SORRIR: A Resilient Self-organizing Middleware for IoT Applications [Position Paper]

Jörg Domaschka
joerg.domaschka@uni-ulm.de
Ulm University
Germany

Christian Berger
Hans P. Reiser
Philipp Eichhammer
University of Passau
Germany

Frank Griesinger
Jakob Pietron
Matthias Tichy
Franz J. Hauck
Gerhard Habiger
Ulm University, Germany

ABSTRACT

The increasing societal pervasion and importance of the Internet-of-Things (IoT) raises questions regarding the fault tolerance and robustness of IoT applications as these increasingly become part of critical infrastructures. In this position paper, we outline novel ideas that focus on the design of a resilient and self-organizing execution platform for IoT applications called SORRIR. Its main ambition is to simplify, alleviate and accelerate the development, configuration and operation of resilient IoT systems. We follow a holistic approach which is based on a novel design process, a library containing resilience mechanisms and a robust execution platform that is equipped with monitoring and self-organizing capabilities. The goal is that developers only need to specify the desired resilience degree without having to worry about the technical, implementation-level details of employed resilience mechanisms.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**; Embedded systems; Redundancy; • **General and reference** → *Reliability*.

KEYWORDS

Internet of things, resilience, self-organization, middleware

1 INTRODUCTION

Over the last decade cloud computing has tremendously changed the way distributed and large-scale applications are being implemented and operated. As part of this evolution, cloud computing has been providing a well-suited environment for the operation of backend systems for Internet-of-Things (IoT) deployments. This, in turn has led to an increase in importance of IoT installations:

(i) IoT systems are not only becoming more widespread in everyday life, but part of critical infrastructures in Industry 4.0, product service systems, and distributed control systems such as autonomous driving support. The spread of IoT systems also reaches the eHealth sector. This creates an increasing social dependence on reliable operations of these systems. (ii) IoT systems are significantly growing in size (with respect to number of nodes and the geographical area they span) turning them into complex systems. (iii) Latency-critical applications require that parts of the functionality of IoT systems are offloaded to edge devices. (iv) The development cycle for software in general, but also for IoT software decreases while (v) in parallel, the diversification of the sensor actuator, and IT infrastructure landscape increases.

In summary, the development of large-scale IoT systems gets increasingly complex and uses shorter development and innovation cycles. Conversely, the importance of reliability, resilience, and fault tolerance increases making testing and development more time-consuming and complicated. We claim that these contradicting demands can only be satisfied through the separation of concerns: developers shall focus on the business logic of IoT applications, while domain experts provide building blocks for fault tolerance. We suggest that fault-tolerance mechanisms are automatically and dynamically woven into IoT applications so that the system can evolve over time adapting to changing requirements.

This position paper outlines goals and principles of the SORRIR middleware tackling these challenges. It presents the early SORRIR architecture, cf. Figure 1, and introduces the conceptual components that enable the holistic SORRIR approach to resilient, distributed IoT systems. For doing so, SORRIR spans aspects from development time, deployment time, and operation time: The SORRIR programming model fosters the separation of resilience and business logic; the application orchestrator supports the composability of business and fault-tolerance domain; the SORRIR run-time system realises a fault-tolerant, resilient platform for running and monitoring distributed IoT applications in geo-distributed infrastructures.

The remainder of this paper is structured as follows: Section 2 sketches the state of the art for IoT architectures, programming, and operation. It derives requirements for SORRIR, whose architecture, and eco-system are subject to Section 3. In Section 4 we briefly discuss related work before we conclude.

2 BACKGROUND

This section captures the state of the art in IoT architectures, programming and orchestration and distils requirements for SORRIR.

2.1 IoT architectures

ISO/IEC 30141 defines a high level Internet of Things reference architecture [9] similar to others found in literature [4, 7, 10, 16]. It

M4IoT '19, December 9–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3365379

characterises IoT installations as highly distributed systems with physically separated and remotely connected sub-systems (sites). Networking per site is based on heterogeneous technology and may incorporate non-IP proximity networks. The main entities in an IoT landscape are IoT devices (sensors and actuators), IoT gateways, and services: IoT devices bridge digital and analogue world. IoT gateways provide access to wide-area networking (WAN). Applications are built from services with well-defined interfaces.

