Towards Adaptive Fault Tolerance on ROS for Advanced Driver Assistance Systems

Matthieu Amy, Jean-Charles Fabre, Michaël Lauer

Towards Adaptive Fault Tolerance on ROS for Advanced Driver Assistance Systems

M. Amy[1], J.-C.Fabre[2], M. Lauer[3]
CNRS-LAAS, Ave du Colonel Roche, F-31400 Toulouse, France
Univ de Toulouse, [2]INP, [3]UPS, LAAS, F-31400 Toulouse, France
[1]Technocentre RENAULT, F-78280 Guyancourt, France

***Abstract*— The use of over-the-air updates has attracted very much interest these last few years with the software-intensive development of embedded systems in the car industry. The development of autonomous driving and ADAS (*Advanced Driver Assistance Systems*) renders over-the-air updates mandatory, for both user satisfaction and economic reasons. How to make sure that remote updates of critical ADAS do not have an impact on safety? This is the question we tackle in our work with a major car manufacturer. This paper is a progress report. We summarize our approach involving AFT (*Adaptive Fault Tolerance*) implemented on ROS (*Robot Operating System*), describe the simulation platform we have developed to experiment and validate over-the-air updates of ADAS and AFT, and finally draw some lessons learnt and perspectives.**

I. INTRODUCTION

Automotive embedded systems are expected to evolve during their service life, in order to cope with changes of various nature due to maintenance activities or additional features requested by users. For many reasons including economic reasons, over-the-air updates, e.g. additional features installed remotely into cars, are of prime interest for the car manufacturers. This capability has been demonstrated by *TELSA* and is currently one of the motivations for *Adaptive AUTOSAR*.

To this aim, the first challenge is to have a runtime support enabling dynamic software updates to be carried out. ROS is a possible candidate. ROS is a middleware for implementing distributed applications that is used in many applications, from robots (e.g. Robonaut developed by NASA – *National Aeronautics and Space Administration*) to autonomous vehicles (e.g. the Crusher military off-road ground autonomous vehicle developed by NREC – *National Robotics Engineering Center*). A second challenge is related to the side effect of a functional update on dependability. It is of course mandatory to adjust the fault tolerance mechanisms of the updated application to maintain its dependability properties. This requires separation of concerns, isolation of both application code and fault tolerance code into error confinement areas, and dynamic binding facilities between runtime components. We have proposed a framework and developed several conventional fault tolerance mechanisms on ROS to analyze to what extend they can be easily updated.

Our objective is to validate the approach with critical *Advanced Driver Assistance Systems* (ADAS). We are developing a simple *Traffic Jam Pilot* system (TJP) able to drive a car autonomously in a traffic jam. Our experimental platform is composed of a redundant hardware system running the TJP control system on ROS and controlling a virtual car using a simulator, the GAZEBO Sim. Based on an FMECA (*Failure Modes Effects Critical Analysis*), the TJP was equipped with several fault tolerance mechanisms.

Our on-going work consists in applying the approach of *Adaptive Fault-Tolerance* (AFT) we have investigated on ROS to the update of ADAS. We aim at analyzing the effect of a fault in the control system on the behavior of the vehicle. We plan to analyze the impact of functional updates of the dependability of the system, and implement adaptive fault tolerance to make the system resilient.

In this paper, we summarize recent results and draw some perspectives of our on-going work. In section II we describe our approach for implementing adaptive fault tolerance on ROS. In Section III we describe the experimental simulation platform to experiment AFT on *Advanced Driver Assistance Systems*. In Section IV we draw the initial lessons learnt from this on-going work, and mention our future plans.

II. ADAPTIVE FAULT TOLERANCE WITH ROS

A. Basic concepts of AFT

Adaptive fault tolerance means that fault tolerance mechanisms attached to applications need to be updated when conditions change during the service life in the system. The conditions are related to application characteristics; fault tolerance requirements consecutive to a risk analysis and FMECA leading to determine the criticality level of the application and the required fault tolerance mechanisms (FTM); fault tolerance mechanisms assumptions related to the application structure and behavior; and related fault models, namely the type of faults it is able to tolerate.

In this paper, we do not analyze AFT in detail and we refer the interested reader to several papers on the subjects [1,2,3,4].

