HAL
open science

Resilient Computing on ROS using Adaptive Fault Tolerance

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.. Resilient Computing on ROS using Adaptive Fault Tolerance. Journal of Software: Evolution and Process, John Wiley & Sons, Ltd., 2018, 30 (3), pp.e1917. 10.1002/smr.1917 . hal-01703968

HAL Id: hal-01703968

https://hal.laas.fr/hal-01703968

Submitted on 16 Feb 2018

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.

Resilient Computing on ROS using Adaptive Fault Tolerance

Michaël Lauer[1] *, Matthieu Amy, Jean-Charles Fabre[2], Matthieu Roy, William Excoffon and Miruna Stoicescu[3]

LAAS-CNRS, Université de Toulouse, CNRS, [1]*UPS,* [2]*INP, Toulouse, France*
[3]EUMETSAT, Darmstadt, Germany

SUMMARY

Computer-based systems are now expected to evolve during their service life in order to cope with changes of various nature, ranging from evolution of user needs, e.g., additional features requested by users, to system configuration changes, e.g., modifications in available hardware resources. When considering resilient embedded systems that must comply with stringent dependability requirements, the challenge is even greater, as evolution must not impair dependability attributes. Maintaining dependability properties when facing changes is, indeed, the exact definition of resilient computing.
In this paper, we consider the evolution of systems with respect to their dependability mechanisms, and show how such mechanisms can evolve with the system evolution, in the case of ROS, the Robot Operating System. We provide a synthesis of the concepts required for resilient computing using a component-based approach. We particularly emphasize the process and the techniques needed in order to implement an adaptation layer for fault tolerance mechanisms. In the light of this analysis, we address the implementation of Adaptive Fault Tolerance (AFT) on ROS (Robot Operating System) in two steps: firstly, we provide an architecture to implement fault tolerance mechanisms in ROS, and secondly, we describe the actual adaptation of fault tolerance mechanisms in ROS. Beyond the implementation details given in the paper, we draw the lessons learned from this work and discuss the limits of this run-time support to implement AFT features in embedded systems.

KEY WORDS: Adaptive fault tolerance; ROS; Resilience

1. INTRODUCTION

Evolution during service life is very frequent in many systems nowadays, including dependable systems. Such an evolution leads to modifications of the system software and hardware configuration. A challenge for the dependability community is to develop systems that remain dependable when facing changes (new threats, change in failures modes, application updates). The persistence of dependability when facing changes —defining the resilience of the system [1]— encompasses several aspects, among which evolvability is a key concept. Handling evolution involves new development processes, such as agile development methods, but also run-time supports that enable modifications at run-time.

At run-time, dependability relies on fault-tolerant computing, i.e., a collection of Fault Tolerance Mechanisms (FTMs) attached to the application according to its criticality level . In this context,

*Correspondence to: LAAS-CNRS 7, avenue du Colonel Roche BP 54200, 31031 Toulouse cedex 4, France

one of the key challenges of resilient computing is the capacity to adapt the FTMs attached to an application during its operational life.

In resilient systems, faults lead to failure modes that may violate dependability properties. The role of the safety analysis (e.g., using FTA, *Fault Tree Analysis*, or FMECA, *Failure Modes, Effects and Criticality Analysis*) is to identify the failure mode, the fault model 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, that are based on *Fault Tolerance Design Patterns* (FTDP) that can be combined together.

During the operational life of the system, several situations may occur. For example, new threats may lead to revise the fault model (electromagnetic perturbations, obsolescence of hardware components, software aging, etc.). A revision of the fault model has of course an impact on the fault tolerance mechanisms. In other words, the validity of the fault tolerance mechanisms or the safety mechanisms depends on the representativeness of the fault model. In a certain sense, a bad identification of the fault model may lead first, to pay for useless mechanisms in normal operation and, second to observe a very low coverage of erroneous situations. This has an obvious side effect on the dependability measures (reliability, dependability). A change in the definition of the fault model often 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 loss of processing units, but simply a loss in network bandwidth may invalidate some fault tolerance mechanisms from a timing viewpoint.

Application changes are more and more frequent during the operational lifetime of a system. This is obvious for conventional applications (e.g., mobile phones) but it is becoming also needed for more critical embedded systems. Today, it 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 for commercial reasons. 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 on the lower level behavior, and the importance of assumptions coverage [2] is known for decades in the dependability community. Whatever the system's evolution during its whole lifetime, the safety mechanisms must remain consistent with all assumptions and operational conditions in terms of fault model, resources availability and application characteristics. Thus, the FT mechanisms must be adapted accordingly, leading to the notion of *Adaptive Fault Tolerance* (AFT).

Contributions: This work provides the following three contributions: $i)$ we describe a concise synthesis of concepts required by any Adaptive Fault Tolerant system. This synthesis is oriented to derive the required support from the underlying operating system or middle-ware, $ii)$ we propose an architectural model to implement generic and composable FT mechanisms on ROS, in a way that their integration is transparent to application, a prerequisite to their dynamic adaptation $iii)$ we analyze in details to what extent the adaptation at run-time of FT mechanisms in ROS is feasible, and discuss the cost involved by this adaptation.

In a first part, we summarize our approach to implement Adaptive Fault Tolerance, enabling partial updates of FTMs to be carried out on-line. We take advantage of Component Based Software Engineering technologies for implementing the adaptation of fault tolerance mechanisms. The minimal run-time support for implementing adaptive fault tolerance must provide one-to-one mapping of components to run-time units, segregation between components, and dynamic binding between components.

In the second part, we analyze to what extent AFT can be implemented on ROS. ROS is presently used in many applications (robotics applications, automotive applications like ADAS *Advanced Driver Assistance Systems*, or military applications). We show how ideal components can be mapped to ROS components and give implementation details of adaptive composable FTM at run-time.

