Automatic Integration of BFT State-Machine Replication into IoT Systems

Christian Berger
University of Passau
Passau, Germany

Hans P. Reiser
Reykjavik University
Reykjavik, Iceland

Franz J. Hauck
Ulm University
Ulm, Germany

Florian Held
Ulm University
Ulm, Germany

Jörg Domaschka
Ulm University
Ulm, Germany

Abstract—**Byzantine fault tolerance (BFT) can preserve the availability and integrity of IoT systems where single components may suffer from random data corruption or attacks that can expose them to malicious behavior. While state-of-the-art BFT state-machine replication (SMR) libraries are often tailored to fit a standard request-response interaction model with dedicated client-server roles, in our design, we employ an IoT-fit interaction model that assumes a loosly-coupled, event-driven interaction between arbitrarily wired IoT components.**

In this paper, we explore the possibility of *automating and streamlining the complete process of integrating BFT SMR into a component-based IoT execution environment*. Our main goal is providing simplicity for the developer: We strive to decouple the specification of a logical application architecture from the difficulty of incorporating BFT replication mechanisms into it. Thus, our contributions address the automated configuration, re-wiring and deployment of IoT components, and their replicas, within a component-based, event-driven IoT platform.

Index Terms—**IoT, Middleware, Byzantine Fault Tolerance, Replication, Automation, Deployment**

I. INTRODUCTION

Internet-of-Things (IoT) systems consist of a variety of heterogeneous components which are connected with each other. These components may generate and consume data, provide and demand services, and collaboratively compose an application that is used by machines or people.

Over the last years, IoT systems not only became popular within the public at large, e.g., by smart-home installations, but they were also incorporated in some resiliency-critical infrastructures, e.g., of the industrial sector [1]. A broad field of ongoing research (e.g., see [2]–[4]) aims for building resilient IoT systems by equipping these systems with resilience mechanisms—often by adding redundancy to mask faults.

A well-conceived resilience mechanism is state-machine replication (SMR): Independent and connected server replicas remain functional as a group even if up to a specific number of replicas fail [5]. A fault model describes the type of considered faults. For example, Byzantine fault tolerance (BFT) can tolerate arbitrary and malicious behavior, even collusion among faulty replicas [6]. When employing BFT replication, an intruder cannot break the availability or integrity of a replicated component as long as the attacker is unable to compromise more replicas than the specified bound.

Moreover, BFT SMR can serve as an important resilience mechanism for IoT infrastructures [7]. For instance, it can help to protect components from random data corruption as often devices such as the Raspberry Pi are not equipped with error correcting memory. Furthermore, BFT can even mask malicious behavior which is in particularly relevant if devices are likely to be deployed in dispersed (and potentially untrustworthy) administrative domains.

Although researchers have built their own BFT SMR frameworks, there are only few available for public use, e.g., [8]–[11], and they are not easy to use. In fact, these frameworks do not specifically target the requirements of IoT applications due to the following reasons:

(1) A firm separation of *client-server* roles is assumed instead of considering an architecture of individual IoT components that can be arbitrarily connected (and due to the high dynamics of IoT, this architecture may easily evolve over time thus requiring flexible adaptations).

(2) The interaction model is often assumed to be a *request-response* pattern instead of a loosely-coupled, event-driven interaction model, which is typically used in the IoT domain.

(3) Replication has to be considered from the beginning *during development* and cannot be configured later as an orthogonal option, although the latter would hide the burden of integrating desired replication mechanisms into the software.

(4) The deployment of replicas is completely disregarded—yet it is important in the IoT to be capable of mapping a desired resilient software architecture to the available hardware landscape, where devices may impose certain constraints towards the deployment process making it non-trivial.

Overall, we conclude that "classical" SMR frameworks are no perfect fit for rapid development of IoT applications due to these shortcomings. We propose that *development can be both simplified and accelerated* by automating the relevant steps necessary to integrate BFT requirements and mechanics, e.g., library configuration, replica management, key distribution, input dissemination, output consolidation and deployment decisions (distribution of replicas to available hardware devices).

By employing the commonly used SMR library BFT-SMART [8] as building block, we present a method that simplifies and automates the integration of BFT SMR in a component-based, event-driven IoT framework, thus alleviating the incorporation of resilience in IoT applications.

Further, we provide a concrete, practical implementation of our method, that is publicly available for the open-source SORRIR framework [12], which particularly serves as a model of a component-based and event-driven IoT framework.

Contributions and Structure of this Paper

Our approach addresses the mentioned challenges by starting with a *logical software architecture* (LSA) describing application components and their connections. As a second and independent step, an LSA is transformed into a *replication-enriched software architecture* (ReSA), enhancing the LSA by replication mechanisms, additional components and their configurations, finally mapped to the hardware landscape of available devices. Our approach

- proposes a building-block principle for state-machine replication that dissects replication into its functional parts which can be flexibly glued to IoT components;
- seamlessly integrates customizable BFT replication, based on the replication library BFT-SMaRt, into a component-based event-driven IoT architecture;
- automatically deploys component replicas on the hardware architecture, recognizing the constraints of the available IoT devices.

At first, we provide related work in Section II and relevant background knowledge helpful to track our approach in Section III. Then, we present the technical details of our solution in Section IV, and later evaluate our implementation in Section V. Finally, we draw our conclusions in Section VI.

