Towards Adaptive Fault Tolerance: From a Component-Based Approach to ROS

Michaël Lauer, Matthieu Amy, William Excoffon, Matthieu Roy, Miruna Stoicescu

Towards Adaptive Fault Tolerance: From a Component-Based Approach to ROS

M. Lauer[1], M. Amy[2], W. Excoffon[2], M. Roy, M. Stoicescu[3]

CNRS-LAAS, Toulouse, France
Univ de Toulouse, [1] UPS, [2] INP
[3] ESOC/ESA, Darmstadt, Germany, on behalf of GMV

A system that remains dependable when facing changes (new threats, failures, updates) is *resilient*. In this paper, we report on an approach taking advantage of Component Based Software Engineering technologies for tackling a crucial aspect of resilient computing, namely the on-line adaptation of fault tolerance mechanisms. The second part of this paper shows how this approach can be implemented on ROS that is presently used for implementing automotive applications, e.g. ADAS. 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.

Keywords: **fault tolerance; on-line adaptation; CBSE; ROS**

I. INTRODUCTION

Systems have to evolve during their service life in order to cope with additional features requested by users. For dependable embedded systems, the challenge is greater, as evolution must not impair dependability attributes. The persistence of dependability when facing changes is called resilience [1]. Dependability relies at runtime on Fault Tolerance Mechanisms (FTMs or Safety Mechanisms) attached to the application. A challenge of resilient computing is the capacity to adapt the FTMs during the operational life of the system.

Adaptive Fault Tolerance (AFT) [2] is gaining importance and on-line adaptation of FTMs has attracted research efforts for some time now. However, most solutions [3,4,5,6] tackle adaptation in a preprogrammed manner: all FTMs necessary during the service life of the system are 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. Clearly, 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.

In automotive systems, remote changes are of interest for different purposes: maintenance but also updates and upgrades of embedded applications (e.g. ADAS). Remote changes should be partial as it is unrealistic to reload completely an ECU from small updates. This idea is promoted by some car manufacturers like Renault, BMW but also TESLA Motors stating in its website *"Model S regularly receives over-the-air software updates that add new features and functionality"*. Remote changes will become very important for economic reasons, for instance to sell options a posteriori, since most of the evolution in the next future will rely on software for the same hardware configuration (sensors and actuators). Car-to-car applications also imply rapid adaptation of onboard software to remain consistent with the network of cars. Consequently, changes imply revisiting the safety analysis (e.g. FMECA) and thus safety mechanisms may evolve a posteriori.

We propose an alternative to preprogrammed adaptation denoted *agile adaptation of FTMs*. The term "agile" is inspired from agile software development [7]. 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.

Evolvability has long been a prerogative of the application business logic, leading to the development of adaptive software [8]. Consequently, our approach for the agile adaptation of FTMs leverages advances in this field such as Component-Based Software Engineering (CBSE) technologies [9]. Using such concepts and technologies, the idea is to design FTMs as "Lego"-like brick-based assemblies that can be methodically modified at runtime through fine-grained changes affecting a limited number of bricks. This approach maximizes reuse and flexibility, contrary to monolithic replacements of FTMs. In automotive embedded systems, the runtime support does not rely on these software engineering concepts. As shown in [10], AUTOSAR does not provide today much flexibility.

ROS is a of middleware for robotics applications but also used in the automotive industry. This open-source component-based middleware provides means to dynamically manipulate system configuration. The ROS community is very large and the system is used in several critical application, e.g. for unmanned military vehicles at NREC (*National Robotics Engineering Center* in Pittsburgh, e.g. the *Crusher*).

Section II summarizes our CBSE-based approach for adaptive fault tolerance. The mapping of this approach to ROS is described in Section III. Conclusion is given in Section IV.

II. ADAPTIVE FAULT TOLERANCE USING CBSE

A. Adaptation and Change Model

The choice of an appropriate fault tolerance mechanism (FTM) for a given application depends on several parameters: 1) fault tolerance requirements *FT*; 2) application characteristics *A*; 3) available resources *R*. At any point in time, FTM(s)

attached to an application component must be consistent with the current values of (*FT, A, R*). These three parameters enable to discriminate FTMs and represent our change model.