We finally draw the lessons learned from our first experiments that rely on a small case study to identify the limits of ROS as a run-time support for Adaptive Fault Tolerance. We discuss the limits of the exercise and identify some promising directions for future work.

In Section 2 we describe the motivations and the problem statement. We give in Section 3 our definition and understanding of resilient computing. Our Component-Based Software Engineering (CBSE) approach for adaptive fault tolerance is summarized in Section 4. A full account of this approach can be found in [3]. The mapping of this approach to ROS is described in Section 5 and in Section 6, with the latter focusing on dynamic adaptation. The lessons learned are given in Section 7 before concluding.

2. MOTIVATIONS AND 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 [4]. 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 [5]. For Dependable systems that cannot be stopped for performing off-line adaptation, on-line adaptation of Fault Tolerance Mechanisms (FTMs) has attracted research efforts for some time now. However, most of the solutions [6, 7, 8] 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 expected. This explains why static system configurations with all possible FTMs and all possible combinations (FTMs composition) are not tractable. A form of differential FTM updates is proposed in this work to tackle unanticipated dependable systems evolution.

In both aeronautical and automotive systems, the ability to perform remote changes for different purposes is essential: maintenance but also updates and upgrades of big embedded applications. The remote changes should be partial as it is unrealistic to reload completely a processing unit for 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).

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 Adaptive Fault Tolerance leverages advancements in this field such as *Component-Based Software Engineering* [9], *Service Component Architecture* [10], and *Aspect-Oriented Programming* [11].

Functional and configuration changes may have a strong impact on dependability, and fault tolerance mechanisms must be updated to remain efficient in the presence of faults.

To this aim, our 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 tolerant computing. Using such concepts and technologies, we design FTMs as Lego-like brick-based assemblies that can be methodically modified at run-time 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. [6, 7, 8] .

However, most of software run-time supports used in embedded systems today do not rely on dynamic CBSE concepts. AUTOSAR, for instance, relies on very static system engineering

concepts and does not provide today much flexibility [12]. A new approach enabling remote updates to be carried out, including for safety mechanisms, is required. To the best of our knowledge, componentization and dynamic configuration of fault tolerance mechanisms has not been addressed in previous works.

ROS seems an appealing candidate for the dynamic composition of safety mechanisms. ROS is described as[†]: [...] *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 can be viewed as a middle-ware running on top of a Unix-based operating system (typically Linux). ROS is used in robotics applications (e.g., Robonaut 2 from NASA within the International Space Station) but also in other industry sectors, the automotive industry for instance. This open-source middle-ware provides a *weak* component approach and means to dynamically manipulate the system configuration.

3. RESILIENT SYSTEM AND DESIGN PATTERNS

3.1. Basic principles and definitions

A resilient system architecture is similar to a conventional dependable system architecture, but exhibits additional services, like an *Adaptation Engine* and a *Monitoring Engine*. Due to some changes in operation, an FTM may have to evolve and its development is carried out off-line. The Adaptation Engine objective is to update the implementation of the FTM on-line with necessary and sufficient only modifications to make it adequate. The Monitoring Engine controls that the running FTMs are consistent with their assumptions according to system state observation. Any inconsistency detected must trigger an adaptation of the FTM. Monitoring issues are out of the scope of the work reported in this paper.

In our framework, an application component C is attached (*bounded*) to an FTM (possibly a composition of several FTMs) following the well-known *Separation of Concerns* (SoC) principle. The *Adaptation Engine* is thus responsible for the management of the dynamic link between C and FTM, but also between components within a composite FTM component. It keeps track of components assemblies for both the application and the FTM.

Fault Tolerance Design Patterns represent solutions to a given fault tolerant computing problem. In Figure 1 we show an extract of FTDP classification with respect to fault models (F) and application characteristics (A). The fault model F has to be considered in a first place, distinguishing hardware and software faults here. Regarding hardware faults, patterns can deal with crash faults, permanent and transient value faults. In a second step, we refine the selected pattern, a duplex strategy for our example in Figure 1, with application characteristics regarding determinism and state issues. Determinism of execution implies that identical input values lead to identical output results, a key point for active replication. State, if any, also involves the capability to capture the state, which is required for passive and checkpointing-based applications.

A Fault Tolerance Mechanism (FTM) is then an implementation of a selected pattern. This classification is obviously very incomplete, but its merit is to show how to select a given FTM. We rely on such criteria, more precisely assumptions, to illustrate adaptive fault tolerant computing.

In the remainder of this paper, we will consider FTMs dealing with hardware faults only (permanent or transient). We recognize that software faults are more difficult to handle, both for detection and recovery, and mainly depend on application semantics. In our case, FTM handling hardware faults are sufficient to perform our analysis targeting ROS as a run-time support for adaptive fault tolerant computing.

[†] http://wiki.ros.org/ROS/Introduction

3.2. FTM Selection Criteria

As soon as the fault model is determined, then several solutions can be investigated depending on the application characteristics that have an impact on the implementation and the validity of the FTM. Depending on determinism and state issues, one implementation of the FTDP is chosen, leading thus to a concrete FTM. The resource aspect comes last. Any FTM needs resources to execute. Among the several FTMs that satisfy F and A assumptions, the selection can be based on local or system wide criteria. An FTM can be chosen because it requires the smallest set of resources among the valid FTMs candidates, or a more complex algorithm can be run to check if more resources can be granted to an FTM in order to improve some other criteria like response time in normal operation, recovery time, etc.