II. RELATED WORK

Existing BFT SMR frameworks [8]–[11] can not directly be used for rapid development of component-based, event-driven IoT applications. There are a few research works that touch the questions of either making SMR more viable for component-based architectures, or integrating replication mechanisms (though not necessarily BFT) into IoT environments.

An early system providing a platform for replicated software components is *Delta-4* [13]. A modern approach is to use containerised service replicas and manage them by *Kubernetes* [14]. Both approaches support a very vague definition of software component, whereas SORRIR has a well-defined component model that is here transparently enhanced by replication. Within the component-based cloud platform *COSCA* [15] active replication of components was supported by special communication measures [16]. Derecho [17] is a software library for creating structured cloud services that can be replicated using the SMR approach, but cannot cope with Byzantine failures. CEFIoT [18] explains a crash fault-tolerant data-as-a-stream architecture that allows placing processing components in both edge and cloud. It uses Apache Kafka for publish-subscribe messaging, so data streams can be buffered and easily replicated across a cluster. Furthermore, SMaRt-SCADA [19] implements BFT SMR into a SCADA system, which is also an event-driven system with publish-subscribe architecture. Further, a publish-subscribe interaction model for BFT SMR has also been studied for web applications [20].

To the best of our knowledge, our approach is the first effort to streamline and automate the whole process of integrating BFT SMR mechanisms in component-based IoT architectures.

III. BACKGROUND AND PRELIMARIES

We first explain our underlying system model (Section III-A), review some relevant background on state-machine replication, in particular BFT-SMART (Section III-B), which is the replication library we are using, and subsequently summarize both architecture and core characteristics of the SORRIR IoT framework (Section III-C).

A. System Model

Fault Model: Our method allows for a *tunable* fault model. On the granularity level of single components, the system designer can specify whether crash faults or Byzantine faults are assumed. The Byzantine fault model includes both crash faults as well as accidental (e.g., data corruption due to memory errors) or malicious (e.g., intrusions, tampering with devices) non-crash faults and thus may be a preferable choice when component replicas are deployed in dispersed administrative domains. Tolerating f faulty replicas needs a replica group of n replicas composing the replicated component with $n \geq 3f + 1$ (BFT) or $n \geq 2f + 1$ (CFT) respectively [8], and the system designer can choose f as a trade-off between fault tolerance and the replication overhead.

Network Model: We assume the partially synchronous system model [21], in which the system can behave initially asynchronous, but after some unknown time period, called *global stabilization time (GST)*, eventually some upper bound δ on communication delay holds. As soon as *GST* is reached the used replication library guarantees that progress can be made, i.e., messages can be delivered in total order. The communication is point-to-point, authenticated and reliable.

B. State-Machine Replication

State-machine replication is based on independent servers hosting replicas of the same service [5]. Client requests are delivered to all correct replicas in identical (total) order, so that their deterministic execution in each replica leads to equal replica states. There are multicast protocols that are able to support CFT and BFT. In case of BFT, the client needs to receive $f + 1$ identical answers from different replicas.

BFT-SMaRt is a framework library to develop SMR-based applications in Java [8]. It can be used for CFT and BFT, and provides appropriate consensus-based multicast protocols. In particular, we employ the decision forwarding extension [22]. Communication is authenticated and encrypted by using TLS, where each replica possesses an RSA key pair (sk_i, pk_i) to construct secure channels with all the other replicas.

C. The SORRIR Framework

SORRIR is a research framework for developing resilient IoT applications [12]. Applications are composed from software components that interact by sending and receiving events [23]. Components may be organised in a hierarchy so that some
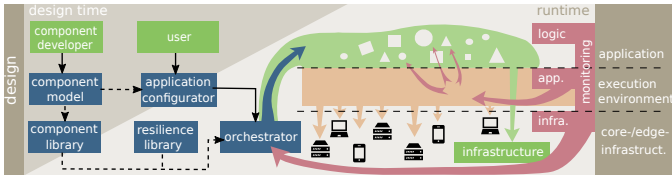
Fig. 1: High-level overview of the SORRIR approach [12].

(a) Logical SA. (b) Replication-Enriched SA.

Fig. 2: Component-Defined Software Architecture (SA).

can be grouped to form a composite component that can be used like any other component. Components define ports over which they can be wired together by connections, following a building-block concept. Each component can be attached to resilience mechanisms from a resilience library. The main idea is to keep application code as independent from resilience mechanisms as possible. Developers can build applications from components with an application configurator (LSA), and attach resilience mechanisms to each component, leading to a RESA. At runtime, an orchestrator takes care that components and resilience mechanisms are deployed on the available infrastructure of an IoT application. A symbolic overview can be found in Fig. 1.

Additionally, SORRIR has a component-dependent degradation concept. For the case of failures, developers can specify levels of degraded behaviour. SORRIR then automatically switches components into degradation levels, if certain conditions occur, e.g., dependent components fail or degrade [24].

