational life of the system, several situation may occur. New threats may lead to revise the fault model (electromagnetic perturbations, obsolescence of HW components, software aging, etc.). A revision of the fault model has consequences on the fault tolerance mechanisms to be used. In other words, the validity of the fault tolerance mechanisms of safety mechanisms (whatever you want to call them) depends on the representativeness of the fault model. In a certain sense, a bad choice of the fault model may lead to pay for useless mechanisms in both normal operation and erroneous situations. This has an obvious side effect on the performance and on the dependability measures (reliability, dependability) respectively. This means that a change in the definition of the fault model implies a change in the fault tolerance mechanisms.

Beyond the fault model, there are other sources of changes.

Resources changes may also impair some safety mechanisms that rely on hardware resources. A typical example is the lost of processing units, but simply a loss in networks bandwidth may invalidate some fault tolerance mechanisms from a timing viewpoint.

Application changes are more and more frequent during the operational lifetime. This is obvious for many conventional applications (e.g. mobile phones) but it is becoming also needed for more critical embedded systems. This is the case for long living systems like space or avionics systems, but also in the automotive domain, not only for maintenance purposes but also of commercial reasons. The evolution of the specification during the lifetime of a system is a fact; it follows the evolution of the user requirements or needs. The notion of versioning (updates) or the loading of additional features (upgrades) may lead to change the assumptions on top of which the implementation of FT mechanisms rely. Such change implies revisiting the FMECA spreadsheets but also the implementation of the FT mechanisms. Some FT mechanisms rely on strong assumptions regarding the behavior of the application, and everybody knows in the dependability community the importance of the coverage of such assumptions [16].

As a conclusion, the safety mechanism must remain compliant with all assumptions in terms of fault model, resources and application characteristics during the whole lifetime of the system. Their efficiency relies on this statement.

In this paper, we first motivate the issue and then report on an approach taking advantage of Component Based Software Engineering technologies for tackling this crucial aspect of

resilient computing, namely the adaptation of fault tolerance mechanisms. We defined a minimal runtime support for implementing adaptive fault tolerance. The second part of this paper shows how this minimal runtime support can be implemented on ROS, presently used in many applications (robotics applications, automotive applications like ADAS, or military applications). We illustrate the mapping of ideal components to ROS components and give implementation details of a fault tolerance design pattern that is adaptive at runtime. We finally draw the lessons learnt from our first experiments, discuss the limits of the exercise, and identify some promising directions.

In Section II we present the problem statement, and then summarize our CBSE-based approach for adaptive fault tolerance in Section III. A full account of this approach can be found in [13]. The mapping of this approach to ROS is described in Section IV. The lessons learnt are given in Section V before concluding.

II. PROBLEM STATEMENT

The need for *Adaptive Fault Tolerance* (AFT) rising from the dynamically changing fault tolerance requirements and from the inefficiency of allocating a fixed amount of resources to FTMs throughout the service life of a system was stated in [2]. AFT is gaining more importance with the increasing concern for lowering the amount of energy consumed by cyber-physical systems and the amount of heat they generate [3]. For Dependable systems that cannot be stopped for performing off-line adaptation, on-line adaptation of FTMs has attracted research efforts for some time now. However, most of the solutions [4], [5], [6] tackle adaptation in a preprogrammed manner: all FTMs necessary during the service life of the system must be known and deployed from the beginning and adaptation consists in choosing the appropriate execution branch or tuning some parameters, e.g., the number of replicas or the interval between state checkpoints. Nevertheless, predicting all events and threats that a system may encounter throughout its service life and making provisions for them is impossible. The use of FTMs in real operational conditions may lead to slight updates or unanticipated upgrades, e.g., compositions of FTMs that can tolerate a more complex fault model than initially considered.

In both aeronautical and automotive systems, the ability to perform remote changes for different purposes is essential: maintenance but also updates and upgrades of embedded applications. The remote changes should be partial as it is unrealistic to reload completely an processing unit from small updates. This idea is recently promoted by some car manufacturers like Renault, BMW but also TESLA Motors in the USA stating in its website *"Model S regularly receives over-the-air software updates that add new features and functionality"*. It is important to mention that performing remote changes will become very important for economic reasons, for instance selling options a posteriori since most of the evolution in the next future will rely on software for the same hardware configuration (sensors and actuators). In addition to this, the X-to-X applications (X being cars or planes) will imply rapid adaptation of onboard software to remain consistent with the network of X.