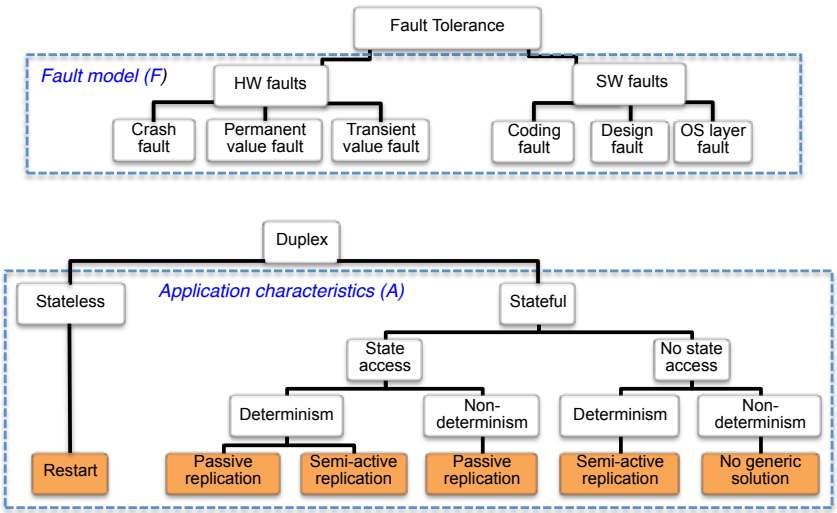

Figure 1. Extract of FTDP classification

The fault model can obviously be very much extended with more detailed types of faults including undesirable events identified in safety analysis. The mechanisms are identified according to the fault model, but their implementation depends very much on the application characteristics. The example given here shows the implication of state and determinism in the selection of a given implementation of duplex strategy. An extended definition of the fault model, including accidental physical faults both permanent and transient, programming faults, application undesirable events considered in safety analysis, may lead to the composition of several FTM. This issue is considered in this paper and illustrated in Section 5.

The next Section focuses on describing the basic concepts that underly an adaptive fault tolerant system.

4. ADAPTIVE FAULT-TOLERANCE

In this Section, we synthesize the essential concepts to address the problem of Adaptive Fault Tolerant computing. The extensive discussion is out of the scope of this paper and can be found in [3, 13, 14, 15].

4.1. Basic concepts of AFT

Three software development concepts are, in our view, essential for adaptive fault tolerance [13, 14]:

- *Separation of Concerns:* this concept is now well known, it implies a clear separation between the functional code, i.e. the application, and the non-functional code, i.e. 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 concept means that any software components can be decomposed into smaller components. Each component exhibits 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 of high interest for AFT.
- *Design for adaptation:* the adaptation of software systems implies that $i)$ the software itself has been analyzed with adaptation in mind for later evolution using componentization (although all situations cannot be anticipated), and $ii)$ software systems have been designed to simplify 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 [3].

The main benefits of AFT with respect to pre-programmed adaptation is that it provides means to define and update dependability mechanisms later during the lifetime of the system. Pre-programmed adaptation implies that all possible undesirable situations are defined at design time, which is difficult to anticipate regarding new threats (attacks), new failure modes (obsolescence of components), or simply adverse situations that have been ignored or forgotten during the safety analysis. In short, fine grain adaptation of FTMs improves maintainability of the system from a non-functional viewpoint.

4.2. 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: $i)$ fault tolerance requirements (F); $ii)$ application characteristics (A); $iii)$ available resources (R). We denote (F, 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 (F, A, R).

The three classes of parameters enable to discriminate FTMs. Among fault tolerance requirements F, we focus, for the time being, on the fault model that must be tolerated. Our fault model classification is based on well-known types [2], e.g., crash faults, value faults, development faults. In this work, we focus on hardware faults but the approach is perfectly reproducible for FTMs that target development faults.

The application characteristics A that we identified as having an impact on the choice of an FTM is: application statefulness, state accessibility and determinism. We consider 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 F 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 interference or hardware aging may change the initial model to a more complex one.

4.3. FT Design Patterns and Assumptions

To illustrate our approach, we consider some fault tolerance design patterns and briefly discuss their underlying assumptions and resource needs (a full coverage of this point can be found in [15]).

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 run-time support faults using repetition of the computation and voting. This is a 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 that could be violated by a value fault or by a random run-time support error. Such assertions can be user-defined and used to parameterize the FTM. In a certain sense it is a hybrid mechanism, since its overall behavior is customized by application-dependent assertions. Other mechanisms fall in this category, like *Recovery Blocks* and *N-Version programming*. *Adjudicators* and multiple *Versions* are examples of user-defined software blocks used in these generic fault tolerance design patterns.

In the work reported in this paper, we use simple implementations of a sub-set of FTMs (see Table I). More complex implementations have been proposed in other works, as described in [16].

Assumptions / FTM		PBR	LFR	TR	A&D
Fault Model (F)	Crash	√	√		√
	Transient			√	√
Application Characteristics (A)	Deterministic		√	√	(√)
	State Access	√		√	(√)
Resources (R)	Bandwidth	high	low	nil	(TDB)
	# CPU	2	2	1	2

Table I. Assumptions and fault tolerance design patterns characteristics

The underlying characteristics of the considered FTMs, in terms of (F, A, R), are shown in Table I. For instance, PBR and LFR tolerate the same fault model, but have different A assumptions and R needs. PBR allows non-determinism of applications execution 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 Figure 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 (e.g., safety property verification). Transitions can occur in both directions, according to parameter variation.

The priority is the fault model, the selection of the solution (i.e., the composition of several FTMs) depending on the application characteristics and the available resources. The final objective is always to comply with the dependability properties during the service lifetime.

4.4. Design for adaptation of FTMs

Our *design for adaptation* aims at producing reusable elementary components that can be combined to implement a given fault tolerance 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 Table II.

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

Table II. 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.

4.5. Run-time support

The software run-time support must provide key features to manipulate the component graph. Any application or an FTM is perceived as a graph of components. From previous experiments reported in [13], the following primitives 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 middle-ware, FRASCATI [17], which features a scripting language to manipulate the component graph, FScript [18].

In the following section, we describe how fault tolerance mechanisms can be implemented in ROS [19] in a way that is transparent to applications. Then, in Section 6, we implement the above-described concepts in ROS.