In SORRIR, each component is modeled as a state machine. In the execution engine underlying the SORRIR framework

(1) events are executed step-wise in a loop until no longer a state transition can be applied and all components deployed on the SORRIR unit have reached their stable state,
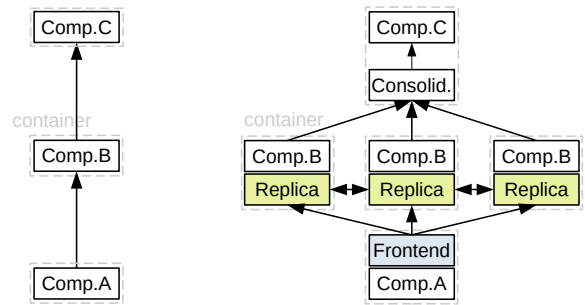
(2) the execution of any incoming event (that arrived on any arbitrary port) needs to complete before the execution of the next incoming event is allowed to start, and

(3) the execution of any event itself is deterministic which is a strict requirement for all components.

The SORRIR framework is implemented in TypeScript on Node.js and provides different protocols for transmitting events between remote IoT components, i.e,. HTTP, MQTT and BLE. Components are placed into so called *units* that are finally implemented as containers. Thus, units are the smallest distributable entities and can contain one or more components.

IV. THE SORRIR APPROACH TO SMR

Our approach for incorporating replication mechanisms for desired components consists of a streamlined process of solving individual challenges: First, we utilize a building-block principle (Section IV-A) that allows us to modularly swap-in *component replicas* within the logical software architecture (LSA). Second, we employ a resilience configuration (Section IV-B) that lets developers customize replication mechanisms for individual components. Third, we subsequently conduct an automated integration procedure of the specified replication mechanisms by transforming the LSA into a replication-enriched software architecture (RESA) that

contains instances of component replicas and dedicated output-consolidating components as well as instances of the BFT-SMART middleware (Section IV-C). In this step, components are also automatically re-wired to account for swapping in the newly generated component replicas and consolidators. Fourth, we use the k3s[1] Kubernetes distribution to manage the deployment of *units* (instances of SORRIR that execute components) on available hardware devices (Section IV-D).

A. A Building-Block Principle for State-Machine Replication

The core idea of our architecture is making the existing BFT replication library BFT-SMART [8] viable for practical IoT applications while satisfying the goals listed before.

Our solution employs modularity by utilizing a building-block principle in which different building blocks implement BFT replication mechanics. There is the FRONTEND (used for input dissemination), the REPLICA (framework instance of BFT-SMART used as a proxy to the actual component) and the CONSOLIDATOR (used for output consolidation). These components are inserted *on demand* into the architecture. Subsequently, connections between components are re-wired to account for both newly inserted and replicated components. To explain the interplay of these building blocks as well as their interaction with the SORRIR framework and the BFT-SMART library, we use in the following a simple example of three components COMP A, COMP B, and COMP C sending events from A to B and from B to C (Fig. 2a). In the corresponding RESA, COMP B is replicated (Fig. 2b). Fig. 3 shows the technical details of this configuration.

COMPONENT: Application components (in Fig. 3) are usually created by the developer. A necessary requirement is that they provide a deterministic step() function which is used to transition the component state (see SORRIR programming model [23]). No additional interfaces are required or needed. The communication and execution of components is handled by the underlying SORRIR framework ① that offers a variety of different communication technologies. Since COMP A sends an event to the replicated COMP B, the SORRIR framework delegates this event to an instance of FRONTEND ②, which disseminates it as input to all replicas.
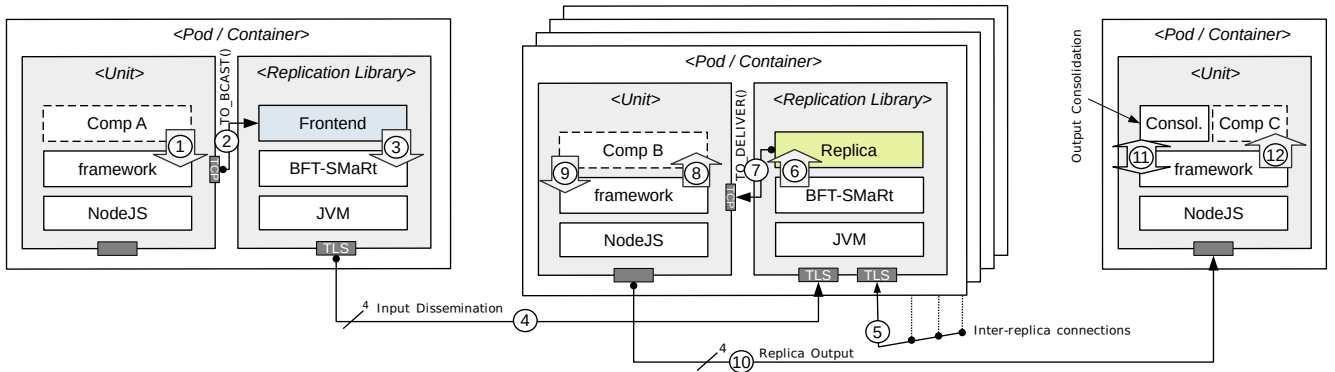
3

Fig. 3: A modular architecture for interaction between SMR building blocks that are instantiated on demand.

FRONTEND: The SORRIR framework treats the FRONTEND like a *communication technology* over which it can send events. Specifically, the FRONTEND acts as a local proxy when an application component is speaking to a replicated component, thus broadcasting the event to all replicas. Thus, it hides the fact that the event is sent to a replicated component. For the SORRIR framework, the forwarding of an event to a FRONTEND looks like ordinary event communication. The FRONTEND calls the `invokeOrdered()` method of the client-side BFT-SMART library ③ so that the event passes the *total-order multicast* layer. Each REPLICA instance of the replicated component COMP B receives the event ④.