The main interest of AFT is its ability to update FTMs to maintain compliance with some dependability requirements and assumptions. An FTM should remain consistent with the safety analysis when a change occurs, in particular after an over-the-air update of an embedded application. Such flexibility is essential, we would say mandatory, to keep the system resilient, i.e. dependable in the presence of changes [5].

Two basic concepts are essential to implement Adaptive Fault Tolerant computing, as demonstrated in [6]:

- *Separation of Concerns at runtime*: this concept is now well-known at design time, but it is also very important at runtime; it implies a clear separation between the application code and the fault tolerance mechanisms. The connection between the application code and the FTM must be clearly defined. The FTMs should be disconnected and replaced by a new one through standardized connectors.
- *Componentization and dynamic binding*: the first idea is that fault tolerance software are 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.

The main benefits of component-based AFT with respect to pre-programmed adaptation is clear: separation of concerns at runtime, componentization and dynamic binding enable FTMs to be more easily updated *a posteriori* during the system lifetime. Pre-program adaptation implies that all possible undesirable situations are known at design time, which is difficult to anticipate regarding new threats (attacks), new failure modes (obsolescence of components), or simply adverse situations ignored or forgotten during the safety analysis.

In short, fine grain adaptation of FTMs improves maintainability of the system from a non-functional viewpoint. Over-the-air updates of ADAS may have an impact on fault tolerance requirements, a strong argument in favor of AFT.

B. Component model and reconfiguration with ROS

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 involves the assembly of nodes, which is akin to component-based approaches. Such an assembly is referred to as the computational 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 topic. The client/server model relies on bidirectional synchronous communications through the concept of service. These high-level communication models introduce modularity and flexibility in software systems.

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 services and topics to the ROS Master. It is the only node that has a comprehensive view of the computational graph. When a node issues a service call, it queries the master for the address of the node providing the service and then it sends its request to this address.

In order to be able to add fault-tolerance mechanisms to an existing ROS application in the most transparent manner, we need to implement interceptors. An interceptor provides a means to insert functionality, such as safety or monitoring nodes, into the invocation path between two ROS nodes. To this end, a relevant ROS feature is its remapping capability. 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 to interceptor nodes.

ROS provides two computational models: client-server (by mean of services) and publish-subscribe (by means of topics). The proposed approach is illustrated with the client-server model in the paper. The application of the proposed framework to the publish-subscribe computational model is on-going work. In short, it requires the capture of the termination of the computation within a ROS node to synchronize replicas.

C. Implementing Componentized FTMs

In this section, we first present the generic computational graph we use for implementing FTMs on ROS. An implementation of a duplex FTM, a *Primary Backup Replication (PBR)* combined with a *Time-Redundancy (TR)* mechanism has been done to validate our proposal.

We assume that the reader is familiar with conventional replication techniques for fault tolerance (see. [7] or [8] for more details about well-known replication techniques). The objective is not to present and compare such techniques. The objective is to show the capabilities of our framework to combine, compose, decompose, adjust FT mechanisms. Depending on a large number of performance criteria (e.g. coverage, timing, communication overhead, HW resources, etc.), the system manager may prefer one FTM instead of another. This analysis is out of the scope of this paper.

1) Generic Computational graph

We have identified a generic pattern for the computational graph of a FTM. Fig. 1 shows its application in the context of ROS. All components are ROS nodes. A node, the *Client,* uses a service provided by a *Server* node. The FTM computational graph is inserted between the two nodes thanks to the ROS remapping feature. Since *Client* and *Server* must be re-launched for the remapping to take effect, the insertion is done off-line, i.e. the binding between nodes is static. The FTM

nodes, topics, and services are generic for every FTM. Implementing an FTM consists in specializing the *Before*, *Proceed*, and *After* nodes with the adequate behavior of the required FTM.

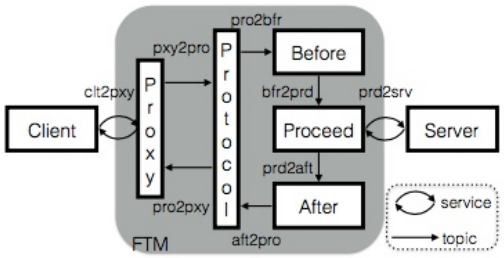

Fig. 1. Generic computational framework for FTM

2) Application to Primary-Backup Replication

We briefly illustrate here the approach and the *Before-Proceed-After* framework, through the use of a *Primary-Backup Replication* (PBR) mechanism. Three computers are needed: the CLIENT site hosting the *Client* node and the ROS Master, the MASTER site hosting the ***primary replica,*** and the SLAVE site hosting the ***backup replica***.

