Transparent Fault Tolerance for Real-Time Automation Systems

Invited Paper

Burak Hasircioglu
Turkish Aerospace Industries (TAI) Inc., Turkey
burakhasircioglu@gmail.com

Yvonne-Anne Pignolet
ABB Corporate Research, Switzerland
yvonne-anne.pignolet@ch.abb.com

Thanikesavan Sivanthi
ABB Corporate Research, Switzerland
thanikesavan.sivanthi@ch.abb.com

ABSTRACT

Developing software is hard. Developing software that is resilient and does not crash at the occurrence of unexpected inputs or events is even harder, especially with IoT devices and real-time requirements, e.g., due to interactions with human beings. Therefore, there is a need for a software architecture that helps software developers to build fault-tolerant software with as little pain and effort as possible. To this end, we have designed a fault tolerance framework for automation systems that lets developers be mostly oblivious to fault tolerance issues. Thus they can focus on the application logic encapsulated in (micro)services. That is, the developer only needs to specify the required fault tolerance level by description, not implementation. The fault tolerance aspects are *transparent* to the developer, as the framework takes care of them. This approach is particularly suited for the development for mixed-criticality systems, where different parts have very different and demanding functional and non-functional requirements. For such systems highly specialized developers are needed and removing the burden of fault tolerance results in faster time to market and safer and more dependable systems.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; **Real-time systems**; **Dependable and fault-tolerant systems and networks**;

KEYWORDS

automation systems, real-time, fault-tolerance, transparency, dependability

ACM Reference Format:
Burak Hasircioglu, Yvonne-Anne Pignolet, and Thanikesavan Sivanthi. 2018. Transparent Fault Tolerance for Real-Time Automation Systems: Invited Paper. In *IoPARTS'18: 1st International Workshop on Internet of People, Assistive Robots and ThingS , June 10, 2018, Munich, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3215525.3215538

IoPARTS'18, June 10, 2018, Munich, Germany

ACM ISBN 978-1-4503-5843-9/18/06…$15.00
https://doi.org/10.1145/3215525.3215538

1 INTRODUCTION

When building today's automation systems, human beings always need to be taken into account, since they need interact with a plant's devices and its robots, either for supervision or maintenance task, or in collaborative work environments, where robots support human beings [5]. To prevent hazardous conditions, it is of prime importance, that real-time behavior is not violated and that the system can react to changes quickly, even when parts of the system fail. Therefore, the main design goal of our framework is to make monitoring and fault tolerance transparent to application services. This is achieved by a set of interfaces that need to be implemented by the developer and some code running in the background. This code ensures that the state of different replicas of a service stay synchronized and that a standby service takes over when the primary service crashes.

As a consequence, application developers do not have to write code for monitoring and fault tolerance management, as long as the services implement the corresponding APIs and declare the required supervision and fault tolerance properties as illustrated in Figure 1.

Does this all sound a bit a like magic? Let us look under the hood of this fault tolerance framework to understand how it works. The ingredients to make this happen are

- A service-oriented architecture
- A fault tolerance configuration mechanism
- A supervision service for monitoring and redundancy management
- Automatic state transfer mechanisms (cold, warm and hot standby, workby).

The service-oriented architecture provides the foundation for building applications using loosely coupled services. The fault tolerance configuration mechanism provides customization of required fault tolerance properties by configuration rather than implementation. The first two items provide the software developers the possibility to ignore, how the fault tolerance and state transfer mechanism work exactly. All they need to do is to ensure the services conform to the API and configure the services according to their required fault tolerance properties. This allows them to dedicate all their focus on the functional and non-functional properties of the services they are responsible for.

