Adaptive Fault Tolerance: Is ROS a Relevant Executive Support ?

Matthieu Amy

Adaptive Fault Tolerance: Is ROS a Relevant Executive Support ?

Matthieu AMY
LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France
Email: mamy@laas.fr

***Abstract*—Every system evolves in operation. A system that remains dependable when facing changes (new threats, failures, updates) is called resilient. We present advantages of Component-Based Software Engineering technologies for tackling a crucial aspect of resilient computing, namely the on-line adaptation of fault tolerance mechanisms. Then we show how this approach can be implemented on ROS, presently used for robotic and automotive applications, e.g. ADAS. We give some implementation details and result of different experiments to validate the solution. We finally report the lessons learned and the future work targeting resilient computing for automotive applications.**

I. Introduction

A computer system is resilient when it remains dependable when facing changes (new threats, change of fault models, updates of applications) [1]. At runtime, dependability relies on *Fault Tolerance Mechanisms* (FTMs). The FTMs are designed and developed to match a fault model specific to an application. This fault model considers both hardware and software faults that may lead to failure modes which impair the correct behavior of the system. In resilient systems, the fault model may evolve during its operational lifetime. Therefore, *Adaptive Fault Tolerance* (AFT) [2] is required to accommodate changes. However, solutions are mostly preprogrammed [3], FTMs are loaded at the beginning for the entire operational lifetime. Adapting the mechanisms means selecting the appropriate one or tuning them with few parameters (e.g. number of replicas). Adapting provisions for all events and threats a system may encounter throughout its service life is one of the best way to achieve the dependability of the system.

The agile adaptation of FTMs is investigated as an alternative to preprogrammed FT solutions. The term "agile" is inspired from agile software development [4]. 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. In order to develop an agile solution, *Component-Based Software Engineering* (CBSE) techniques [5][6] are some of the most relevant approaches in developing adaptive FTMs. The idea is to design an FTM as a graph of software bricks that can be removed or changed at runtime. The purpose of this design approach is to change the least amount of "software bricks" for an easier adaptation. This approach maximizes reuse and flexibility, contrary to monolithic replacements of FTMs. Using CBSE techniques, software bricks are components.

The objective of the work reported in this paper is to analyze to what extent ROS (Robot OS) [7] is an appropriate executive support for AFT based on CBSE design concepts.

ROS is an open-source middleware for robotic applications providing a component-based system architecture. Its user base is very large and this middleware is already used for critical applications in industry, e.g. for unmanned military vehicles at NREC *(National Robotics Engineering Center, Pittsburgh)*.

In this paper, we consider two FTM mechanisms tolerating cash faults, *Primary Back-Up Replication* (PBR) and *Leader Follower Replication* (LFR), and one targeting transient value faults, namely *Time Redundancy* (TR). These mechanisms are designed using a CBSE approach and implemented on ROS. We investigate the on-line adaptation of these mechanisms using ROS capabilities and services. We consider three types of adaptations: i) the update of a FTM, ii) the substitution of a FTM by another one, and iii) the composition of several FTMs.

In section II we describe more precisely the concept of Adaptive Fault Tolerance. We detail the software architecture of an FTM on ROS and the various means to make them adaptive in section III. In the next section IV we illustrate our approach on a simple case study. We draw the lessons learnt, mention on-going work and conclude in section V.

II. Adaptive Fault Tolerance

Adaptive Fault Tolerant relies on three essential concepts: *Separation of Concern*, implying a clear separation between the functional code (i.e. the application) and the non-functional code (i.e. the fault tolerance mechanisms), *Componentization*, implying the decomposition of the software into a graph of software components, and the *Design for adaptation*, implying that the software is designed to facilitate its adaptation.

A. Fault tolerance assumptions and requirements

The choice of an FTM attached to an application depends on three class of parameters: The FaulT model (FT), the characteristics of the Application (A) and the Resources of the system (R).

As mentioned in introduction, three FTM have been used in our experiments. PBR (*Primary Back-Up Replication*) and LFR (*Leader Follower Replication*) are two variants of a duplex strategy to tolerate crash faults. With PBR, just one replica is active and processes input requests, the backup replica handles state checkpoints. In LFR, both replicas are active and process input request; the leader delivers the outputs. To tolerate transient faults, TR (*Time Redundancy*) repeats the execution a number of time to detect and tolerate faults.