We propose an alternative to preprogrammed adaptation that we denote *agile adaptation of FTMs*. The term "agile" is inspired from agile software development [7] that emphasizes the importance of accommodating change during the lifecycle of an application at a reasonable cost, rather than striving to anticipate an exhaustive set of requirements. Agile adaptation of FTMs enables systematic evolution: according to runtime observations of the system and of its environment, new FTMs can be designed off-line and integrated on-line in a flexible manner, with limited impact on the existing software architecture.

Evolvability has long been a prerogative of the application business logic. A rich body of research exists in the field of software engineering consisting of concepts, tools, methodologies and best practices for designing and developing adaptive software [8]. Consequently, our approach for the agile adaptation of FTMs leverages advancements in this field such as Component-Based Software Engineering (CBSE) technologies [9], Service Component Architecture [10] and Aspect-Oriented Programming [11].

The basic idea is the following. Fault Tolerance or Safety Mechanisms are developed as a composition of elementary mechanisms, e.g. basic design patterns for fault tolerance computing.

Using such concepts and technologies, **we design FTMs as "Lego"-like brick-based assemblies** that can be methodically modified off-line or at runtime through fine-grained changes affecting a limited number of bricks. This is the basic idea of our approach that maximizes reuse and flexibility, contrary to monolithic replacements of FTMs found in related work, e.g., [4], [5], [6].

However, most of software runtime supports used in embedded systems today do not rely on CBSE concepts. AUTOSAR, for instance, relies on very static system engineering concepts and does not provide today much flexibility [12]. A new approach is of interest today enabling remote updates to be carried out, including for safety mechanisms.

ROS is a middleware for robotics applications (e.g. *Robonaut 2* from NASA within the ISS) but also used in industry, the automotive industry for instance. This middleware provides a *weak* component approach. It is open-source, its user community is very large and it is used for critical application e.g. at NREC (The *National Robotics Engineering Center* in Pittsburgh) for unmanned military vehicles (e.g. the *Crusher*).

III. ADAPTIVE FAULT TOLERANCE

A. Basic concepts for AFT

Some basic concepts must be discussed to address the problem of Adaptive Fault Tolerant computing. Three essential concepts must be discussed beforehand:

- *Separation of concerns*: this concepts is now well known, it implies a clear separation between the functional code, i.e. the application, and the non-functional code, the fault tolerance mechanisms in our

case. The connection between the application code and the FTM must be clearly defined as specific connections. This means that the FTMs can be disconnected and replaced by a new one provided the connectors remains the same.

- *Componentization*: this concepts means that any software components can be decomposed into smaller components. Each component exhibit interfaces (services provided) and receptacles (services required). This means that any FTMs can be decomposed into smaller pieces, and conversely that an FTM is the aggregation of smaller ones. The ability to manipulate the binding between components (off-line but also on-line) is essential for AFT.
- *Design for adaptation*: the adaptation of software systems imply that (i) the software itself has been analyzed with adaptation in mind for later evolution using componentization (although all situations cannot be anticipated) and (ii) designed to simplify their adaptation including from a programming viewpoint (e.g. using object-oriented, aspect-oriented programming concepts).

Such basic concepts have been established and validated through various steps of analysis of fault tolerance design patterns and after several design and implementation loops, as discussed in [17].

B. Change Model

The choice of an appropriate fault tolerance mechanism (FTM) for a given application depends on the values of several parameters. We consider three classes of parameters: 1) fault tolerance requirements (FT); 2) application characteristics (A); 3) available resources (R).

We denote (FT, A, R) as *change model*. At any point in time, the FTM(s) attached to an application component must be consistent with the current values of (FT, A, R).

The three classes of parameters enable to discriminate FTMs. Among fault tolerance requirements *FT*, we focus, for the time being, on the fault model that must be tolerated. Our fault model classification is based on well-known types [14], e.g., crash faults, value faults, development faults. In this work, we focus on hardware faults but the approach can be extended to FTMs that target development faults.

The application characteristics *A* that we identified as having an impact on the choice of an FTM are: application statefulness, state accessibility and determinism. We consider here that the FTMs are attached to a black-box application. This means there is no possibility to interfere with its internals, for tackling non-determinism, for instance, in case an FTM only works for deterministic applications. Resources *R* play an important part and represent the last step in the selection process. FTMs require resources such as bandwidth, CPU, battery life/energy. In case more than one solution exists, given the values of the parameters FT and A, the resource criterion can invalidate some of the solutions. A cost function can be associated to each solution, based on R.

Any parameter variation during the service life of the system may invalidate the initial FTM, thus requiring a transition towards a new one. Transitions may be triggered by new threats, resource loss or the introduction of a new application version that changes the initial application characteristics. A particularly interesting adaptation trigger is the fault model change. Incomplete or misunderstood initial fault tolerance requirements, environmental threats such as electromagnetic interferences or hardware aging may change the initial model to a more complex one.