5. ADDING FAULT-TOLERANCE TO ROS

Said it concisely, ROS has not been designed to run safety critical systems, despite the fact that robots may be safety critical. Rather, 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 in ROS involves describing an assembly of nodes, a process that is in line with the component-based architecture we described in the previous section. Such an assembly is referred to as the computation graph of the application.

5.1. 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, and asynchronous communications through the concept of *topics*. When a node publishes a message on a topic, it is delivered to every node that has subscribed to this topic. A publisher is not aware of the list of subscribers to its topics and does not know other publishers. The *client/server* model defines bidirectional transactions (one request/one reply) implemented as 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 ease the addition, substitution, or deletion of nodes in a transparent manner, be it offline or online.

To provide this level of abstraction, each ROS application has to include a special node called the *ROS Master*. It provides registration and lookup services to the other nodes. All nodes register services and topics to the ROS Master. The master is the sole node that has a comprehensive view of the computation graph. When another node issues a service call, it queries the master for the address of the node providing the service, and then 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 a functionality, such as a monitoring node, in the invocation path between two ROS nodes. To this end, a relevant ROS feature is its *remapping* capability. At launch time, it is possible to reconfigure the name of any service or topic used by a node. Thus, requests and replies between nodes can be rerouted to interceptor nodes.

5.2. Implementing a componentized FT design pattern

In this section, we first present the generic computation graph we use for FTMs on ROS ; then the full implementation on ROS of a duplex FT design pattern, a *Primary Backup Replication* (PBR) combined with a *Time-Redundancy* (TR) design pattern is developed.

5.2.1. Generic Computation Graph We have identified a generic pattern for the computation graph of a FTM. Figure 2 shows its application in the context of ROS. Node *Client* uses a service provided by *Server*. The FTM computation graph is inserted 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.

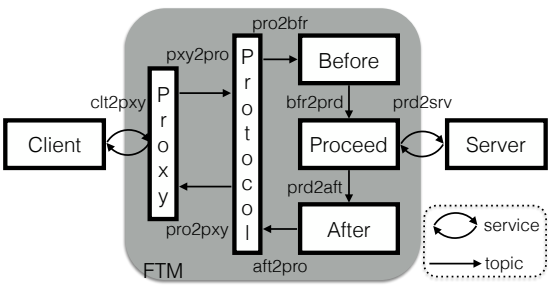

Figure 2. Generic computation graph for FTM

The FTM nodes, topics, and services are generic for every FTM discussed in section II. Implementing a FTM consists in specializing the *Before*, *Proceed*, and *After* nodes with the adequate behavior in the FTM.

5.2.2. Implementing PBR We illustrate the approach, through a *Primary-Backup Replication* (PBR) mechanism added to a Client/Server application in order to tolerate a crash fault of the Server. Figure 3 presents the associated architecture. Three machines are involved: the CLIENT site, which is hosting the *Client* node and 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, and the topics and services used through a request/reply exchange between a node Client and node Server (see Figure 3):

- *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 *crash detector* of the slave notifies the *recovery* node (topics *CD_S2rcy*). This node has two purposes: (i) in order to enforce the fail-silent assumption, it must ensure that every node of the MASTER are removed; (ii) 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.

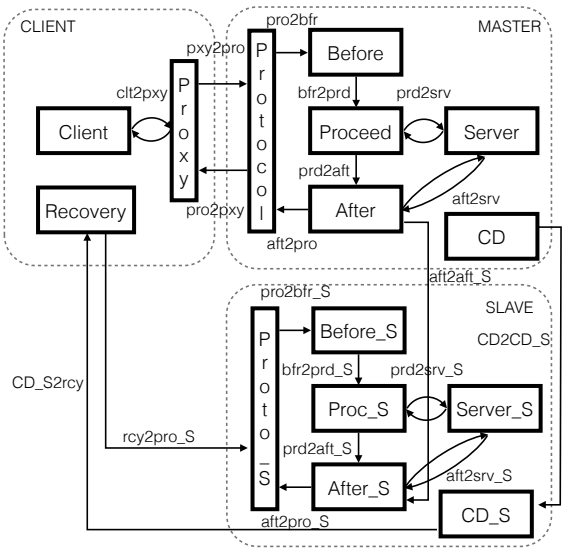

Figure 3. Computation graph of a PBR mechanism

ROS does not provide a command to change bindings between nodes after their initialization. The node developer must implement the transition logic. The SLAVE *Protocol* spins waiting for a notification from *Recovery* (topic *rcy2pro_S*). This is done using the ROS API: background threads, within a node, check for messages independently of the nodes main functionality. Upon reception of this topic, the SLAVE *Protocol* subscribes to topic *pxy2pro* and publishes to topic *pro2pxy*. After this transition, the proxy forwards the Clients requests to the SLAVE *Protocol*.

5.2.3. Impact on 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* nodes 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*. From an object-oriented viewpoint any server inherits from an abstract class *stateManager* providing two virtual methods, *getState* and *setState*, overridden during the server development.

5.3. *Composition of FT mechanisms*

The generic computation graph for FTM is designed for composability. In this section, the composition scenario is two-fold. We first illustrate the composition of two FTMs, PBR for crash faults and TR for transient value faults. Initially the application was installed with PBR. From an operational standpoint, at a given point in time, transient faults impacting numerical calculations appeared due to hardware components aging or sudden increase of environmental radiations. In a second step, later on, we consider that the communication channel between client and server can be the target for intrusions. Cryptographic protocols, based for instance on a simple *Public Key Infrastructure* (PKI), can be used to cipher communications and add cryptographic signatures.

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 substitute the Proceed node of a mechanism by a *Protocol* and its associated *Before/Proceed/After* nodes, as shown in Figure 4. 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.