According to [10] MQTT, CoAP, AMQP, and HTTP are the primary IoT communication protocols outside proximity networks. All of them make use of IP. The NIST Network of Things architecture [16] stresses the importance of movable devices and the need for aggregation while data is sent from devices towards services.

2.2 IoT components and stacks

Truong et al. [14], just as ISO/IEC 30141 [9], stress the importance of composing IoT systems from components. Node-RED[1] is a tool for composing flows from individual building blocks, called nodes. Node-RED flows interact with external entities such as IoT devices using dedicated io nodes. The scope of a flow is limited to one entity (gateway, server), even though attempts for distribution exist.

AWS IoT[2] provides an IoT operating system with add-on services such as cloud and database access as well as handling of communication errors. Besides, IoT functionality integrates well with the AWS EC2 landscape making Serverless a preferred approach for introducing custom logic. Similar initiatives exist for other public cloud providers such as Google[3] and Microsoft Azure[4].

2.3 Orchestration and operation

Cloud computing and containerisation have been dominating the orchestration of software components over physical infrastructures mostly through IaaS, PaaS, and CaaS abstraction layers. Orchestration comprises all steps necessary to acquire resources from cloud platforms and deploy software on them [2]. Hence, cloud orchestrators take a specification of the application as well as deployment instructions and constraints about the deployment as input.

TOSCA [3] is an open standard for describing topology and orchestration of cloud applications. Its goal is to support portability and operational management of cloud applications and services across their entire lifecycle. While it comes with its own eco-system such as vinery, it has also been adopted by OpenStack in its orchestrator Heat[5] and Cloudify[6]. CAMEL follows a similar approach [12], but adds run-time information to the overall model. Also, it supports cloud provider agnostic specifications of applications.

Docker swarm[7] is a basic orchestrator for Docker containers using Docker compose as a specification language. Kubernetes[8] is the most wide-spread orchestrator. For specifications it uses a custom format or Helm. k3s[9] is a lightweight Kubernetes distribution suited for IoT devices.

[1]https://nodered.org/
[2]https://aws.amazon.com/iot
[3]https://cloud.google.com/solutions/iot/
[4]https://azure.microsoft.com/en-us/services/iot-hub/
[5]https://wiki.openstack.org/wiki/Heat
[6]https://cloudify.co/
[7]https://docs.docker.com/engine/swarm/
[8]https://kubernetes.io
[9]https://k3s.io/

2.4 Requirements

The discussion so far shows that IoT systems span large geographic domains and are vulnerable to failures of hardware, communication systems, and others. Programming models for IoT systems are either limited (cf. serverless) or very open, while the management and orchestration approaches are mostly based on containers or virtual machines and do not incorporate IoT gateways or even IoT devices.

For realising the vision of SORRIR to increase the fault tolerance of IoT systems without burdening application developers, we identified four requirements based on literature review and conceptual considerations. All of them are assume that IoT applications are composed from services and further aim at the separation of concerns between business logic and fault-tolerance mechanisms:

Requirement R.1: In order to achieve fault tolerance for IoT applications and infrastructures, it is necessary to identify failures in the infrastructure. These events need to be acted upon such that erroneous conditions are mitigated. This demands for a resilient orchestration, execution environment, and monitoring system.

Requirement R.2: The reference architecture [9] foresees resilience and fault tolerance as a cross-domain issue. From this consideration, it follows that fault-tolerance mechanisms should not be realised on a per-service basis, but rather be provided as building blocks to be added to applications and components based on their individual needs. This separation of concerns also eases the re-use of application components in different environments.

Requirement R.3: Different fault-tolerance mechanisms exist for addressing different kinds of failures. Each of them demands different pre-requisites from the business logic. A fault-tolerant execution environment needs to understand the interdependencies between business logic and fault-tolerance mechanisms. This demands for a fault-tolerant programming and component model that should also address the need for composability as requested by [9].

Requirement R.4: A configuration mechanism should (i) provide the capability to compose IoT applications from components developed with the given programming model; (ii) allow the IoT operator to configure the desired reliability at the level of individual components and at the application level.

3 SORRIR

Based on the requirements from Section 2.4, SORRIR starts off to fulfill the following four primary goals: (i) to decouple fault-tolerance mechanisms from application logic (R.3); (ii) to foster the re-use of fault-tolerance mechanisms by providing libraries of often-used and generally applicable mechanisms (R.2); (iii) shift the decision for fault tolerance from development-time to run-time (R.4) due to a (iv) fault-tolerance–enabled orchestration, run-time, and monitoring (R.1).