In this work, we consider that a monitoring service provides accurate information on the resource usage and the error rate in operation through logs analysis (both at runtime and off-line). The monitoring service providing the triggers for FTM adaptation is out of the scope of this paper. Some triggers can also come from the system manager when an update is done at the application level.

C. FT Design Patterns and Assumptions

To illustrate our approach, we consider some fault tolerance design patterns (FTMs) and discuss their underlying assumptions and resource needs. Any change that invalidates an assumption or implies an unacceptable resource change calls for an update of the FTMs.

Duplex protocols tolerate crash faults using passive (e.g. *Primary-Backup Replication* denoted PBR), or active replication strategies (e.g. *Leader-Follower Replication* denoted LFR). In this case, each replica is considered as a *self-checking* component, the error detection coverage is perfect. The fault model includes hardware faults or random operating system faults (no common mode faults). At least 2 independent processing units are necessary to run this FTM.

Two design patterns tolerating transient value faults are briefly discussed here. *Time Redundancy* (TR) tolerates transient physical faults or random runtime support faults using repetition of the computation and voting. This is way to improve the self-checking nature of a replica, but it introduces a timing overhead. *Assertion&Duplex* (A&D) tolerates both transient and permanent faults. It's a combination of a duplex strategy with the verification using assertions of safety properties derived from safety analysis that could be violated by a value fault or by a random runtime support error.

Assumptions / FTM		PBR	LFR	TR	A&D
Fault Model (FT)	crash	✓	✓		✓
	transient			✓	✓
Application behaviour (A)	Deterministic		✓	✓	(✓)
	State access	✓			(✓)
Resources (R)	Bandwidth	high	low	nil	(TDB)
	# CPU	2	2	1	2

Fig. 1. Assumptions and fault tolerance design patterns charateristics

The underlying characteristics of the considered FTMs, in terms of (FT, A, R), are shown in Fig. 1. For instance, PBR and LFR tolerate the same fault model, but have different A and R. PBR allows non-determinism of applications because only the Primary computes client requests while LFR only works for deterministic applications as both replicas compute all requests. LFR could tackle non-determinism if the application was not considered a black box, as in our approach. PBR requires state

access for checkpoints and higher network bandwidth (in general), while LFR does not require state access but generally incurs higher CPU costs (and, consequently, higher energy consumption) as both replicas perform all computations.

During the service life of the system, the values of the parameters enumerated in Fig. 1 can change. An application can become non-deterministic because a new version is installed. The fault model can become more complex, e.g., from crash-only it can become crash and value fault due to hardware aging or physical perturbations. Available resources can also vary, e.g., bandwidth drop or constraints in energy consumption. For instance, the PBR→LFR transition is triggered by a change in application characteristics (e.g. inability to access application state) or in resources (bandwidth drop), while the PBR→A&D transition is triggered by a change in the considered fault model (requiring a safety mechanism extension). Transitions can occur in both directions, according to parameter variation.

D. Design for adaptation of FTMs

Our *"design for adaptation"* aims at producing reusable elementary components that can be combined to implement a given FTM or safety mechanism. Any FTM follows the generic *Before-Proceed-After* meta-model. Many FTMs can be mapped and combined using this model, as shown in Fig. 2.

FTM	Before	Proceed	After
PBR (primary)		Compute	Checkpointing
PBR(backup)			State update
LFR (leader)	Forward request	Compute	Notify
LFR (follower)	Handle request	Compute	Handle notification
TR	Save/restore state	Compute	Compare
A&D		Compute	Assert

Fig. 2. Generic execution scheme for FT design patterns

Composition implies nesting the *Before-Proceed-After* meta-model. This approach improves flexibility, reusability, composability and reduces development time. Updates are minimized since just few components have to be changed.

E. Runtime support

The software runtime support must provide key features to manipulate the component graph. Any application or FTM is a graph of components at runtime. From previous experiments reported in [17], the following primitive are required.

- Dynamic creation, deletion of components;
- Suspension, activation of components;
- Control over interactions between components for the creation and the removal of connections (bindings);

Our first implementation was done on a reflective component-based middleware, FRASCATI [14] providing a scripting language to manipulate the component graph, FScript [15]. The proposed approach is reproducible on any other support that provides these features.

IV. Adaptive Fault Tolerance on ROS

A. Introduction to ROS

ROS is described as[1]: … *an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly used functionality, message passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.* ROS is a middleware running on top of a Unix-based operating system (typically Linux). The main goal of ROS is to allow the design of modular applications: a ROS application is a collection of programs, called nodes, interacting only through message passing. Developing an application involve the assembly of nodes, which is akin to component-based approaches. Such an assembly is referred to as the software computation graph.