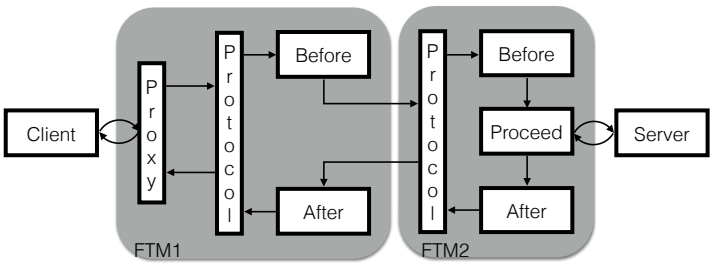

Figure 4. Principle of composition for FT mechanisms

5.3.1. Composition of PBR and TR The composition of PBR with TR can be triggered by a change in the fault model F. Lets suppose that, at a given point in time during the system lifetime, transient faults need to be tolerated because of hardware aging or due to some changes in the run-time environment, like electromagnetic perturbations.

The architecture of the composite FTM made of PBR and TR is given in Figure 5. This figure is an extension of Figure 3 where the *Proceed* node of the PBR has been replaced with the *Protocol* node of the TR implementation.

5.3.2. Composing FTMs with Cryptographic protocols Suppose now that some passive attacks are considered in the fault model F, requiring thus the inclusion of some ciphering mechanisms, in addition to the crash and transient fault tolerance mechanisms. The generic computation graph presented in Figure 2 enables cryptographic protocols to be seamlessly added to an application, already equipped with accidental fault tolerance mechanisms, PBR and TR in our example. The cryptographic mechanism (called SEC for security) is located at both the client (SEC_C) and the server side (SEC_S) as shown in Figure 6. On the server side, SEC operates before PBR and TR.

In this example, we only deal with possible intrusions between the client and the server.

We assume that a node implements the *Certification Authority* (CA). Three topics are used to communicate with the CA, namely *Cli2CA* for the *Client*, *Master2CA* for the Master and *Slave2CA* for the Slave. The topic *Cli2CA* enables the *Before* node of the *Client* to collect the certificate of the *Server*. Similarly, the topic *Master2CA* and *Slave2CA* enable *Before* of the Master, respectively the Slave, to collect the certificate of the *Client*. We assume that all parties know CA's public key. We assume that, for each participant, *Client* or *Server*, *Before* and *After* of the SEC mechanism share the pair of private and public keys they received when initialized.

- *Before* of the *Client* can cipher the request with K^S_{pub}, the *Server*'s public key, and adds a signature, using K^C_{priv} the *Client*'s private key;

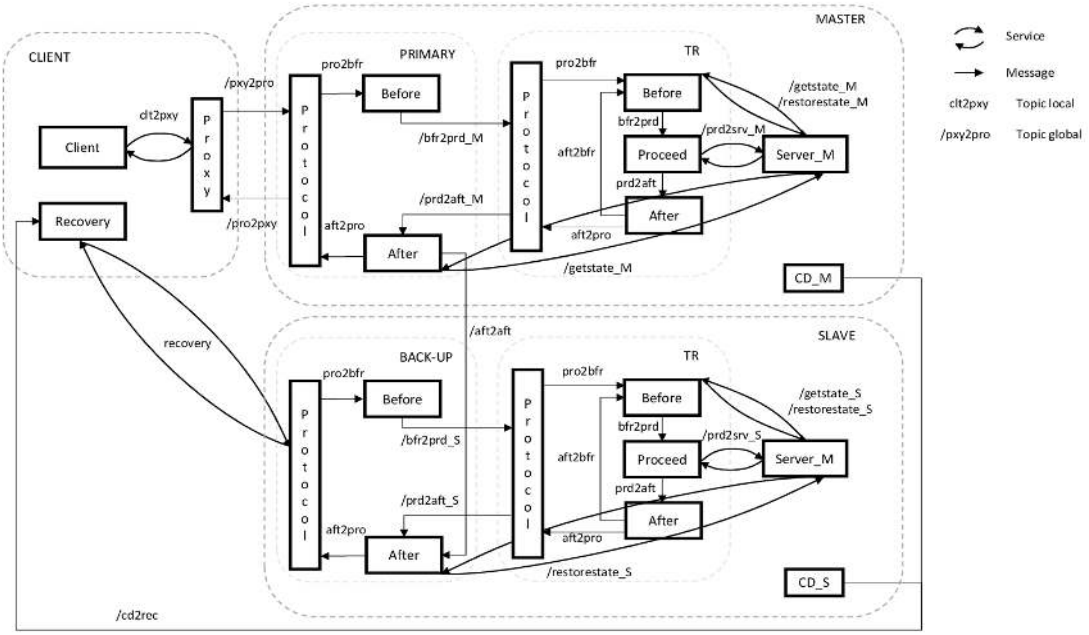

Figure 5. Composition of PBR and TR mechanisms on ROS

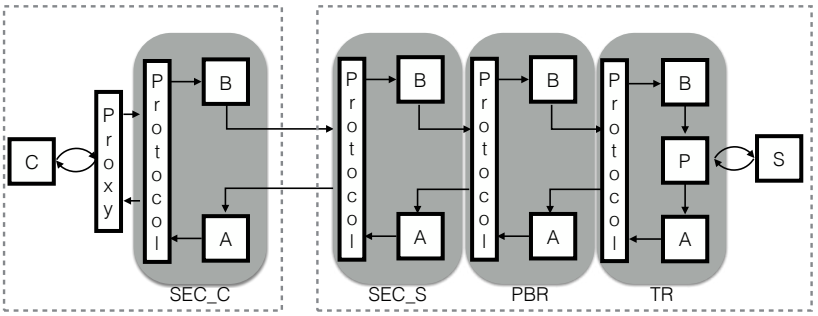

Figure 6. Principle of composition of SEC with other FT mechanisms

- Using the generic scheme given in Figure 6, a message is sent by the client to the server side through a new topic (called *Client_2_Server*) connecting *Before* of SEC_C to *Protocol* of SEC_S.
- *Before* of the Master deciphers the request with K^S_{priv}, the Server's private key, and checks the signature, using K^C_{pub}, the Client's public key;
- The Server can then proceed with a valid deciphered request through PBR and TR.