In this paper, we shed light on the transparency aspects of this fault tolerance framework. We focus on the description of the supervision service and its state transfer mechanisms mentioned above. Since we heavily rely on the runtime environment's support to

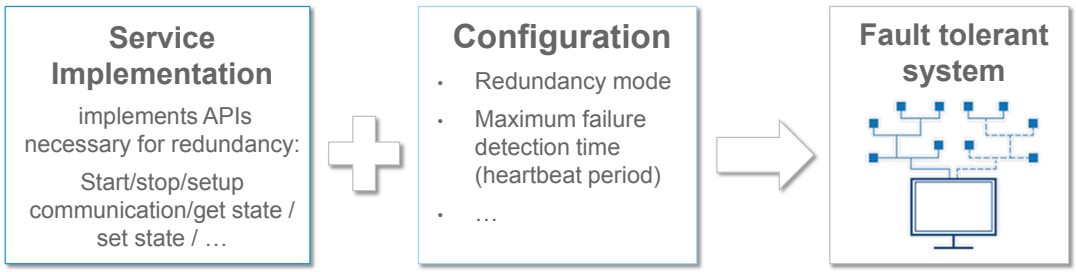

Figure 1: Application service developer does not have to deal with fault tolerance and redundancy. Customization of required fault tolerance properties by configuration, not implementation.

achieve our goals, we first describe the service-orientation characteristics in our automation system, which facilitates transparent fault tolerance.

2 SERVICE-ORIENTED ARCHITECTURE

Automation systems realize monitoring and control of factories or other plants with fewer and fewer human work involved. Typically, such systems comprise several devices that are interconnected via a communication network to carry out system-level or local automation functions. Some devices interface directly with the so-called primary equipment for the purposes of sensing and/or actuation. To cope with the complexity and heterogeneity of the tasks involved, *service orientation*[1] can be applied as an architectural principle when designing automation systems.

In this context, a *service* is a set of functionalities that can carry out some logic, e.g., run control applications, interfaces to robots, HMI/Web interfaces, etc. We consider an *application*, i.e., a certain function of an automation system (e.g., control of a production task), to be composed of a set of *service instances*, each configured to carry out a part of the function. An application functionality can be implemented on a single device or on multiple devices. In order to carry out an application logic, services need to be able to communicate using a flexible, lightweight pub/sub *service bus* that allows services to be decoupled from one another. This decoupling includes the architectural requirement that no service will enter an unsafe state due to the failure of another service. In other words, each service needs to guarantee that it will either work correctly or fail in a safe manner, regardless of the messages exchanged with other services. The architecture comprises an *execution environment* (light gray boxes in Figure 2) which provides the environment to execute services on a given device, and also a *service runtime environment* (white boxes in Figure 2) which provides an environment that allows a service and its runtime to interface to the underlying execution environment and to communicate with other services (i.e., using a pub/sub service bus). Together with a hardware and an operating system abstraction layer, this forms an architecture for an automation device that can execute services.

The service runtime environment, in white in Figure 2, includes the service runtime management and in general the service interfaces. The runtime environment is basically the only common part

[1]cf. micro-services architectural style which has gained a lot of popularity in the Internet/Enterprise/Cloud world (https://martinfowler.com/articles/microservices.html).

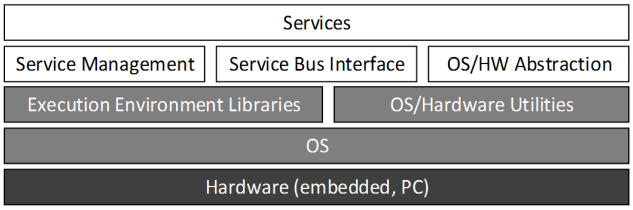

Figure 2: Service-oriented architecture.

to all services. The *service runtime manager* (SRM) uses the API functionality that all services must implement. The service runtime management contains service execution management (e.g., start, stop), monitoring and diagnostics (e.g., reporting health), and configuration management. The SRM does not perform the actual operations for the services, it simply triggers them, while the service provides the appropriate implementation. That is, when a service instance is launched, it is in fact an SRM executable that is launched as a separate process in the underlying operating system. The SRM has the responsibility to coordinate the sequence of events with respect to loading the service code; initializing it; managing its execution (start, stop commands); monitoring the status and sending heartbeats on service health; coordinating more complex processes such as performing a dynamic update of the service code, performing a fail-over or initiating a self-testing procedure. All of these operations are triggered or coordinated by the SRM, but they all result in calling the service model API that the service itself needs to implement. For instance, only the service knows what it means to initialize its logic/state, to start, to configure itself or to gracefully stop. To enable mixed-criticality systems, services need to tolerate pre-emption and/or provide worst case execution time guarantees for each part of the service. Together with declarative configuration files and compute resource models, this allows to compute a device allocation and schedule for the service execution.