REPLICA: The REPLICA implements the server-side of the BFT-SMART protocol. It receives messages from frontends and communicates with the other replicas in a three phase agreement pattern to decide a consensus instance. The BFT-SMART leader drains messages from its in-queues and puts them in a batch, which the leader subsequently proposes to all other replicas, thus determining the order of execution ⑤.

Once a consensus is decided, messages contained in the batch are delivered in order to the `executeOrdered()` method of BFT-SMART ⑥. The REPLICA implementation forwards ordered events back to the SORRIR framework ⑦, which subsequently pushes them one after another into the event queue of the receiving component ⑧ (e.g., a replica of the application component COMP B). In our example, all replicas of COMP B process the same event, and produce some outgoing event to be sent to COMP C, which is handled by the SORRIR communication system ⑨. This could result in a HTTPS POST request ⑩, and the same event is received multiple times by the framework of the receiver COMP C.

CONSOLIDATOR: The CONSOLIDATOR is a SORRIR-provided component that is automatically instantiated for any component that *receives* from a replicated component. Its purpose is to provide the abstraction of receiving events as if a non-replicated sender was involved. It consolidates the output of several replicas into a single input towards the receiving application component (here: COMP C). On a technical level, the consolidator checks for a certain threshold (depending on the configured fault model) of *matching* events ⑪ (e.g., same sequence number, same content), de-duplicates them,

and subsequently puts the *consolidated event* into the event queue of the intended receiver—in our example COMP C ⑫.

We provide two implementations, used for the BFT or CFT model, respectively. The BFT consolidator employs a strict comparison of an event's content, thus masking deviating output of faulty components. This modular approach allows developers to inject their own consolidator component to re-alize application-specific behavior such as consolidating non-deterministic input. This is useful, e.g., when reading from a replicated sensor and checking if obtained values are within a certain interval (thus relaxing the strict consistency of outputs as they are expected in traditional SMR). The communication and execution of a consolidator component is handled by the underlying SORRIR framework.

SORRIR framework: The SORRIR framework is the middle-ware in which all other building blocks are plugged into. It serves as an execution platform that steps through components, executing events until no further state transition is possible. Also, it handles the exchange of messages with other components, either deployed locally or located on a remote unit. It comes with an configurator that allows developers to wire components together, thus accelerating development.

B. Configuration of the Replication Mechanisms

A *resilience configuration* is a simple specification of which resilience mechanisms should be applied to which compo-nents. This configuration is thus a declarative description how to transform an LSA to a ReSA (cf. Fig. 2). Note, that we make SMR configurable on the granularity of individual components. This means we can easily employ different fault models, replica group sizes or consolidation techniques for different components within our software architecture.

The mechanism-specific properties specify the intended behavior, e.g., number f of faulty component-replicas to be tolerated, the employed fault model (Byzantine or Crash) and as an option an alternative consolidator component. In Figure 4, we show the example of a simple configuration which specifies that component COMP-B is being replicated to tolerate a single Byzantine component replica.

```
┌──────── resilience configuration ────────┐
│ {"components": [                          │
│   {"id": "Comp-B",                        │
│    "mechanisms": {                        │
│      "activeReplication": {               │
│        "enabled": true,                   │
│        "f": 1,                            │
│        "faultModel": "BFT",               │
│        "consolidator":  "BFTConsolidator" }│
│ } },   ...  ]}                            │
└──────────────────────────────────────────┘
```

Fig. 4: Replication can be configured for each component.

C. Automatic Integration of Replication Mechanisms

Automatically integrating replication mechanisms within a component-based, event-driven IoT framework demands to provide glue code for the framework to setup the additional building blocks (see Section IV-A) and rewiring components accordingly. We call this process SETUP-REPLICATION, the *transformation* of the LSA into the RESA. This transformation happens step-wise by iterating over the list of components, checking if a replication mechanism is specified, inserting necessary building blocks, and rewiring connections.

1) Insertion of Necessary Building Blocks: The transformation starts with the setup of additional building blocks on a unit. The differentiation which building blocks need to be inserted into some unit u follows a set of rules:

- CONSOLIDATOR: Insert for each component on u that receives from a replicated component (to be more concrete: for each replicated component it receives from).
- FRONTEND: Insert for each component on u that sends to a replicated component (in particular: for each replicated component it sends to).
- REPLICA: Insert iff a component replica instance of some replicated component is to be deployed on u (this depends on deployment decisions, which we will explain in more detail in Section IV-D).

In SORRIR the LSA specifies for each unit a locally deployed configuration CONF which consists of components and their connections. This CONF is expanded by our provided CONSOLIDATOR components and component replicas. Also, SETUP-REPLICATION creates BFT-SMART configuration files and bootstraps necessary FRONTENDs and REPLICAs.

After the setup procedure, the configuration still contains the old connections and does not yet consider the newly inserted building blocks. To complete the transformation to the RESA, rewiring of the components is necessary.