3.1 System model and Terminology

The SORRIR system model and terminology widely aligns with ISO/IEC 30141 [9], but is enhanced with terminology from cloud and edge computing. From an infrastructure perspective, we differentiate between IoT devices, IoT gateways, and execution sites. We do not impose a strict hierarchy on the communication model; in particular, we allow that messages travel vertically, e.g., from device to gateway, and horizontally, e.g., from gateway to gateway.

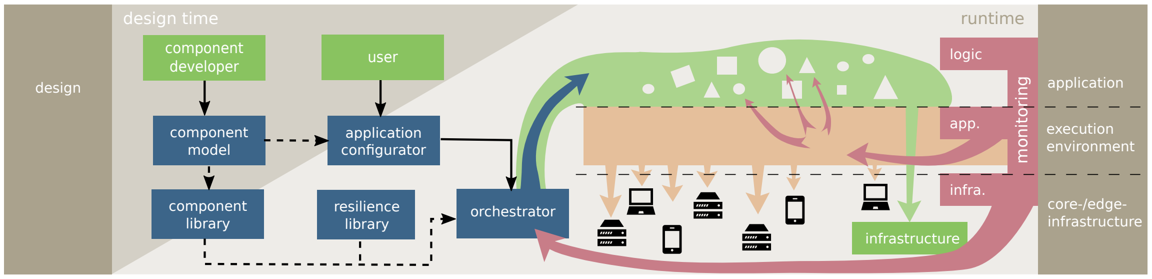

Figure 1: Overview of SORRIR components and their interplay.

IoT devices comprise sensors and actuators and reside in the field. They provide little configuration capabilities, and re-programming their software usually requires manual steps. IoT devices either have direct access to WAN, communicate with the WAN through one or multiple gateways, or communicate with other IoT devices in a mesh network to reach a gateway.

IoT gateways receive messages from IoT devices and relay them upstream (and vice versa). Depending on their computational capabilities they may preprocess and aggregate data. Due to their stable connection they are easier to access for re-programming. Yet, gateways do not provide a uniform API for this kind of access. For SORRIR we allow that IoT gateways interact with each other.

Execution sites represent any type of computational resources accessible from IoT gateways over network. Execution sites pool computational resources and allow the instantiation of services; hence they provide an IaaS or PaaS abstraction. The compute resources provide better computational capabilities than gateways and due to a well-defined API it is relatively easy to roll out new functionality. Any execution site resides at a specific geographical location and has a specific location in the overall network topology. Concretely, execution sites may come for instance as regions of public cloud providers, private clouds and compute clusters owned by the user, and edge computing sites provided by network operators.

Applications and components An application encapsulates the business logic. From a user perspective, we assume that an application is composed of a set of instantiable components. Components can be directly instantiated at IoT gateways and on execution sites, after compute resources have been allocated at that site. The scale out factor of a component (at a site) is the overall number of instance of that component (at that site). Components may define communication dependencies between them that are mapped to the instance level in various ways (1:n, 1:m<n, m:n, ...).

3.2 Architecture and Eco-system

In order to achieve its goals, SORRIR provides mechanisms at design/development time, composition/deployment time, and finally during run-time as sketched in the overview in Figure 1. This is compliant with [11] that claims that IoT systems should amongst others support application development, deployment, device management, service management, monitoring.

Design-time For design-time, SORRIR provides a dedicated programming model. This programming model supports the specification and implementation of SORRIR components. Depending on the use of specific SORRIR design patterns, base classes and APIs, each component is capable of supporting a specific failure model and allows that different types of fault-tolerance components be combined with the application component. Similarly, matching fault-tolerance mechanisms can be derived for groups of SORRIR components and their communication dependencies. The SORRIR component library provides commonly used components.

Configuration and composition Application operators make use of the SORRIR configurator to compose distributed applications from SORRIR components. The output of the configurator is a SORRIR application configuration that defines the application structure including the communication topology. What is more, the application configuration defines desired resilience levels per-component, per communication channel, and for groups of components. Taking SORRIR's focus into account, we expect to the configuration to be concrete about the placement of components over IoT gateways and execution sites.