Every service implements the service model to allow it to be managed by the service runtime manager SRM. The model includes a constructor which initializes the name, ID, path to configuration files, service version (needed for dynamic updates) and methods for initialization, to shut down the service, to start the service, and to stop the service. The question is of course, who or what manages services. The answer comes from the way we separate

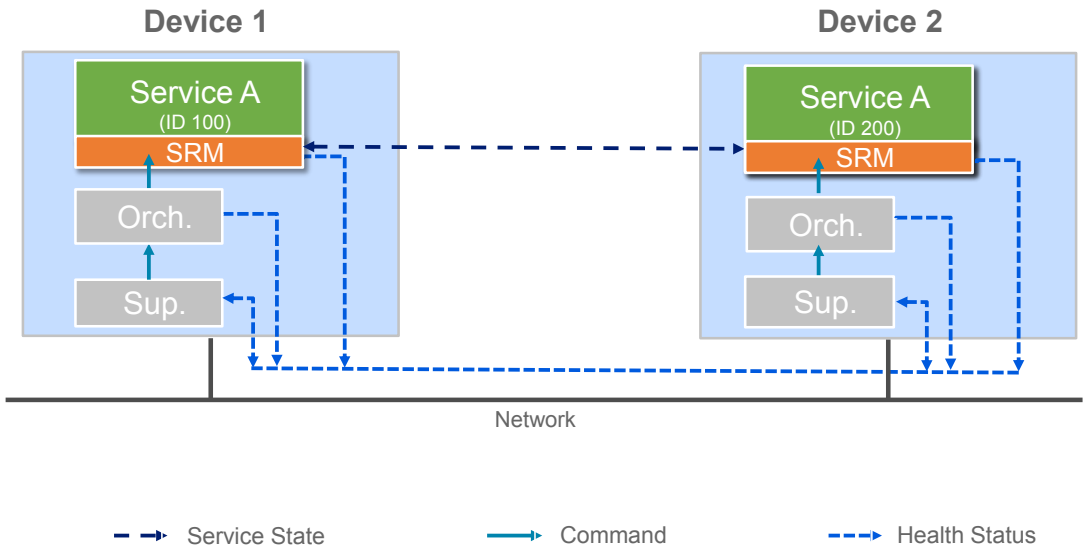

Figure 3: Information flow between application, supervision and orchestrator service. All traffic (commands, state transfer, health status messages) is transmitted over the service bus, within and between devices.

services between *support* and *application* services. The latter are the ones that carry out the application logic while the former are specialized services meant to provide the means for management at the service-level and execution environment level (monitoring, diagnostics, testing). Note that support services abide by the same service API as application services.

Among the support services, the *Orchestrator* plays an important role in the coordination of all service runtime management (start stop, configure, reconfigure, etc.) operations including launching all services as part of the bootstrapping procedure or when services fail and require re-launching. In fact, this is the first service that is brought up by default on boot. Another important support service, the *Supervisor* is responsible for monitoring the status of all services and systems. This is necessary to achieve transparent fault tolerance, while it can be exploited for other purposes as well, e.g., for monitoring information and diagnostics.

To summarize, the Orchestrator service can be used for system features relying on a centralized authority within a device, while the Service Runtime Managers associated with each service serve can enable distributed coordination features. Furthermore, it facilitates the reuse of code for lifecycle management, dynamic updates, and recovery management

In the next sections, we describe how the SRM and, the orchestrator and the supervisor cooperate for transparent fault tolerance.

3 SUPERVISOR SERVICE

Each service reports its health status via heartbeats (with a configurable frequency) that the Supervisor is listening to. This knowledge allows the Supervisor to enforce different levels of fault tolerance. For instance, when services stop responding (e.g., health status messages timeout), the Supervisor signals the Orchestrator that the service instance needs to be relaunched and/or that a replica needs to become active as part of the fail-over mechanism (if replicas exist). The Orchestrator can further impose policies on relaunch, i.e., service can relaunch in isolation, or other dependent services need to be relaunched as well.