Among *fault tolerance requirements FT*, we focus on the fault model. Our classification is based on well-known fault types [14]: crash faults, value faults, and development faults. We focus here on hardware faults but the approach is adaptable to any type of faults, e.g., 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. Mastering non-determinism or state access for black boxes is impossible!

The *available resources R* play an important role in the selection process. FTMs require resources such as bandwidth, CPU, battery life/energy. In case more than one solution exists, for a given set of parameters *FT* and *A*, the resource criterion can invalidate some of the solutions (notion of cost function).

Any parameter variation during the service life of the system may invalidate the running FTM, thus requiring a transition towards a new one. Transitions may be triggered by new threats (i.e., fault model change), resource loss or a new application version with different characteristics.

B. FT Design Patterns and Assumptions

To illustrate our approach, we consider some fault tolerance design patterns and discuss their underlying assumptions and resource needs.

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). Each replica is considered as a *self-checking* component. The fault model includes hardware faults or random operating system faults (no common mode faults). At least 2 independent processors are necessary to run this FTM.

Time Redundancy (TR) tolerates transient physical faults or random runtime support faults using repetition of the computation and comparison. This is a way to improve the self-checking nature of a replica.

The above *Duplex strategies* can be combined to TR to tolerate both transient and permanent faults.

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

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. PBR requires state access for checkpoints and higher network bandwidth (in general), while LFR does not require state access but generally incurs higher CPU cost as both replicas execute the request.

C. Possible transitions

During the service life of the system, the values of 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. 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→PBR+TR (FTM composition) transition is triggered by a change in the considered fault model (e.g. value faults). Transitions can occur in both directions, according to parameter variation. A transition implies remote loading of additional individual FTM bricks, and on-line reconfiguration of the FTM bricks assembly.

D. Design for adaptation of FTMs

Our approach relies first on an analysis of the FTMs in order to extract common parts and variable features between them. We follow a *"design for adaptation"* approach consisting of several design loops. We combine Object-Oriented concepts and Aspect-Oriented-Programming concepts to produce componentized fault tolerance design patterns. The result is a set of elementary reusable components that can be combined to implement a given fault tolerance or safety mechanism. For instance, duplex mechanisms can be designed using the *Before-Proceed-After* scheme, because an inter-replica synchronization takes place before request processing and another one after (Fig. 2).

FTM	Before	Proceed	After
PBR (primary)	Forward request	Compute	Checkpointing
PBR(backup)	Handle request		State update
LFR (leader)	Forward request	Compute	Notify
LFR (follower)	Handle request	Compute	Handle notification
TR	Get/SetState	Compute	Compare & repeat

Fig. 2. Generic execution scheme for FT design patterns

Composition implies nesting the *Before-Proceed-After* scheme. 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 above execution scheme was implemented on the FraSCAti [11], a reflective component-based middleware providing runtime support for applications designed according to the Service Component Architecture (SCA) specifications. FraSCAti enriches the basic SCA specification with support for on-line exploration and reconfiguration of component-based assemblies. Our approach is reproducible on any other support providing control over components and bindings [12].

III. Adaptive Fault Tolerance on ROS

A. Introduction to ROS

ROS can be viewed as 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 *computation graph* of the application.

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 *topics*. When a node publishes a message on a topics, it is delivered to every nodes subscribing to this topics. Note that neither a publisher is aware of the subscriber to its topics nor the other publishers. 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 offline or online.

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, having thus a comprehensive view of the computation graph. When a node issues a service call, it queries the master for the address of the service node and then it sends its request to this address.