B. Component model and reconfiguration

Two communication models are available in ROS: a publisher/subscriber model and a client/server one. The publisher/subscriber model defines one-way, many-to-many, asynchronous communications through the concept of topic. When a node publishes a message on a topic, it is delivered to every nodes subscribing to this topic. Note that a publisher is not aware of the nodes subscribing to its topic. The client/server model defines bidirectional transaction (one request/one reply) synchronous communications through the concept of service. A node providing a service is not aware of the client nodes that may use its service. These high-level communication models allow adding, replacing or deleting nodes in a transparent manner, either off-line or on-line.

To provide this level of abstraction, each ROS application includes a special node called the ROS Master. It provides registration and lookup services to the other nodes. All nodes register their services and topics to the ROS master. It is the only node that has a comprehensive view of the computation graph. When a node issues a service call, it queries the master for the address of the node providing the service and then it sends its request to this address.

In order to be able to add fault-tolerance mechanisms to an existing ROS application in the most transparent manner, we need to implement interceptors. An interceptor provides a means to insert functionality, such as safety or monitoring nodes, into the invocation path between two ROS nodes. To this end, a relevant ROS feature is its remapping capability. When a node is launched, it is possible to reconfigure the name of any services or topics it is using. Thus, requests and replies between nodes can be rerouted to interceptor nodes.

C. Implementing a componentized FT design pattern

1) Generic Computation Graph: We identified a generic pattern for the computation graph of a FTM. Figure 3 shows its application in the context of ROS. Node *Client* uses a service provided by *Server*. The FTM computation graph is inserted

1 http://wiki.ros.org/ROS/Introduction

between the two thanks to the ROS remapping feature. Since *Client* and *Server* must be re-launched for the remapping to take effect, the insertion is done offline. The FTM nodes, topics, and services are generic for every FTM discussed in section II. Implementing a FTM consist in specializing the *before*, *proceed*, and *after* nodes with its corresponding behavior (see Fig. 3).

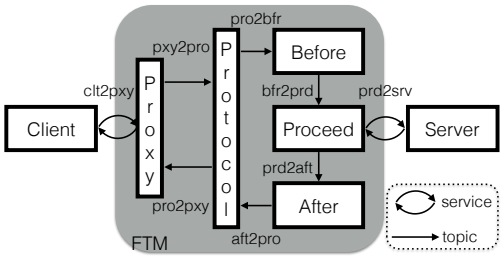

Fig. 3. Generic computation graph for FTM

We illustrate the approach, through a Primary-Backup Replication (PBR) mechanism added to the Client/Server application in order to tolerate a crash fault of the Server. Fig. 4 presents the associated architecture. Three machines are involved: the *Client*, which is also hosting the ROS, master, the *MASTER* site hosting the primary replica and the *SLAVE* site hosting the backup replica. For the sake of clarity, the symmetric topics and services between *MASTER* and *SLAVE* are not represented. Elements of the slave are suffixed with "_S".

We present the behavior of each node, the topics/services used through a request/reply exchange between a node *Client* and node *Server* (see Fig. 4).

- *Client* sends a request to *Proxy* (service *clt2pxy*);
- *Proxy* adds an identifier to the request and transfers it to *Protocol* (topics *pxy2pro*);
- *Protocol* checks whether it is a duplicate request: if so, it sends directly the stored reply to *Proxy* (topics *pro2pxy*). Otherwise, it sends the request to *Before* (service *pro2bfr*);
- *Before* transfers the request for processing to *Proceed* (topics *bfr2prd*); no action is associated in the PBR case, but for other duplex protocol, *Before* may synchronize with the other replicas;
- *Proceed* calls the actual service provided by *Server* (service *prd2srv*) and forwards the result to *After* (topics *prd2aft*);
- *After* gets the last result from *Proceed*, captures *Server* state by calling the state management service provided by the server (service *aft2srv*), and builds a checkpoint based on this information which it sends to node *After_S* of the other replica (topics *aft2aft_S*);
- *Protocol* gets the result (topics *aft2pro*) and sends it to *Proxy* (topics *pro2pxy*);
- On the backup replica, *After_S* transfers the last result to its protocol node *Proto_S* (topics *aft2pr_S*) and sets the state of its server to match the primary.