2) Rewiring Connections: In the next step, the overall architecture of every unit is being rewired to satisfy two goals: (1) input dissemination and (2) output consolidation. In particular, this concerns the following connections between *components* at each *unit*:

a) Input dissemination: If component c has an outgoing connection to some replicated component r then c's connection is re-configured to employ the *total-order multicast* communication technology. This means an event is disseminated to *all* (n_r) component replica instances using the FRONTEND $F_{c,r}$.

Further, if component c is a replicated component, then all of its incoming connections are configured to employ *total-order delivery* of events over the local REPLICA R_c instance that is connected to the SORRIR framework.

b) Output consolidation: If component c has an incoming connection from replicated component r then c is being re-wired to receive from the newly created consolidator component $Cons_{r,c}$ instead. Further, component $Cons_{r,c}$ is being wired to receive from *all* component replicas of r which creates n_r additional connections.

c) Interaction between two replicated components: We support the interaction between two replicated components. To illustrate the behavior (see Fig. 5), let us suppose that in an application a replicated component, defined as a set of state machines $B = \{b_1, .., b_n\}$, is wired over a directed connection to another replicated component $C = \{c_1, .., c_m\}$. We denote this connection by $B \rightarrow C$.
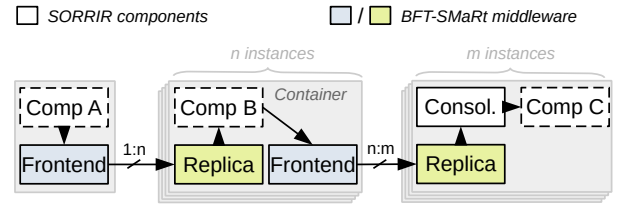

Fig. 5: Interaction between two replication groups.

For each i with $1 \le i \le n$ and each j with $1 \le j \le m$ a connection from the Fronted $F_{i,j}$ that is instantiated for b_i to a Replica instance R_j co-located with component c_j is created. Any event e that is sent through $B \rightarrow C$ arrives up to n-times at each of the m Replicas. After ordering, events are passed to the SORRIR framework, de-duplication is performed by the consolidators and subsequently $\forall j \in \{1..m\}$ only once e is delivered to each component instance c_j.

D. Automated Deployment

Our automated deployment approach is general enough for most IoT systems. The only restriction is that the system must be managed by *k3s* and that associated bundled platform components (i.e. *kubelet, kube-proxy*) are installed on the individual devices. *k3s* is a special *Kubernetes* distribution, to manage and execute containerized workloads in IoT edge environments. Today, Kubernetes is seen as the de-facto standard solution for container orchestration technologies [25] and helps us to manage an IoT *testbed* that is built from multiple heterogeneous devices like *Jetson TX2* or *Raspberry Pi 4b* single board computers. Further, k3s provides meta-information on devices like *device name, type, architecture* and custom tags like *location*, which can be exploited for the selection and the deployment of BFT component replicas.

Next, we describe the workflow from configuring an application that incorporates BFT replicated state machines up to the orchestration step complying to aforementioned boundary conditions associated with BFT replication. On a conceptual level, we describe how in the orchestration step the software

5

architecture as defined in the RESA gets decoupled from the decision of component placement on individual devices.

1) Configuration and Orchestration: We use the SORRIR Configurator[2] in the form of a REST API to configure the logical software architecture of the application. Precisely, we build the application by adding components and their connections and specify their assignment to units. Then, we add the BFT state-machine replication as resilience mechanism for some specific component (by referring to its identifier) to the resilience configuration (see Fig 4). Additionally, we can set the BFT parameters for n and f (see Section III-A).

In the next step, the SORRIR *Orchestrator*[3] uses this configuration to create specific resources in the k3s ecosystem. These are *deployments* that create *pods* in which the units get encapsulated. As k3s uses containerized software components, the units get deployed in the form of containers (see Fig. 3).

Kubernetes *Services* act as a frontend to individual pods, respectively units, in the sense that they route incoming messages to the individual pods. In the SORRIR case, we always map one service to one pod (i.e. unit). Eventually, resources named *configMaps* are created alongside. These serve as an abstraction to inject configuration files for our SORRIR application inside the containers.

2) Deployment of BFT-associated Components: The orchestrator deploys all different entities as shown in Fig. 3 encapsulated in *pods* on different devices. In particular, the orchestrator takes care that different software parts for BFT replication, e.g., FRONTEND, REPLICA and CONSOLIDATOR are correctly utilized and wired together. Apart of the connection inside a container, that is between a *<Unit>* and the *<Replication Library>* via a local socket, the orchestrator exploits built-in k3s functionality for setting up a proper IP-based communication system. This is done through an integrated DNS server and the created *services*, which are bound to individual pods. The *services* can be addressed by means of DNS resolvable names that are identical to the associated pod's unit name with the main task to forward received messages to its assigned pods. The service names are used within the BFT-SMART host configuration file which is deployed by the orchestrator on all components that are part of a specific replication group (either as invoker or replica). This ensures that component replicas can establish connections to each other by utilizing their service names, as well as the sender component can establish a connection to every component replica. The *deployments* are responsible for setting the container image URI and, if needed, for pointing to a *Kubernetes secret* that encrypts user credentials for accessing the container registry. Finally, *configMaps* enable the creation of configuration files, that get injected into the containers. For example, the component instance to be replicated and its parameters n and f are specified by this.