- We present the behavior of each node, the topics and services used through a request/reply interaction between a node *Client* and node *Server* (cf. Fig. 2).
- *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 other action for PBR.
- *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 SLAVE replica (topics aft2aft S);
- *Protocol* gets the result (topics aft2pro) and sends it to *Proxy* (topics pro2pxy);

The *Before-Proceed-After* (BPA) framework synchronizes replicas in normal operation, i.e. in the absence of faults. It also runs the recovery procedure when the failure detector (an external/independent node) signals the crash of a replica.

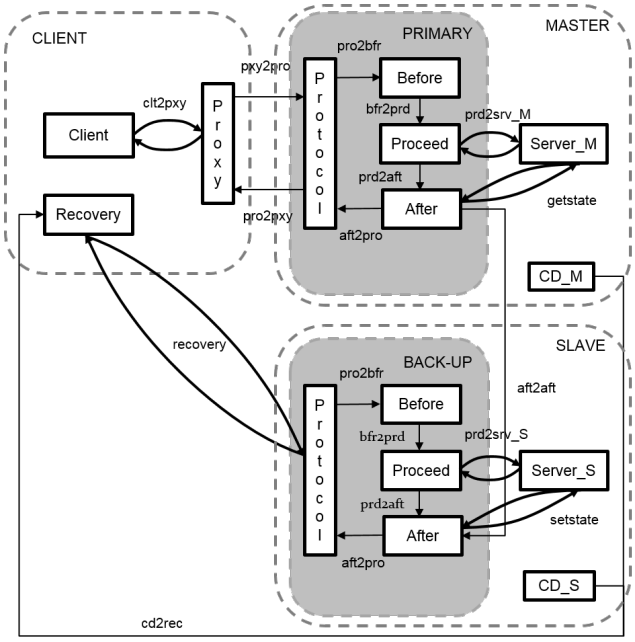

Fig. 2. Before-Proceed-After framework applied to PBR

The main advantage of this approach is that a slight change in the protocol can be performed easily just by replacing/updating one of the *Before, Proceed, After* nodes. A second advantage of the approach is that the inter-replica protocol is clearly independent of the application service. The main drawback is that ROS does not provide command to change bindings between nodes after their initialization.

3) Composition of several FT Mechanisms

The generic computational graph for FTM given in Fig. 1 is designed for composability. The key feature is that a *Protocol* node can substitute for a *Proceed* node.

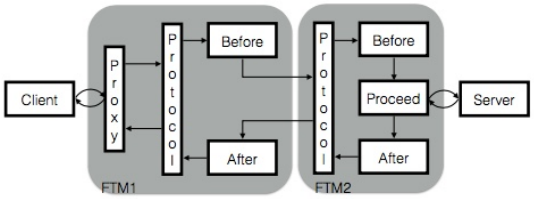

Fig. 3. Principle of composition for FT mechanisms

With respect to request processing, a *Protocol* node and a *Proceed* node exhibit the same interfaces: in short, a request as input, a reply as output. Hence, the composition of several FT mechanisms relies on replacing the *Proceed* node of a mechanism by a *Protocol* and its associated *Before-Proceed-After* nodes of a second mechanism, as shown in Fig. 3. 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.

Two composition scenarios are shortly described below.

PBR+TR. PBR is of interest to tolerate crash faults whereas TR tolerates transient value faults. TR tolerate transient fault by repeating the computation and voting on the results. As a second FTM (FTM2), the *After* node of TR is responsible for triggering the repetition of the computation (involving *Before* and *Proceed*) and the vote on the various results produced before forwarding the reply to the *After* node of FTM1, which implements PBR.