Conversely, *After* of the Master ciphers the reply and computes a signature. *After* of the Client deciphers the reply, checks the signature, and finally delivers the reply to the Client.

The communication between Master and Slave can also be secured using a similar security protocol.

6. DYNAMIC ADAPTATION: TO WHAT EXTENT WITH ROS

6.1. FTM Adaptation principles and ROS

Dynamic adaptation requires remote loading and removal of individual elements of a software component architecture, and dynamic binding facilities to reorganize a graph of components. It also requires control features to suspend and activate individual components. *To what extent ROS provides such features to safely adapt an FTM at runtime?*

We have considered three types of adaptations : $i)$ updating the current FTM, for instance updating the inter-replica synchronization protocol, $ii)$ switching from one FTM to another because some dramatic change occurred in the fault model, or because an application update leads to new application characteristics, and $iii)$ composing two FTM, for instance because the fault model has been extended to consider other types of faults.

We recall that the design, development, and validation of a new FTM configuration is performed off-line. The first type of adaptation implies a revision of the design or the implementation of the FTM. The other two are used to comply with parameters evolution (F, A or R). In all cases, the same features are required. Some are provided by ROS or by the underlying OS and some have been developed in-house.

A set of minimal API required to guarantee the consistency of the transition between two different FTMs has been established in previous work [13]:

- 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 have to be stopped in a quiescent state, i.e. when all internal processing has completed.
- Incoming requests on stopped components must be buffered.

ROS provides means to add and remove nodes, to buffer messages, and to control binding when a node is launched (using ROS remapping capability as presented in section 5). There is no ROS command to start or stop a node. Also ROS does not provide API to control the bindings of a node at runtime.

However, the good news is that these APIs can be emulated with dedicated logic added to some nodes. For instance, this is what we use to control the bindings in the Primary-Backup Replication to switch to the Backup when the primary fails.

To analyse to what extend run-time adaptation is possible with ROS, we need to describe in more details how topics work. Topics are the central concept in the publish/subscribe communication model used in ROS. A Topic is defined by:

- A *name*: ports are connected through a named Topic.
- *Sending ports*: used by publishers to send messages.
- *Receiving ports*: used by subscribers to receive messages.
- A *data type*: a unique data type is assigned to a topic for messages.

In ROS, when a node wants to publish or subscribe to a topic, it uses methods provided by the NodeHandle. The NodeHandle is an object instantiated in each ROS node and serves as the main interface to interact with the ROS master from within a node. The NodeHandle manages the start and the shutdown of the node. It also manages the instantiation of the sending and receiving ports. Creating a publisher or a subscriber is done in the following manner:

- The NodeHandle instantiation:
  `ros::NodeHandle nh`

- Publisher instantiation:
  ```
  ros::Publisher pub =
  nh.advertise<Data_type>("topic_name", queue_size)
  ```
- Subscriber instantiation:
  ```
  ros::Subscriber sub =
  nh.subscribe("topic_name", queue_size, callback_function)
  ```

Publishers and Subscribers are ROS objects. The callback function is triggered by the reception of a message and includes the data type as an argument.

ROS allows to remap the names of Topics a node uses, by substituting the name of the Topics hard coded in the node by new names provided as parameters of the command launching the node. Therefore, when a new node is launched, we are able to reconfigure the Topics of this node to communicate through any Topic matching the data type of its initial topic. Remapping arguments can be passed to any node and use the syntax `topic_name:=newname`. For example, a *Protocol* node which subscribed to a Topic named "pxy2pro" can be remapped at initialization to subscribe to an existing Topic "bfr2prd" by using two methods:

- either using an XML script:
  ```
  <node pkg="package name" type="node type" name="node name">
  <remap from="initial topic name" to="final topic name"/>
  ```
- or, with the user command line:
  ```
  rosrun package node initialTopicName:=finalTopicName
  ```

With this ROS features we can launch and link nodes to the graph of component so we can adapt or compose FTMs.

We illustrate adaptation through a composition example. Our FTMs architecture 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 substitute the *Proceed* node of a mechanism by a *Protocol* and its associated *Before/Proceed/After* nodes, as shown in Fig. 2 and Fig. 6.

Since ROS does not provide services to manipulate a component graph at runtime, we have developed an *Adaptation Engine* node. Its purpose is to run a script controlling the adaptation of an FTM. For instance, the composition of a PBR with a TR mechanism goes through the following steps:

- The Primary *Protocol* is suspended using the Unix signal SIGSTOP;
- The *Proceed* node is killed using a ROS command:
  `rosnode kill Primary/Proceed`
- The TR nodes (*Protocol-B-P-A*) are launched (on each replicas) using a script in XML and a ROS command: `roslaunch TR TR.launch`;
- The TR *Protocol* links itself to the PBR *Before* topic and the PBR *After* one using the Topic names parameters provided in the `TR.launch` script;
- The Primary *Protocol* is restarted using the Unix signal SIGCONT.

Note that ROS ensures that messages are not lost during adaptation. A publisher node buffers all on-going messages until all its subscriber nodes read them. Thus stopping a node is safe with respect to communication. The other types of adaptation are based on a similar sequence of steps: suspend, substitute, link, and restart. For an update, only one node may be replaced. For a transition between two mechanisms only the *Before* and *After* nodes need to be changed.

With the above described ROS/Unix features, we are able to compose or adapt our FTMs. However, we cannot dynamically adapt the communication between two nodes at run-time. The following section describes how we overcome this limitation.

6.2. Implementing Dynamic Binding on ROS

Dynamic binding is the ability to configure on-line the communications between two nodes. It is an important feature for AFT in order to manipulate the graph of components. Indeed, we need to be

able to manage the nodes but also the communications between them. Thus, the dynamic binding consists in being able to manage the communication of nodes.