Orchestration The SORRIR orchestrator bridges the design-time aspects and run-time aspects. Its input is a SORRIR application configuration. For putting this specification into operation the orchestrator needs to fulfil three basic tasks: (i) acquire compute resources over the execution sites defined in the application specification and provision a SORRIR run-time system on these resources. (ii) Enhance the SORRIR components defined in the application specification with matching resilience mechanisms that are provided through a resilience library (see below). (iii) Deploy the enhanced components over the acquired infrastructure as well as IoT gateways. Where applicable it may also reconfigure sensors and IoT gateways. For deployed applications, the SORRIR orchestrator provides the ability to change the current configuration and resilience specifications.

Operation and monitoring While it is the task of the orchestrator to acquire compute resources and instantiate components on these resources, the run-time system is the part of SORRIR that surveils all parts of the system: Its basic task is to monitor computational resources, IoT gateways, and IoT devices. The SORRIR run-time system is a decentralised, distributed application that, considering its crucial role in the overall architecture, needs to be resilient and fault-tolerant. An instance of the SORRIR run-time system runs on each of the IoT gateways and compute resources. We refer to these instances as nodes.

Each node permanently shares its current state with its peers. The current state at least contains information on the run-time status of components scheduled on this node and the current load; it may also include application-specific information in case this has been defined in the application specification. When problems or errors occur, the SORRIR run-time system notifies both the orchestrator as well as running components through the APIs defined by the SORRIR component model. For instance, it could notify a component that some sensor is no longer accessible, that some other component has failed, or that the system is under attack.

4 RELATED WORK

Recent research works propose a broad variety of approaches to fault tolerance, in particular investigating on how certain aspects of fault tolerance can be brought to IoT systems to increase their resilience towards software or hardware faults [1, 5, 6, 8, 13, 15, 17].

Tsigkanos et al. present a roadmap for resilient IoT systems which identifies state-of-the-art techniques and methods to establish resilience in the face of disruption as well as future directions for engineering resilient IoT systems, e.g., having the edge infrastructure consumed as a full-fledged utility, abstracting business logic management from infrastructure capabilities as well as autonomous control and self-healing [15].

Employing redundancy is also a possibility for making IoT systems fault-tolerant. Terry et al. argue that within IoT systems, devices like sensors or actuators are often fail-stop, thus application fault tolerance can often be provided without resorting to active replication (even with only an additional component) [13]. Further, the necessary costs for redundancy can be reduced by having the IoT platform discover and select nearby devices that can report similar events, such as motion or presence, and that support similar actions, like turning on a light, for instance, devices could automatically connect to nearby operating hubs and then switch hubs as soon as failures occur [13]. Moreover, the concept of virtual services which use data from more than one sensor devices to replace an actual service on some faulty device can also be leveraged to incorporate fault tolerance in service-oriented IoT architectures [17]. Fault-tolerant routing between a possibly large number of small, inter-connected devices can be achieved by constructing, recovering and selecting k-disjoint multipath routes that guarantee connectivity even after the failure of up to $k-1$ paths [8].

Abreu et al. propose a modular IoT middleware for smart cities, which comprises a *Resilience Manager* that has the main task of permanently supervising activities using a *Monitor* module and, additionally, employs a *Protection and Recovery* module to trigger proper actions in case of faults, e.g., relying on other modules such as topology control as well as placement and migration modules [1].

In the domain of e-Health systems, Gia et al. describe an approach to ensure reliability which is constructed on top of a 6LoWPAN communication infrastructure and utilizes backup routing between nodes and advanced service mechanisms to maintain connectivity in case of failing connections between system nodes [6].

5 CONCLUSIONS AND FUTURE WORK

The growing spread of Internet-of-Things (IoT) systems increases their societal pervasion and importance. At the same time fault tolerance of these installations is challenged by shorter development cycles as well as larger and more complex systems. In this position paper, we present key ideas and concepts around SORRIR, a resilient, self-organising middleware for IoT applications.

Its main ambition is to simplify, alleviate and accelerate the development, configuration and operation of resilient IoT systems. SORRIR is constructed around the core idea of decoupling business logic from resilience logic. Components developed using the SORRIR programming model can be dynamically bundled with matching fault-tolerance mechanisms provided through a library. The SORRIR run-time system monitors the overall execution of applications, reacts on failures and triggers reconfigurations. Using SORRIR developers only need to specify the desired resilience degree for an application without having to worry about the technical, implementation-level details of employed resilience mechanisms.