In parallel with request processing, the node crash detector on the *MASTER* (noted CD) periodically gives a proof of life to the crash detector (CD_S) on the *SLAVE* to assert its liveliness (topics *CD2CD_S*). If a crash is detected, then the slave crash detector notifies the recovery node (topics *CD_S2rcy*). This node has two purposes: (1) in order to enforce the fail-silent assumption, it must ensure that every node of the *MASTER* are removed; (2) it switches the binding between the *Client* proxy and the *MASTER* protocol to the *SLAVE* protocol. Thus, the *SLAVE* will receive the *Client's* requests and will act as the *Primary*, changing its operating mode.

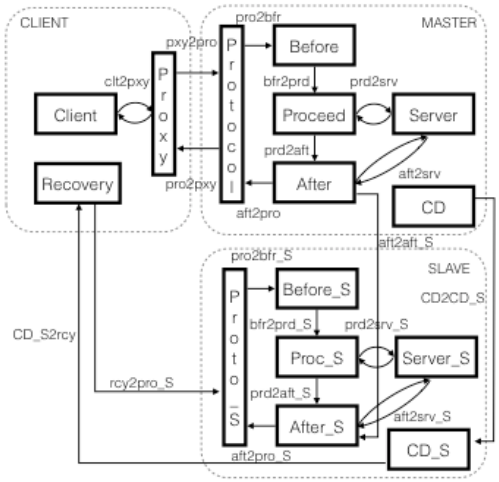

Fig. 4. Computation graph of a PBR mechanism

ROS does not provide APIs to dynamically change bindings between nodes. The node developer must implement the transition logics. The *SLAVE protocol* spins waiting for a notification from *recovery* (topics *rcy2pro_S*). This is done using the ROS API: background threads, within a node, check for messages independently of the node's main functionality. Upon reception of this topic, *protocol* subscribes to topic *pxy2pro* and publishes to topic *pro2pxy*. After this transition, the proxy forwards the *Client*'s requests to the *Slave* protocol.

2) Impact on the existing application: From the designer viewpoint, there are two changes required to integrate a FTM computation graph to its application. First, Client will have to be remapped offline to call the *proxy* node's service instead of directly the Server. Second, state management services, to get and set the state of the node, must be integrated to the *Server*. Form an object-oriented viewpoint any server inherits from an abstract class *stateManager* providing two virtual methods, *getState* and *setState*, overridden during the server development.

D. Adaptation of FT mechanisms

The previous PBR example validates the implementation of our design on ROS. However, our main goal is to provide safe and agile adaptation of FTMs in order to improve the resilience of the system. We illustrate our approach by adapting the previous example in response to a change in the fault tolerance requirement (change in the FT dimension of the change model). For instance, suppose that we need to adapt the current mechanism in order to tolerate both crash and

transient faults. Since PBR mechanism already takes care of the crash fault, we propose to compose it with a Time Redundancy (TR) mechanism to add the transient fault-tolerance capability. With our approach, three steps are required: (1) designing a stand-alone TR mechanism; (2) composing the PBR and TR mechanisms; and finally (3) installing the composite FTM.

1) Design of the TR mechanism:

Our generic design pattern isolates the specific behavior of a FTM within the *Before*, *Proceed*, and *After* nodes. Thus we can fully reuse the Proxy and Protocol nodes, already developed for PBR mechanism, in the new mechanism. This decrease significantly the amount of work required. The computation graph of the stand-alone TR mechanism is given in Fig. 5. Note that the server offers the same interfaces as before and thus does not require any adaptation. This aligns well with our separation of concern objective. We summarize the behavior of the components specific to the TR mechanism:

- *Before* receives a new request and retrieve the state of the *Server*;
- The request is forwarded to *Proceed*, which call the *Server's* service. The reply is transferred to *After*;
- *After* stores this reply and synchronize with *Before*;
- *Before* restores the state of the *Server* and forward again the initial request to *Proceed* which in turn call the *Server*. The new reply is transferred to *After*;
- *After* compares the two replies. If they are identical one of them is sent to *Protocol*, everything is executed a third time. If two replies among three are identical, then one of them is return to *Protocol*, otherwise an exception is raised

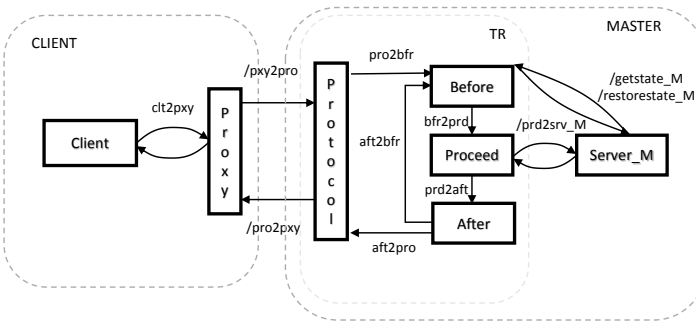

Fig. 5. Computation graph of a TR mechanism

2) Composition of mechanisms:

The generic computation graph for FTM is designed for composability. With respect to request processing, a Protocol node and a Proceed node present the same interfaces: a request as input, a reply as output. Hence, a way to compose mechanisms is to replace the Proceed node of a mechanism by a Protocol and its associated Before/Proceed/After nodes, as shown in Fig. 6. Our approach enables developing a new mechanism on the foundation of several existing ones. This improves the development time and the assurance in the overall system, since all mechanisms have been validated off-line by test and fault injection techniques. It is worth noting here that composition of mechanisms is also validated off-line in accordance with standard development processes (ISO26262, DO178C) to comply with certification if needed.

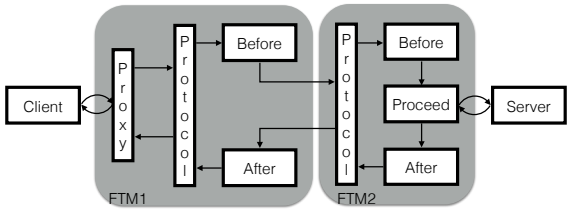

Fig. 6. Composition principle of FT mechanisms.

The architecture of the composite FTM made of PBR and TR is given in Fig. 7.

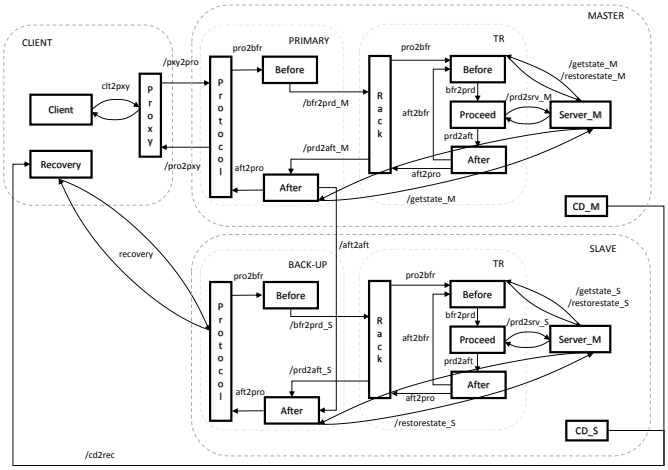

Fig. 7. Composition of PBR and TR mechanisms.

3) Installation of the mechanism

Installing an FTM within a ROS application or adapting an existing FTM does not incur technical difficulties as long as the system's nodes (application + FTM) can be stopped and re-launched. Indeed, using the remapping capability of ROS implies rewriting some configuration files, which are taken into account only during the initialization of the nodes. For system where interruption of service is not an option, adaptation has to be done at runtime. In the context of ROS, this requires some additional software development. This may not be ideal with respect to the availability of the system. As presented in the following, adaptation at runtime minimizing the disruption of the service is more demanding.

E. Dynamic Adaptation of FTM

Dynamic adaptation of FTM is required to provide continuity of service in resilient systems. The question is then: is it possible to safely adapt a FTM at runtime in the context of ROS? A set of minimal API required to guarantee the consistency of the transition between two different FTMs has been established in previous work [14]:

- Control over components *life cycle* at runtime (add, remove, start, stop).
- Control over *interactions* between components at runtime, for creating or removing bindings.

Furthermore, ensuring consistency before, during and after reconfiguration, requires that no requests or replies are lost:

- Components are stopped in a quiescent state, i.e. when all internal processing has finished
- Incoming requests on stopped components must be buffered

With the exception of *add* and *remove*, ROS does not provide these APIs. However, these APIs can be emulated with dedicated logics in some nodes. For instance, we are using some binding control in the Primary to Backup switch described in our example. Controlling node lifecycle is more complex but can be done in the same manner and these principles can be applied in the context of dynamic adaptation, i.e. add new nodes at runtime and binding them in the computation graph.

The *protocol* node plays a central part to provide proper consistency during a transition. Indeed, our design pattern for FTM is such that only stateless nodes, namely *before*, *proceed* and *after,* need to change in order to switch from one FTM to the next. Thus, *protocol* does not need to be changed during a transition and it can be used to buffer messages and detect when the changing nodes are in quiescent state. To do this, *protocol* is extended to deal with three new messages. The first one is used to signal *protocol* that a transition is about to happen and it has to start storing incoming requests. The second one is published by *protocol* and confirms that the FTM is in a safe state and transition can be safely executed. In particular, the safe state is reached when *protocol* has received the replies of all pending requests. The third message is used to signal *protocol* that the transition has been executed and it can resume normal operation and release the requests stored during the transition.