Dynamic binding is crucial to the proposed architecture of FTMs. A good example of dynamic binding usage is the transfer of the connection linking the *Client* to the *Primary* to the *Backup* when the *Primary* crashes (see section 5.2.2 – Implementing PBR). We cannot kill and relaunch the *Backup* nodes (loss of its internal state) therefore we cannot use the remapping at initialization. The Topic on which the *Client* publishes still exists after the crash of the *Primary*. We need to instantiate the communication ports of the *Backup* to communicate on this existing Topic.

We have added to the node a function in order to control the instantiation of the communication ports. New data types for topics cannot be defined at run-time, however new topics based on pre-defined data types can be instantiated. For example, we have implemented a service (simplified in Fig. 7) to activate or deactivate the communication between the *Backup* and the *Client* at runtime.

```
bool recover(Request &req, Response &res){
 //deactivation of the ports
 if(req.activation==0){
   pub.shutdown();
   sub.shutdown();
 }
 //instantiation of the ports to reconnect the node
 else if(req.activation==1){
   pub = nh.advertise<Data_type>(req.topic_pub, req.queue_size);
   sub = nh.subscribe(req.topic_sub, req.queue_size, callback);
 }
 return true
}
```

Figure 7. Example of a dynamic binding service

This service is triggered by an external node, in this example the *Recovery Node*. The input request must contain multiple parameters such as Topic name, publish or subscribe, activation or deactivation. We choose to use a service (synchronous message) to have an acknowledgment of the correct service execution. When the crash of the *Primary* is detected, the *Recovery Node* call the service implemented in the *Backup* and thus the connection to the *Client* is dynamically established, without using remapping at initialization.

In Fig. 7 the function `recover` reinitializes the publisher or the subscriber to manage the dynamic binding with an external node. The function has two objectives: i) to shutdown a port (this function will use ROS API `pub.shutdown()` or `sub.shutdown()`) and ii) to initialize the port (this function will use ROS API `advertise` or `subscribe` presented in 6.1). In any case, an external node is mandatory to trigger the function and to pass to the node the various parameters it needs.

In our example, we chose to use an existing Topic to bind the *Client* and the *Backup*. Thanks to this approach it is possible to create a totally new Topic between them.

In summary our dynamic binding approach enables solving two situations:

1. Activation/shutdown of a Topic in an existing node (switch Primary/Back-Up)
2. The insertion of a node between two communicating ones (insertion of the FTM)

In our prototype, AFT is realized through a combination of ROS features, Unix features, and some custom services. In particular, a nodes life cycle (stop, start) is controlled directly through UNIX signals. Dynamic binding is achieved through implementation of custom methods in the nodes and through external nodes, here the *Adaptation Engine* or the *Recovery node*, to orchestrate the adaptation. In conclusion, even if ROS lacks some essential features, AFT is possible with ROS.

7. LESSONS LEARNED SUMMARY

The general requirements for an executive support suitable for implementing AFT we have exhibited in former work relies on the following features: i) control over component's life cycle at run-time (add, remove, start, stop), ii) control over interactions at run-time for creating or removing bindings. In addition to separation of concerns, these features are related to the degree of observability and control the software platform provides over the component-based architecture of applications (including FTM) at run-time. Furthermore, to ensure consistency before, during and after reconfiguration of the component-based software architecture, several issues must be carefully considered: i) components must be stopped in a quiescent state, i.e., when all internal processing has finished, ii) incoming requests on stopped component must be buffered.

This specification is our frame of reference to discuss the adequacy of ROS as a run-time support for AFT.

In our experiments, a component was mapped to a node at run-time providing memory space segregation. The binding between components relied on topics managed by the ROS Master. Dynamic binding was possible but ROS does not provide a specific API to manage such connection between components. As we have seen in the previous section, additional code is required to manage dynamic bindings, using facilities provided by the underlying Linux operating system.

Control over component's life cycle:

- ROS provides commands to delete and create nodes
- Thanks to UNIX commands nodes can be stopped and restarted

Control over interactions at run-time:

- ROS enables nodes to connect or disconnect to/from topics
- A specific service must be added to all nodes to trigger these connections/disconnections
- The topics a node can connect or disconnect to/from is defined at the initialization of the node
- ROS enables new topics to be created
- Topics store outgoing messages when subscribers are not available

Regarding the features of ROS for implementing AFT, we consider them not entirely satisfactory, as ROS does not provide dynamic binding between nodes, and the API to control components lifecycle at run-time is too weak. However, although imperfect, resilient computing using AFT can be implemented on ROS. Dynamic binding is possible on existing topics by adding some specific services in the nodes. For new topics, a customized solution was proposed in this work. ROS provides separation of concerns, since components can be mapped to nodes (Unix processes) that have their own address space. The model of communication used in ROS is also a benefit to design and implement resilient distributed application.

It is worth noting that, as soon as some change is identified, the adaptation of the FTM attached to the application is carried out off-line and by the way validated according to the development process standard in the domain (automotive/ISO26262, aerospace/DO178C, IEC61508, etc.). The dynamic adaptation of the mechanisms is a service to avoid complete reload of the system.

Using ROS in a dependable and resilient system is hindered by the fact that the ROS master is a single point of failure in the architecture. The ROS Master must be operational when installing an application and during its execution. When the ROS master fails, the whole software architecture must be restarted. We are currently investigating a replicated implementation of the ROS master using the DMTCP *(Distributed MultiThreaded Checkpointing)* library developed at NorthEastern University Boston [20]. This is however very complex and having multiple ROS masters running in parallel is currently not possible. For the time being our software architecture, as any ROS application, is linked to a unique ROS Master. This problem should be solved by the ROS community and this is something that should be addressed in ROS 2. Indeed, the next major revision of ROS (ROS2) is based on a DDS (Data Distribution Service) communication system that should help solving this problem by distributing the ROS master functionalities among the nodes of the system. This approach would however require reliable multicast protocols properly implemented and validated.