In Fig.1, we summarize the fault model, the application characteristics and some required resources for each of these

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

Fig. 1: Assumptions and fault tolerance mechanisms

FTMs. We consider in the rest of this paper simple implementation of these mechanisms tolerant to hardware faults. It is worth noting that solutions to the same fault model (e.g. PBR and LFR for crash faults) comply with different application characteristics (e.g. determinism, state access) and require a different amount of resources (e.g., number of CPU, bandwidth usage). This means that any evolution leading to a change in the fault model or the application characteristics implies an adaptation of the FTMs accordingly.

B. Componentization of FTMs

The idea is to decompose FTM into elementary components. Based on Object-Oriented and Aspect-Oriented Programming concepts, and following a CBSE approach, an FTM is divided into three components, *Before - Proceed - After*.

- *Before* is responsible for coordination among replicas (e.g. client request agreement).
- *Proceed* triggers the execution of the application attached to the FTM.
- *After* is responsible for the post coordination and synchronisation among replicas (e.g. checkpointing).

The decomposition of the FTMs is 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

Fig. 2: Generic design for Fault Tolerance Mechanisms

Thanks to this decomposition, only the *Before* and *After* components have possibly to be modified during a transition from one FTM to another.

C. On-line Adaptation

The concept of AFT implies transitions between FTM when we observe a change in the parameters FT, A or R. The transition may involve updating the Before and After components, or compose mechanisms together dynamically. Applying CBSE techniques allows us to adapt the FTM while minimizing the number of changes.

For instance, suppose that at a given point in time an application update lead to invalidate the state access assumption. If the current strategy is PBR then, it must be substituted to LFR. This implies changing both Before and After components.

Suppose now that, transient faults need to be tolerated. The LFR strategy must be combined with TR. The Proceed component of LFR must delegate the computation to the Before-Proceed-After of TR.

The challenge now is to select a runtime support able to :

- Map components to runtime units and control their life cycle.
- Manipulate components interaction channels dynamically.

The aim of the rest of this paper is to evaluate to what extent ROS provides theses features.

III. Adaptive FT on ROS

ROS is a middleware for robotics systems such as the PR2[1]. It allows the development of a modular software architecture where each process, known as *Node*, is a separate entity in both time and space. Each *Node* can communicate with another by Asynchronous messages (*Topic*) through a publish-subscribe communication or Synchronous messages (*Service*). A special node, called the *ROS master* is launched in the background to control communications between application *Nodes*.

A. Componentization of FTMs with ROS

We consider a $Server$ delivering a service to a $Client$. The Server is attached to a FTM at initialization according to the initial values of FT, A and R parameters. Our main challenge is to implement the FTM between the $Client$ and the $Server$ without modifying these nodes *(Separation of Concerns)*.

The $Client$ node interacts with the $Server$ node through a $Proxy$ node. The $Server$ node uses a $Protocol$ node to communicate with the $Client$ proxy. The FTM is implemented according to the B-P-A framework, each components of this framework being a node on ROS (cf Fig.2). In practice the final FTM implementation is composed of four nodes (*Protocol-B-P-A*) in addition to the $Server$ node. A $Watchdog$ node is used on each computer to detected crash faults.

The behavior of the PBR strategy on ROS is the following:

- *Client* sends a request to *Proxy* (service *clt2pxy*);
- *Proxy* adds an identifier to the request and transfers it to *Protocol* (topic *pxy2pro*) of the *Primary* replica;
- *Protocol* checks whether it is a duplicate request: if so, following the *"at_most_once"* semantics, it sends directly the stored reply to *Proxy* (topic *pro2pxy*). Otherwise, it sends the request to *Before* (service *pro2bfr*);

[1] https://www.willowgarage.com/pages/pr2/overview

- *Before* transfers the request for processing to *Proceed* (topic *bfr2prd*);
- *Proceed* calls the service provided by the *Server* (service *prd2srv*) and forwards the result to *After* (topic *prd2aft*);
- *After* gets the last result from *Proceed*, captures *Server* state (service *aft2srv*), and builds a checkpoint which is sent to node *After* of the backup replica (topic *aft2aft_S*);
- *Protocol* gets the result (topics *aft2pro*) and sends it to Proxy (topic *pro2pxy*);
- In the backup replica, *After* transfers the last result to its *Protocol* node (topics *aft2pr_S*) and updates its own state with the checkpointed state.
- If the *Primary* crashes, the *Recovery Node* reestablishes the connection between the *Client* and the *Backup* becoming now *Primary* alone (while a new *Backup* is not created). The *Recovery Node* uses a service (*recovery*) to reconnect the *Protocol* node of the *Primary* alone server the *Proxy* of the *Client*.