The orchestrator takes over control of the key generation and distribution. Since the authenticated and encrypted communication between involved component replicas as described in Section III-B uses symmetric secrets which are negotiated through the usage of *Diffie-Hellman* in TLS, we need the a-priori existence of asymmetric key material: The orchestrator generates an RSA key pair (sk_i, pk_i) for each component replica and deploys the private key sk_i only on the respective container i while all public keys are distributed to all containers that are part of the replication group.

Further, the orchestrator assures that replicas of the same component must not be placed on the same host. Violating this restriction, which is orthogonal to defined restrictions in the RESA, would break the assumption of independent faults.

V. EVALUATION

In this section, we first give a brief overview of our experimental setup and methods. In our first experiment we benchmark the ordering performance of the used replication library, BFT-SMART. Subsequently, we quantify how a replication-enriched system architecture performs compared to an architecture without replicas as a baseline to reason about the induced performance overhead. Then, we conduct an experiment in which we monitor the effects of node failure within the system. Finally, we discuss the obtained results.

A. Setup and Method

Setup. We use a small testbed that consists of 6 Raspberry Pi 4-B single-board computers which are connected over a 100 Mbps switch. Each device is equipped with at least 4 GB RAM and is running Raspberry Pi OS 64-Bit.

Benchmarking BFT-SMART. We first conduct benchmarks for the BFT-SMART middleware to reason about the performance of ordering events. For this purpose, we use the micro-benchmark suite of BFT-SMART. This benchmark employs synchronous clients with request-response interaction.

A *request* is a wrapper for an event (which acts as payload). Later, in Sections V-C and V-D, we only send requests (carrying events) one-way, towards the replica without caring about replies. To estimate the performance boundaries of ordering requests, we employ request payload sizes of 0 bytes and 1 kb. We deploy 4 replicas on 4 different Pis and launch an increasing number of clients on the other 2 devices to stimulate the system with higher workloads.

Benchmarking SORRIR. We developed a benchmarking application in the SORRIR programming model, which consists of three components: A *loadgenerator* maintains a fixed-size backlog of events that are sent simultaneously to the *processor*. Increasing the size of this backlog leads to higher workloads in the system. The processor is an empty implementation that only forwards all received events to the *reporter*. Moreover, the reporter measures and logs the system throughput and sends events back as feedback (used for controlling the backlog) to the loadgenerator: Essentially, for each event that is fed back, a new event is created. Using the feedback, the loadgenerator can calculate and log the end-to-end latency of events.

Benchmarking a ReSA in SORRIR. To evaluate the ReSA, we replicate the processor in $f = 1$ BFT mode, with component replicas being deployed on four separate Pis. In this setup,
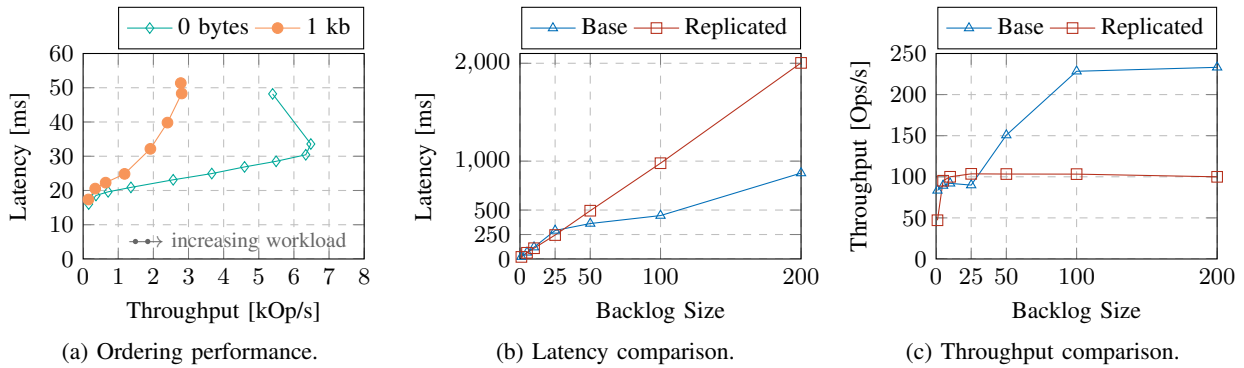
[2]https://github.com/sorrir/configurator
[3]https://github.com/sorrir/orchestrator

(a) Ordering performance.

(b) Latency comparison.

(c) Throughput comparison.

Fig. 6: Evaluation results of conducted benchmarks.
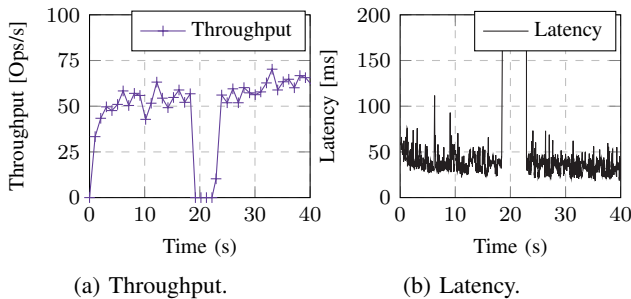

(a) Throughput.

(b) Latency.