Note that the described transition technique requires that an FTM is already in place in the system, meaning that the Client and the Server are already configured to use our proxy nodes. Installing an FTM in an application without interruption is not possible with ROS since control over binding at runtime is only possible with dedicated code within the nodes.

V. LESSONS LEARNT

ROS is a candidate for embedded applications in automotive systems. As already said in introduction, it is currently investigated by Renault for implementing ADAS and also used by NREC and BMW for embedded applications.

In this paper, we analyze the use of ROS for embedded applications through a different angle. We consider critical applications that need to be modified during the lifetime of the system. This may be due to many events such as versioning, configuration changes, new threats and/or evolution of the fault model due to hardware aging or environment changes.

The question key is: to what extent safety mechanisms attached to critical applications can be adapted with ROS?

As far as adaptation is concerned, ideally, several requirements must be considered. A first requirement is *separation of concerns*, at design time but also at runtime. The second requirement relates to the capability to update, compose basic building blocks to realize/customize a safety mechanism: the key concept here is *componentization* thanks to CBSE approaches. The third requirement relates to the runtime support (OS and/or middleware) that should enable *component mapping to tasks* at runtime. The fourth relates to the *dynamic binding* between components at runtime to manipulate dynamically connections. Last but not least, a fifth requirement relates to the *control over components* corresponding to functions like activate, suspend, etc.

A. Concepts for adaptation

The 5 concepts mentioned above will be used here as a set of criteria to judge ROS with respect to the objective expressed in the above question. The first 3 criteria concern off-line adaptation: (i) separation of concern, (ii) componentization, (iii) component mapping to tasks. The last 2 criteria relates to the dynamic adaptation of the software on-line: (iv) dynamic binding and (v) control over components.

The design for adaptation issues addressed in section III.A is out of the scope of this analysis since it relates to the development process for software adaptation and not to the runtime support. Based on former work [17], safety mechanisms or FTMs can be developed as a collection of building blocks, i.e. elementary FT design patterns. Such FT building blocks can then be composed together to realize a new safety mechanism during the system lifetime. From a runtime viewpoint, such building blocks are aggregated with our framework *before-proceed-after*.

1) Off-line adaptation

Separation of concern can be achieved with ROS, since an application is implemented as a collection of nodes, some implementing functional aspects other fault tolerance aspects. We have shown that for a given application server, a replication protocol can be implemented as a set of nodes. The composition of different mechanisms has been shown as well. The connection between mechanisms relies on the notion of topic and promotes the *publish-subscribe* interaction model. As far as dependability is concerned, the separation of concern is not always possible for all dependability mechanisms, some being either intrinsically embedded into the code (CRC, defensive programming, exception handling…) and/or application dependent. ROS cannot be criticized from this viewpoint; it provides separation of concern for coarse grain FT mechanisms (e.g. replication strategies). More fine grain control over code execution requires more reflective features at the OS/interpreter level.

Componentization has two facets, namely at design time and at runtime. At design time, the solution resides on the use of CBSE techniques during the development process. At runtime it is related to the runtime model provided by ROS. Again the notion of node is essential. Our componentization finally corresponds at runtime to *before*, *proceed* and *after* components. In our experiments these components are simple, their mapping to task at this level is clearly possible as shown.

Mapping to tasks become then a natural consequence of the ROS runtime model. Any component in the design can be

mapped to a node, which is a task in practice. This is has been illustrated in the experiments carried out and reported in this paper. The mapping of components to nodes has also a major benefit; it provides an error confinement area for the components (single or composite). A ROS node corresponds to a Unix process, so it is associated to a protected memory space. *Time and Space Partitioning* (TSP) is a very attractive concept as far as dependability is concerned (see. ARINC 653 for avionics real-time systems), but this is not provided by ROS. Indeed, a middleware cannot provide this kind of facility that belongs to operating system kernels.

2) On-line adaptation

Dynamic binding between components at runtime is essential to manipulate an application composed as a set of nodes. Fine-grain updates or composition of mechanisms implies modifications in the component graph, i.e. the graph of nodes at runtime that has been designed off-line. This is something difficult with ROS, although the publish-subscribe model could provide such facility. Topics correspond to pre-defined communication channels between nodes that cannot be manipulated and changed on-line.

Control over components is also an important aspect of dynamic adaptation. New components must be created while others must be removed. More importantly component must be suspended in quiescent state before being removed or updated. The consistency of the whole graph of nodes representing the application depends on the state of components. The quiescent state is defined as a state in which all requests processed by the node are terminated and that all incoming requests are buffered. In this state, a component can be removed or updated. ROS does not provide this kind of facility that can be found in reflective component based middleware like FraSCAti [14].