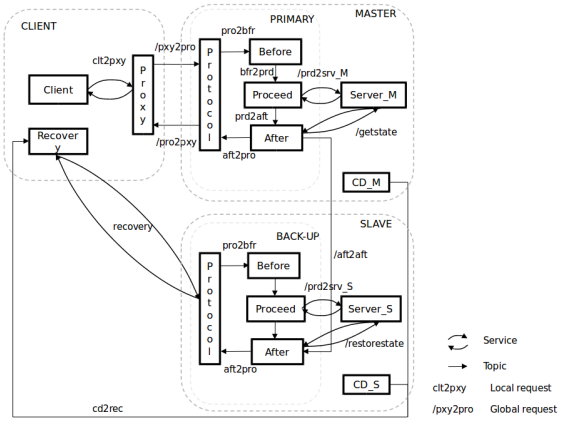

Fig. 3: Implementation of PBR on ROS

This example shows that ROS provides a convenient executive support to map a graph of software components, such as our FTMs. The key point now is to check to what extent ROS enables the dynamic adaptation of the graph for AFT.

B. Adaptation of FTM

Three types of adaptation can be identified: i) updating the current FTM, ii) switching from one FTM to another, and iii) composing the an FTM with another one. The first type implies a revision of the design or the implementation of the FTM. The other two are used to comply with parameters evolution (FT, A or R). In all cases, the same services are required. Some are provided by ROS or the underlying OS and some have been developed in-house.

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. 4.

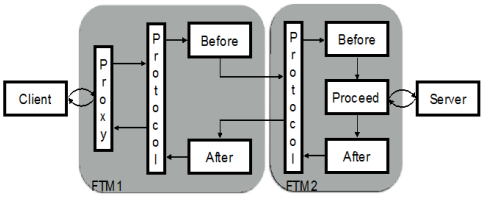

Fig. 4: Composition of two FTMs

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;
- 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 stoping 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.

A key issue during adaptation is dynamic binding between nodes. Two situations may arise: i) a node must subscribe and/or publish to an existing topic, ii) nodes needs to communicate through a new topic. These two situations are treated in the same manner: if a topic does not exists, it is automatically created by the *ROS master* when a node starts to publish to it. However, these bindings are realized during a node initialization and ROS does not provide commands to change them afterwards. Thus, to achieve dynamic binding we have added some ad-hoc APIs to our nodes. With these APIs, accessed through specific topics, we are able to select to which topics a node publishes or subscribes to. Note that these ad-hoc APIs are also useful beside adapation. For example, we use these APIs to enable the transition that occurs after the *Primary* fails (cf. section III-A), in order to dynamically bind the backup server, and its FTM, to the client.

In summary, AFT is possible on ROS. However, it lacks some essential features. In our prototype, a node's life cycle

(stop, start) is controlled directly through Unix signals. Dynamic binding is achieved through implementation of custom methods in the nodes and we have developed a specific node, called the *Adaptation Engine*, to orchestrate the adaptation.

IV. Case Study

This section gives some performance results of our approach on a case study, a *Car Control* simulation. The experiments have been run on a PC with the following characteristics: Ubuntu 14.04 Trusty, Processor Intel i5 Dual Cores 2,5 GHz (20 000 BogoMIP), 16 Go DDR3 of RAM. In the experiments, initialization, processing, recovery, and adaptation times have been collected.

We have developed an application to prevent car crashes like an embedded front radar does in a real car. Several FTMs have been evaluated. The initialization time is almost the same for every FTM and it is due to the ROS launch command: the launch time is around $0.5s$ ($0.4s$ for LFR and PBR, $0.3s$ for TR). The difference is due to the launch time of the master which controls every communication and establishes the Topics and the Services.