PBR+Assertion. Assertions are often derived from safety analysis. For instance, "*the electronic lock of the steering column must not activate when the speed of the vehicle is over 10 km/h*". This safety rule can easily be translated into a logical expression, i.e. a Boolean function. The second FTM (FTM2) is responsible for the verification of such assertion implemented in its *After* node. When the assertion is false it may raise an alarm and return an error signal to FTM1 that will send it back to the *Client* for emergency action.

D. Lessons learnt

The main advantage of ROS is to provide concepts for componentization and separation of concerns. This is important for the design of adaptive fault tolerance mechanisms, but also for their implementation. The proposed framework *Before-Proceed-After* inspired from Aspect Oriented Programming [9] enables various fault tolerance mechanisms first to be decomposed into isolated components that can be customized according to the needs, but also to facilitate the composition of several mechanisms in a row.

Separation of concerns enables the FTM to be externalized with respect to the functional code, namely the application code. The generic FTM mechanisms we propose are independent of the nature of the application. This independence between FTMs and application simplifies their i) externalization and ii) their composition. The benefits of separation of concerns have been demonstrated in many ways for non-functional properties (replication, security, tracing, etc.) using Meta-Object Protocols [10] in the past as in [11] and was the main motivation for Aspect Oriented Programming. The main interest is to avoid gluing non-functional mechanisms with application code, an approach making maintenance and evolution very difficult to achieve. Separation of concerns has a lot of merits at design, implementation and validation time, but also at runtime since the application and the attached FTM can be located into isolated components. Isolation is a key feature for dependable computing.

From an implementation viewpoint, ROS nodes provide isolation in a protected address space for error confinement. The services and the mechanisms can be isolated from each other, and thus an error within the application (e.g. memory violation) does not impact the FT mechanism. Although we assume that the implementation of any FTM is zero-default (huge validation effort following ISO 26262), this isolation property also applies to nodes implementing the FTMs.

The static binding between nodes is a drawback because it can only be manipulated a priori and off-line. This is a weakness of ROS regarding fine grain over-the-air updates of componentized FTM: an update can only be finalized after restarting the application.

It is worth noting however that the validation of a new mechanism or even an updated version of it, must be carried out off-line following an intensive validation process, in particular fault injection as far as fault tolerance is concerned.

Ideally, dynamic binding would improve the efficiency of over-the-air updates of ADAS for instance. As we have shown previously, only few or even just one node belonging to our *Before-Proceed-After* framework may need to be updated. So, why restarting the whole application? Just uploading a new node and binding it to its companion nodes would suffice. This is not possible at present with ROS, version 1. There is no API to manipulate nodes and bindings at runtime. However, these APIs can be emulated with dedicated logic added to some nodes, using underlying Unix features and commands.

Last but not least, the ROS master is a single point of failure in the current version of ROS. This problem could be tackled using DMTCP [12], a library for checkpointing Unix multi-threaded processes as a whole. This might be of interest in the short term since a POSIX compliant kernel is part of the upcoming Adaptive Autosar platform whose aim is to facilitate dynamic reconfiguration and updates of embedded software.

However, 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.

III. EXPERIMENTAL PLATFORM FOR AFT & ADAS

The objective of the platform is to provide the support for several activities: i) the simulation of critical advanced driver assistance systems, ii) a target for implementing over-the-air update of ADAS, ii) a set of use cases for safety analysis, iv) the implementation of adaptive fault tolerance techniques and v) their validation by fault injection.

The use of ROS for the implementation of any ADAS is essential to validate our AFT approach and our *Before-Proceed-After* framework.

Instead of performing functional updates and related FTM adaptation on a real car, we have used a simulator to implement the car behavior. The GAZEBO-Sim tool enables a vehicle and its environment to be simulated with a quite interesting level of detail. Sensors and actuators can be developed and integrated into a model of vehicles on roads.

A. *The GAZEBO Simulator*

GAZEBO is a very well designed open-source tool for a 3D robot simulation [13] that is very well connected to ROS. It is based on the Open Dynamics Engine (ODE) and provides many libraries of simulated components.

It is thus possible to represent different items for modeling very realistic situations; each item (*links*) can be parameterized with physical realistic characteristics like, mass, inertia, stiffness, coefficient of friction, damping factor, etc. The physical connections between items are of course part of the model (*joints*). Several *joints* are available: fixed or enabling sliding or rotation between solids. All items, *links,* must be interconnected by connectors, i.e. *joints*.