B. Practical aspects

From a practical standpoint, ROS is easy to use. Tutorial can be found and, as an open source middleware, the community is present and very reactive. The use of ROS is simple, as it is a library that can be used in C/C++ programs.

The development of application as a set of nodes and topics as communication channels is easy. Some functionalities are however only available in some languages (e.g. Python) and the documentation is sometime weak. The inter-node communication is fine but static. It could be extended with the notion of dynamic topics.

The centralized and essential role of the *ROS Master* is an issue but this could be alleviated using conventional fault tolerance techniques to address such single point of failure.

VI. Conclusion

The adaptation of embedded application required an adequate runtime support. Beyond design for adaptation issues that relate more to the development process, the runtime support must fulfill 5 requirements: (i) separation of concern, (ii) componentization, (iii) component mapping to tasks. The last 2 criteria relate to the dynamic adaptation of the software on-line: (iv) dynamic binding and (v) control over components.

ROS enables the 3 first requirements to be satisfied, but fails to provide the last two. Adaptation can be easily done off-line using a CBSE design approach. On-line adaptation is far more difficult as shown by our experiments.

As a runtime support for Resilient Computing, ROS in its current version is not a good candidate. ROS is a development platform to test concepts for adaptive fault tolerance off-line. The mapping of component to ROS is roughly simple. ROS is well suited for agile development processes. However, its capabilities for dynamic dependability are limited. Anyway, the insights gained with this work will help us to develop a suitable runtime support for Adaptive Fault Tolerance.

References

[1] J.-C. Laprie. From Dependability to Resilience. In 38th *IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN), 2008.

[2] K. H. K. Kim and T. F. Lawrence. Adaptive Fault Tolerance: Issues and Approaches. In *Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computing Systems*. IEEE, 1990, pp. 38–46.

[3] C. Krishna and I. Koren. Adaptive Fault-Tolerance for Cyber-Physical Systems. In *IEEE International Conference on Computing, Networking and Communications* (ICNC), 2013, pp. 310–314.

[4] J. Fraga, F. Siqueira, and F. Favarim. An Adaptive Fault-Tolerant Component Model. In 9th *Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, 2003, pp. 179–186.

[5] L. C. Lung, F. Favarim, G. T. Santos, and M. Correia. An Infrastructure for Adaptive Fault Tolerance on FT-CORBA. In 9th *International Symposium on Object and Component- Oriented Real-Time Distributed Computing*. IEEE, 2006.

[6] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards Adaptive Fault-Tolerance for Distributed Multi-Agent Systems. In 4th *European Research Seminar on Advances in Distributed Systems*, 2001, pp. 195–201.

[7] J. Highsmithand and A. Cockburn. Agile Software Development: The Business of Innovation. In *Computer*, vol. 34, no. 9, pp. 120–127, 2001.

[8] P. McKinley, S. Sadjadi, E. Kasten, and B. H. C. Cheng. Composing Adaptive Software. In *Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[9] C. Szyperski. Component Software: Beyond Object-Oriented Programming. In *Addison-Wesley Longman Publishing* Co., Inc., 2nd ed. Boston, MA, USA: 2002.

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

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97 Object-Oriented Programming*, pp. 220– 242, 1997.

[12] H. Martorell, J.-C. Fabre, M. Lauer, M. Roy and R. Valentin. Partial Updates of AUTOSAR Embedded Applications — To What Extent?. In *European Dependable Computing Conference (EDCC),* 2015, Paris, France.

[13] M. Stoicescu, J.-C. Fabre, M. Roy. From Design for Adaptation to Component-Based Resilient Computing. In *PRDC* 2012: 1-10

[14] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. In *SP&E*, 2011.

[15] M. Leger, T. Ledoux, and T. Coupaye. Reliable Dynamic Reconfigurations in a Reflective Component Model. In *13th International Conf. on Component-Based Software Engineering*, 2010.

[16] D. Powell. Failure Mode Assumption and Assumpion Coverage. In *Proc. of the IEEE Int. Symp. on Fault-Tolerant Computing* (FTCS-22), Boston (USA), 1992, pp.386-395. (revised in the book Predictably Dependable Computing Systems, ISBN 3-540-59334, 1995.)

[17] M. Stoicescu. Architecting Resilient Computing Systems: A Component-based Approach. In *PhD thesis, National Polytechnic Institute of Toulouse (INP)*, 2013. www.theses.fr/en/2013INPT0120.