8. CONCLUSION

The adaptation of embedded application requires an adequate run-time support. Beyond design for adaptation issues that relate more to the development process, the run-time support must fulfill 5 requirements: (i) separation of concerns, (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 provides efficient solution for the last two. On-line adaptation is possible as demonstrated in this paper. We have been able to overcome the limitations of ROS thanks to underlying OS features and some additional logic implemented into the nodes.

As a run-time support for resilient computing, ROS is an interesting development platform to test concepts for Adaptive Fault Tolerance. The mapping of components to ROS is simple (component $\rightarrow$ node) and on-line modification of FTMs during the lifetime of the system is possible. The insights gained with this work should help to develop a suitable run-time support for Adaptive Fault Tolerance in the context of safety critical real-time systems.

Our current work is done in collaboration with Renault-Nissan Group, especially targeting remote updates for ADAS. The basic principles of our approach are consistent with the framework proposed in Adaptive AUTOSAR. The basic operating system is based on Posix and several services are defined to master adaptation, like the *Software Configuration Management* service and the *Platform Health Management*, which can be related to our *Adaptation Engine*. We believe that a run-time support like Linux or any Posix-based OS is not dynamic enough to implement fine-grained adaptation. ROS is providing an additional layer to this aim as a middle-ware, but the granularity remains coarse and the dynamic binding is still difficult to handle. We hope that ROS2 will provide a more powerful and reliable platform, using DDS (Data Distribution Service) for industrial applications. Solving the dynamic binding problem implies revisiting the publish-subscribe implementation to manipulate communication channels at run-time. Finally, the middle-ware should provide additional features to suspend, activate run-time entities, save/restore internal state and buffer inter-entities communications. In the next future, we plan to address those issues, taking advantage of the development of the Adaptive AUTOSAR Platform.

REFERENCES

1. Laprie JC. From dependability to resilience. *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, Citeseer, 2008; G8–G9.
2. Powell D. Failure mode assumptions and assumption coverage. *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, 1992; 386–395, doi:10.1109/FTCS.1992.243562.
3. Stoicescu M. Architecting resilient computing systems : a component-based approach. PhD Thesis 2013. URL http://www.theses.fr/2013INPT0120/document, 2013INPT0120.
4. Kim K, Lawrence TF. Adaptive fault tolerance: Issues and approaches. *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*, IEEE, 1990; 38–46.
5. Krishna CM, Koren I. Adaptive fault-tolerance: fault-tolerance for cyber-physical systems. *Computing, Networking and Communications (ICNC), 2013 International Conference on*, IEEE, 2013; 310–314.
6. Fraga J, Siqueira F, Favarim F. An adaptive fault-tolerant component model. *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. The Ninth IEEE International Workshop on*, 2003; 179–179, doi: 10.1109/WORDS.2003.1267506.
7. Lung LC, Favarim F, Santos GT, Correia M. An infrastructure for adaptive fault tolerance on FT-CORBA. *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, 2006; 8 pp.–, doi:10.1109/ISORC.2006.16.
8. Marin O, Sens P, Briot JP, Guessoum Z. Towards adaptive fault tolerance for distributed multi-agent systems. *Proceedings of ERSADS* 2001; :195–201.
9. Szyperski C. *Component Software: Beyond Object-Oriented Programming*. 2nd edn., Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2002.
10. Marino J, Rowley M. *Understanding SCA (Service Component Architecture)*. 1st edn., Addison-Wesley Professional, 2009.
11. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*, chap. Aspect-oriented programming. Springer Berlin Heidelberg: Berlin, Heidelberg, 1997; 220–242, doi:10.1007/BFb0053381.
12. Martorell H, Fabre JC, Lauer M, Roy M, Valentin R. Partial updates of autosar embedded applications – to what extent? *Dependable Computing Conference (EDCC), 2015 Eleventh European*, 2015; 73–84, doi:10.1109/EDCC.

2015.18.
13. Stoicescu M, Fabre JC, Roy M. From design for adaptation to component-based resilient computing. *Dependable Computing (PRDC), 2012 IEEE 18th Pacific Rim International Symposium on*, IEEE, 2012; 1–10.
14. Enard Q, Stoicescu M, Balland E, Consel C, Duchien L, Fabre JC, Roy M. Design-driven development methodology for resilient computing. *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, Vancouver, BC, Canada, June 17-21, 2013*, 2013; 59–64, doi:10.1145/2465449.2465458.
15. Stoicescu M, Fabre J, Roy M. Architecting resilient computing systems: A component-based approach for adaptive fault tolerance. *Journal of Systems Architecture - Embedded Systems Design* 2017; **73**:6–16, doi:10.1016/j.sysarc.2016.12.005.
16. Wiesmann M, Pedone F, Schiper A, Kemme B, Alonso G. Understanding replication in databases and distributed systems. *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, 2000; 464–474, doi:10.1109/ICDCS.2000.840959.
17. Seinturier L, Merle P, Rouvoy R, Romero D, Schiavoni V, Stefani JB. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience* May 2012; **42**(5):559–583, doi:10.1002/spe.1077. URL https://hal.inria.fr/inria-00567442.
18. Léger M, Ledoux T, Coupaye T. Reliable dynamic reconfigurations in a reflective component model. *Proceedings of the 13th International Conference on Component-Based Software Engineering*, CBSE'10, Springer-Verlag: Berlin, Heidelberg, 2010; 74–92, doi:10.1007/978-3-642-13238-4_5.
19. Lauer M, Amy M, Fabre JC, Roy M, Excoffon W, Stoicescu M. Engineering adaptive fault-tolerance mechanisms for resilient computing on ROS. *17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7-9, 2016*, 2016; 94–101, doi:10.1109/HASE.2016.30.
20. Ansel J, Arya K, Cooperman G. DMTCP: Transparent checkpointing for cluster computations and the desktop. *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.