Fig. 4. Simple graphical example with GAZEBO for modeling the TJP

The model format with GAZEBO is SDF (*Simulation Description Format*) deriving from XML. The use of GAZEBO with ROS requires a conversion of the models into URDF format (*Unified Robot Description Format*). The interaction of the models with the environment, e.g. sensors, is developed using plugins in C++.

B. *Simulation of the TJP*

The TJP (*Traffic Jam Pilot*) is a control and command system providing autonomous driving in traffic jam conditions. Its role is to drive a vehicle without any intervention of the driver at low speed, i.e. below 35 km/h. The TJP automatically adjusts the speed of a vehicle (the *follower car*) to maintain a safe distance from the vehicle that is ahead (the *leader car*). The development of the TJP is based on three simple use cases: i) vehicle positioning on the road, ii) vehicle control in traffic jam, and iii) emergency braking.

1) Simplified specifications

Gap and Safety Distance: The positioning of the car on the road is essential and is based on several sensors used, in particular to implement the following features:

- computation of the distance between the follower and the leader car, denoted *gap distance*.
- computation of the speed of the car that is used to adapt the *safety distance* (the faster the speed, the greater the safety distance)

Vehicle control: The control system must be able to accelerate and brake the vehicle to implement the TJP. This aspect must consider different parameters of the physical object in real life to make our simulation realistic. We also need to take into account the side effect of level of acceleration and strength of braking on the passengers of the vehicle, for their own comfort and safety. A too severe braking may injure passengers or trigger the airbag system by mistake!

Emergency braking (EB): in our specifications we have included an additional Emergency Braking system (EB) that will be implemented independently of the TJP for two reasons. The first reason is a failure of the TJP should not impair the EB capacity to stop the vehicle in case of emergency. The second reason is that some external unanticipated event not detected by the sensors (pedestrian crossing between the two vehicles) should also lead to an emergency stop. Such situation can easily be simulated using physical sensors.

2) Architecture and implementation

The experimental platform is composed of two parts: the simulator of the car and the control system running on ROS. The TJP was designed using UML tools and is composed of the following functions:

- the *controllerPid* implementing the control algorithms;
- the *distanceSecurityCalculator* responsible for the processing of distances between vehicles;
- the *measureManager* responsible for the computation of the speed of the vehicle;
- the *commandManager* responsible for the management of the various commands delivered to the actuators of the vehicle;

These functions are ROS nodes implementing the control system on a redundant hardware platform. The physical redundancy enables implementing various FTMs. Virtual *laser sensors* and *speed actuators* are used in GAZEBO to control the car for the TJP. One physical *Ultrasonic sensor* is used for the Emergency Braking system. The management of the *Ultrasonic sensor* is also a ROS node in addition to the 4 mentioned above.

The interaction between GAZEBO and the physical platform running the TJP on ROS is realized thanks to additional features enabling a seamless connection between GAZEBO and ROS application software.

Functional testing was performed through a series of experiments. A dynamic driving profile was assigned to the *Leader* car, the *Follower* objective being to follow the *Leader*, observing the safety distance, the limits in deceleration and braking. In case of an unanticipated event detected by the

ultrasonic physical sensor, the emergency braking is activated. The priority of this task is higher that any command sent to the car by the *controllerPid*.

The implementation on ROS is illustrated in Fig. 5.