To this end, the SRM which is common to all services and enabling the management of services at runtime is exploited. The SRM of each launched service, publishes heartbeat messages with the health status of the service. The Supervisor subscribes to these health status messages of all service replicas. The Supervisor then manages the replicas by issuing commands to the SRMs of the replicas via the orchestrator. These mechanisms can be used within and across devices and thus increase the service availability. In addition, the SRM lets stateful services exchange information about their state so replicas are kept up-to-date, as illustrated in Figure 3.

How the supervisor should react to the failures of services is described in a configuration file. In this file, the following information is declared for each supervised service instance

- If the service runs on the same device (local)
- If it is to be relaunched upon failure
- Its priority level (to determine the active replica)
- Its redundancy mode (none, cold, warm, hot standby and workby).

The SRMs of alive services, send information about their health status in so-called *heart-beat messages* periodically, in a configurable interval. When a service is started, the Supervisor sets the service to ACTIVE or PASSIVE status. An ACTIVE service executes its application logic and produces outputs. A PASSIVE service is a service that is launched and may or may not run the application logic but it does not produce outputs.

The Supervisor is responsible for the following functionalities:

- Failure detection: monitors the status of services, devices and systems using health status messages.

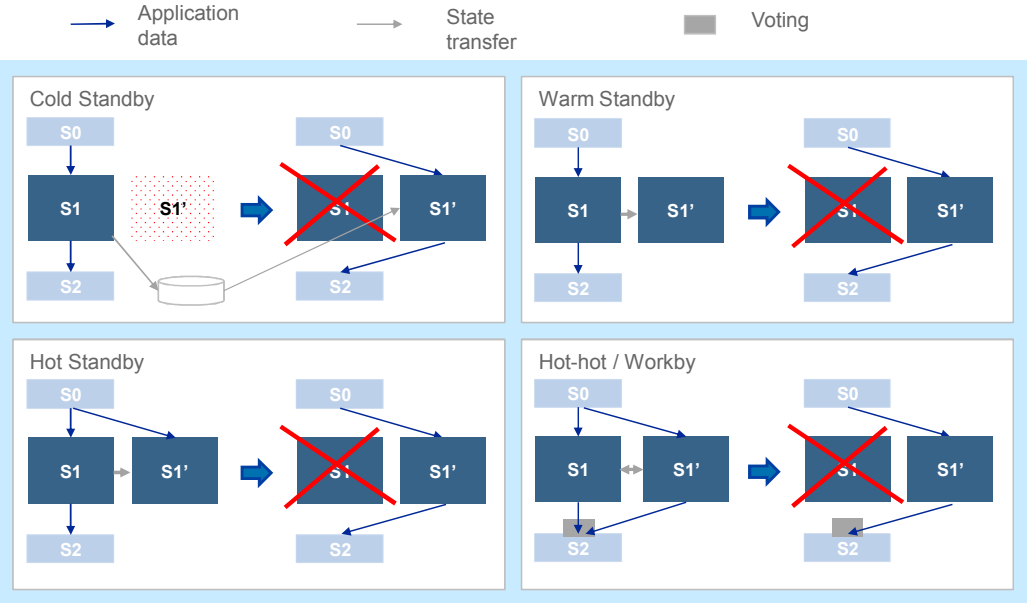

Figure 4: Patterns supported by our framework. S1 and S1′ are replica of the same service (same type). For each pattern the scenario before and after the failure of S1 is depicted. The arrows indicate the information flow: dark blue for application-level traffic, light grey for state transfer. For the workby pattern, a voting module has to be used in the Service S2 to decide how to process the output of S1 and S1′.

- Group membership: maintains a list of alive services, devices and systems.
- Fault tolerance state machines: maintain current state of services, devices and systems, trigger actions when their states change.
- Replication management: react to failure detection and group membership events using state machines and issuing commands to the orchestrator service.