Even if PBR and LFR have the same number of components, the cycle of PBR is around $4ms - 5ms$ whereas the cycle of LFR is around $5ms - 6ms$. This small difference is due, in our implementation, to the synchronization between the $Leader$ and the *Follower* (requests and notifications). For instance, when the car is driving at $50km.h^{-1}$ the impact on the distance estimation is about $1,4cm$.

The recovery time for PBR or LFR corresponds to the time spent to connect the *Protocol* node of the slave replica (*Backup* or *Follower*) to the *Proxy*. The value varies between $1ms$ and $2ms$. Consequently, a Client request won't be handled during this time window. For a car driving at $50km.h^{-1}$, the car will go on moving forward for $3cm$ at most before the front radar request is processed.

The composition of PBR or LFR with the TR mechanism increases the number of component running in parallel from 17 to 23. The computing time is around $7ms$ to $10ms$. The recovering time also correspond to the duration of the reconnection of the slave replica (*Backup* or *Follower*) with the *Client*. The composition of FTMs implies launching and connecting *Nodes*, so the duration is of the same order of magnitude of the initialization time (i.e. $0.3s$ and $0.1s$).

This time overhead is one of the main problems with ROS. In our application, $300ms$ is an incredibly long reaction time when a failure occurs. The given results were obtained with all the cores activated. Reducing the number of cores will scale up the results accordingly. The performance issue calls for an optimized implementation of the ROS middleware on (*ideally*) a very efficient real-time microkernel.

V. Lessons Learned and on-going work

This work aimed at evaluating to what extent ROS was an appropriate execution support to implement Adaptive Fault Tolerance and component-based FTMs. Two positive aspects emerge: firstly, ROS provides a notion of components at runtime (*Node*), a node being a Unix process ; and secondly both asynchronous and synchronous communication models are available for nodes interaction.

The negative points are essentially related to the management of components and their interaction channels at runtime. The components are activated even if unused which slows down execution. So, having multiple applications on the same computer attached to complex FTMs (e.g. more than 17 processes running in parallel per application) is an issue regarding the implementation of AFT on ROS. A second negative point is the dynamic binding between ROS nodes. A solution was found by implementing custom services in nodes. Last but not least, the control over components execution (e.g. suspend, activate) required by AFT to modify the component graph at runtime was missing in ROS. Again, a solution was based on Unix features and some custom services.

To conclude this paper, ROS is not a perfect executive support for AFT. While it facilitates the implementation of our component model for FTM, it lacks some key features for adaptivity, especially regarding dynamic binding. However, we have shown in the paper how these weaknesses can be circumvent with some Unix services and some ad-hoc implementation. The main drawback is the performance overhead. A minimized version of ROS with an optimization of the *Nodes* execution is required for industrial embedded systems. Nevertheless, ROS is still a good middleware for a proof of concept for CBSE concepts and Adaptive Fault Tolerance.

Future work will be carried out in cooperation between the LAAS-CNRS and Renault-Nissan. The main objective is the design of an agile development process for fault tolerance mechanisms in embedded systems. A second objective is to apply AFT in safety critical applications, in particular, considering the results of safety critical systems analysis to determine and classify safety mechanisms. From the classification, we aim at defining possible adaptation patterns with respect to realistic evolution scenario. Last but not least, a special attention will be paid to the executive support since the performance of AFT is real issue.

References

[1] Jean-Claude Laprie. From dependability to resilience. In *38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*, pages G8–G9. Citeseer, 2008.

[2] KH Kim and Thomas F Lawrence. Adaptive fault tolerance: Issues and approaches. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*, pages 38–46. IEEE, 1990.

[3] C Mani Krishna and Israel Koren. Adaptive fault-tolerance fault-tolerance for cyber-physical systems. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 310–314. IEEE, 2013.

[4] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

[5] V Kozaczynski and Jim Q Ning. Component-based software engineering (cbse). In *icsr*, page 236. IEEE, 1996.

[6] Clemens Szyperski, Jan Bosch, and Wolfgang Weck. Component-oriented programming. In *Object-oriented technology ecoop'99 workshop reader*, pages 184–192. Springer, 1999.

[7] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.