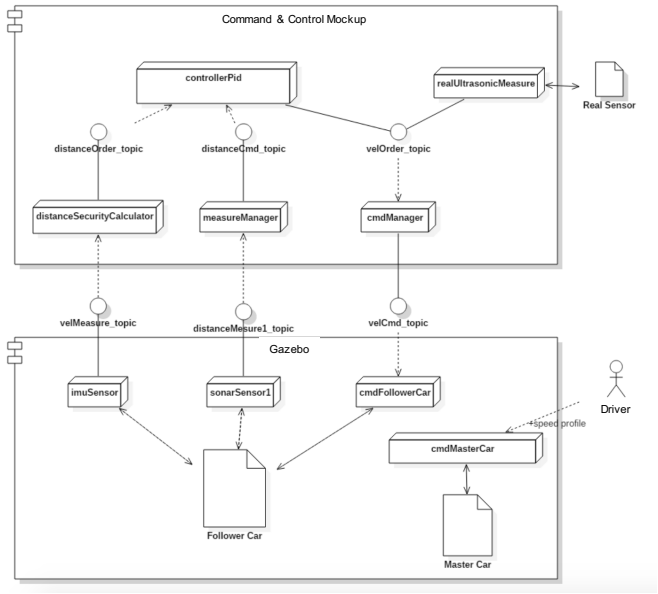

Fig. 5. Component view of the TJP implementation

C. *Safety analysis and fault tolerance*

The safety analysis has considered simple fault models affecting the major functions and equipment for the TJP. Physical faults leading to a crash of a computer or a transient fault affecting the sensors were considered (value errors).

Entity	Failure mode	Risk	F	G	Safety mechanism
Raspberry	Crash	No speed command received by the car.	2	4	Passive replication
Simulator: Gazebo	Crash	No more car, the TJP does not exist anymore.	2	4	Alarm message + stop application
Roscore	Crash	No communication between nodes and topics anymore.	1	4	Alarm message + stop application
Virtual sensor: IMU	Crash	Set point distance no longer adaptive.	1	3	Alarm message + stop application
	Inconsistent data	Wrong set point distance.	3	3	Error message + Temporal redundancy
Virtual sensor: Laser	Crash	Collision with the front car.	1	4	Alarm message + Physical redundancy
	Inconsistent data	Wrong distance to the front car.	3	4	Error message + Physical redundancy
Real sensor: Ultrasonic	Crash	No detection of a close obstacle.	1	4	Alarm message + stop application
	Inconsistent data	Emergency braking wrongly activated.	3	4	Temporal redundancy

Fig. 6. Extract of the simplified FMECA Analysis.

This simplified safety analysis led to the identification of several FTMs (see Fig. 6). The frequency (F) and the gravity (G) range from 1 to 4, 4 being the most serious for the gravity. The result can be summarized as follows:

- the crash of a computer running the TJP (a Raspberry PI in our mockup) leads to a loss of the service; the solution was based on a PBR replication strategy;
- erroneous data delivered by the virtual sensor IMU (*Inertial Measurement* Unit) used to measure the speed of the vehicle was solved using TR and by computing an average value on a sliding window of values;
- erroneous information delivered by virtual laser sensors was solved by triplication and voting.

The impact of such problems on the safety of the TJP is classified ASIL D or ASIL C according to *RENAULT* experts, combining *Frequency* and *Gravity*.

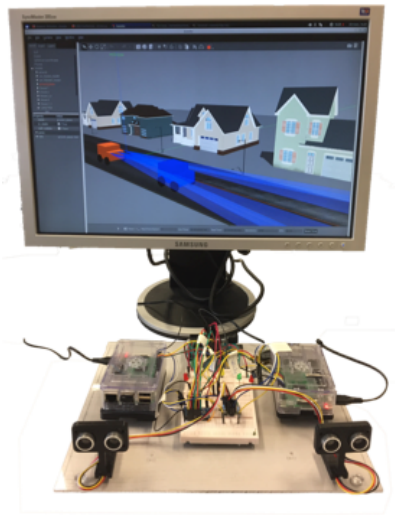

Fig. 7. Overview of the experimental platform

D. *Mockup of the experimental platform*

The Mockup (see Fig. 7) is composed of a PC running the GAZEBO simulator and an experimental platform for the execution of the TJP. The experimental platform is composed of two *Rapsberry PI 3* computers running ROS and an *Arduino* playing the role of a watchdog. Physical Ultrasonic sensors are replicated and attached to each computer.

The *controllerPid* is replicated using the PBR strategy. It is running as a primary on one *Raspberry PI* while it runs as a backup on the second one. Both replicas can receive the inputs from the virtual and physical sensors. When one Raspberry PI crashes, the watchdog triggers the switch to the backup that takes over the processing of sensor data and the computing of the commands.

Physical fault injection (loss of power) was used to simulate a crash of a computer. SWIFI was used to simulate transient faults of the IMU and the laser sensors.