ACKNOWLEDGMENTS

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany (grant nr. 01IS18068, SORRIR). We thank Christine Giss for her research on technical background.

REFERENCES

[1] David Perez Abreu, Karima Velasquez, Marilia Curado, and Edmundo Monteiro. 2017. A resilient Internet of Things architecture for smart cities. *Annals of Telecommunications* 72, 1 (01 Feb 2017), 19–30. https://doi.org/10.1007/s12243-016-0530-y
[2] D. Baur, D. Seybold, F. Griesinger, A. Tsitsipas, C. B. Hauser, and J. Domaschka. 2015. Cloud Orchestration Features: Are Tools Fit for Purpose?. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. 95–101. https://doi.org/10.1109/UCC.2015.25
[3] T. Binz, G. Breiter, F. Leyman, and T. Spatzier. 2012. Portable Cloud Services Using TOSCA. *IEEE Internet Computing* 16, 3 (May 2012), 80–85. https://doi.org/10.1109/MIC.2012.43
[4] F. Carrez, T. Elsaleh, D. Gómez, L. Sánchez, J. Lanza, and P. Grace. 2017. A Reference Architecture for federating IoT infrastructures supporting semantic interoperability. In *2017 European Conference on Networks and Communications (EuCNC)*. 1–6. https://doi.org/10.1109/EuCNC.2017.7980765
[5] K. Christidis and M. Devetsikiotis. 2016. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access* 4 (2016), 2292–2303.
[6] T. N. Gia, A. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen. 2015. Fault tolerant and scalable IoT-based architecture for health monitoring. In *2015 IEEE Sensors Applications Symposium (SAS)*. 1–6.
[7] Jasmin Guth, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Lukas Reinfurt. 2016. Comparison of IoT Platform Architectures: A Field Study based on a Reference Architecture. In *2016 Cloudification of the Internet of Things (CIoT)*. IEEE, 1–6. https://doi.org/10.1109/CIOT.2016.7872918
[8] M. Z. Hasan and F. Al-Turjman. 2017. Optimizing Multipath Routing With Guaranteed Fault Tolerance in Internet of Things. *IEEE Sensors Journal* 17, 19 (Oct 2017), 6463–6473. https://doi.org/10.1109/JSEN.2017.2739188
[9] ISO/IEC 130141:2018 2018. *Internet of Things (IoT) – Reference Architecture*. Standard. International Organization for Standardization, Geneva, CH.
[10] N. Naik. 2017. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*. 1–7. https://doi.org/10.1109/SysEng.2017.8088251
[11] Partha Pratim Ray. 2016. A survey of IoT cloud platforms. *Future Computing and Informatics Journal* 1, 1 (2016), 35 – 46. https://doi.org/10.1016/j.fcij.2017.02.001
[12] Alessandro Rossini, Kiriakos Kritikos, Nikolay Nikolov, Jörg Domaschka, Frank Griesinger, Daniel Seybold, and Daniel Romero. 2015. D2.1.3—CAMEL Documentation.
[13] D. Terry. 2016. Toward a New Approach to IoT Fault Tolerance. *Computer* 49, 8 (Aug 2016), 80–83. https://doi.org/10.1109/MC.2016.238
[14] H. Truong and S. Dustdar. 2015. Principles for Engineering IoT Cloud Systems. *IEEE Cloud Comput.* 2, 2 (Mar 2015), 68–76. https://doi.org/10.1109/MCC.2015.23
[15] Christos Tsigkanos, Stefan Nastic, and Schahram Dustdar. 2019. Towards resilient Internet of Things: Vision, challenges, and research roadmap. In *Proc. 39th IEEE Int. Conf. Distrib. Comput. Syst.(ICDCS)*. 1–11.
[16] Jeffrey Voas. 2016. *Networks of 'Things'*. NIST Special Publication 800-183. National Institutee of Standards and Technology. https://doi.org/nistpubs/SpecialPublications/NIST.SP.800-183.pdf
[17] S. Zhou, K. Lin, J. Na, C. Chuang, and C. Shih. 2015. Supporting Service Adaptation in Fault Tolerant Internet of Things. In *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*. 65–72. https://doi.org/10.1109/SOCA.2015.38