Fig. 7: System performance under a leader failure.

a consolidator is deployed on the same unit as the reporter to receive and consolidate the output from the four replicas, which adds additional work on the reporter unit (all handled by a single Node.js thread). An event of the benchmarking application has roughly the size of 150 bytes (which acts as payload for the underlying BFT-SMART middleware).

B. Performance of Ordering Events

In our first experiment we evaluate how the underlying replication middleware, BFT-SMART, that is used to order events, performs on the Raspberry Pi 4-B devices. The main purpose is to understand whether the process of ordering events (which is necessary for replication) may induce a bottleneck in systems that make use of replication.

Measurements. We systematically increase the workload by increasing the number of synchronous clients (each client sends 5000 requests to the replicas) until we reach the saturation point. Throughput is measured at the leader replica, while latency is measured on the client side at a chosen client.

Results. We show our results in Figure 6a. We observe a peak throughput of $6,482$ operations per second (op/s) at a latency of 34 ms for 0 bytes requests and still up to $2,806$ op/s at a latency of 48 ms for requests with 1 kb payload. These numbers indicate that BFT-SMART replicas are able to handle the ordering of events in a sufficiently efficient way even on constrained ARM devices like the Pi 4-B.

C. Replication Overhead

In our second experiment we compare how a ReSA performs compared to a standard non-replicated system as baseline. This is to investigate if replication mechanisms may introduce a performance drop in the system.

Measurements. We systematically increase the size of the backlog to generate higher workloads in the system until we reach the saturation point. For this purpose we use increasing backlog sizes for our measurements. For each backlog size, we configure the loadgenerator to send a total of $k = 10,000$ events and use all events in the interval of [2,500, 7,500] to calculate the average latency and the average throughput. Latency is measured as the end-to-end latency as observed by the loadgenerator which receives feedback back from the reporter. Throughput is measured in the reporter component.

Results. We display our results in Figure 6b and 6c. When benchmarking the non-replicated SA as baseline, then we observe throughput up to 233 op/s (however at a latency of 877 ms). Generally, we observe that latency increases fast when increasing the workload in the system (see Figure 6b). The system with replicated processors displays a noticeable drop in peak performance: The saturation point is 104 op/s; after that only latency increases with higher workloads. Because of our observation from the last experiment, we think that this performance loss is not induced by the BFT-SMART middleware, but rather by the overhead of consolidating outputs as the consolidator runs in SORRIR's Node.js thread (which displays 100% CPU core utilization during the experiment). However, at the level of an acceptable system latency (below 100 ms) both systems achieve comparable throughput.

D. Leader Failure

In our third experiment we observe the impact of a replica failure in the replicated system. For this purpose, we crash the leader, which leads to a re-election after some timeout. Our expectation is that the system resumes working correctly.

Measurements. We use the same benchmark application as in the previous experiment and employ a workload of circa 50 op/s. During the experiment, roughly at time $t_{crash} = 18$ s we terminate both the Node and Java process at the leader replica.

Results. Figure 7b and 7a show the impact of this crash on system performance. During the re-election time of slightly above 4 s, the throughput drops to zero. As expected, we observe system performance bouncing back after a new leader is elected (roughly at $t_{new} = 22.8$ s). This also leads to an observable peak latency of 4456 ms (see Fig. 7b). During this process no events were lost, they were only delayed.

E. Discussion of Results

From the obtained results we got the following insights: First, employing BFT-SMART as replication middleware was an efficient design decision. Due to its implementation in Java, it is easily portable and works on a variety of devices. On the ARM-Cortex-A72-based Raspberry Pi 4B, we reached competitive throughput levels of over 2.8 kop/s even with 1 kb payload. Second, the replication overhead on throughput on the SORRIR component level is noticeable (see Figure 6c), which is caused by the SORRIR component-based implementation of consolidation, that seems to bottleneck the system. However, consolidating events in the SORRIR framework was a design decision that allows (1) developers to implement customizable consolidation behavior, e.g., relaxing strict consistency of outputs and (2) flexible instantiation of consolidators, even on devices that do not run a Java environment, or can not establish a TCP connections as they might be wired over Bluetooth. This design lowers the entry barriers for IoT components to interact with replicated components. For low system latency (below 100 ms) replicated, and non-replicated systems achieve comparable throughput, that we consider practical for many IoT applications. Third, we tested the effectiveness of our replication method to tolerate faults, concluding that the replicated system can swiftly return back to its intended behavior.

VI. CONCLUSION

BFT SMR allows IoT infrastructures to tolerate random data corruption (IoT devices may lack error correcting memory) or mask malicious behavior if components are deployed in dispersed (and potentially untrustworthy) administrative domains. However, using existing SMR libraries to develop an IoT-fit replication-enriched software architecture is far from trivial. We show how to automate the whole process of integrating BFT SMR into a component-based IoT execution environment. As solution, we introduce a building-block principle for SMR which flexibly glues replication functionalities to IoT components. Our solution allows for customizable BFT replication mechanisms to be seamlessly integrated in component-based, event-driven execution models like SORRIR. Furthermore, our solution also addresses the automated orchestration and deployment of component replicas on the available hardware landscape. Experiments conducted on a testbed of Raspberry Pis demonstrate the practicability of our solution.

REFERENCES

[1] T. Simon. (2017) Critical infrastructure and the internet of things. [Online]. Available: cigionline.org/sites/default/files/documents/GCIG%20no.46_0.pdf

[2] D. Terry, "Toward a new approach to IoT fault tolerance," *Computer*, vol. 49, no. 8, pp. 80–83, Aug 2016.