Our on-going work consists first in improving the BPA framework for asynchronous interactions between nodes, i.e. following the publish-subscribe computational model. The solution we have today is based on the capture of the termination of processing of an input message by a subscriber using a *Terminate* statement corresponding to a library function. All ROS nodes developed in this simulation have a main loop, a terminate statement is invoked at each iteration.

The second activity consist in revising and extending the FMECA analysis to investigate new fault tolerance strategies, ranging from simple restart to user-defined assertion-based strategies. The interesting work consists in analyzing to what extent the ROS implementation of the BPA framework provides sufficient flexibility at design and runtime. Some measurements will be performed concerning the development time, concerning the uploading time (complete or partial) and the suspension time for the system activity.

IV. CONCLUSIONS AND PROSPECTIVES

An ideal runtime support for Adaptive Fault Tolerance should provide separation of concerns, componentization and dynamic binding at runtime. As shown in previous work [6], this ideal executive support should exhibit the following features at runtime: i) control over component's life cycle (add, remove, start, stop), ii) control over interactions for creating or removing bindings. This is our frame of reference to discuss the adequacy of ROS as a runtime support for AFT.

In our approach, a component is mapped to a ROS node providing memory space segregation. The binding between components relied on topics managed by the ROS Master. The remapping facilities were used to manipulate the software configuration, off-line only, to adjust the FTM mechanisms. Although it is not a core feature of ROS at present, dynamic binding was possible but ROS does not provide a specific API to manage such connection between components. Additional code is required to manage dynamic binding, using facilities provided by the underlying Linux operating system. The ROS master is a single point of failure in the architecture. Solutions exist to overcome this problem and new versions of ROS should provide new solutions.

The proposed (*Protocol)Before-Proceed-After* framework was of high interest to design FTM for later adaptation and to customize them easily according to the needs. This framework is also of interest to compose FTMs on a case-by-case basis without any impact on the functional software. More details can be found in [14]. We are convinced that AFT is essential for a safe management of over-the-air updates of ADAS. Many ADAS are currently available and one objective of car manufacturers is to maintain, update, but also sell *a posteriori* new software-implemented ADAS. The proposed approach enables FTMs to be easily specialized for a given ADAS release and it matches the *Agile* development processes [15] considered today in the car industry. Over-the-air updates are thus of interest for both functional and non-functional software.

The integration of our approach in the development process of ADAS at Renault and SDK is one of our objectives. The mockup should help us to validate the AFT approach with several versions of ROS and ADAS of different nature, including safety critical ones targeting autonomous driving.

ACKNOWLEDGEMENTS

The authors wish to deeply thank our master students for the implementation of the mockup: Sarah AMAR, Jules LE BRETON, Daniel LOCHE, Hélène PHOURATSAMAY, and David RAYMOND, from the department of *Electrical Engineering and Automation* of ENSEEIHT (www.enseeiht.fr), the *School of Engineering* of the *Toulouse Institute of Technology*.

REFERENCES

[1] 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.

[2] 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.

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

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

[5] J.-C. Laprie, "From Dependability to Resilience," in 38th IEEE/IFIP International Conf. on Dependable Systems and Networks (DSN), 2008.

[6] M. Stoicescu, J.-C. Fabre, M. Roy, "Architecting Resilient Computing Systems: A Component-Based Approach For Adaptive Fault Tolerance", Journal Of Systems Architecture, Elsevier Eds, Ref. Jsa-D-16-00131R1, Nov. 2016.

[7] Delta-4: A Generic Architecture for Dependable Distributed Computing, David Powell (Eds), Springer, ISBN 978-3-642-84696-0.

[8] 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.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar et Maeda, « Aspect-Oriented Programming », *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, 1997, p. 220–242

[10] G. Kiczales, J. d. Rivières and D. G. Bobrow, *The Art of the Metaobject Protocol,* MIT Press, 1991.

[11] J.-C. Fabre and T. Pérennou, "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach", *IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems*, pp. 78-95, 1998.

[12] 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.

[13] Official Gazebo website http://gazebosim.org to get started with Gazebo.

[14] M. Lauer, M.Amy, J.-C. Fabre, M. Roy, W.Excoffon, M. Stoicescu, "Engineering Adaptive Fault Tolerance Mechanisms for Resilient Computing on ROS", IEEE HASE 2016 (High Assurance System Engineering), Orlando, USA, , Jan. 2016.

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