Fault tolerance (FT) is realized using different state machine classes for different redundancy modes or patterns [8], see Figure 4. We explain the approach for services; device and system fault tolerance can be provided in a similar manner. Multiple state machine objects, one for each service, contain the service fail-over and fail-back logic. To illustrate the fault tolerance mechanism, we consider the following scenario with two redundant services (e.g., Service A with instance IDs 100 and 200), running on the same or different devices in warm standby mode. After bootstrapping, both these services are in passive (standby) mode (i.e. service status is PASSIVE). In passive mode, both Service A (ID 100) and Service A (ID 200) receive the same input (i.e., they subscribe to the same topic). The service publishes data only when it is set to active mode (i.e. service status is ACTIVE). The SRM of each service publishes health status messages periodically using Pub/Sub channels of the service bus. The Supervisor service subscribes to these messages to monitor the health of every service. Then it chooses the active service, let's say Service A (ID 100) in our example, among the alive services according to their priorities specified in the XML file, updates its internal state (i.e. the FT state machine corresponding to the service and its replicas) and sends an activation command for Service A (ID 100) to the action topic. This command will be received by the Orchestrator service and will be forwarded to services via the command topic. The SRM of Service A (ID 100) will then set Service A (ID 100) to the ACTIVE status, which will activate the output. If Service A (ID 100) crashes, the Supervisor will notice that it does not receive health status messages from Service A (ID 100) anymore. As a consequence, it will update its FT state machine and send an activation message to Service A (ID 200) via the orchestrator. When receiving this message, the SRM of Service A (ID 200) will set the service STATUS to active, which causes Service A (ID 200) to take over the role of the primary service and produce output. The supervisor also has the option to relaunch a failed replica, as specified in the configuration file.

The Group membership functionality provides a list ("view") of all operational elements in the system (services, devices, systems..) a given point in time. It can be used for online arbitration and reconfiguration (as an alternative for static configuration descriptions). To deal with networking errors, an algorithm like Viewsnoop [6] can offer consistency guarantees.

We decided to implement the functionality of supervision, fault-tolerance and recovery in one service so we can use calls and callbacks instead of sending messages over the service bus for lower latency. In other words, regarding the trade-off between modularity and communication overhead we gave more weight to the communication overhead.

Note that the supervisor service not necessarily has to be hosted on every node. Its functionality can be carried out from a service on a different (physical or logical) device. Each supervision service has its own configuration. This configuration determines not only

which services are run on which devices, but also which supervisor service is responsible for them. For hot and warm standby, the relaunch attribute indicates if the service will be relaunched (on the same device) after failure. When an active service fails, the new active service is determined according to priorities.

Note that with this approach, the fault tolerance of orchestrator and supervisor can be handled as for any other service. The failure detection is also done based on the health status messages. All replicas of the supervisor will hear each other's heartbeats and the orchestrators' heartbeats and monitor their aliveness. The relaunch of a failed orchestrator or supervisor service is then performed by the active orchestrator. The detection of a situation where both orchestrators are failed, causes a relaunch of the orchestrators, achieved by a hardware watchdog.

4 STATE TRANSFER

Once running, a service may develop a state (e.g., a set of objects stored in memory), which can include for instance, the value of key variables or connection objects to use the service bus. This state can be serialized/de-serialized to allow state synchronization for fail-over purposes. It can also be transferred between two versions of the service code to support (close to) seamless dynamic updates. How to serialize, de-serialize, transfer, including the definition of state is implemented by the service. The mechanisms to synchronize and transfer state are implemented in the SRM. This has the effect of alleviating the burden on the service developer to implement the fail-over/dynamic update mechanisms.

We now look at how snapshots of a service's state are taken and transferred to another service or to a local/remote storage location. This is useful for standby fault tolerance patterns and dynamic updates for service upgrades/patches. In the first use case, snapshots of the state of a service are taken, e.g., periodically or event based, and sent either to another service that might be running elsewhere, or saved in a database etc.

When the standby service has to take over, it can use this stored information to continue as seamlessly as possible. In the second use case, a service can be updated with new features or patches without restarting the device on which they run, not affecting other services, and (depending on the actual change) even keeping the operation of the service uninterrupted. The design goal of this feature is to enable such a state transfer in a generic manner, without having the developers of application services worrying about the details of state transfer. The application service developer is responsible to define states and events that define when the state should be transferred or stored. If necessary, they can also provide the application-specific logic to ensure consistency between replicas.