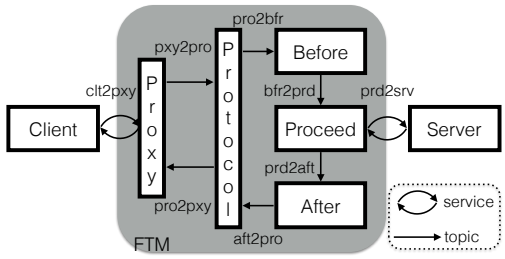

Fig. 3. Generic computation graph for FTM

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. At launch time, it is possible to reconfigure the name of any services or topics used by a node. Thus, requests and replies between nodes can be rerouted easily to interceptor nodes.

B. 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 between the two thanks to the ROS remapping feature. 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. 2).

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* hosting the primary replica and the Slave 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.

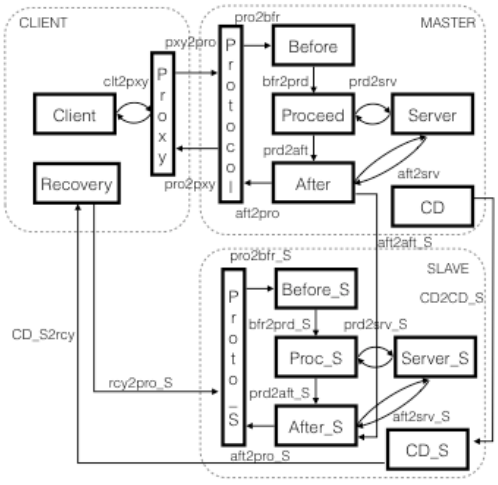

Fig. 4. Computation graph of a PBR mechanism

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 by background threads within a node independently of its main functionality, using the ROS API. Upon reception of this topic, *protocol* subscribes to topic *pxy2pro* and publishes to topic *pro2pxy*. Further requests from the *Client* will now be forwarded by the proxy 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 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*. 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.

C. Composition of FT 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. 5.

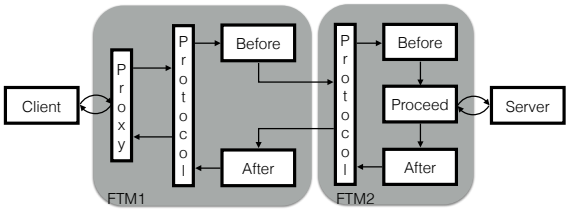

Fig. 5. Composition of FT mechanisms.

D. Dynamic Adaptation of FTM

A set of minimal API required for dynamic adaptation of 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 run- time, for creating or removing bindings. Furthermore, to ensure consistency before, during and after reconfiguration, 1) components must be stopped in a quiescent state, i.e. when all internal processing has finished, and 2) incoming requests on stopped components must be buffered, to ensure consistency of request processing.

With the exception of add and remove, ROS does not provide this API. This API can be emulated with dedicated logics in some nodes. For instance, we have described binding control in the Primary to Backup switch in our example. Controlling node lifecycle 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. In future work, we intend to emulate the required API to guarantee consistency of reconfiguration.

IV. Lessons Learnt and Conclusion

In previous work, we have defined a complete process for Adaptive Fault Tolerance, starting from a Design for Adaptation strategy. Then, we have shown that CBSE technologies and associate reflective middleware enable FTMs to be adapted off-line and on-line, by changing few components.

The work reported in this paper aims at evaluating to what extent ROS is an appropriate runtime support to implemented Adaptive Fault Tolerance. The result is two-fold. The positive aspect is that ROS provides a notion of component at runtime (node) providing time and space partitioning (based on the notion of Linux process). A second benefit is that it is possible to map both synchronous and asynchronous interactions on services and topics, respectively. The negative points are essentially related to the manipulation of the bindings on-line. This involves additional customization of the nodes to reach this aim.

The manipulation of the suspension/activation of the nodes is problematic at runtime. An ad-hoc solution has to be developed to handle the quiescent state of the node. Last but not least, at this stage, performance issues have not been investigated.

References

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

[2] K. H. K. Kim and T. F. Lawrence. Adaptive Fault Tolerance: Issues and Approaches. In *Procs 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 Int. Symp. on Object and Component-Oriented RT Dist. 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 *2nd Computer*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[10] 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 Conf. (EDCC),* 2015, Paris, France.

[11] 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.

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