[3] C. Tsigkanos, S. Nastic, and S. Dustdar, "Towards resilient internet of things: Vision, challenges, and research roadmap," in *39th IEEE Int. Conf. Distr. Comp. Sys. (ICDCS)*, 2019, pp. 1–11.

[4] D. Ratasich, F. Khalid, F. Geissler, R. Grosu, M. Shafique, and E. Bartocci, "A roadmap toward the resilient internet of things for cyberphysical systems," *IEEE Access*, vol. 7, pp. 13 260–13 283, 2019.

[5] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comp. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[6] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Prog. Lang. Sys.*, vol. 4, no. 3, p. 382–401, Jul. 1982.

[7] C. Berger, P. Eichhammer, H. P. Reiser, J. Domaschka, F. J. Hauck, and G. Habiger, "A survey on resilience in the iot: Taxonomy, classification, and discussion of resilience mechanisms," *ACM Comp. Surv.*, vol. 54, no. 7, sep 2021. [Online]. Available: https://doi.org/10.1145/3462513

[8] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with BFT-SMaRt," in *44th Annu. IEEE/IFIP Int. Conf. on Dep. Sys. and Netw. (DSN), 2014*, 2014, pp. 355–362.

[9] A. Barger, Y. Manevich, H. Meir, and Y. Tock, "A Byzantine faulttolerant consensus library for Hyperledger Fabric," in *IEEE Int. Conf. on Blockch. and Cryptocurr. (ICBC)*, 2021, pp. 1–9.

[10] R. C. Aksoy and M. Kapritsos, "Aegean: replication beyond the clientserver model," in *27th ACM SOSP*, 2019, pp. 385–398.

[11] C. Stathakopoulou, T. David, and M. Vukolić, "Mir-BFT: highthroughput BFT for blockchains," *arXiv:1906.05552*, 2019.

[12] J. Domaschka, C. Berger, H. P. Reiser, P. Eichhammer, F. Griesinger, J. Pietron, M. Tichy, F. J. Hauck, and G. Habiger, "SORRIR: A resilient self-organizing middleware for IoT applications [position paper]," in *6th Int. Worksh. on Middlew. and Appl. for Internet of Things (M4IoT)*, 2019, pp. 13–16.

[13] P. Barret, A. Hilborne, P. Bond, D. Seaton, P. Verissimo, L. Rodrigues, and N. Speirs, "The Delta-4 extra performance architecture (XPA)," in *20th Int. Symp. on Fault-Tol. Comp. (ISFTC)*, 1990, pp. 481–488.

[14] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. Sá de Souza, "State machine replication in containers managed by Kubernetes," *J. of Sys. Arch.*, vol. 73, pp. 53–59, 2017.

[15] S. Kächele, J. Domaschka, and F. J. Hauck, "COSCA: An easy-to-use component-based paas cloud system for common applications," in *1st Int. Worksh. on Cloud Comp. Platf. (CloudCP)*. New York, NY: ACM, 2011.

[16] S. Kächele and F. J. Hauck, "COSCAnet-FT: Transparent network support for highly available cloud services," in *Int. Conf. and Worksh. on Netw. Sys. (NetSys)*, 2015, pp. 1–5.

[17] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman, "Derecho: Fast state machine replication for cloud services," *ACM Transactions on Computer Systems (TOCS)*, vol. 36, no. 2, pp. 1–49, 2019.

[18] A. Javed, K. Heljanko, A. Buda, and K. Främling, "CEFIoT: a faulttolerant IoT architecture for edge and cloud," in *IEEE 4th World Forum on Intern. of Things (WF-IoT)*. IEEE, 2018, pp. 813–818.

[19] A. Nogueira, M. Garcia, A. Bessani, and N. Neves, "On the challenges of building a bft scada," in *48th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 163–170.

[20] C. Berger and H. P. Reiser, "Webbft: Byzantine fault tolerance for resilient interactive web applications," in *IFIP Int. Conf. on Distributed Applications and Interoperable Systems*. Springer, 2018, pp. 1–17.

[21] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[22] C. Berger, H. P. Reiser, and A. Bessani, "Making reads in BFT state machine replication fast, linearizable, and live," in *40th Int. Symp. on Rel. Distr. Sys. (SRDS)*. IEEE, 2021, pp. 1–12.

[23] M. Tichy, J. Pietron, D. Mödinger, K. Juhnke, and F. J. Hauck, "Experiences with an internal DSL in the IoT domain," in *4th Int. Worksh. on Model-Driven Eng. for Internet of Things (MDE4IoT)*, 2020.

[24] A. Heß, F. J. Hauck, D. Mödinger, J. Pietron, M. Tichy, and J. Domaschka, "Morpheus: A degradation framework for resilient IoT systems," in *STAF 2021 Worksh. Proc.: Int. Worksh. on MDE for Smart IoT Sys.*, June 2021, pp. 105–114.

[25] A. P. Ferreira and R. Sinnott, "A performance evaluation of containers running on managed Kubernetes services," in *IEEE Int. Conf. on Cloud Comp. Techn. and Sci. (CloudCom)*. IEEE, 2019, pp. 199–208.