For each service, the important variables that describe its current state can be declared as part of a `State` class belonging to this service. The service runtime manager can be configured to take snapshots periodically or when triggered by the service, transfer the state and update the service with a new state, using the interfaces defined. The generic parts for the state transfer and recovery are the responsibility of SRM's recovery manager (RM), which is a submodule of SRM.

We illustrate the interaction of these components with an example where the state is stored in a file for cold standby, depicted

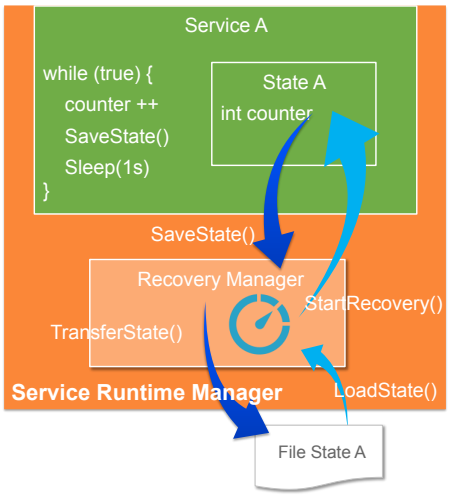

Figure 5: Example service and interaction with recovery manager of SRM.

in Figure 5 (of course the state could also be stored in a different manner and place). The green box represents Service A, here simply incrementing a counter, saving its state and then sleeping for a second. The state of Service A thus consists of an integer value. Around Service A, we have the service runtime manager, of which we highlight the recovery manager functionality. When triggered, the recovery manager requests the current state of service and stores it to a file. If Service A needs to be relaunched, e.g., after a crash, the recovery manager reads the state from the file and updates the service state. For other patterns, the recovery manager can transfer the state to another service, see Figure 6.

In addition to saving state, services can also log events, e.g., the receipt of messages before processing them [8]. Services can log events and save their state at any place in the implementation. Upon activation due to a failed service, a passive service will load the last saved state, replay the log, continue with its service logic, thus replacing the failed service.

To avoid domino effects and inconsistent global states, dependencies between services are not allowed. In other words, it is the responsibility of each service to adhere to a publish/subscribe paradigm and ensure that it is always is in a safe state, regardless of what the state of other services is. That means it must run correctly even when the data stream it relies on is corrupted or missing. In such cases it must adapt its output accordingly and inform the diagnostics/alarm services of the automation system.

5 SERVICE BUS REQUIREMENTS

In order to provide fault tolerance, the service bus has to be able to satisfy some QoS requirements, at least for some of the control traffic, regardless of the technology used below. First, the service bus should offer an abstraction that re-sends messages if they are not received and that the temporal order for each sender can be maintained. To ensure that services are not shutdown or restarted

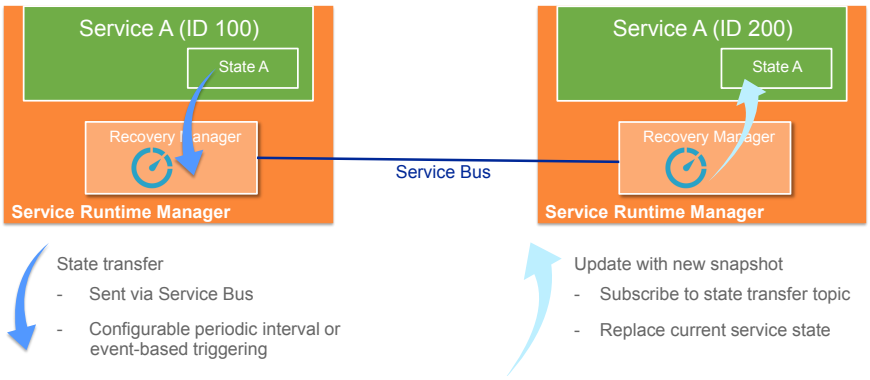

Figure 6: State transfer between services, for warm and hot standby patterns. For the workby pattern, both services send their state and process the other replica's state before updating their state and continuing with the application logic.

by accident by receiving a command message twice, an exactly-once-semantics should be established for such messages. This can be achieved by waiting an amount of time which is the multiplication of "the number of retries" and the heartbeat period between consecutive commands and setting the heartbeat period to a low value. This has performance penalties and uses more bandwidth. To avoid this, a special QoS for commands can be defined, in which the "age" of the service and the message are compared before delivery (no messages older than age should be delivered).

6 RELATED WORK

Transparent fault tolerance is an important design goal for many dependable systems, as it is notoriously hard to develop, debug and test distributed systems. Different approaches that alleviate the implementation hurdles of active and passive replication based on various message passing systems have been proposed in the past. Prior work offering developers a wide range of replication and failover strategies has focussed on stateless functional languages (Haskell [9], Erlang [2]), RPC-based architectures [3, 7] and more recently Web Services [1, 4]. They take the same approach, where the applications need to implement an interface and a special layer takes care of (re)starting and synchronization according to a specified fault tolerance policy. However, they are either tightly coupled to a language, communication layer, or virtual machine implementation. Furthermore, none of the related work targets automation systems with a publish-subscribe service bus and real-time requirements, to the best of our knowledge.

7 CONCLUSIONS

In this paper, we gave a brief introduction to our fault tolerance framework. We have described how the supervisor monitors services and how it interacts with the orchestrator to guarantee fault tolerance, despite real-time constraints. Thus our approach can be used in demanding cyber-physical automation systems. Moreover, we showed how the recovery manager of the service runtime manager enables a transparent state transfer between services running on the same or on different devices. These features are useful for both time-triggered and event-driven systems. For critical services, it is of great advantage to be able to save and transfer state to replica, while for others a more light-weight approach where a service is relaunched automatically when its failure is detected is more suitable. With our approach, the fault tolerance level is chosen explicitly during engineering by configuring the system accordingly. This has the additional benefit, that the real-time behaviour in case of faults can be analysed statically before the deployment and performance bottlenecks can be identified and mitigated. Due to the fact that the fault tolerance policies can be configured independently for each service, no re-design for fault tolerance is necessary when a service is used in a different context with different requirements.

This framework has been implemented and tested and can serve as a basis for automation systems with mixed criticality requirements, e.g., for the collaboration of robots and human beings. Due to the generality and modularity of its architecture, it can also be integrated in other code bases to provide the benefit of a transparent supervision and management approach for fault tolerance.

REFERENCES

[1] Navid Aghdaie. 2005. *Transparent Fault-Tolerant Network Services Using Off-the-Shelf Components.* Ph.D. Dissertation. University of California, Los Angeles.
[2] Joe Armstrong. 2010. erlang. *Commun. ACM* 53, 9 (2010), 68–75.
[3] Thomas Becker. 1994. Application-transparent fault tolerance in distributed systems. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems.* IEEE, 36–45.
[4] Vijay Dialani, Simon Miles, Luc Moreau, David De Roure, and Michael Luck. 2002. Transparent fault tolerance for web services based architectures. In *European Conference on Parallel Processing.* Springer, 889–898.
[5] Jeff Fryman and Bjoern Matthias. 2012. Safety of industrial robots: From conventional to collaborative applications. In *Robotics; Proceedings of ROBOTIK 2012; 7th German Conference on.* VDE, 1–5.
[6] Rachid Guerraoui, David Kozhaya, Manuel Oriol, and Yvonne-Anne Pignolet. 2016. Who's On Board?: Probabilistic Membership for Real-Time Distributed Control Systems. In *35th Symposium on Reliable Distributed Systems (SRDS).* IEEE, 167–176.
[7] Nitya Narasimhan. 2001. *Transparent fault tolerance for Java remote method invocation.* University of California, Santa Barbara.
[8] Titos Saridakis. 2002. A System of Patterns for Fault Tolerance. In *EuroPLoP.*
[9] Robert Stewart, Patrick Maier, and Phil Trinder. 2016. Transparent fault tolerance for scalable functional computation. *Journal of Functional Programming* 26 (2016).